arXiv:2606.22283v1 [cs.AR] 21 Jun 2026

Apple Neural Engine

Architecture, Programming, and Performance

Spencer H. Bryngelson

School of Computational Science & Engineering
Georgia Institute of Technology, Atlanta, GA 30332, USA
ORCID 0000-0003-1750-7265

A11–A18 • M1–M5

June 2026

Apple Neural Engine: Architecture, Programming, and Performance.

Copyright © 2026 Spencer H. Bryngelson.

This work is licensed under a Creative Commons Attribution 4.0 International License (CC BY 4.0). You are free to share and adapt the material for any purpose, including commercially, provided you give appropriate credit. The full license is at `creativecommons.org/licenses/by/4.0/`.

Apple, Apple silicon, Core ML, and the Apple Neural Engine are trademarks of Apple Inc. This is an independent reference work and is not affiliated with, authorized by, or endorsed by Apple Inc. The private interfaces it describes are undocumented, unsupported, and may change without notice.

How to cite.

Spencer H. Bryngelson. *Apple Neural Engine: Architecture, Programming, and Performance.* 2026.

Abstract

The Apple Neural Engine (ANE) is the fixed-function matrix accelerator that has shipped in Apple systems-on-chip since the A11-class iPhone and iPad chips and the M1-class Mac chips, exposed to applications only through the Core ML model framework. This guide reports a reverse-engineered account of the engine, based on direct measurement on Apple silicon and static analysis of the private runtime, compiler, kernel driver, and firmware. It documents the datapath and the roofline that bound the engine's throughput and energy, the dispatch route that reaches it below Core ML, the compiler and on-disk program format, the weight-compression scheme, and the kernel driver, firmware, and command protocol beneath them. The account covers the A11 through A18 and M1 through M5 families, with per-chip target tables and an operation-by-device matrix; the direct measurements are on the M1 and M5. Claims are labeled as measured, decompile-derived, or predicted, and the methodology and open questions are recorded. The direct route is callable from ordinary user space but remains undocumented, unsupported, and version-fragile; it is intended for measurement, research, and on-device work, not for shipping software, where Core ML remains the supported path.

Contents

Introduction 1

Part I. The Machine 3

1 What the ANE is 4

2 Execution model 8

3 Numerics 13

4 Capability surface 21

Part II. Reaching the ANE 28

5 Software stack 29

6 Dispatching without Core ML 37

7 Weights and compression 41

8 Entitlement boundary 47

Part III. Performance and Fit 51

9 Roofline 52

10 Power and efficiency 58

11 ANE, GPU, and CPU 64

12 Across the chip family 69

Part IV. Workloads 74

13 Vision, convolution, and encoders 75

14 LLM case study 82

15 Training on the engine 87

16 Numerical and scientific computing 92

Part V. Practice 97

17 Model-design rules 98

18 Optimization and the cost model 104

19 Pitfalls and limits 110

Interlude. Below the API: how the engine works 116

Part VI. The Silicon 117

20 Datapath and MAC geometry 118

21 Memory hierarchy 126

Part VII. The Toolchain and Encoding 132

22 Compiler 133

23 Program and container format 140

24 HAL and capability gates 151

25 Compression internals 158

26 Hidden layers and direct netplist authoring 165

Part VIII. System Internals 170

27 Kernel driver and IOKit ABI 171

28 Address translation and the DART 180

29 Firmware 187

30 Host-to-firmware command protocol 196

31 Power and thermal 203

32 Security and isolation 208

33 Telemetry and hardware counters 212

Part IX. Cross-Silicon Reference 217

34 Cross-silicon targets 218

35 Per-family code generation 222

36 Predicted upper tier 230

Back Matter 237

Methodology 238

Open questions 242

Statements 245

Appendices 246

A Operation-by-device matrix 247

B Hidden-layer catalog 258

C Decoded reference tables 263

D Glossary 278

E Provenance 283

References 295

Introduction

The Apple Neural Engine (ANE) is among the most widely deployed machine-learning accelerators in existence and among the least documented. Apple has built it into every Apple system on chip since the A11 in 2017 and the M1 in 2020, so nearly every iPhone and iPad sold since and every Apple-silicon Mac contains one. Apple's active installed base passed 2.5 billion devices in early 2026 [AppleActiveDevices2026], the large majority of them on silicon new enough to include a Neural Engine. There it runs the on-device vision, speech, and language models that the operating system and its applications rely on. Yet of the programmable engines on an Apple chip it is the most opaque: there is no public instruction set, no driver interface, and no documented way for a program even to confirm that a computation ran on it. An application reaches it only indirectly, through the Core ML model framework, which treats the engine as one option behind a placement hint. Its datapath and numerics, performance and energy envelope, and compiler, program format, kernel driver, firmware, and command protocol are all undocumented.

This guide reports a reverse-engineered account of the engine, from the silicon datapath up to the system interface. Two lines of evidence check each other: direct measurement on Apple silicon and static decompilation of the private runtime, compiler, kernel driver, and firmware. The engine is reachable directly, below Core ML and from an ordinary unprivileged process. The compiler lowers a graph to the engine's own program format, the runtime loads and dispatches it without the model framework, and the operations the compiler accepts need no special entitlement. That direct route is what makes the rest measurable. It is undocumented, unsupported, and fragile across operating-system updates, and it is meant for measurement, research, and on-device experimentation, not for shipping software, where Core ML remains the supported path.

Performance begins with the roofline. On the M1 the engine holds about 12 fp16 TFLOP/s of compute against a DRAM-bandwidth ceiling. The roofline has a ridge point near 141 FLOP per byte, a 2 MB working-set threshold, a 0.23 ms floor under any single dispatch, and efficiency near 0.37 picojoules per FLOP at the compute optimum. On a 256-channel 3x3 convolution it runs about 3.8 times faster than the same chip's GPU and 9 times more energy-efficient. The roofline pairs the engine's throughput ceilings with its measured power.

Reaching the engine is not the same as running an arbitrary graph on it. The operations the engine executes are distinct from the ones a capability bit only advertises. A feature attested in the hardware tables or accepted by the compiler frontend counts only once a compile-and-run confirms it, and several advertised operations, three-dimensional convolution among them, never lower to the engine at all. Weight compression on the direct path cuts bandwidth, not only stored size. On the unentitled engine, int4 lookup-table weights run about 2.37 times faster than fp16, and structured sparsity 1.55 to 1.64 times faster at 0.43 times the bytes.

Beneath the datapath lies the private stack the engine runs on: the compiler and its backend dialect, the on-disk program and container format, the kernel-driver IOKit ABI, the unencrypted firmware and its ninety-three-command host protocol, and the address-translation path that maps host buffers into the engine. Twenty-eight compiler targets are decoded as well, spanning the A11 through A18 and M1 through M5 families, with the rule that maps each M-series part to its internal H-series identity, the per-family operation floors, and an operation-by-device matrix. The cross-generation predictions checked on a second physical chip held, and a seeded training run reproduced across generations to within 0.001 in final accuracy.

Related work

Reaching the engine below Core ML has both concurrent and prior precedent. The closest is [Orion2026], concurrent work that characterizes and programs the engine for large-language-model training and inference. A line of community reverse engineering precedes it: the tinygrad project recovered the HWX program format and the AppleH11ANEInterface IOKit path [tinygrad], Yoon's `ane` project built a reverse-engineered Linux driver and the `anecc` compiler [eilnANE], and Singh's recent series decodes the M4 engine [Singh2026].

Runtime and application work builds on that access. The `libane` native runtime exposes the engine to ordinary programs [libane], whisper.cpp reaches it through the Core ML-routed path [whispercpp], and the long-maintained catalog of Hollemans [Hollemans] and Apple's own engineering note on deploying Transformers [AppleANETransformers] collect what is publicly known. Several further repositories document direct access and runtime APIs [CommunityANE], and an earlier thesis treats decoupling on-device intelligence from the application on IoT hardware [Plyenkov2019].

The performance treatment draws on the roofline literature. It is organized around the roofline model [Williams2009] and its later refinements: the energy roofline [Choi2013], the cache-aware roofline [Ilic2014], the instruction roofline [Ding2019], hierarchical roofline analysis [Yang2020], and its application to machine-learning accelerators [Verhelst2025]. It sits within a wider body of work that measures neural accelerators and edge inference: the in-datacenter analysis of the first tensor processing unit [Jouppi2017], smartphone deep-learning benchmarks [Ignatov2019], edge-platform inference benchmarking [Jayanth2024], energy-and-time roofline studies on edge accelerators [Prashanthi2025], NPU energy efficiency on microcontrollers [Fanariotis2025], LLM inference trade-offs across mobile NPU and GPU under sustained load [Tummalapalli2026], and on-device LLM roofline benchmarking [Bi2026]. Work closest in workload studies heterogeneous on-device LLM inference: fast NPU inference [Xu2025], GPU-NPU hybrid serving for long context [Moon2025], and mobile-SoC characterization for heterogeneous execution [Chen2025]. On Apple silicon specifically, Benazir and Lin study mixture-of-experts inference on the NPU [Benazir2026], Hübner and colleagues evaluate the M-series for HPC efficiency [Hubner2025], and the ML.ENERGY project [Zeus2025] treats the programmatic energy measurement the power figures here depend on.

This guide does not claim to be first to reach the engine. The direct route it measures is released as the open-source ANEForge runtime [ANEForge2026], which this guide accompanies. It differs from prior work in scope and in method: it covers the full stack from the fp16 datapath down to the firmware and command protocol, not a single access path. Beyond the access result it adds a reachable-operation census, unentitled weight streaming, a validator-based prediction of an operation's reachability from a callable compiler validator, and an account of how the engine's compiler fuses a whole graph into a single program. Earlier Apple-silicon rooflines are GPU-only; this treatment covers the engine, pairs it with measured power, and gives a batched-serving energy crossover.

The references give full bibliographic detail for these works.

How to read this guide

A reader can take the two halves independently. The front half, Parts I through V, covers using the engine: what the hardware is, how to reach it, how it performs, and how to fit real workloads to it. The back half, Parts VI through IX, covers the architecture and system internals beneath that surface, down to the firmware and command protocol. The appendices collect the operation-by-device matrix, decoded reference tables, glossary, and provenance record, and the references close the guide.

Every substantive claim has one of three evidentiary marks. A measured claim was observed directly on Apple silicon, primarily the M1 and M5. A decompile-derived claim was read out of the disassembled runtime, compiler, kernel driver, or firmware. A predicted claim was inferred from a model or a per-chip table and is not yet confirmed on silicon. Appendix E records the mark on every claim, the methodology describes how the engine was reached, and the open questions name what remains unmeasured.

Part I

The Machine

01 What the ANE is

The fixed-function fp16 datapath, its wide accumulator, and the route below Core ML.

02 Execution model

Compile once and dispatch many; the per-call latency budget and the program cache.

03 Numerics

fp16 end to end, the wide accumulator, saturation, and where precision breaks down.

04 Capability surface

Which operations the engine runs, against which a capability bit only advertises.

1 What the ANE is

SUMMARY

The Apple Neural Engine is a fixed-function fp16 matrix accelerator with a wide accumulator, reachable directly below Core ML. The fp16 product path and wide accumulator explain much of the engine's numerical behavior and contribute to its efficiency. It is faster than the GPU on compute-bound vision and convolution, 3.8 times faster and 9 times more efficient on a 256-channel 3x3 convolution; the GPU leads only on bandwidth-bound decode. The overhead-isolated compute slope reaches near 12 fp16 TFLOP/s against 85 GB/s of DRAM bandwidth, with a 2 MB on-chip working-set threshold as the primary design limit.

The Apple Neural Engine is a fixed-function matrix accelerator built into every recent Apple system on chip, from the A11 and the M1 onward. It runs the feed-forward neural networks of on-device perception, vision and speech, at low power, and leaves the CPU and the GPU free for the rest of the system. Apple distributes it in volume and documents almost none of it. The engine is beside the CPU and the GPU on the same chip, where the three share one pool of DRAM, as figure 1.1 shows.

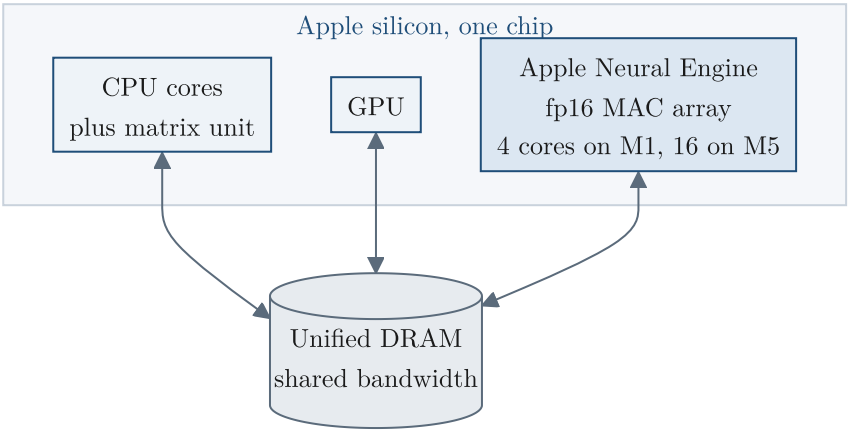

Figure 1.1. The Apple Neural Engine beside the CPU and GPU on one chip, sharing unified DRAM.

1.1 Reachable surface

The public way to use the engine is Core ML [AppleCoreML], whose load-and-predict call shape Listing 1.1 shows.

Listing 1.1. The Core ML load-and-predict path, where the compute-unit field is a placement hint rather than a guarantee.

```
// The public path: ask for the engine, but Core ML still places the work.
let config = MLModelConfiguration()
config.computeUnits = .cpuAndNeuralEngine  // a hint, not a guarantee
let model = try MLModel(contentsOf: url, configuration: config)
let out = try model.prediction(from: input)  // planner splits ops across CPU, GPU, ANE
// The caller is not told which device ran each segment.
```

A cost-driven placement planner segments a model handed to Core ML across the CPU, GPU, and ANE. An eligibility check decides which operations the engine can accept, and a roofline and transfer cost model decides where each segment runs. The planner is opaque. A caller does not choose the device and is not told which device ran the work.

A direct route reaches the engine without Core ML. The same private Espresso runtime that Apple's own dispatchers use is callable from ordinary user space. It compiles a network to the engine's program format, loads it, and drives an execution stream, with no placement planner in the path and no special entitlement for the operations the compiler accepts. This route makes the engine a target a developer can address on purpose rather than a scheduling hint. It is not a supported or App-Store-safe path; see the status note in the front matter. Part II covers it in full.

The stack from a graph to the silicon is a fixed set of layers, and the direct route enters one level below Core ML at the runtime, as table 1.1 gives layer by layer.

Table 1.1. The software and firmware stack from a compute graph to the silicon, with the role of each layer.

Layer	Role
Core ML, the GPU graph framework, the direct runtime route	three independent front ends
Espresso and the runtime	the universal compile, load, and execution-stream layer
the engine framework and its daemon	program lifecycle, brokered over cross-process messaging
the kernel driver	the coprocessor endpoint, mailbox transport, signed program load
the firmware	the engine's own real-time operating system, the dispatch loop, the data-movement and power control
the silicon	the multiply array, the on-chip memory, and the address-translation unit

The compile path requires the modern intermediate language, which the compiler accepts only from the A13 and M1 generation up. The pre-A13 parts have no path through this toolchain, which is why the M1 is the practical floor for addressing the engine directly.

1.2 One fact that explains the machine

The products are fp16 while the accumulator is wide, of fp32 class, and that property accounts for most of what the engine does. The datapath multiplies in fp16 end to end: fp16 inputs, fp16 weights, fp16 outputs. The frontend accepts fp32, int32, and bf16 type annotations, but the backend does not implement them. The datapath reconstructs compressed weights to fp16 before they reach the multiplier.

The running sum is not fp16. Input tiles round to fp16 on the way in, the output port rounds to fp16 on the way out, and the accumulation between those two points is held in a wide accumulator. Representable sums thus come back near exact. A reduction of sixteen thousand ones is bit exact, where a naive fp16 running sum would stall near two thousand once the partial total exceeds the spacing of its own increments. A cancellation

probe settles the question: a sum of a large value, its negation, and one, taken near sixteen thousand, returns the one intact, which a fp16 running sum would have swallowed. The sum is physically wider than fp16.

The wide accumulator is what suits the engine to vision, audio, and encoders. Those workloads are convolutions, matrix multiplies, and normalizations whose partial sums stay in range and whose results are representable, so the accumulator holds their precision. The same arithmetic explains where a transformer decoder loses precision in fp16. The loss comes from the per-product fp16 rounding of the inputs and weights under heavy cancellation, in the down-projection in particular, not from the accumulator. The accumulator is wide enough; the inputs to it are already quantized. A cancellation-heavy step has no fp16-safe form on the engine and needs a wider anchor on the CPU or the GPU.

The fp16 datapath also accounts for the power efficiency. A narrow multiply and a fixed-function pipeline move and compute far fewer bits per result than a general-purpose vector unit.

1.3 Two core counts

The engine reports two different core counts, and only one is the throughput unit. The advertised 16-core figure is the count Apple markets, 16 on the M1, exposed at runtime as the device property for the number of engine cores. The architectural figure is the core count the compiler tiles work across and the cost model scales by, read from the hardware-abstraction-layer offset `0x238`, which is 4 on the M1. The core count is 4 on the M1 base part, 8 on the M1 Pro and Max, and 16 on the M5. Each core emits up to 4 output channels per cycle in the default fp16 path. The M1 thus reaches about 16 output-channel multiply-accumulates per cycle in fp16, and int8 doubles the lanes to reach about 32, consistent with the overhead-isolated compute slope near 12 fp16 TFLOP/s.

1.4 Where it leads

On compute-bound work the engine is faster than the GPU outright. A 3x3 convolution at 256 channels runs about 3.8 times faster than the same work on the GPU and about 9 times more energy efficient, and batched matrix multiply is more efficient at every batch size, as table 1.2 records for both workloads.

Table 1.2. Engine versus GPU speed and efficiency on the convolution and batched matrix multiply workloads.

workload	engine vs GPU speed	engine vs GPU efficiency
3x3 convolution (256 channels)	3.8x faster	9x more efficient
batched matrix multiply	faster below N of 2048	more efficient at every batch size

The case where the GPU leads is real but limited: it holds for bandwidth-bound autoregressive decode, where the work is moving weights rather than computing on them. The measured envelope on the M1 fixes the scale. The overhead-isolated compute slope reaches near $P \approx 12$ fp16 TFLOP/s against a DRAM bandwidth of $B \approx 85$ GB/s, at about 0.5 pJ per FLOP sustained, near 0.37 at the compute optimum. The roofline ridge point, the arithmetic intensity above which a kernel is compute-bound rather than bandwidth-bound, is at $I^* = P/B \approx 141$ FLOP per byte. A hard 2 MB on-chip working-set threshold is the primary design limit: a kernel whose live tiles exceed it stalls on off-chip streaming rather than running on the multiplier. Newer chips scale the core count and the clock, but the form of the roofline and the fp16 datapath apply across the family.

1.5 Reference: the M1 envelope

Table 1.3 collects the M1 envelope figures, the two distinct core counts, and the family core-count span.

Table 1.3. The M1 envelope figures, the two distinct core counts, and the family core-count span.

Quantity	Symbol	M1/H13 value
Compute roof	P	12 fp16 TFLOP/s
DRAM bandwidth roof	B	85 GB/s
Energy per FLOP		0.5 pJ sustained, 0.37 pJ optimum
Roofline ridge point	I^*	141 FLOP/byte
On-chip working-set threshold		2 MB
fp16 maximum finite magnitude		65504
Advertised 16-core figure, M1		16
Core count, M1 (HAL `0x238`)		4
Core count, M1 to M5		4 to 16
Output channels per cycle per core, fp16		4

2 Execution model

SUMMARY

The engine runs as an autonomous coprocessor: a network compiles once into the engine's program format, then dispatches many times against that one program. The compile phase is costly and belongs out of the hot loop; the dispatch phase binds operands, posts one mailbox command, and waits. The compiled program is a static graph the hardware walks, so control flow is fixed at compile time and cannot depend on runtime values. A buffer can stay resident across dispatches, so a key-value cache or optimizer state persists in place without a host round-trip.

2.1 Compile once, dispatch many

Work reaches the engine in two phases whose costs are far apart. The public surface exposes only a load-and-predict view of this split [AppleCoreML]; the two phases below are the mechanism beneath it. The compile phase turns a network into the engine's program format: the compiler lowers the operation graph, lays out weights for the streaming datapath, and produces a loadable program. The dispatch phase runs that program against a set of operands and reads back the output.

The compile phase is costly. It runs the full lowering and layout pipeline, writes a program to a content-addressed cache on disk, and the first dispatch of a freshly compiled program pays a further one-time cost to produce the loadable hardware form. The dispatch phase binds operand buffers, hands the program to the engine, and waits for completion.

Table 2.1 sets the two phases against the runtime call surface, with when each runs, its cost, and the calls in each.

Table 2.1. The compile and dispatch phases, when each runs, its cost, and the runtime calls in each.

Phase	When it runs	Cost	Example calls
Compile	once, ahead of the loop	full lowering and layout, written to disk	compile the network, open the program library, retain the program function, load the function for execution
Dispatch	once per frame, token, or request	bind operands, post command, wait	encode the operation, execute the stream, read the output

The runtime exposes the two phases as distinct call families: Listing 2.1 gives the compile-side calls, which run once, and Listing 2.2 the dispatch-side calls, which run on every request.

Listing 2.1. The one-time compile-side calls that turn a network into a loaded program.

```
/* once, out of the hot loop */
e5rt_e5_compiler_compile(compiler, model_path, options, &library);
e5rt_program_library_retain_program_function(library, name, &function);
e5rt_program_function_load_for_execution(function);
```

Listing 2.2. The per-call dispatch-side calls that encode and run the loaded program.

```
/* once per frame, token, or request */
e5rt_execution_stream_encode_operation(stream, op);
e5rt_execution_stream_execute_sync(stream);
```

Chapter 6 gives the full create-bind-encode-execute sequence, including the compute operation and the input and output port binding.

The compile phase does not produce one program form but three, lowered at successive layers, which table 2.2 names with what each is and where it lives.

Table 2.2. The three program representations a compiled network passes through, from the cached bundle to the firmware load format.

Representation	What it is	Where it is
the bundle	a flat-buffer container whose fused graph collapses to a three-op chain, a cast in, the engine inference, a cast out, with a parametric per-op descriptor whose size does not grow with the tensor	the on-disk content-addressed cache
the program image	a signed executable the kernel loader parses, magic `0xbeefface`, with a register-write text section, a weight section, constants, and scratch	materialized below the host boundary at load
the firmware container	the firmware's load format, a three-level container keyed by program identity, with generic, kernel, text, operation, and procedure sections	resident on the engine

The loader expands the parametric descriptor in the bundle into the explicit register-write program below the host boundary, so the shape-specific program appears in no host buffer. This is why the compile cost is paid once on disk and a further one-time cost on the first dispatch, when the loadable hardware form is produced.

2.2 Host drives an autonomous coprocessor

The engine is on the system on chip with its own controller and its own local memory. The host never reads or writes the engine's compute registers and never steps it instruction by instruction. It hands over a compiled program and the operand buffers, signals the engine through a command mailbox, and waits for a completion notification.

The mailbox is a ring buffer of command records shared between the host and the engine's controller. To start work, the host writes a command that names the program and its operands and rings a doorbell. The controller picks the command up, drives the datapath through the work, and signals completion back across the same channel. Operand buffers are mapped through the engine's own address translation unit, so the engine reads inputs and writes outputs directly in memory the host prepared. The host waits on the completion signal; it does not supervise the computation.

Once the command is posted, the host CPU is idle with respect to that work and can prepare the next operands or post more commands. Several programs from several processes can have work outstanding at once; the engine time-shares itself across them without host involvement.

Each inference is a procedure call that contains its operand buffers, its optional wait, signal, and shared events, and a set of task-descriptor partitions. The firmware pushes one engine request per partition onto a

task queue, and the engine preempts at task-queue granularity with a mid-flight abort, so a higher-priority program does not wait for a lower-priority one to drain. The host posts a command through a header that names the command, its size, a priority in the range 0 to 7, and the program, process, and procedure identities. A secure mode can claim the engine exclusively by quiescing and power-cycling around the boundary, which is how protected-content work is isolated.

A single dispatch stream keeps one operation in flight reliably, but overlapping two or more streams in one process is the unfinished path on the M1. The completion event for the first stream signals and its waiter returns, while the completion notification for a second concurrently overlapped stream does not fire, so its waiter blocks. The runtime has the controls that would change this, a low-latency event path and a submit call with a timeout, but the default serialized path is the sound one. A caller that needs aggregate throughput runs independent streams rather than overlapping them in one, since sequential decode cannot overlap with itself in any case.

2.3 What one dispatch costs

A single dispatch is governed by the cost of getting to the engine and back, not by the engine compute itself. Measured live on the M1 with a read-only trace, a tiny graph of a 3-by-3 convolution from 8 channels to 8 with padding 1, then a relu, then a mean, runs in a hot loop of about 2000 iterations. Each call costs about 190 microseconds of wall-clock time. About 98 percent of that is software and firmware dispatch overhead rather than engine compute.

Table 2.3 breaks the per-call budget into its stages, with the cost of each from the user-space binding through the firmware round trip to kernel-side completion.

Table 2.3. The per-call latency budget of a single small dispatch on the M1, from the user-space binding through the firmware round trip to kernel-side completion.

Stage	Cost
User-space binding, runtime, and host fp16 input and output copy	about 25 microseconds
Building the firmware request	about 16 microseconds
Firmware kick: the doorbell, which returns asynchronously	about 2 to 3 microseconds
Firmware round trip	about 130 microseconds
Kernel-side completion processing	about 10 microseconds

The kernel user-client submit, the `ANE_ProgramSendRequest` external method, takes about 163 microseconds from entry to return. Completion is interrupt-driven: an interrupt handler fires about twice per inference, and the asynchronous-message completion path is not used on this synchronous small-model path. During that firmware round trip the firmware wakes, picks the command off its queue, executes, and signals back. It is not sub-splittable from user space with read-only tools, because the firmware per-task-descriptor latency profiler is gated.

2.4 Submissions serialize at one in flight

The driver keeps at most one firmware command in flight at a time, a single-pending-queue scheduler. Two concurrent submission threads thus serialize, measured at 1.04 times, so the round trip is not hidden by overlapping requests.

2.5 A walked graph, not a decoded stream

The compiled program is a static graph of work segments that the hardware walks, not a stream of instructions that a processor decodes. There is no program counter to read and no microcode to dump. The compiler fixes the order and shape of every segment ahead of time, and the engine's controller advances through that fixed structure, programming the data movement engines and the multiply array for each segment in turn.

A direct consequence is that control flow must be static. The graph has a shape decided at compile time, so the work the engine performs cannot depend on values computed during the run. Data-dependent branching does not execute on the engine: no path through the walked graph selects itself from a runtime value. A network that needs such a branch must resolve it on the host or restructure it so the branch becomes a fixed computation, for example a mask applied to both sides rather than a choice between them. A loop with a fixed trip count unrolls into a fixed graph and is admissible; a loop whose length depends on the data is not.

The same property explains the absence of a readable instruction trace. The unit the engine executes is a precompiled segment of data movement and multiply work, parameterized by operand addresses and shapes. The fine-grained register program that drives the silicon is materialized below the host boundary at load time and never appears in a host buffer. The program format is the subject of a later chapter.

2.6 State kept resident across dispatches

A dispatch does not have to round-trip every tensor through the host. The engine can keep a buffer resident in its working set across calls, so a value produced by one dispatch is available to the next without a copy back to the host and a copy forward again. The mechanism aliases an output buffer of one call to an input buffer of the following call, so the data persists in place between dispatches.

The aliasing reuses the same port-binding calls that chapter 6 uses for ordinary I/O. One buffer object is bound to the output port of the operation and to the input port of the next dispatch, so the dispatch that writes the held tensor and the dispatch that reads it name the same memory, as listing 2.3 shows call by call.

Listing 2.3. Keeping a state buffer resident across dispatches by binding it to both an output port and the next step's input port.

```
/* One buffer object holds the resident state (a KV-cache or optimizer state). */
e5rt_buffer_object_alloc(&state_buf, nbytes, /*type=*/0);

/* Bind it to BOTH the output port that writes the new state ... */
e5rt_execution_stream_operation_retain_output_port(op, "state_out", &out_port);
e5rt_io_port_bind_buffer_object(out_port, state_buf);

/* ... and the input port that reads it on the next step: same buffer object. */
e5rt_execution_stream_operation_retain_input_port(op, "state_in", &in_port);
e5rt_io_port_bind_buffer_object(in_port, state_buf);

for (int step = 0; step < n; step++) {
    /* Send only the small per-step input (a token or a minibatch). */
    /* schematic: write the step input into its own bound port, not state_buf */
    e5rt_execution_stream_encode_operation(stream, op);
    e5rt_execution_stream_execute_sync(stream);
    /* state_buf now holds the updated state; it is never re-sent from the host. */
    e5rt_execution_stream_reset(stream);
}
```

Any multi-step computation that holds state uses this mechanism. An autoregressive decoder keeps its key and value cache resident, so each step appends the new entry in place rather than restreaming the whole cache through the host every token. A training loop keeps its optimizer state resident across steps for the same reason. The host sends only the small per-step inputs, a new token or a minibatch, and reads the

resident buffers back at a checkpoint. The large held tensor stays on the engine instead of crossing the host boundary twice per step.

The output-to-input aliasing is the reachable face of the firmware's data-chaining subsystem, which keeps an output set resident and chains it as the next call's input. This is the route on the M1, rather than the engine's native persistent-state operations. The M1 task descriptor has no in-place resident-state data-movement engine: the encoders for that path are stubbed with the message that the data-movement form is not supported on this architecture, and the engine's native state operations are rejected when the program compiles. The held tensor is thus written as one call's output and read as the next call's input through one bound buffer, with the positional write done as a standard masked update against a small position vector sent each step. A resident accumulator built this way returns 1, 2, 3, 4 over four dispatches with no host re-send, and a resident cache fills each slot in place across steps, both confirmed on the M1.

2.7 Compile out of the loop, dispatch inside it

The cost split dictates the loop structure: compile once before the loop, then dispatch the loaded program against fresh operands on every iteration. A content hash keys the compiled program, so recompiling an unchanged network is a cache hit rather than a second lowering pass.

Listing 2.4 compiles and loads once, then dispatches the loaded program per frame, per token, or per request.

Listing 2.4. The compile-once, dispatch-many loop, with compile and load above the loop and bind and dispatch inside it.

```
/* Once, out of the hot loop: compile and load. A cache hit skips the lowering. */
e5rt_e5_compiler_compile(compiler, model_path, options, &library);
e5rt_program_library_retain_program_function(library, fn_name, &function);
e5rt_program_function_load_for_execution(function);
e5rt_precompiled_compute_op_create_options_create_with_program_function(&op_opts, function);
e5rt_execution_stream_operation_create_precompiled_compute_operation_with_options(&op, op_opts);
e5rt_execution_stream_operation_retain_input_port(op, "x", &in_port);
e5rt_io_port_bind_buffer_object(in_port, in_buf);   /* bound once, refilled per frame */
e5rt_execution_stream_create(&stream);

for (int frame = 0; frame < n_frames; frame++) {
    /* Inside the loop: write the next frame, then encode, execute, reset. */
    e5rt_execution_stream_operation_prepare_op_for_encode(op);
    e5rt_execution_stream_encode_operation(stream, op);
    e5rt_execution_stream_execute_sync(stream);
    e5rt_execution_stream_reset(stream);
}
```

3 Numerics

SUMMARY

The datapath is fp16 end to end with a wide accumulator of fp32 class, so representable sums come back near exact. Convolution, matrix multiply, and normalization keep their precision; a cancellation-heavy step such as the transformer down-projection loses it to per-product fp16 input rounding, not to the accumulator, and needs a wider anchor off-engine. The fp16 ceiling is 65504, and a width-axis slice with a nonzero begin offset applies a fixed gain of sixteen, so a fill above 4094 overflows silently to infinity on the M1 and the A14. Activation lookup tables are accurate to about half a unit in the last place, but they coerce a NaN to a clamp value and have a small origin bias a bit-exact oracle must model.

The Apple Neural Engine computes in fp16 with a wide accumulator. That property, stated in chapter 1, decides which computations keep their precision on the engine and which lose it.

3.1 fp16 datapath

The multiply array is fp16 end to end. A fp16 value decodes from a sign bit, five-bit exponent, and ten-bit fraction as

$$x = (-1)^s \, 2^{e-15} \left(1 + \frac{m}{2^{10}}\right),$$

with a maximum finite magnitude of 65504. Inputs are fp16, weights are fp16, outputs are fp16. The frontend accepts fp32, int32, and bf16 as type annotations, but the backend does not implement them, so those annotations do not reach the silicon as wider arithmetic. Bias is the single exception in the descriptor: it may be held as fp32, but it is added in the fp16 datapath. The datapath reconstructs compressed weights to fp16 before they reach the multiplier; it dequantizes an int8, int4, or palettized weight to fp16 on the way in, and the multiply that follows is the same fp16 multiply as for an uncompressed weight. The dequantization is affine, $w = s\,(q - z)$, with the scale s scalar or per-output-channel, and agrees with the documented conversion-tool quantization relation [AppleCoreMLTools]. On the M1 generation the quantization is symmetric, so $z = 0$ and $w = s\,q$; the inverse encode is $q = \mathrm{round}(x/s) + z$.

The order of rounding in one multiply-accumulate pass is fixed. A weight is dequantized to fp16, the fp16 multiply runs, the products accumulate in the wide register, an optional per-channel scale and bias apply in fp16, an optional activation table applies, and the result stores to memory as fp16. Two rounding points bracket the accumulation: the inputs round to fp16 going in, and the output rounds to fp16 going out.

3.2 Wide accumulator

The running sum between those two rounding points is not fp16. The input port rounds tiles to fp16 on the way in and the output port rounds to fp16 on the way out, but the reduction in between is held in a wide register of fp32 class on every ANE device. The accumulator width is a fixed hardware property, not a per-program or per-chip setting.

Representable sums thus come back near exact. A reduction of sixteen thousand ones is bit exact, where a naive fp16 running sum would stall near two thousand once the partial total passes the spacing of its own increments. The spacing of fp16 at a partial of magnitude p is $\mathrm{ulp}(p) = 2^{\lfloor \log_2 p \rfloor - 10}$, so once $p > 2048$ the spacing exceeds 1 and an added unit increment falls below half a step and is swallowed.

The worked case that shows the accumulator is wider than fp16 yet supplied by fp16-rounded inputs is the sum of one value of 4096 followed by 1024 ones. The exact sum is

$$[4096] + [1] \times 1024 = 5120,$$

and the engine returns 5116, between the naive-fp16 result 4096 and the exact total. A naive fp16 running sum returns 4096, because each added one falls below half the spacing at 4096 and is swallowed. The engine instead holds the 1024 ones that fp16 would drop, and the small deficit from the exact 5120 is the fp16 rounding of the single input tile that holds the 4096, not a narrow running sum.

The discriminating test is a cancellation triple. A sum of a large value, its negation, and a one, repeated sixteen times near a magnitude of four thousand, returns all sixteen ones intact. An fp16 running sum holding four thousand has a spacing of two there, so each one would round away. The ones survive, so the accumulator is physically wider than fp16. The first reduction stage groups input lanes into tiles of four, fp16-rounded, which then supply the one wide accumulator. Swept across magnitudes, the survivor count is layout-independent: the same result whether the one precedes or follows the negation, so the hardware re-associates over a fixed lane lattice rather than the source order. Table 3.1 records the survivor count against the cancellation magnitude and the fp16 spacing there.

Table 3.1. The cancellation-threshold survivor sweep, M1/H13 measured, which fixes the first-stage tile width at four.

Cancellation magnitude	fp16 spacing at that magnitude	Survivors of sixteen ones
1024	1	16
3000	2	16
4090	2	16
4096	4	4
8000	4	4
16000	8	4
30000	16	4

The survivor count holds at sixteen up to a magnitude of 4090, then drops to a hard floor of exactly four at 4096 and stays at four out to 30000. The threshold is at 4096 because that is the first magnitude where the fp16 spacing reaches four, so a one sharing a four-lane tile with a rounded partial at or above 4096 falls below the in-tile rounding threshold and vanishes. The flat floor of four is the signature of a four-lane tile: a three-element triple period beating against a four-lane tile leaves exactly four unaffected lanes out of sixteen. A single one added next to one large resident value cannot expose the tile, because that value forces the whole wide partial onto its own coarse spacing and the one rounds away regardless of which tile it is in.

Both behaviors are reproducible on the engine by reducing a fp16 vector. The cancellation probe routes a sum through a matmul against a ones vector so the reduction is in the wide accumulator, and the slice trigger exercises the width-axis crop gain, as listing 3.1 gives both probes.

Listing 3.1. Probes for the wide accumulator's cancellation behavior and the width-slice saturation that overflows to infinity above the fp16 ceiling.

```
# Pseudocode for the engine's wide accumulator and the width-slice saturation.
# Each reduce_via_matmul(v) runs on the engine: v is fp16, summed against a
# fp16 ones-vector so the reduction accumulates in the wide (fp32-class) register;
# the result is read back as fp16.

# (a) Cancellation probe: a large value, its negation, and ones near the threshold.
#     A naive fp16 running sum at 4000 has ulp = 2 and swallows every one.
big = 4000.0
v = ([big, -big, 1.0] * 16)               # 16 ones survive the cancellation
assert reduce_via_matmul(v) == 16.0       # engine keeps all 16; fp16 loop -> 0

#     The accumulator is wide, but the inputs still round to fp16 going in:
v = [4096.0] + [1.0] * 1024               # one tile of 4096, then 1024 ones
assert reduce_via_matmul(v) == 5116.0     # between naive-fp16 4096 and exact 5120

# (b) Width-axis slice saturation: a nonzero begin offset applies a x16 crop gain.
#     The fp16 ceiling is 65504, so 4094*16 == 65504 passes but 4096*16 overflows.
def width_slice_with_offset(value):       # slice begin offset != 0 on the width axis
    return value * 16.0                   # the fixed crop-DMA gain, then fp16 store

assert width_slice_with_offset(4094.0) == 65504.0          # at the ceiling, finite
assert width_slice_with_offset(4096.0) == float("inf")     # 65536 > 65504 -> inf
```

3.3 Compressed weights and the streaming gate

The datapath reconstructs a compressed weight to fp16 before it reaches the multiply array, so the arithmetic is fp16 regardless of the storage format. What differs by format and by chip is whether the compressed bytes stream to the engine compressed and decompress on the way in, or fold to dense fp16 in memory first. A format that streams moves fewer bytes across the DRAM boundary; a format that folds yields the storage saving but not the bandwidth. Table 3.2 marks which formats stream and which fold across three chip generations.

Table 3.2. Which compressed-weight formats stream natively versus fold to dense fp16, by generation, M1 measured with A14 and M5 rows from the per-format gate.

Format	M1 / A13	A14 / M2	A16 / M5
int4 palette lookup table	stream	stream	stream
int8 affine	fold	stream	stream
sparse	stream	stream	stream
blockwise	fold	fold	stream

On the M1 the int4 palette form streams natively, measured at about 2.37 times the bandwidth of the dense weight, and a sparse weight with at least half its values zero streams as a one-bit keep-mask with the packed fp16 nonzeros. The int8 affine and blockwise forms fold to dense fp16 before the data-movement step. The fold expands the int8 weight to a dense fp16 constant in DRAM before the multiply, so the bytes that cross the DRAM boundary are full-width fp16 and the layer gets no bandwidth gain on the M1. The int8 fold is a stored-size saving only on the M1. The weight is half the size on disk, but it is reconstructed to fp16 in DRAM before the data-movement step, so a weight-streaming-bound matmul moves the same bytes as fp16 and runs at the fp16 latency. The int8 form first streams as int8 on the A14 and M2 generation, where it is dispatched as int8 and dequantized at the multiply port so it moves half the bytes of fp16. A bandwidth-bound matmul reaches about 0.52 times the fp16 latency at a weight of 8192 by 8192. The compressed-weight quantization adds error only at the input rounding: an int8 conv weight tracks the fp16 result at a cosine near 1.0, with a relative error near 0.6 percent against an fp32 reference, against about 0.02 percent for fp16.

3.4 Where precision holds and where it is lost

Convolution, matrix multiply, and normalization keep their precision when their partial sums stay in fp16 range and their results are representable. The wide accumulator holds the reduction, and the only quantization is the fp16 rounding of the inputs and the output. Vision, audio, and encoder workloads are in this regime, and run on the engine without a precision penalty.

Precision is lost on cancellation-heavy steps. The transformer decoder down-projection is the case that fails: a large positive and a large negative contribution nearly cancel, and the result is a small difference between two large numbers. The loss does not come from the accumulator, which is wide enough to hold the partial. It comes from the per-product fp16 rounding of the inputs and weights before they enter the accumulator. Once the operands are quantized to fp16, the cancellation amplifies that quantization into the result. A cancellation-heavy step has no fp16-safe form on the engine and needs a wider anchor computed on the CPU or the GPU.

3.5 Activation functions

Lookup tables evaluate the nonlinear activations. Identity and plain ReLU are not table-driven, since ReLU is a max, but sigmoid, tanh, gelu, swish, erf, exp, and the rest route through a piecewise-linear table of fixed knots. The table is a 33-knot piecewise-linear curve, not a dense sample grid. The input maps affinely onto the 32 segments between 33 knots, the bracketing segment evaluates as a slope times the input plus an intercept in fp16, and the value clamps to the end-knot asymptote past the table domain. Accuracy comes from the piecewise fit and the per-function domain, not from sample density. A user sigmoid does not lower to the plain sigmoid table. It lowers to a high-precision sigmoid table by default, which pushes the domain to a wider range with a finer subdivision near the linear region, so a sigmoid-heavy or attention-gate-heavy model stays accurate under fp16.

The decoded table is accurate enough that it is not a meaningful error source. On the standard set, measured on device, the worst absolute error is at the level of the fp16 storage floor: sigmoid 0.0034, tanh 0.0017, gelu 0.0059, each under 0.4 percent of the function range. The on-device value matches the fp16-rounded exact function to about half a unit in the last place. The table adds nothing measurable on top of fp16 storage rounding. The exceptions are sin, cos, and atan, which have up to about 0.04 to 0.12 absolute error near the seams of their argument reduction; a model that evaluates trig directly near a magnitude of pi should fold the range upstream.

3.6 Edge behavior a correctness oracle must model

Several edge behaviors of the fp16 datapath depart from a host IEEE reference, and a model that reproduces engine results bit for bit has to encode them. The engine coerces a NaN to positive infinity at the input boundary, and never produces a NaN anywhere. A NaN sent in and echoed through the identity $x + 0$ returns positive infinity, with the same bits as sending positive infinity directly, and every downstream op then behaves as if the value had been positive infinity. A NaN into relu returns infinity, a NaN into sigmoid or tanh returns 1.0, a NaN into erf returns 1.0, and a NaN into exp returns infinity. A NaN into the maximum of two values returns infinity, and a softmax of a lane holding a NaN puts all the mass on that lane. The one case where a NaN is not bit-identical to positive infinity is the variance reduction inside layer normalization, where the NaN enters as a large finite magnitude rather than as literal infinity. Rms normalization gives identical results for a NaN input and an infinity input. An upstream NaN through any gate thus leaves the engine as a finite or infinite value and does not surface as a NaN to downstream code.

The engine flushes to positive zero all the indeterminate forms that produce a NaN under IEEE. The infinity minus infinity case returns positive zero, the zero times infinity case returns positive zero, sqrt(-1) returns positive zero, rsqrt(-1) returns positive zero, and log(-1) returns positive zero.

The engine preserves denormals elementwise but flushes them inside the multiply-accumulate. A fp16 denormal down to 2^{-24} (about 5.96×10^{-8}) echoes bit-exactly through $x + 0$ and $x \times 1$ on the elementwise path, and scales correctly, so the common assumption that the engine flushes denormals globally is false for that path. A pair of denormals summed inside a matmul, with a representable denormal total, returns positive zero, so the flush-to-zero is a property of the accumulator stage and not of the datapath as a whole. The M5 accumulator instead preserves denormals, so a denormal input and a product of two denormals both survive a matmul reduction unflushed. The flush is thus an M1-generation property rather than a fixed engine behavior, and the M2 through M4 parts between them were not measured, so the generation at which it changed is unknown.

Signed zero loses its sign before a reciprocal or a reciprocal square root. The reciprocal of negative zero returns positive infinity where IEEE returns negative infinity, and the reciprocal square root of negative zero returns positive infinity where IEEE returns negative infinity, while sqrt of negative zero stays positive zero per IEEE. A negative zero echoes as positive zero through $x + 0$, so the engine drops the sign bit of a zero before the reciprocal path.

A few activation tables collapse a large input rather than tracking it: softplus and softsign of positive infinity return positive zero, where the values should be positive infinity and a unit magnitude. The logarithm of positive zero returns a finite sentinel of -45440 rather than negative infinity. These are properties of the table approximations, distinct from the NaN coercion above.

Softmax subtracts a hardware maximum before the exponential, so it does not overflow even when a raw exponential would. A softmax of $[1000, 1, 2, 3]$ returns $[1, 0, 0, 0]$ and matches a wide reference, despite $\exp(1000)$ being far past the fp16 range, and a softmax of four equal values returns four quarters. A bare exp with no max-subtraction overflows to infinity at an input near $11.094 = \ln(65504)$, so the stable route is the fused softmax rather than a hand-rolled exponential over a sum.

Rounding on the fp16 output grid is round half to even. A tie at the midpoint between two representable fp16 values rounds to the value with an even last bit, not away from zero. A partial of 2049 at a grid spacing of 2 rounds to 2048, the even neighbor, and a trailing half above the 2048 threshold rounds the $L + 0.5$ accumulation to even rather than up. The M5 returns 2050 for this case, rounding up where the M1 rounds to even. Whether the wide accumulator presents a value just above 2049 rather than an exact tie, or the rounding differs by generation, is unresolved.

Some activation tables have a small constant bias at the origin. The decoded gelu table returns -0.000543 at $x = 0$ where the exact gelu is 0, and the swish table returns -0.001259 at $x = 0$ where the exact swish is 0. The bias is below the fp16 storage floor and does not affect the per-op accuracy figures, but a bit-exact oracle has to hold it.

3.7 Saturation hazard

The fp16 maximum finite value is 65504. The compute datapath has no fp32 margin, so a value above that saturates to infinity silently.

The multiply-accumulate output stage saturates earlier than the storage format, at exactly $2^{15} = 32768$, half of the fp16 ceiling. This is a different axis from the width-slice gain below: it is a property of the accumulator output port, and it fires on matrix multiply, linear, and any convolution that accumulates two or more taps, whatever the number of accumulation terms. The threshold is pinned to the bit: the largest fp16 value below 2^{15}, which is 32752, passes through a linear, and the next fp16 value, 32768, returns infinity. It holds for $K = 1$, for $K = 2$, and for a two-channel convolution, so it tracks the would-be output magnitude at the accumulator port and not the count of accumulated terms. An interior partial that exceeds 2^{15} overflows to infinity even when a later cancellation would have brought the final result back into range. The paths that drive a dedicated reduction or a single elementwise multiply hold the full fp16 range instead. A reduce-sum rounds to 65504 first and then overflows, an elementwise multiply overflows at the true fp16 limit near 65536, and a pointwise one-by-one convolution with a single input channel passes a fill of 60000. This earlier

accumulator ceiling is consistent with the multiply-accumulate datapath having about one bit less margin than the fp16 storage format on the M1, so the guard for a matrix multiply or a multi-tap convolution is to keep the output magnitude below 2^{15}.

The second saturation a developer encounters is on the slice path. A width-axis slice with a nonzero begin offset routes through a crop DMA that applies a fixed gain of sixteen. A value at or below 4094 passes through bit exact, because $4094 \times 16 = 65504$ is the fp16 ceiling. A value of 4095 or above becomes infinity, because it rounds to 4096 on the fp16 grid and $4096 \times 16 = 65536 > 65504$, past the ceiling. The control case, a slice with a zero begin offset, is free of the saturation at the same fill values, so the trigger is the nonzero width-axis offset and not the magnitude alone. The hazard is on the width axis only: a nonzero begin offset on the height, channel, or batch axis stays finite at the same fill values. The saturation is measured on both the M1 generation (H13) and the A14 generation (H14), where a fill at width offset 4094 stays finite ($4094 \times 16 = 65504$) and a fill at 4096 overflows to infinity ($4096 \times 16 = 65536$). The non-saturating route arrives on the A15 generation and later, so the hazard is not fixed by the immediate next family. The guard is to avoid nonzero last-axis begin offsets on a width slice, or to route the offset onto a different axis.

3.8 Determinism

For a fixed graph and a fixed input the M1 engine is bit-deterministic: the raw fp16 output bytes are identical across reruns, across an independent recompile, and across a fresh process. The lowered program fixes the accumulation order, so there is no run-to-run drift to round away.

Re-executing one compiled program on the same input returns the same fp16 bytes every time, measured at zero units in the last place over 200 repeats for a matrix multiply, two-layer convolution, and long reduction. The same holds over 50 repeats for softmax, reduce-mean, and a large matrix multiply. Compiling the same graph twice at optimization level zero, the byte-identical lowering path, produces two programs whose outputs agree to the bit, including a real-tiled $[128, 1024]@[1024, 1024]$ matrix multiply. Running the same graph in a fresh subprocess, with a new dispatch client, returns the same output digest as the in-process run, so daemon and process state do not perturb the result.

The result is also independent of batch size and batch position. A row computed alone is bit-identical to the same row computed inside a batch, and the same row placed at every position of a batch of sixteen gives sixteen identical outputs. Row zero is invariant across batch sizes of one, two, eight, thirty-two, and one hundred twenty-eight. The engine computes each batch element identically regardless of its neighbors.

The one thing that changes the bits is changing the math. Writing a graph with a different association order, $(a + b) + c$ against $a + (b + c)$, gives different fp16 rounding. About 31 percent of the elements differ, by about one fp16 unit in the last place at that magnitude, with a maximum absolute difference near 7.8×10^{-3}. Each ordering is itself perfectly reproducible, so this is ordinary fp16 non-associativity of the graph as written and not hardware nondeterminism. Two high-level ops that lower to the same kernel agree to the bit. A dot product written as a matrix multiply and the same dot product written as an elementwise multiply followed by a reduce-sum return identical bytes here, because the compiler lowered both to the same accumulation order. Reassociation bites only when the graph dictates a different order, so the engine is safe to treat as reproducible, and the only nondeterminism source is the accumulation order the graph chooses, never the silicon.

3.9 Keeping a graph inside the fp16 envelope

Two numeric decisions belong before a graph is compiled. A developer anchors a cancellation-heavy reduction off-engine, on the CPU or the GPU, because it has no fp16-safe form on the datapath. A width-axis slice avoids a nonzero begin offset, because the fixed gain of sixteen overflows a fill above 4094 to infinity.

The cancellation-heavy step routes to a wider unit, and the rest stays on the engine, as listing 3.2 works the three rules through one graph.

Listing 3.2. The three rules that keep a graph inside the fp16 envelope before compile.

```
# Keep a graph numerically safe BEFORE compiling it, by applying three rules
# to every reduction and every cropped tile in the graph.

# fp16 limits the whole datapath uses:
fp16_max         = 65504        # largest finite fp16 value; above this -> infinity
accum_out_max    = 32768        # multiply-accumulate output port saturates here
width_slice_gain = 16           # a width-axis crop with a nonzero begin offset
                                #   multiplies its values by this fixed gain

# RULE 1: route a reduction through a matmul against a ones-vector, so the
#         sum accumulates in the wide (fp32-class) accumulator and stays near exact.
#         A plain elementwise running sum rounds at every step in narrow fp16.
function safe_reduce_sum(vector v):                  # v is fp16
    ones = vector_of_ones(length(v))                 # fp16, same length as v
    return matmul(v, ones)                           # one dot product, wide accumulator

# RULE 2: a cancellation-heavy step (a large positive nearly canceling a large
#         negative) has no fp16-safe form on the engine, because the operands
#         were already rounded to fp16 before the subtract. Scale it up so the
#         small difference clears the fp16 grid, OR split it out to a wider
#         unit (CPU or GPU) and feed the result back as a graph input.
function place_cancellation_step(step):
    if is_cancellation_heavy(step):
        return compute_off_engine(step)              # wider anchor, fed back as input
    else:
        return keep_on_engine(step)                  # representable sums stay near exact

# RULE 3: keep every tile value under the limit that applies to its path,
#         so nothing saturates silently to infinity mid-graph.
function check_tile_value(value, path):
    if path == "matmul_or_multitap_conv":
        require value <= accum_out_max               # output-port ceiling, ~half fp16_max
    if path == "width_slice_with_nonzero_offset":
        require value * width_slice_gain <= fp16_max # 4094*16 == 65504 passes; 4096*16 overflows
    otherwise:
        require value <= fp16_max                    # plain elementwise / single reduction

# Assemble the safe graph: representable conv/matmul/norm stay on the engine,
# reductions go through safe_reduce_sum, the cancellation step is anchored wide.
graph G:
    input  x                                         # fp16 activations
    feats  = conv(x, weights = W)                    # representable sums, stays on engine
    pooled = safe_reduce_sum(feats)                  # RULE 1: wide-accumulator reduction
    output pooled
program P = compile(G, target = H13)                 # fp16 datapath, wide accumulator
result = dispatch(P, features)
# The cancellation-heavy down-projection (RULE 2) is computed off-engine and
# fed back as an input, never lowered into this graph.
```

3.10 Reference: fp16 numeric constants

Table 3.3 collects the fp16 numeric constants, the saturation thresholds, and the activation-table error figures.

Table 3.3. The fp16 numeric constants, M1/H13 measured, with the saturation reproduced on A14/H14.

Constant	Value
fp16 maximum finite magnitude	65504
Multiply-accumulate output ceiling	32768
Width-slice crop-DMA gain	16
Width-slice finite fill ceiling	4094

Constant	Value
Width-slice overflow fill	4096
Wide-accumulator bit-exact reduction	16000 ones
Worked sum, $[4096] + [1] \times 1024$	5116
First reduction-stage tile width	4 lanes
Sigmoid worst absolute error	0.0034
Tanh worst absolute error	0.0017
Gelu worst absolute error	0.0059
Gelu origin bias at $x = 0$	-0.000543
Swish origin bias at $x = 0$	-0.001259
Trig seam absolute error (sin, cos, atan)	0.04 to 0.12
Activation table knot count	33
Sigmoid table domain clamp	$[-9.938,\ +8.320]$
Exp input where output first reaches infinity	11.094
Square input where output first reaches infinity	256
int4 palette native stream speedup, M1	2.37x
int8 fold on M1 (stored-size saving, no stream gain)	1.0x fp16 latency
int8 weight-stream latency, A14/M2 (8192 weight)	0.52x fp16

4 Capability surface

SUMMARY

The engine runs the operations on-device perception networks are built from: convolution, matrix multiply, fused attention, the normalizations, the activations, and data movement, all native from the M1 onward. A confirmed set has no hardware path on any family, including reduce_prod, the scatter family, and the recurrent cells; a network that needs one rewrites it. Some operations are family-gated: the texture-engine sampler arrives on the A14, and sin and cos arrive on the A15. A capability attested at one layer is not a reachable operation: three-dimensional convolution has a capability byte yet fails backend lowering on every device, so only a compile-and-run confirms an operation.

The Apple Neural Engine computes a fixed vocabulary of tensor operations. The full operation-by-device table is Appendix A; this chapter gives its shape and the rules for fitting work to the engine.

4.1 What fits

The engine runs the operations that on-device perception networks are built from. Convolution and matrix multiply are at the center, the normalizations and activations around them, and data-movement operations between the compute steps.

Two-dimensional convolution is native, including the transpose (deconvolution) and the dilated, depthwise, and grouped forms, with kernel and tensor sizes bounded by the per-chip limits in Appendix A. The compiler selects a Winograd path for eligible 3x3 stride-1 convolutions without any caller action. Matrix multiply and fully-connected layers are native and fold into the convolution datapath when the operand fits the on-chip working set, tiling when it does not. Attention runs as a fused operation: scaled dot-product attention runs on the matrix-multiply and softmax path, native on every family from the M1 onward and not gated behind the texture engine.

The common normalizations all run: layer, instance, group, and L2 normalization, plus batch normalization folded to an affine at inference. Pooling (average, max, and L2) is native, as are the elementwise arithmetic and comparison operations. The activation set covers ReLU and its variants, sigmoid, tanh, gelu, swish, softmax, erf, exp, and log, which evaluate through the lookup tables described in chapter 3; that table is a programmable piecewise-linear curve, and chapter 26 covers reaching an arbitrary pointwise function through it. Data movement is native for reshape, flatten, expand, squeeze, transpose, concat, split, stack, constant pad, and slice, all descriptor edits or DMA operations rather than compute. A reduction or transpose over a large axis switches to a tiled route at a per-chip threshold, at no change to the result.

The backend represents these operations in its own dialect, `anec.*`, one level below the frontend intermediate language. Listing 4.1 gives the signatures for the core operations.

Listing 4.1. The backend dialect signatures for convolution, matrix multiply, and normalization, with the operands and attributes the compiler expects.

```
// 2D conv: weights and bias fold in; rank 4 or 5; M1 kernel at most 29x29 (13x13 fp16).
anec.convolution(%input) {
    strides, dilation_rates, groups, kernel_sizes,
    explicit_padding, padding_style, weights_layout
}

// matrix multiply: contraction direction is set by the transpose flags, not by operand order.
// depth D must be 1 on both operands ("depth > 1 is not supported for MatMult inputs").
anec.matmul(%lhs, %rhs) { transpose_lhs, transpose_rhs }

// layer and group normalization are one atom: group_norm is layer_norm with num_groups > 1.
// channels must be divisible by num_groups; the output format must be Float.
anec.layer_norm(%input) { groups, epsilon, axis }   // gamma and beta fold in
```

The tensor layout the compiler reasons about is five-dimensional, `N` batch, `D` depth, `C` channel, `H`, `W`, with a maximum rank of five, and transforms act only on the last four dimensions.

4.2 Hard limits

Some operations have no hardware path on any current family. They do not run on the engine and must be computed off-engine or reformulated. The confirmed cases, unsupported on every family from the M1 through the M5, are: reduce_prod, the scatter family (scatter, scatter along axis, scatter ND), mod, one_hot, non_zero, band_part, reverse_sequence, shape, sliding_windows, the logical and, or, and xor operations, the recurrent gru, lstm, and rnn cells, the inverse and hyperbolic trigonometric functions, and the random-sampling operations other than the uniform generator. A network that needs one of these rewrites it: a product reduction through log-sum-exp, logical-and through a minimum, recurrent cell through an unrolled fixed-trip-count graph.

A second class is family-gated rather than universal, and Appendix A gives the chip that enables each. Gather on the M1 takes a software path valid only for a batch of one, depth of one, and three-element index channel, and is rejected outside it. The texture-engine operations (resize as a hardware sampler, crop-resize, resample, affine) are absent on the M1 and arrive on the A14 generation. Trigonometric sin and cos are native only from the A15 generation; the M1 and the A14 reject them and decompose on the host.

Table 4.1 groups the operation classes by status, from native on every family through the family-gated paths to those with no hardware path.

Table 4.1. Operation classes by status, from native on every family through limited or family-gated paths to those with no hardware path.

Class	Examples	Status
Convolution	2D, transpose, dilated, depthwise, grouped	Native, M1 onward
Matrix multiply, fully-connected	`matmul`, `linear`	Native, M1 onward
Fused attention	scaled dot-product attention	Native, M1 onward
Normalization	layer, instance, group, L2, batch-fold	Native, M1 onward
Pooling	average, max, L2	Native, M1 onward
Elementwise, activation	add, mul, ReLU, sigmoid, tanh, gelu, swish, softmax	Native, M1 onward
Data movement	reshape, transpose, concat, split, pad, slice	Native, M1 onward
Product reduction	`reduce_prod`	No path on any family

Class	Examples	Status
Scan	`cumsum`	Native on the M1 through a curated runtime path
Scatter	scatter, scatter along axis, scatter ND	No path on any family
Indexing, shape	`one_hot`, `non_zero`, `shape`, `band_part`	No path on any family
Recurrent cells	gru, lstm, rnn	No path on any family
Trig, inverse and hyperbolic	tan, asin, sinh, atanh	No path on any family
Gather	limited software envelope	M1 (batch 1, depth 1, 3-element index)
Texture-engine	resize, crop-resize, resample, affine	A14 onward
Trig sin and cos	`sin`, `cos`	A15 onward

4.3 Type limits

The datapath is fp16, as chapter 3 shows. The frontend accepts fp32, int32, and bf16 as type annotations, but the backend does not implement them, so those annotations do not reach the silicon as wider arithmetic. The compiler rejects a cast to int32 on the M1, and bf16 is not usable as a program input or output dtype. Requesting a wider type gains no precision: the multiply, activation tables, and output port are fp16 regardless, and only the accumulator is wide.

A requested dtype routes by type, as figure 4.1 traces the fp16 and wider-type paths.

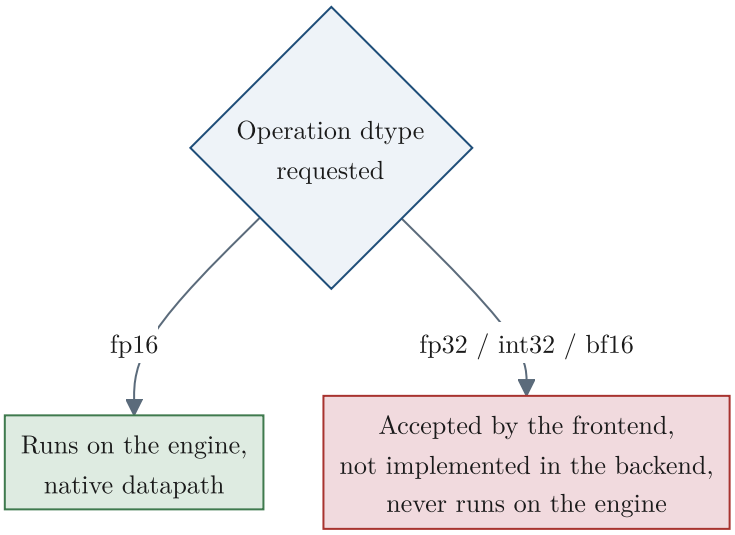

Figure 4.1. How a requested operation dtype routes on the engine.

4.4 Attested is not reachable

A hardware capability bit, or an operation the compiler appears to accept, does not guarantee a reachable, correct operation on the direct path. Capability is attested at one layer and must be confirmed at the layer that runs the work.

The case that fixes the rule is three-dimensional convolution. It is advertised by a hardware capability byte and recognized by the compiler frontend, yet it fails backend lowering on every device mask: the operation exists in the attestation and does not run. The gap appears in the other direction too. On the M1 the

top-k, sort, and dynamic-slice validators are all callable, yet the code generator rejects all three. A capability advertised in a table, recognized by a frontend, or validated by a checker is a claim about one layer; only a compile-and-run on the target confirms the operation at the layer that executes it.

The guide reports a reachable surface rather than the advertised one. Every operation marked native in Appendix A was compiled and run on the M1, not inferred from a capability bit. The advertised surface is the larger, and the difference is the operations that pass an earlier check and fail at code generation or backend lowering. This account extends and partly corrects the documented convertible operation set, which lists operations the public converter accepts rather than the operations that lower to the engine [AppleCoreMLTools].

4.5 Two compile routes

The engine has two compile routes with different capability gates, and an operation's availability can differ between them on the same chip. The intermediate-language route is gated by the per-operation family floor: an operation is native when the chip family is at or above the operation's floor, and below the floor the compiler decomposes or rejects it. The bridge route authors a fused layer directly and is gated by the hardware-abstraction-layer feature bytes rather than the family floor. The clear example is the whole-tensor argument-maximum. On the intermediate-language route it is gated to the A15 generation and is rejected on the M1, yet on the bridge route it runs correctly on the M1, because the feature byte that gates it is already set on the A13. The same structure explains the other direction. Native sin and cos have no bridge route, so they stay on the intermediate-language route and are rejected on the M1, while the rank and sort operations have a bridge route that is rejected at code generation on the M1.

Table 4.2 lists the hardware-abstraction-layer feature bytes that gate the bridge-route capabilities, with the per-generation value of each.

Table 4.2. The hardware-abstraction-layer feature bytes that gate the bridge-route capabilities, with the per-generation value of each.

Capability	Gate byte	older	A13 / M1	A14	A15
softmax	`0x815`	0	1	1	1
instance normalization	`0x816`	0	1	1	1
argument-maximum, hardware form	`0x4f2`	0	1	1	1
square-after-reduction fusion	`0x494`	0	0	1	1
texture engine	`0x81d`	0	0	1	1
dropout and random	`0x4a9`	0	0	0	1
kernel-memory streaming select	`0x48f`	0	1	1	1

4.6 Confirming an operation on the target

Because attestation is not reachability, an operation is confirmed by compiling the graph to the chip target and running it, not by reading a capability bit. A graph that lowers and runs on the target is reachable there; a graph that the compiler rejects names the operation and the layer that refused it.

The graph compiles against the target and runs before an operation is relied on, the procedure listing 4.2 carries out for one operation at a time.

Listing 4.2. Confirming one operation on a target by compiling a single-op graph and reading the compiler's verdict.

```
# Confirm one operation on a target by compiling a graph that contains ONLY
# that operation, then reading the compiler's verdict. Do not trust a
# capability byte: a byte can be set for an op that still fails to lower.

# Build the smallest legal graph that exercises the operation under test.
function one_op_graph(operation, target):
    graph G:
        input  x                                # a dummy input of a legal shape
        output operation(x)                     # the single op we want to confirm
    return G

# Ask the compiler to lower the graph against the target. Two outcomes only:
#   - it lowers and loads        -> the op is REACHABLE on this target
#   - the compiler rejects it    -> it names the op and the layer that refused
function confirm_op(operation, target):
    G = one_op_graph(operation, target)
    try:
        P = compile(G, target = target)         # lower + load against this chip
        return NATIVE                           # reached the silicon: reachable
    catch reject as r:
        # r holds the rejecting layer and message, e.g.
        #   "Some ops are not supported on any of the specified backends"  (no path)
        #   "<op> requires family >= N"                                    (family-gated)
        report r.layer, r.message               # the reject string IS the signal
        return REJECTED

# Worked checks against the M1 target (family H13):
status_conv = confirm_op(conv_2d,     target = H13)   # NATIVE: 2D conv lowers from M1 onward
status_sin  = confirm_op(sin,         target = H13)   # REJECTED: sin/cos are gated to A15+
status_prod = confirm_op(reduce_prod, target = H13)   # REJECTED: no hardware path on any family

# An op gated to a later family is rejected when compiled against an earlier
# target, so the target argument is how a caller checks the family floor
# before building anything on top of the operation.
```

4.7 Reference: the per-chip shape limits

The operations that run have per-chip shape and kernel limits, read from the compiler's hardware-abstraction-layer tables. The headline limits separate the M1 from the M5 by a factor of four on the tensor extent and by a step on the kernel size; the full table is Appendix A. Table 4.3 gives the per-generation kernel and tensor shape limits decoded from those tables.

Table 4.3. The per-generation kernel and tensor shape limits, decoded from the compiler's hardware-abstraction-layer tables, M1 measured.

Limit	older	A13 / M1	A14	A15	A16+ (M5)
max kernel width, default format	29	29	32	32	32
max kernel width, fp16	13	13	16	16	16
min kernel width, large mode	16	16	1	1	1
max kernel depth	1	16	16	16	16
max tensor width, height	16384	16384	16384	16384	65536
max tensor depth	1	16384	16384	16384	65536
max tensor batch	4096	65536	65536	65536	65536
reduction-to-transpose threshold	none	192	192	384	384
matrix-multiply working set	2 MB	2 MB	2 MB	2 MB	2 MB
has texture engine	no	no	yes	yes	yes

Two per-operation envelopes are hard limits a caller meets early. The matrix-multiply operation requires the depth axis to be 1 on both operands, and the fully-connected operation rejects an input of rank 5 or more. The software gather path on the M1 is valid only inside a small envelope: the data batch and depth must be 1, the index channel must be 3, the index width and depth must be 1, and the gather-axis count must be 3. Outside it the compiler aborts rather than falling back.

4.8 Reference: the compile-legal envelope

The shape limits above bound the operations that run; a second set of rules bounds the programs the compiler accepts at all. These were measured compile-only on the M1, building each graph and calling the compiler inside a guard, never running the result, so a rejected program names its layer without provoking the firmware. Two layers refuse a program: a Python-side check in the frontend, before anything reaches the device, and the on-device compiler, which validates and returns an Espresso exception.

A tensor has a maximum rank of five, the `N, D, C, H, W` layout of the per-chip limits, and the frontend rejects a graph that builds a rank-6 tensor. Per-operation support is narrower than the generic rank cap: a rank-5 batched matrix multiply builds yet the on-device matrix-multiply backend rejects it, and convolution is pinned to four-dimensional `NCHW` input. Each axis is capped at 2^{14}, the 16384 tensor extent of the M1, applied per-axis rather than to the last axis alone: the matrix-multiply output and contraction dims, both elementwise axes, and the rank-1 length all accept 16384 and reject 16385. The convolution input-channel and output-channel dims escape this cap, both `Cin` and `Cout` of 16385 compile, while the convolution input spatial axis obeys it. Size-1 dims and the rank-0 scalar are legal and compile; a zero-size dim is the one degenerate shape the on-device validator rejects, as a type mismatch, since the frontend does not pre-screen it.

Program inputs are `fp16` or `uint8` only. The `fp16` input is the compute type; the `uint8` input supplies only the in-graph dequantization path for an integer image input, not an `add` or `matmul` directly, and the frontend rejects any other input dtype. Weight constants accept any float numpy dtype, with `fp32` constants down-cast to `fp16`, while the compiler rejects integer-typed weights. Convolution pre-screens the kernel width in the frontend, rejecting a width above 15 on the M1, while the fp16 datapath caps it lower at 13 as the limit table above shows, so an fp16 kernel of width 14 or 15 still rejects at lowering. The kernel height is unconstrained, so a 16x3 kernel compiles and a 3x16 kernel does not. Stride, dilation, grouped and depthwise forms, and padding wider than the input all compile; the on-device compiler rejects an indivisible group count.

Table 4.4 records each compile-legal probe with its result, the layer that ruled on it, and the message the rejecting layer returns.

Table 4.4. The compile-legal envelope on the M1, separating the frontend rejects from the on-device compiler rejects.

Probe	Result	Layer	Compiler message
tensor rank 1 through 5	accept		
tensor rank 6	reject	frontend	`tensor rank 6 exceeds the ANE maximum of 5`
batched matmul rank 5	reject	on-device	`Some ops are not supported on any of the specified backends`
any axis = 16384	accept		
any axis = 16385	reject	frontend	`exceeds ANE family 2's max dimension 16384`
conv `Cin` or `Cout` = 16385	accept		
size-1 dims, rank-0 scalar	accept		
zero-size dim `(0, 8)`	reject	on-device	`Expected tensor<fp16,[1,8]>; got tensor<fp16,[0,8]>`

Probe	Result	Layer	Compiler message
input dtype `fp16`	accept		
input dtype `uint8` into arithmetic	reject	on-device	`Param 'y' ... got tensor<uint8,...>`
input dtype `fp32`, `int32`, `bf16`, `int8`	reject	frontend	`dtype must be 'fp16' or 'uint8'`
conv kernel width `kW` $<=$ 15	accept		
conv kernel width `kW` $=$ 16	reject	frontend	`kW must be <=15`
conv kernel height `kH` $=$ 16	accept		
conv indivisible groups	reject	on-device	`KernelChannels (2) != InputChannels (8) / Group (3)`

Part II

Reaching the ANE

05 Software stack

The layers from a graph to the engine, and the daemon that brokers access.

06 Dispatching without Core ML

The five-step direct route: build, compile, load, bind, and dispatch.

07 Weights and compression

The int4, sparse, int8, and blockwise forms, and which stream natively.

08 Entitlement boundary

The signed-load gate and the entitlements that bound the reachable surface.

5 Software stack

SUMMARY

Four user-space layers separate an application from the engine, and the execution runtime beneath the model framework is reachable directly with no placement planner and no entitlement for accepted operations. The model framework segments work across the processor, graphics processor, and engine by a shortest-path solve over a per-operation cost graph, reported through the public model-plan read-out. A system daemon holds the single privileged device gate, a content-hashed program cache, and one time-shared request queue that arbitrates every client.

Part I treated the engine as a machine: a fixed-function matrix accelerator with an fp16 datapath, driven as an autonomous coprocessor through a command mailbox. Part II treats it as a target a developer can address. Between an application and the silicon are several layers of system software, each with a defined job. This chapter maps those layers and marks the one this Part enters.

5.1 Layered stack

Four user-space layers separate an application from the engine, with a system daemon and a kernel driver below them out of the dispatch band. Figure 5.1 shows the full stack from the application down to the silicon, with the two routes that reach the runtime.

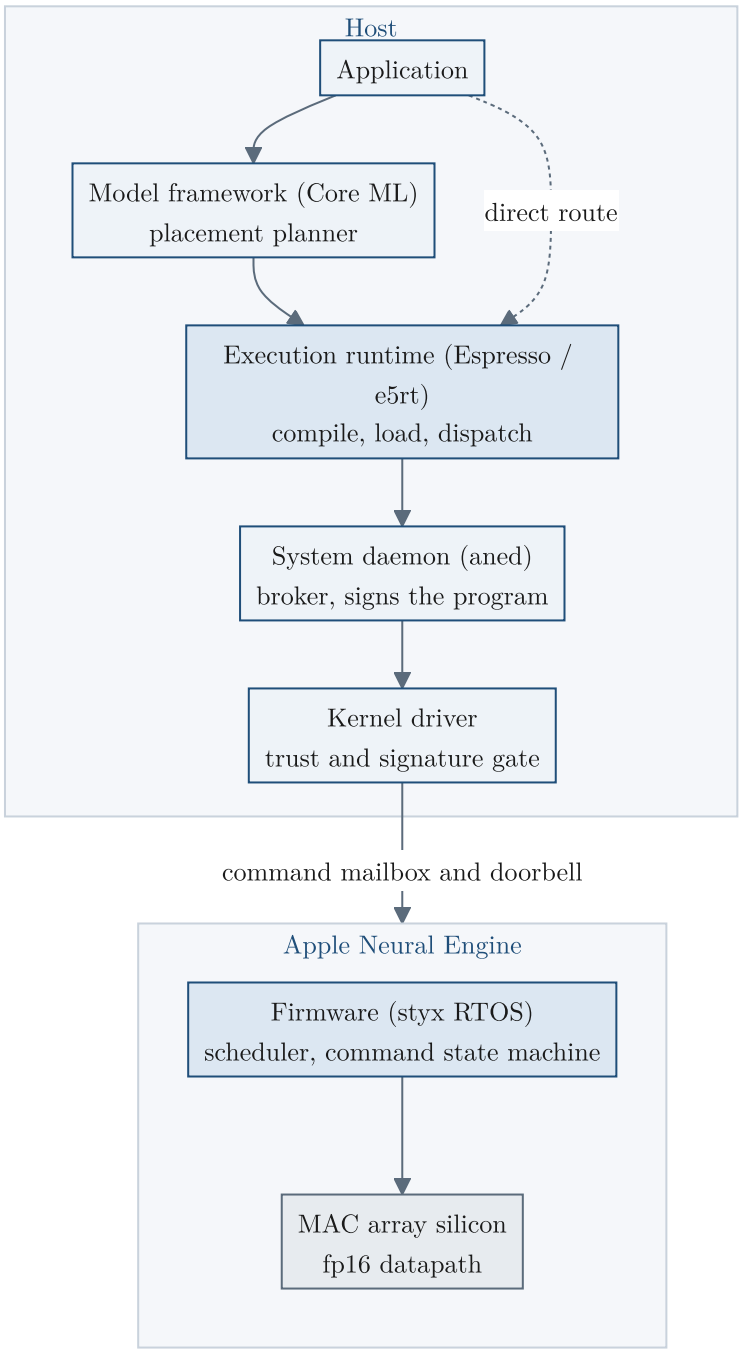

Figure 5.1. The full stack from an application down through the kernel driver, the engine firmware, and the silicon, with the two routes to the runtime.

The top layer is the model framework. It accepts a trained network, decides which operations the engine can accept, and segments the work across the central processor, graphics processor, and engine under a cost-driven placement planner. A caller at this layer does not choose the device and is not told which device ran the work. Chapter 1 describes this public surface.

Beneath it is the execution runtime, the layer every path passes through. It owns compilation: it turns a network in the intermediate representation into the engine's program format, writes the result to a content-addressed cache on disk, and produces a loadable program. It owns the program library and the callable functions inside it. It owns the execution stream, the queue against which a compiled program is encoded and dispatched, and the descriptors that bind operands to a program's named ports. The same runtime hosts the central-processor and graphics-processor executors as alternative backends, so a single stream can hold operations placed on different devices. The engine is one backend among those three.

Beneath the runtime, on the path to the engine, is the client layer. It splits into two duties. The client layer hands program lifecycle, meaning compile, load, instantiate, cache, and purge, to the system daemon over an interprocess channel. Per-inference dispatch goes directly to the kernel driver once a program instance exists. The daemon holds the privileged device handle and the on-disk program cache; the client process holds the connection it uses for the hot path.

The lowest user-space layer is a thin shim over the kernel driver. Its calls translate into driver method invocations on the engine's user client. Below that is the kernel driver itself, then the firmware and the silicon.

Table 5.1 lists the four user-space layers and their reachability from ordinary user space, with the daemon and kernel driver below them out of the dispatch band.

Table 5.1. Layers of the engine software stack, each layer's role, and whether a caller can reach it directly.

Layer	Role	Reachable directly
Model framework	Cost-driven placement planner; segments work across processor, graphics, and engine	No: device is a scheduling outcome, not a choice
Espresso runtime	Network loading and operator graph beneath the framework	Yes, but the framework drives it
Execution runtime	Owns compile, program library, execution stream, and operand descriptors	Yes: names the engine as target, no entitlement for accepted ops
Client / user-client layer	Splits program lifecycle to the daemon from per-inference dispatch to the driver	Yes, for the hot dispatch path
System daemon (out of band)	Holds the privileged device gate and the content-hashed program cache; arbitrates clients	No: reached through its interprocess channel
Kernel-driver shim (out of band)	Thin user-space translation into driver method invocations on the engine's user client	Yes, as the lowest user-space call

Each layer has a distinct symbol family in the binaries, and the family name identifies the layer a call is on. The execution runtime exports a flat C facade over a C++ core in the `E5RT::` namespace, with every entry point prefixed `e5rt_`. The client layer is Objective-C, every class prefixed `_ANE`. The daemon communicates over a private interprocess protocol whose selectors hold the arguments verbatim. The shim into the kernel driver is the `ANEServices` C and C++ entry points, which lower to the IOKit `IOConnectCall*` calls on the engine's user client. Table 5.2 gives the entry-point symbol family that marks each layer, from the execution runtime down to the kernel-driver shim.

Table 5.2. The entry-point symbol family that marks each layer of the engine software stack, from the execution runtime down to the kernel-driver shim.

Layer	Symbol family (entry points)
Execution runtime (`Espresso.framework`)	`e5rt_e5_compiler_compile` (compile MIL to `.e5`), `e5rt_program_library_retain_program_function`, `e5rt_program_function_load_for_execution`, `e5rt_execution_stream_execute_sync` / `_submit_async`; C++ core `E5RT::E5Compiler`, `E5RT::ExecutionStream`
Client layer (`AppleNeuralEngine`)	`_ANEClient`, `_ANEModel` / `_ANEInMemoryModel`, `_ANERequest`, `_ANEIOSurfaceObject`, `evaluateWithQoS:options:request:error:`
System daemon (`aned`, NSXPC `com.apple.aned`)	`createProgramInstanceForModel:modelToken:...:statsMask:memoryPoolID:enableLateLatch:...:error:`
Kernel-driver shim (`ANEServices.framework`)	`_ANEServicesProgramCreate`, `ANE::ANEServicesDevice::ANE_ProgramSendRequest(...)` to `IOConnectCallAsyncMethod(selector=2)` on `H11ANE`

5.2 Runtime is reachable below the framework

The same execution runtime that the system's own dispatchers use is callable from ordinary user space, below the model framework. A caller can compile a network to the engine's program format, open the resulting program library, load a function for execution, bind operand buffers to its ports, encode the operation onto a stream, and submit it. No placement planner is in this path, and the operations the compiler accepts require no special entitlement. The caller names the engine as the target rather than receiving it as a scheduling outcome.

5.2.1 Runtime surface and its tunable dictionary

The execution runtime is a flat C facade over a refcounted C++ core in the `E5RT::` namespace. The binary exports 292 `e5rt_` entry points, organized into five object families: a compiler and its configuration, program library and the functions inside it, precompiled compute operation, execution stream with its buffer and port objects, and asynchronous event for fences. Each family pairs a `_create` constructor with a `_release` destructor, and each has a create-options object whose typed get/set accessors are the runtime's tunable dictionary.

The dictionary spans four objects. The compile-time options object (`e5rt_e5_compiler_options_`) exposes 21 keys: the backend bitmask (`compute_device_types_mask`, where `0x4` names the engine), the cache controls `force_recompilation` and `force_fetch_from_cache`, the `segmenter` selection, and a set of backend-preference and experimental toggles. A separate compiler-configuration object (`e5rt_e5_compiler_config_options_`) holds the 2 cache keys: `cache_bundle_location`, the directory the daemon writes the compiled bundle into, and `bundle_cache_apfs_purgeable`, which marks that bundle reclaimable under storage pressure. The per-operation options object (`e5rt_precompiled_compute_op_create_options_`) holds 14 keys, including `operation_name`, `allocate_intermediate_buffers`, the resident-weight path `mutable_mil_weight_paths`, and a cross-process IOSurface pool binding. The execution-stream configuration object holds 3 keys: `enable_concurrent_sync_execution`, `enable_low_latency_async_events`, and `skip_io_fences`. A neutral all-engine compile sets only a small subset of these: the engine bitmask, `force_recompilation`, and the `"graph"` segmenter, leaving every other key at its runtime default.

The on-disk program cache the daemon writes is keyed by content, not by source filename. The runtime assembles a `cacheURLIdentifier` from a per-segment key computed over each segment's network structure and weight blob containers, combined with the resolved source URL, options dictionary, and platform. Two compiles of structurally identical graphs with identical weights and options resolve to the same identifier and hit the cache, while changing any weight, shape, operation, the device mask, or the segmenter changes the

key. The `force_recompilation` key bypasses the cache fetch and rewrites the bundle unconditionally, which is the documented inverse of `force_fetch_from_cache`.

5.2.2 Dispatch selector that splits the path

The split between the daemon and the client described above resolves to specific kernel-driver selectors, measured by read-only tracing of `IOConnectCall*Method` on this M1. Per-inference submit is `IOConnectCallAsyncMethod` selector 2, the `ANE_ProgramSendRequest` handler in the H11ANE dispatch table, and the unentitled client issues it directly: a freshly compiled program issued selector 2 exactly once per execute, observed in the client process and never in the daemon. The daemon issues only the lifecycle selectors over its own connection during the same compile and prepare, the synchronous selectors 3 through 6 for program destroy, status, instance create with unprepare, and program create. The division of labor is thus precise: the daemon compiles the network and creates, prepares, and destroys the program object over IOKit, while the client submits each inference over selector 2 itself. A warm re-execute on an already-submitted program issued no additional `IOConnectCall` from the client, since the runtime reuses the armed async submit ring and signals the engine without a fresh external method per call. This is consistent with the warm-eval cost being firmware round-trip bound rather than dispatch-call bound.

5.3 Placement segmenter above the runtime

The model framework adds one stage the direct route does not have: it decides, per operation, which of the three backends runs it. When the framework places a model, it segments the operation graph across the central processor, graphics processor, and engine by solving a shortest path (Dijkstra) over a cost graph with one node per operation-and-backend pair. The per-operation cost on each backend comes from a set of learned regression decision trees, several hundred of them, keyed by operation and by backend. Two coarse compute and bandwidth anchors order the backends, and the trees supply the calibrated per-operation cost the solver minimizes. A fixed launch penalty per segment and a transfer penalty at every backend boundary bias the solution toward fewer and larger engine segments, for the reason given with the cost equation below. An operation the engine cannot accept has no engine node in the cost graph, so the minimum-cost path routes around the engine through the central or graphics processor. This mechanism drives the framework's automatic fallback, and the direct route skips it by authoring a single all-engine graph.

A developer can read this placement decision without running inference, through the supported public model-plan API. `MLComputePlan.load` parses a compiled model and reports the segmenter's choice per operation. `deviceUsage(for:)` returns the set of devices that could run an operation and the one device the planner preferred, with a `Reason` describing why an operation is or is not supported on a device. `estimatedCost(of:)` returns the per-operation cost weight the segmenter compared. These are documented at developer.apple.com/documentation/coreml [AppleCoreML].

The segmenter is a named pipeline inside the ahead-of-time compiler. The compiler canonicalizes the intermediate program for the platform, expands the requested compute-unit mask into a set of backend identifiers, then runs one of three segmenters over the graph. Table 5.3 names the segmenter's stages inside the ahead-of-time compiler, from eligibility through validity and cost to segmentation.

Table 5.3. The named stages of the placement segmenter inside the ahead-of-time compiler.

Stage	Component	Role
Eligibility	`ComputeUnitsToBackends`	expands the compute-unit mask into concrete backend identifiers
Validity	`AneValidator`, `BnnsValidator`, `MpsGraphValidator`	per-backend per-operation legality, one validator per backend
Cost	`EstimatorMILDecisionTree`, `ConstCostEstimator`	the per-operation cost the solver minimizes

Segmentation	`SegmenterCoarse`, `SegmenterGraph`, `SegmenterShortestPath`	greedy, graph-grouped, or shortest-path placement

The cost the shortest-path segmenter minimizes is the larger of compute and bandwidth time plus penalties. The per-operation compute cost is the larger of the compute time and the bandwidth time, drawn from the engine compute rate and bandwidth primitives `GetEngineGflopsPerS` and `GetEngineBwGbPerS`. The placement adds a fixed launch cost per segment and a transfer cost at each backend boundary.

$$\text{cost} = \max\left(\frac{\text{flops}}{\text{gflops}}, \frac{\text{bytes}}{\text{bw}}\right) + \text{launch} + \text{transfer}$$

The transfer cost is why the minimum-cost solution favors long single-backend runs: each boundary charges a tensor repack between the engine channel-interleaved fp16 layout and the host layout, so one engine segment is cheaper than several. The placement audit trail is itself a queryable compile output: the program library exposes a post-compile analytics dictionary, keyed by `selected_backend`, `backend_support`, `estimated_runtime`, `op_type`, `op_path`, and `validation_messages`, so the chosen backend and predicted runtime for each operation are readable without running inference.

5.4 Broker

A system daemon mediates access to the engine. Only that daemon and its per-user sibling hold the kernel gate that opens the device, so every other process reaches the engine through it. A client proves itself to the daemon over the interprocess channel, and the daemon performs the privileged device open on the client's behalf, returning a program handle the client then drives.

The daemon arbitrates across clients. There is one physical engine, so concurrent demand resolves by time-division on a single request queue rather than by partitioning the hardware. The daemon owns the shared program cache, keyed by a content hash of each compiled network, so a network already compiled by one client is a cache hit for the next. It holds a quality-of-service value on every request and adjusts a client's requested queue depth under contention, which keeps one client from depriving another of service. A request from a developer-signed program and a request from a system dispatcher arrive at the same broker, and the same queue arbitrates both.

The daemon is the one process holding the kernel gate, and its interprocess surface is the program-lifecycle protocol every client drives. The program-instance method holds the per-instance parameters verbatim in its selector, as listing 5.1 shows, with the residency, power, and statistics arguments named.

Listing 5.1. The daemon's program-instantiation selectors, with the per-instance residency, power, and statistics parameters held as named arguments.

```
# the daemon's central program-instance method (recovered selector, NSXPC _ANEDaemonProtocol)
createProgramInstanceForModel:modelToken:modelFilePath:qos:isPreCompiled:
  enablePowerSaving:skipPreparePhase:statsMask:memoryPoolID:enableLateLatch:
  modelIdentityStr:owningPid:cacheUrlIdentifier:aotCacheUrlIdentifier:
  optOutOfModelMemoryUnwiring:error:

# the weights-streaming variant: one resident base, a thin adapter per call
createProgramInstanceWithWeights:modelToken:qos:baseModelIdentifier:
  owningPid:numWeightFiles:error:
```

The on-the-wire instance is a C struct the daemon returns to the client, two opaque pointers, two 512-byte name buffers, the input and output port tables, a procedure record, and counters, whose recovered type encoding listing 5.2 gives.

Listing 5.2. The program-instance struct the daemon returns over the interprocess channel.

```
# ANEProgramInstanceStruct (recovered NSXPC type encoding)
^{ANEProgramInstanceStruct=^v^vQQC^[512c]^Q^{ANEProgramIOInfoStruct}
  ^{ANEProgramIOInfoStruct}C^[512c]^QC^{ANEProgramProcedureStruct}QIQQcQQiC}
```

5.5 Access-control model

The device has one user-space entry point. The kernel driver denies opening the user client without the entitlement `com.apple.ane.iokit-user-access`, and exactly two binaries on the system hold it: the daemon and its per-user sibling. Every other process proves itself to the daemon over the interprocess channel instead, gated by a `com.apple.aned.private.*` entitlement family that the daemon checks per connection and per method. Table 5.4 gives that entitlement family, what each member authorizes, and the count of system binaries holding it in this build.

Table 5.4. The entitlement family that gates the engine, with the count of system binaries holding each in this build.

Entitlement	What it authorizes	Holders
`com.apple.ane.iokit-user-access`	the hard kernel gate: privileged device open, compile, cache	2
`com.apple.aned.private.allow`	baseline: compile, load, and instantiate models through the daemon	18
`com.apple.aned.private.ANEAccess.allow`	the inference-client access grant	14
`com.apple.aned.private.adapterWeight.allow`	stream adapter weights onto a shared resident base model	5
`com.apple.aned.private.processModelShare.allow`	share one resident model across processes	4
`com.apple.aned.private.secondaryANECompilerServiceAccess.allow`	use the longer-duration secondary compiler service	1

A privileged subset of 27 binaries also holds a sandbox exception for the class `H11ANEInDirectPathClient`, which lets a latency-sensitive client open the low-latency user client and drive per-inference submission on its own connection rather than round-tripping each inference through the daemon. The exception grants no device access by itself: the daemon still performs the privileged open and returns the program handle. A developer-signed binary cannot assert the kernel gate, since it names a restricted user-client class that ad-hoc and development signing cannot claim. The direct route thus reaches the engine the same way a sanctioned application does, through the daemon. It authors its work at the model and program layer rather than at the kernel interface.

5.6 Compiler is its own gated service

Compilation does not happen in the calling process. The service that turns a network into the engine's program format is a separate sandboxed interprocess service, `com.apple.ANECompilerService`, reached through an `NSXPCConnection` named for it. That service vends a single entry point, the method `compileModelAt:csIdentity:sandboxExtension:options:tempDirectory:...:withReply:`, and admits a connection only through an entitlement gate. The service's `listener:shouldAcceptNewConnection:` delegate calls `valueForEntitlement:` on the connection against the string returned by `+compilerServiceAccessEntitlement`, then logs whether the client holds the entitlement or is missing it.

This is the concrete reason the service compiles a hand-authored or self-compiled program rather than the calling process compiling it. The service holds two entitlements an ordinary process does not: it writes

the system-protected compile cache under `rootless.storage.ane_model_cache`, and it decrypts under `coreml.decypt_allowed`. A caller thus hands its network to the service and receives the compiled bundle back, the same separation the daemon imposes on device access.

The service keeps a warm-start cache so a repeat compile of the same network is not re-paid. The cache is a nested directory tree of the form `cache/com.apple.e5rt.e5bundlecache/<os-build>/<hash>/`, keyed by the model hash `model.anehash`, which is a double SHA-256 over the program. A repeat compile of the same program reuses the cached bundle. The `force_recompilation` option defeats the warm start and compiles from scratch.

5.7 Entering the runtime below the framework

A caller that names the engine as the target enters at the execution-runtime layer rather than the model framework, so no placement planner is in the path. The runtime compiles a graph to the engine program format, loads it, and dispatches it, and the same call works whether or not a chip is in hand for the cost estimate.

The neutral workflow builds a graph, compiles it to a chip target, and dispatches, with no segmenter between the graph and the engine.

```
/* Enter at the execution-runtime layer: compiler -> library -> function -> stream. */
e5rt_e5_compiler_create_with_config(&compiler, config);            /* the runtime object */
e5rt_e5_compiler_compile(compiler, model_path, options, &library); /* to the engine format */

/* The program library vends the callable function, below any model framework. */
e5rt_program_library_retain_program_function(library, fn_name, &function);
e5rt_precompiled_compute_op_create_options_create_with_program_function(&op_opts, function);
e5rt_execution_stream_operation_create_precompiled_compute_operation_with_options(&op, op_opts);

/* The execution stream is the runtime's dispatch queue, with no segmenter above it. */
e5rt_execution_stream_operation_retain_input_port(op, "x", &in_port);
e5rt_io_port_bind_buffer_object(in_port, in_buf);
e5rt_execution_stream_create(&stream);
e5rt_execution_stream_encode_operation(stream, op);
e5rt_execution_stream_execute_sync(stream);
```

The framework layer is one client of this same runtime, and an application that names the engine is another.

6 Dispatching without Core ML

SUMMARY

The engine is reachable in five steps from ordinary user space: build a graph, compile it to the program format, load it, bind operand buffers, and dispatch. No placement planner is in the path, and the operations the compiler accepts require no entitlement. A single submission can drive several on-engine steps without a host round-trip per step, at the same rate as the steps issued one host call at a time, so the unentitled path is performance-complete for dispatch.

6.1 Five steps

Build a network description first. This is the layer-and-wiring graph for the work: the operations, their parameters, the weight blobs, and the input and output ports. The runtime accepts the description directly, so the graph does not have to come from a trained model file or pass through a converter.

Compile that description to the engine's program format. The runtime drives the compiler, which lowers the graph, lays the weights out for the streaming datapath, validates every operation against the native-layer catalog, and writes a loadable program to a content-addressed cache on disk. This is the costly phase, and it runs once per network.

Load the compiled program. The runtime opens the program from the cache and instantiates the function it exposes, preparing the program for execution on the device. A freshly compiled program pays a one-time cost on this first load to produce the loadable hardware form; a program already in the cache is a hit.

Bind operand buffers. Each external input and output of the program is a named port. Binding attaches a buffer to each port so the engine reads its inputs and writes its outputs in memory the caller prepared.

Dispatch last. The caller encodes the loaded program as an operation into an execution stream and submits the stream, synchronously or asynchronously, then reads the output buffers back when the work completes. Dispatch is the low-cost phase, and it runs in the hot loop.

The runtime that runs these steps is the engine's own C dispatch layer, the `e5rt_*` family exported from the Espresso framework. Every entry point returns an `int64_t` error code, zero on success, and an entry point that creates an object returns it through its first out-parameter, while the compile and retain calls return through their last. The five steps map onto the call sequence listing 6.1 gives.

Listing 6.1. The direct route, from compiling a network to dispatching it on an execution stream.

```
/* Compile (step 2): the runtime drives the out-of-process compiler. */
e5rt_e5_compiler_create_with_config(&compiler, config);
e5rt_e5_compiler_compile(compiler, model_path, options, &library);

/* Load (step 3): retain the callable function the program exposes. */
e5rt_program_library_retain_program_function(library, fn_name, &function);
e5rt_precompiled_compute_op_create_options_create_with_program_function(&op_opts, function);
e5rt_execution_stream_operation_create_precompiled_compute_operation_with_options(&op, op_opts);

/* Bind (step 4): attach a CPU buffer object to each named I/O port. */
e5rt_buffer_object_alloc(&buf, nbytes, /*type=*/0);
e5rt_execution_stream_operation_retain_input_port(op, "x", &in_port);
e5rt_io_port_bind_buffer_object(in_port, buf);

/* Dispatch (step 5): encode the op into a stream and submit, in the hot loop. */
e5rt_execution_stream_create(&stream);
for (;;) {
    /* fill input buffer via e5rt_buffer_object_get_data_ptr(buf, &ptr) */
    e5rt_execution_stream_operation_prepare_op_for_encode(op);
    e5rt_execution_stream_encode_operation(stream, op);
    e5rt_execution_stream_execute_sync(stream);   /* or async_submit */
    /* read output buffer back via its data pointer */
    e5rt_execution_stream_reset(stream);
}
```

A network expressed for the compiler is a netplist, the `.espresso.net` representation the compiler accepts alongside `.mil`. Its layers are dictionary entries the binary calls Units, keyed under a network body, with a `ProcedureList` of callable entry points naming the `InputList`, `OperationList`, and `OutputList`. A single matmul lowers to the native `MatrixMultLayerDesc`, whose backend operation is `anec.matmul` with `transpose_lhs` and `transpose_rhs` attributes.

The netplist keys are dictionary string constants the compiler parses verbatim, recovered from the runtime binary. They name the network body (`Networks`, `NetworkName`, `ProcedureList`, `Units`, `Weights`), the per-unit wiring (`Name`, `OperationName`, `OperationList`, `Bottom`, `Top`), the external ports (`InputList`, `OutputList`, `InputName`, `OutputName`, `InputType`, `OutputType`), and the port shape, a five-tuple plus an interleave factor and a dtype (`BatchSize`, `InputChannels`, `InputDepth`, `InputHeight`, `InputWidth`, `InputInterleave`, `OutputChannels`, `OutputInterleave`). Chapter 22 lists each key with its role.

A Unit is one layer. It has a `Type` tag, a `Bottom` wiring list naming its input symbols, the per-bottom and output dtypes, and a `Params` sub-dictionary of typed attributes. The directed-graph wiring is by symbol name, a Unit's `Bottom` referencing another Unit's `Name` or an external `InputName`. The execution order is the network body's `Units` array. Table 6.1 maps the execution-runtime call families onto the compile, load, bind, and dispatch phases of the direct route.

Table 6.1. The execution-runtime call families, mapped onto the compile, load, bind, and dispatch phases of the direct route.

Family	Phase	Representative entry points
`e5rt_e5_compiler_*`	compile	`create_with_config`, `compile`, `is_new_compile_required`
`e5rt_program_library_*`, `e5rt_program_function_*`	load	`create`, `retain_program_function`, `load_for_execution`
`e5rt_precompiled_compute_op_*`	load	`create_options_create_with_program_function`
`e5rt_buffer_object_*`, `e5rt_io_port_*`	bind	`alloc`, `get_data_ptr`, `bind_buffer_object`

Family	Phase	Representative entry points
`e5rt_execution_stream_*`	dispatch	`encode_operation`, `execute_sync`, `submit_async`, `reset`

One network is compiled and loaded a single time, and the same program is dispatched against fresh operands on every call. Figure 6.1 shows the split between the one-time setup and the per-call hot loop.

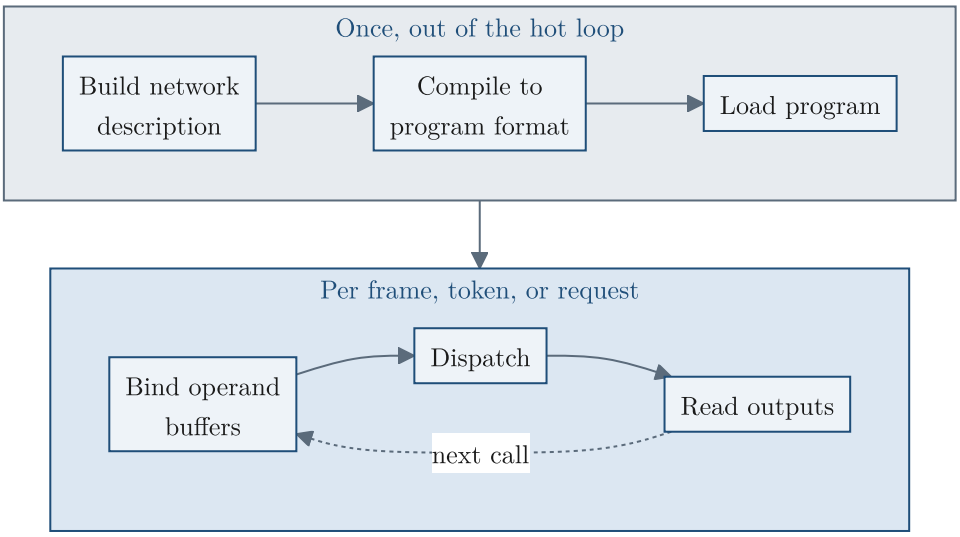

Figure 6.1. The compile-once, dispatch-many split between one-time setup and the hot loop.

6.2 Compiler and runtime options

Compiler and runtime options are not a fixed-layout struct. They are a string-keyed dictionary of `std::any` values, which is why setting an option changes no fixed byte offset in the option handle. Each key has its value type in the name, for example `forceRecompilation<bool>`, `segmenter<std::string>`, and `computeDeviceTypesAllowed<std::vector<ComputeDeviceType>>`, and the compute-device mask value `0x4` selects the engine.

Most of the runtime is reachable but unused by an ordinary dispatch. The runtime exposes 292 exported entry points, of which about 275 are callable and about 52 are exercised by a normal dispatch. The largest reachable but unused capabilities are zero-copy input and output through an image surface or a graphics buffer, firmware-side data chaining, a warm-start cache, a quality-of-service control, mutable weights, and an asynchronous submission with a timeout. These are present on the direct path and are not exercised by the standard flow.

6.3 No planner and no entitlement

The direct route has no placement planner. The public framework path loads a model and selects engine eligibility through a compute-unit setting, segments a model across the processors, and decides per segment where the work runs, and the caller is not told the result [AppleCoreML]. On the direct route the caller has already decided: the network is compiled for the engine and dispatched to the engine, with no segmentation step in between.

The operations the compiler accepts require no special entitlement. The runtime, the compiler, and the dispatch path are all reachable from ordinary user space. The set of operations is the catalog the compiler validates: an operation the compiler accepts compiles and dispatches without a privileged handle, and the compiler rejects an operation outside that set at compile time rather than the dispatch path gating it. Reachable here means no entitlement is required, not that the path is supported: these interfaces are private, unsupported, version-fragile across operating-system updates, and not App-Store-safe. The boundary that does require an entitlement is a separate matter, taken up in chapter 8.

6.4 Unentitled path is performance-complete

Host dispatch is not the limiting cost on the direct path. A single submission can drive several on-engine steps without a host round-trip per step. The mechanism is the resident state from chapter 2: a buffer stays in the engine's working set across steps, so the output of one step is the input of the next in place. The submission advances through the steps without returning to the host between them. The host supplies the small per-step inputs and reads the held buffers back at a checkpoint, rather than copying the full state across the boundary twice per step.

The arithmetic confirms it. A submission that drives the steps on the engine runs at the same rate as the same steps issued one host call at a time, so removing the per-step host calls does not speed the work up. The unentitled direct path is performance-complete for dispatch: it reaches the same throughput as any path with more privilege.

The stream offers one synchronous and two asynchronous submission forms. The synchronous form blocks the caller until the encoded stream completes. The lightweight asynchronous form returns a submit and a complete identifier, and the full asynchronous form delivers an error object and accepts a timeout, the timeout existing because an asynchronous completion can hang. Single-stream pipelining is sound: keep several operations in flight, each with its own completion event, and drain by waiting on the events. Overlapping two or more streams at once is the unsound path, where the completion event of the second and later streams never notifies and the waiter blocks; the low-latency-async-event stream option and the timeout argument are the controls that break that wait. The caller holds resident state across steps by binding a buffer object once and reusing it: the output port of one step is the input port of the next in the same buffer. The caller re-encodes the prepared operation against fresh small inputs without rebinding the held buffer.

6.5 Compiling once, dispatching many

The five real steps collapse to two phases against the neutral API: compile and load once out of the hot loop, then bind and dispatch against fresh operands on every call. The caller reuses the compiled program across calls, so the costly compile phase runs a single time.

Build and compile are above the loop; bind and dispatch are inside it.

```
/* Once, above the loop: compile to a program library, then retain the callable function. */
e5rt_e5_compiler_compile(compiler, model_path, options, &library);
e5rt_program_library_retain_program_function(library, fn_name, &function);
e5rt_precompiled_compute_op_create_options_create_with_program_function(&op_opts, function);
e5rt_execution_stream_operation_create_precompiled_compute_operation_with_options(&op, op_opts);
e5rt_execution_stream_create(&stream);      /* ports bound once via e5rt_io_port_bind_buffer_object */

for (;;) {                                  /* the hot loop: dispatch the same op per call */
    e5rt_execution_stream_operation_prepare_op_for_encode(op);
    e5rt_execution_stream_encode_operation(stream, op);
    e5rt_execution_stream_execute_sync(stream);
    e5rt_execution_stream_reset(stream);    /* read outputs back, then reset for the next call */
}
```

7 Weights and compression

SUMMARY

The engine reconstructs four compressed weight forms at the multiplier input, and on the unentitled direct route a form either streams its compressed bytes for a bandwidth gain or folds to dense fp16 for none. Which outcome applies to a form is set by the target chip: the M1 streams only int4 lookup-table at 2.37 times fp16 and structured sparsity at 1.55 to 1.64 times, and folds int8 and blockwise affine. The M5 streams all four at 1.6 to 1.8 times. Choose the form by what streams on the target, and on the M1 fall back to int8 only where its halved stored size pays, since the int8 fold expands to fp16 in DRAM and yields no bandwidth there.

A compiled network holds its weights as a separate stream that the engine reads from DRAM on every dispatch. The weight stream is the primary cost of any layer whose arithmetic intensity is low, which covers most decode-shaped and projection-heavy work. Compressing the weights moves fewer bytes across that stream; it does more than shrink a stored file.

7.1 Compression is a bandwidth feature on the direct route

Compressed weights reach the engine through the direct runtime described in chapter 5. They are not gated behind an entitlement, and the operations that reconstruct them are accepted by the compiler without special privilege. The reconstruction happens at the multiplier input, consistent with the fp16 datapath of chapter 3: a compressed weight is turned back into fp16 before it reaches the multiply array, and the multiply that follows is the same fp16 multiply as for an uncompressed weight.

The word compression covers two distinct outcomes. A form that streams reaches the engine in its compressed bytes and is decompressed on chip, so fewer bytes cross DRAM and the layer runs faster when it was bandwidth bound. A form that folds is reconstructed to a dense fp16 constant before the dispatch, so the bytes that cross DRAM are full-width fp16 and the layer gets no bandwidth gain. The target chip sets which outcome applies to a form, not the frontend.

7.2 Compression forms

The engine reconstructs four weight-compression forms, the same linear quantization, palettization, and pruning the conversion tools document as a model-size feature [AppleCoreMLTools]. Per-tensor and per-channel int8 hold the weight as a single byte per element with an fp16 scale, one scale for the whole tensor or one per output channel, reconstructed as the scale times the byte. The reconstruction is the affine dequantization

$$w = s\,(q - z)$$

with stored quantized byte q, fp16 scale s, and zero point z, the encode being $q = \mathrm{round}(x/s) + z$. The zero point folds to zero on the M1 generation, so the int8 form there is symmetric and the relation reduces to $w = s\,q$ with $z = 0$, as listing 7.1 shows at the multiplier input alongside the symmetric form.

Listing 7.1. The affine int8 dequantization evaluated at the multiplier input, with the symmetric form the M1 reduces to.

```
# affine int8 dequant, evaluated at the multiplier input
w_fp16 = scale * (q - zero_point)        # general affine form
w_fp16 = scale * q                       # M1: zero_point folds to 0, symmetric only
# q is int8 in [-127, 127]; scale is fp16, scalar or one per output channel.
```

Each compressed form is declared to the compiler as a single `constexpr_*` reconstruction op that folds into the weight descriptor rather than becoming a standalone backend operation, one per form as listing 7.2 names them. The op count per form is fixed, and a violation is a hard validation error.

Listing 7.2. The single reconstruction op that declares each compressed weight form to the compiler.

```
# the constexpr reconstruction op per form (one MIL op each, folded into the conv/linear weight)
constexpr_affine_dequantize(q, scale, zero_point)   # int8 affine:  w = scale * (q - zero_point)
constexpr_lut_to_dense(indices, lut)                # int4 palette: w = lut[indices]
constexpr_sparse_to_dense(mask, nonzeros)           # sparsity:     scatter into mask positions
constexpr_blockwise_shift_scale(q, scale)           # blockwise:    one scale per contiguous block
```

The int4 lookup-table form holds a four-bit index per element into a sixteen-entry fp16 codebook, and reconstruction is a table lookup with no arithmetic since the codebook is already fp16. Structurally it is a palette rather than an arithmetic dequantization, which is why it behaves differently from int8 on the streaming question.

A worked example shows the packing and the decode. The four weights `[1.0, 0.0, 0.0, 1.0]` index a sixteen-entry fp16 codebook whose first two slots hold `0.0` (entry 0, `0x0000`) and `1.0` (entry 1, `0x3c00`). The index stream `[1, 0, 0, 1]` packs two four-bit indices per byte, low nibble first, so the four weights occupy the two bytes `0x01` and `0x10`. Decoding reads each nibble as a table index, recovering `[1.0, 0.0, 0.0, 1.0]`.

Structured sparsity holds a one-bit mask marking the nonzero positions plus the packed fp16 values of those nonzeros. The mask costs one bit per element and the values cost two bytes per surviving element, so a weight that is half zeros or more stores well below its dense size. Reconstruction scatters the values back into the masked positions exactly apart from the fp16 rounding of the kept values. Blockwise affine holds a separate scale for each contiguous block of elements, finer than a per-channel scale and so lower in quantization error.

7.3 What streams on each family

Table 7.1 gives, for each compressed weight form, whether it streams or folds to dense on each chip generation, with the measured speedup.

Table 7.1. Whether each compressed weight form streams or folds to dense per chip generation, with the measured speedup.

Form	M1/H13	A14/M2	A15/M3 and later	M5/H17s	Speedup
int8 (per-tensor / per-channel)	fold	stream	stream	stream	M5 1.6-1.8x; M1 folds, no stream gain
int4 lookup-table	stream	stream	stream	stream	M1 2.37x; M5 1.6-1.8x

Form	M1/H13	A14/M2	A15/M3 and later	M5/H17s	Speedup
structured sparsity	stream	stream	stream	stream	M1 1.55-1.64x at 0.43x dense bytes; M5 1.6-1.8x
blockwise affine	fold	fold	stream	stream	M5 1.6-1.8x; M1 and M2 fold, no stream gain

On the M1, of the H13 generation, only the int4 lookup-table form streams natively. A bandwidth-bound stack of one-by-one convolutions runs 2.37 times faster with int4 weights than with fp16, measured on the M1, because the four-bit indices move at a quarter of the fp16 byte count. Structured sparsity also streams natively on the M1: a convolution stack that is about sixty-three percent zeros runs 1.55 to 1.64 times faster than the same weights stored dense, at 0.43 times the dense weight bytes. The output is bit-faithful to the dense reference apart from fp16 rounding.

The int8 and blockwise forms fold on the M1. The accuracy cost is the per-output int8 quantization: the int8 weight tracks the fp16 result at a cosine near 1.0, with a relative error near one percent against an fp32 reference where fp16 is near two parts in ten thousand.

From the A14 generation the int8 and sparse forms also stream, alongside int4, measured on the M2 of the H14 generation at 0.64 and 0.54 times fp16 on a bandwidth-bound matmul. On the A14 and M2 the int8 weight is dispatched as int8 and reconstructed from half the bytes. The measured latency runs from 0.85 times fp16 at a two-thousand-wide weight down to 0.52 times fp16 at an eight-thousand-wide weight, deeper as the weight grows and the stream dominates. Blockwise affine does not stream on the A14: it still folds there, measured at 0.985 times fp16, a near-zero bandwidth gain, and it first streams from the A15 generation. On the M5, of the H17s generation, all four forms stream, at a measured 1.6 to 1.8 times fp16 on bandwidth-bound layers.

The streaming-versus-folding split is a hardware-abstraction-layer decision, not a property of any single reconstruction operation. Every weight-bearing operation is legal on every chip, so the boundary is in a set of feature bytes the compiler reads from the per-chip table. In that table one master byte enables weight streaming at all and a cluster of per-format bytes admits each compressed form by generation. Table 7.2 gives those feature bytes and the generation each switches on.

Table 7.2. The hardware-abstraction-layer feature bytes that gate compressed-weight streaming, by chip generation.

Hardware-abstraction-layer byte	Role	M1/H13	A14/M2	A15/M3	A16, A17, M5
`+0x48f`	kernel-streaming master	1	1	1	1
`+0x528`, `+0x532`, `+0x537`	per-format gates that switch on at A14	0	1	1	1
`+0x520`, `+0x523`, `+0x533`, `+0x539`	per-format gates that switch on at A15	0	0	1	1
`+0x529`	palette and stride gate	1	1	1	1

The master byte is set from the A13 generation, which is why the M1 streams anything at all, and the palette gate is set on the M1, which is why the int4 lookup-table form streams there. The per-format gates for the

affine int8 and blockwise forms are clear on the M1 and switch on at the A14 and A15 generations, which is why those two forms fold on the M1 and stream on the newer parts. Structured sparsity streams on the M1 by a separate route: it is held as a mask-and-values weight operand under the master byte rather than as a palettized kernel coefficient, so it streams under the master gate independent of the per-format cluster. The int8 floor at A14 is confirmed on the M2 silicon, and the blockwise floor at A15 is read from that gate pattern and not yet confirmed on the intermediate silicon.

On the M1 the int8 fold is a stored-size saving only. The weight is half the size on disk, but it is expanded to a dense fp16 constant in DRAM before the data-movement step, so a weight-streaming-bound matmul moves full-width fp16 bytes and runs at the fp16 latency, with no bandwidth gain. The int8 weight first streams as int8 on the A14 and M2 generation, where it is dispatched as int8 and dequantized at the multiplier input rather than materialized to a dense fp16 constant in DRAM. A matmul-path measurement on the A14 and M2 makes this concrete, and table 7.3 gives it: at a weight matrix wide enough to leave the dispatch floor, the int8 and fp16 weight streams reach the same effective bandwidth against their stored bytes, while the int8 form moves half the bytes.

Table 7.3. The int8 matmul-path weight stream against fp16 on the A14 and M2, the latency ratio approaching one half as the weight grows.

Weight width $K = N$	fp16 latency	int8 latency	int8 / fp16
2048	0.290 ms	0.253 ms	0.87
4096	0.740 ms	0.447 ms	0.60
8192	2.610 ms	1.351 ms	0.52

7.4 Choosing a form

What streams on the target drives the choice of compression form, chip by chip. On the M1, prefer structured sparsity for a weight that is half zeros or more, since it streams and is lossless apart from fp16 rounding, and prefer the int4 lookup table otherwise, since it is the densest form that streams there. Reserve int8 on the M1 for the case where sixteen levels are too coarse, taking it for its halved stored size, since the form folds to fp16 in DRAM and yields no bandwidth there. On the M5 and the newer generations, where every form streams, choose by accuracy per byte rather than by what streams.

The wide accumulator of chapter 3 is what makes streamed low-precision weights safe on the layers that tolerate them. A streamed weight is reconstructed to fp16 and then enters the same fp16 multiply and wide reduction as a dense weight, so the only precision lost is the quantization of the weight itself, not the accumulation. On convolution, matrix multiply, and normalization, whose partial sums stay in range, the reduction holds and the compressed weight keeps the layer's accuracy. A cancellation-heavy step has no fp16-safe form, compressed or not, for the reason given in chapter 3.

7.5 Patching weights in a compiled program

A host can patch a compiled program's weights in place without recompiling. Each weight tensor occupies a decoded region of the program image with a known tiling. A convolution weight is in a `0xC0`-stride layout and a matrix-multiply weight in a `0x40`-stride layout, and editing the weight values leaves the program descriptor unchanged. A host can thus swap new weights into an already-compiled program rather than rebuilding it, which makes a weight-only update inexpensive.

The driver has a per-patch-mutable-buffer accounting path that confirms the route, `ANEScheduler::pendingRequestsPerPatchMutableBuffer`. The descriptor names the operations and binds the buffers, and a weight-value edit touches neither, so the same compiled program runs with the new coefficients.

7.6 Automating the choice

The estimate classifies a layer as compute bound or bandwidth bound so the procedure chooses a form only where the layer is bandwidth bound and a stream would help. The procedure keeps the smallest form that streams natively on the target and clears an accuracy tolerance against an fp32 reference. A form that folds to dense fp16 moves the same bytes as fp16, so it cannot help a bandwidth-bound layer; only a streaming form does. Which forms stream depends on the target: the M1 streams int4 and sparse while int8 and blockwise fold, and the A14 and later stream int8 as well. Sparsity applies only when at least half the weight is zero, and the candidates are tried smallest-bytes-first, int4 before sparse before int8. Here `accuracy_error` round-trips the weight through the candidate form and compares the layer output against the fp32 reference.

```
tolerance = 0.01    # max relative error of the layer vs an fp32 reference

def native_streams(chip):
    if chip == H13:   return [int4, sparse]       # M1: int8 and blockwise fold
    if chip >= H14:   return [int4, sparse, int8]

def choose_weight_form(layer, weights, chip):
    if not is_bandwidth_bound(layer, chip):
        return fp16
    candidates = native_streams(chip)
    if fraction_zero(weights) < 0.5:
        candidates.remove(sparse)
    for form in sorted(candidates, key=bytes_per_weight):
        if accuracy_error(form, weights, layer) <= tolerance:
            return form
    return fp16

form = choose_weight_form(layer, W, chip=H13)     # M1 -> int4 (streams, 2.37x)
program = compile(graph_with(W, form), chip=H13)
```

On the M1 the procedure chooses a folding form, int8 or blockwise, only where the halved stored size or a finer scale pays on the matmul path, not for a stream gain.

7.7 Reference: per-form streaming and measured cost

Table 7.4 collects the per-form streaming behavior and measured cost by chip generation.

Table 7.4. The per-form streaming behavior and measured cost, by chip generation.

Constant	Generation	Value
int4 lookup-table stream speedup	M1/H13	2.37x fp16
Structured sparsity stream speedup	M1/H13	1.55-1.64x fp16
Structured sparsity stored bytes	M1/H13	0.43x dense
int8 fold on M1 (stored-size saving, no stream gain)	M1/H13	1.0x fp16 latency
int8 stored size on disk	M1/H13	0.5x fp16
int8 accuracy versus fp32 reference	M1/H13	cosine near 1.0, relative error near 1%
int8 matmul latency, 2k-wide weight	A14/M2	0.85x fp16
int8 matmul latency, 8k-wide weight	A14/M2	0.52x fp16
int8 stream speedup	A14/M2	0.64x fp16
Structured sparsity stream speedup	A14/M2	0.54x fp16
Blockwise affine fold	A14/M2	0.985x fp16 (no stream gain)
All four forms stream speedup	M5/H17s	1.6-1.8x fp16
Streaming master gate	A13	on
int8 per-format stream floor	A14	confirmed on M2 silicon

Constant	Generation	Value
Blockwise affine per-format stream floor	A15	predicted, not silicon-confirmed

The compressed weight element types are a subset of the engine element-type catalog the kernel descriptor records. The forms that bear on weight encoding are the integer lanes, fp16 dense and codebook entries, and packed palette indices. Table 7.5 gives those weight-relevant entries of the catalog with the width each has.

Table 7.5. The weight-relevant entries of the engine element-type catalog, with the width each has.

Element type	Width	Role in weight encoding
int8	1 byte	affine int8 lane
uint8	1 byte	affine uint8 lane
float16	2 bytes	dense weight, codebook entry, scale, and bias
uint4	4 bits	4-bit palette index
int4	4 bits	4-bit signed, palette-only on the M1
e4m3	1 byte	fp8, gated off on the M1

There is no int4 arithmetic lane in the datapath, so the four-bit value is always a palette index into the sixteen-entry fp16 codebook, which is why the element-type table marks int4 palette-only on the M1.

8 Entitlement boundary

SUMMARY

The direct route reaches the engine's useful compute, and four features are behind the framework loader or an entitlement: three-dimensional convolution, native stateful types, bf16 program input and output, and flexible or symbolic shapes. Each gated feature passes an earlier layer and fails at the one that runs the work, and none of the four runs on the engine on the M1 under either route. One hard limit is below all of this: a hand-built or self-compiled program is rejected at load with error `0xe00002e2`, so the reachable surface is everything the system daemon's compiler will accept.

The direct route reaches the engine through the execution runtime, below the model framework, with no placement planner in the path. The sanctioned route reaches it through the model framework, which adds a segmenter, model container, and set of loader features the direct route does not have. The direct route reaches the engine's useful compute; four features are behind the framework loader or behind an entitlement. This boundary is about technical reachability alone; whether a path is supported by Apple or permitted for distribution is a separate and stricter matter, treated in the front-matter status note.

8.1 What the direct route reaches

The operations that on-device perception and numerics networks are built from compile and run on the direct route without any entitlement. Two-dimensional convolution and its transpose, matrix multiply, fused attention, the normalizations, the activation tables, pooling, elementwise arithmetic, and the data-movement operations all run, bounded only by the per-chip limits of chapter 4. Compressed weights stream where the family supports it and reconstruct to fp16 where it does not. The compiler accepts these operations and the runtime dispatches them, and an entitlement gates none of them. An entitlement gates a separate set of loader-tier features instead, not the compute the direct route already reaches.

8.2 Features that are gated

Four features are attested in the hardware capability tables or recognized by the compiler frontend, yet do not run on the direct route. Each is gated, and the gate is at a different layer in each case, as table 8.1 gives with what gates each and how the direct and framework routes behave.

Table 8.1. Features blocked on the direct path, what gates each, and how the direct and framework routes behave.

Feature	Gated by	Direct path	Entitled/framework path
Three-dimensional convolution	Missing backend lowering on every device mask	Frontend recognizes it, lowering fails with a not-implemented rejection	Segmenter places it on the central or graphics processor, not the engine
Native stateful types (state, ring buffer)	Absent counter-and-event direct-memory-access engine in the M1 descriptor	Compile fails; reserved operation classes do not lower	Model container wires the stateful types; engine still lacks the primitive on the M1
bf16 program input and output	Serialization layer and datapath; bf16 is not among the eleven program-I/O dtype codes	Compile fails with an unsupported-dtype rejection	Framework accepts the bf16 array, then casts to fp16 before the engine segment

Feature	Gated by	Direct path	Entitled/framework path
Flexible and symbolic shapes	Runtime path, not the compiler; the symbolic-shape gate is on for the M1	A symbolic dimension parses, then fails to lower	Compiles a small set of fixed shapes ahead of time and dispatches the nearest

Each gated feature passes an earlier layer and fails at the one that runs the work. Listing 8.1 shows where each is admitted and where it stops, for three-dimensional convolution, a bf16 program output, and a symbolic shape.

Listing 8.1. Where each gated feature is admitted at one layer and rejected at the next, for three-dimensional convolution, a bf16 program output, and a symbolic shape.

```
# 3D convolution: the frontend recognizes a static-weight 3D conv (it enforces
# "3D Convolution does not support dynamic weights"); lowering then fails on every mask.
conv(input, weight, kernel=[d, h, w])
  -> frontend: accepted
  -> lowering: "Not implemented": Some ops are not supported on any of the specified backends

# bf16 program input/output: not among the eleven program-I/O dtype codes.
# ANECIRDataType = {0 int4, 1 uint8, 2 int8, 3 fp16, 4 fp32, 5 int16, 6 uint16,
#                   7 int32, 8 uint32, 9 int64, 10 uint64}   # bf16 absent
function_output dtype = bf16
  -> compile: "Unsupported function output dtype bf16"   # no code to serialize it

# symbolic shape: a "?" dimension parses, then will not lower on the direct path.
tensor<fp16, [1, ?, 64]>
  -> parse: accepted (the symbolic variable s0 binds)
  -> lowering: "Not implemented": Some ops are not supported on any of the specified backends
```

A missing backend lowering gates three-dimensional convolution. The hardware capability tables advertise a kernel-depth dimension, and the frontend recognizes a three-dimensional convolution with a static weight, but lowering then fails on every device mask with a not-implemented rejection. No backend on the M1 has a code-generation path for the three-dimensional convolution operation class. The sanctioned route does not run it on the engine either: its segmenter places the operation on the central processor or the graphics processor instead, so on this silicon there is no on-engine path to it for any caller.

An absent hardware primitive gates native stateful types. The model framework documents the stateful type and flexible input shapes at developer.apple.com/documentation/coreml, and this account corrects the impression that those documented capabilities run on the engine directly [AppleCoreML]. The compiler reserves the operation classes for resident state and ring buffers, the operation table even naming a ring-buffer writer and reader and the counter-and-event direct-memory-access opcodes, but the in-place update needs that counter-and-event engine, and it is stubbed out of the M1 hardware descriptor. Every register setter for it, the source and destination base addresses, shape and stride setters, atomic-operation and counter-mode setters, and wait-event address and value setters, has a body that asserts the engine is unsupported on this architecture. A program that declares native resident state fails the compile, and this gate is stronger than an entitlement: the silicon path does not contain the engine, so no entitlement, property, or compiler option synthesizes it. A resident cache on the M1 is built instead from a shared buffer kept live across dispatches, the documented zero-copy path for this generation. The counter-and-event engine arrives on a later generation, so native resident state is a per-family feature there, not an M1 one.

The bf16 program input and output type is gated by the serialization layer and by the datapath. The dtype enumeration that declares a program's input and output types has eleven codes, covering fp16, fp32, and the integer widths, and bf16 is not among them, so a program that declares a bf16 input or output fails the compile with an unsupported-dtype rejection. The framework accepts a bf16 array at its boundary, but it casts to fp16 before the engine segment, so the engine runs no bf16 datapath under the sanctioned route

either. The fp16 datapath of chapter 3 already holds the accumulation precision a bf16 declaration would otherwise imply.

Flexible and symbolic shapes stop at the runtime path, not at the compiler. The compiler has a full symbolic-shape system, with its master gate on for the M1, yet the direct runtime path will not drive it: a program with a symbolic dimension parses and then fails to lower. The sanctioned route reaches flexible shapes by compiling a small set of fixed shapes ahead of time and dispatching the nearest, with a dynamic remainder on the central processor. That is bucketed fixed-shape specialization, not one symbolic program on the engine, and the direct route matches it by padding to a fixed maximum or compiling a small set of length buckets, with the compile cache making a repeated shape free.

8.3 Image-input boundary

A fifth feature is at the boundary in a sharper form: direct image-format input, where a camera or video surface is sent to the engine in its native four-character-code pixel format with no host-side conversion to fp16. The operation that performs this on-engine conversion, `pixel_buffer_to_tensor`, is present in the intermediate-representation parser on the direct route and parses without an unknown-operation rejection. Its input is a distinct surface type, `pixel_buffer<format, shape, bytes_per_row>`, and the format token is a `FMT_*` enumeration that maps onto the per-chip interchange-format table, so the grammar, format enumeration, and type rules are all reachable and solvable on the direct route.

The boundary is at lowering. On the direct route the intermediate program parses and type-checks and reaches the engine backend, but it does not lower there, so the conversion cannot be compiled on the direct route. The entitled framework route supplies the image-input descriptor and the surface setup that the lowering needs, so direct four-character-code input is a framework-route feature on the M1. The terminal direct-route form is an on-engine integer-to-fp16 dequantization of a plain byte input, which avoids the unsupported operation and still removes the host-side conversion.

8.4 What the boundary means concretely

The choice between the routes is a choice of loader and convenience, not of reachable compute. Vision, encoders, on-device numerics, and training of the supported operation set are built from the operations the direct route compiles and dispatches, none of which the gated set touches. The entitled path adds a bounded list: a segmenter that places arbitrary models across three devices with fallback, model container with flexible-shape and stateful wiring, and loader-tier feature set. It does not add a different engine: the four gated features are absent on the M1 under both routes, or are matched by a direct-route construction that does the same work.

8.5 Load-time signature check

One hard limit is below the compute and defines the whole access model: a hand-built or self-compiled program cannot be loaded onto the engine. The kernel driver verifies every submitted program before it reaches the firmware, by a corecrypto signature check over the program bytes and a trustcache check on the backing file's vnode, and rejects a program that fails either check at load with error `0xe00002e2`. The only program the kernel will load is one the system daemon compiled and signed in place, so a caller cannot author a network binary by hand and submit it. This is why the direct route does not build a loadable binary: it authors a network in the intermediate representation and hands it to the daemon, which compiles and signs it on the caller's behalf, and the caller then drives the resulting signed program. The reachable surface is thus everything the daemon's compiler will accept, not everything the engine could in principle run.

8.6 Unentitled dispatch path

The kernel driver rejects a binary the client signs itself at load with `0xe00002e2`, so the daemon's compile-and-sign step is not optional. The path has three steps. First, the client authors the network as intermediate language rather than a loadable binary, since a self-authored binary never passes the load check and the client does not produce the final program itself. Second, it hands that intermediate language to the system daemon, the one process able to sign: the daemon compiles and signs it in place, and the returned program carries the signature and trustcache trust the kernel load check requires. Third, the client drives the returned signed program over the kernel interface directly, where the corecrypto signature and trustcache checks pass because it is daemon-signed and the program loads onto the engine.

8.7 Reference: the boundary constants

Table 8.2 collects the constants of the entitlement boundary on the M1.

Table 8.2. The constants of the entitlement boundary, M1/H13.

Constant	Value
Program-load rejection code	`0xe00002e2`
Program-I/O dtype codes (`ANECIRDataType`)	11 codes: 0 int4, 1 uint8, 2 int8, 3 fp16, 4 fp32, 5 int16, 6 uint16, 7 int32, 8 uint32, 9 int64, 10 uint64
bf16 program input or output	absent from the 11 codes; compile rejected
Three-dimensional convolution	no backend lowering on any device mask
Native stateful types	counter-and-event direct-memory-access engine stubbed out of the M1 descriptor
Flexible and symbolic shapes	parse and bind, then fail to lower on the direct runtime path
Load-time signature checks	corecrypto signature over program bytes plus trustcache vnode-trust check
Entitlement-rejection return code	`0xe00002c7` (`kIOReturnUnsupported`)
Native counter-and-event engine	every register setter asserts unsupported on this architecture
Direct image-format input	`pixel_buffer_to_tensor` parses, then does not lower on the direct route

The entitlement check the direct route never trips returns a distinct code from the program-load rejection. The program-load check returns `0xe00002e2` from the signature and trustcache check, while the entitlement gate that guards the higher inference-tier features returns `0xe00002c7`, `kIOReturnUnsupported`, the code a client gets when a gated feature's entitlement is absent. The four gated features above fail upstream of either kernel code, inside the compiler or the firmware descriptor, which is why a host-side entitlement moves none of them.

Part III

Performance and Fit

09 Roofline

Two ceilings, the ridge point, the 2 MB working set, and the dispatch floor.

10 Power and efficiency

Engine draw, energy per FLOP, and the efficiency margin over the GPU.

11 ANE, GPU, and CPU

Which processor leads on each workload class, by speed and by energy.

12 Across the chip family

How the roofline scales from the M1 to the M5, predicted then measured.

9 Roofline

SUMMARY

A layer is compute-bound only above 141 FLOP per byte; below it the engine runs on the memory slope. Keep every operation's working set under 2 MB or it streams its full activation from DRAM. Every dispatch pays a 0.23 ms floor, so batch or fuse until the compute clears it.

The Apple Neural Engine is a roofline machine [Williams2009]. Two ceilings bound its performance on any layer: the multiply array sets a peak compute rate, and how fast operands stream from DRAM sets a peak bandwidth. A layer runs at whichever ceiling its arithmetic intensity reaches first. The constants below are the M1 instance, the measured silicon. The roofline has the same form on every chip in the family; only its constants change, scaling with core count and clock, which chapter 12 covers and the M5 confirms.

9.1 Roofline ceilings and ridge point

The compute ceiling depends on how it is measured, and two figures bracket it. The overhead-isolated matmul slope is about 12 fp16 TFLOP/s: a fused matmul chain runs as one dispatch, so deepening it grows the compute at fixed input and output cost. The per-layer slope cancels that fixed cost, leaving the marginal rate. A direct absolute measurement of one large matmul is lower, about 4.8 fp16 TFLOP/s on a 4096-dimension matrix multiply whose call time is far above the dispatch floor. That is about 87 percent of the roughly 5.5 TFLOP/s theoretical peak. A single conv dispatch is lower still, about 1.8 TFLOP/s, because it pays the fixed cost the slope subtracts away.

The M1 fp16 peak is best read as the sequence of consistent numbers listed in table 9.2, from the 1.8 TFLOP/s conservative headline through the 3.25 TFLOP/s effective peak to the 4.8 TFLOP/s saturating ceiling. Apple markets the engine in int8 operations per second, the M1 at about 11 TOPS [AppleANE], and the fp16 rate is by convention about half of an int8 figure, which puts the fp16 theoretical peak near 5.5 TFLOP/s. The 4.8 TFLOP/s figure is the rate one large matrix multiply holds after it crosses the 2 MB working-set threshold, while a fused chain whose activations stay under the threshold sustains more. That is why chapter 10 gives a fused steady state near 8.1 TFLOP/s. The 5.5 TFLOP/s figure is a convention derived from the marketed int8 TOPS rather than a measured hardware ceiling.

The int8 path is faster than fp16 rather than equal to it. A measured int8 convolution runs about 1.4 times faster than the fp16 form and trends toward 2 times at scale. The multiply array has a double-int8 mode that packs two int8 products into one multiply-accumulate, so int8 arithmetic runs at about twice the fp16 rate on the same array. This is distinct from the compressed-weight path of chapter 7, where an int8 weight is dequantized to fp16 and then enters an ordinary fp16 multiply: there the saving is in stored bytes, here the saving is in compute cycles. The int8 path thus cuts compute cycles as well as storage.

The bandwidth ceiling is the rate at which operands cross the DRAM boundary. The engine sustains about 85 GB/s of DRAM bandwidth measured at the memory controller, and the roofline below holds that figure as its B. A direct wall-clock measurement of weight streaming runs lower: a large matrix multiply with one output row reads each weight once and saturates at about 51 GB/s, roughly 60 percent of that 85 GB/s ceiling. That figure matches the compiler's own internal bandwidth constant of 50 GB/s. The achieved-streaming rate is noted alongside the ceiling in table 9.2. A small elementwise op reaches less than either. A single relu stream saturates near 10 GB/s effective, because the per-dispatch overhead caps it well below the DRAM rate.

The attainable rate is the lower of the two ceilings at a layer's arithmetic intensity I, measured in FLOP per byte.

$$R(I) = \min(P,\ I\,B), \qquad P \approx 12 \text{ fp16 TFLOP/s}, \qquad B \approx 85 \text{ GB/s}$$

A layer with intensity I is compute-bound when $I\,B \geq P$ and bandwidth-bound otherwise. The ridge point is where the two ceilings cross, near 141 FLOP per byte, the compute roof over the bandwidth roof, 12×10^{12} over 85×10^{9}.

$$I^{*} = \frac{P}{B} = \frac{12 \times 10^{12}}{85 \times 10^{9}} \approx 141 \text{ FLOP/byte}$$

A layer above 141 FLOP per byte is compute-bound and is on the 12 TFLOP/s roof; a layer below it is bandwidth-bound and is on the memory slope. Figure 9.1 routes a kernel to its bound by comparing its arithmetic intensity against the 141 FLOP-per-byte ridge point.

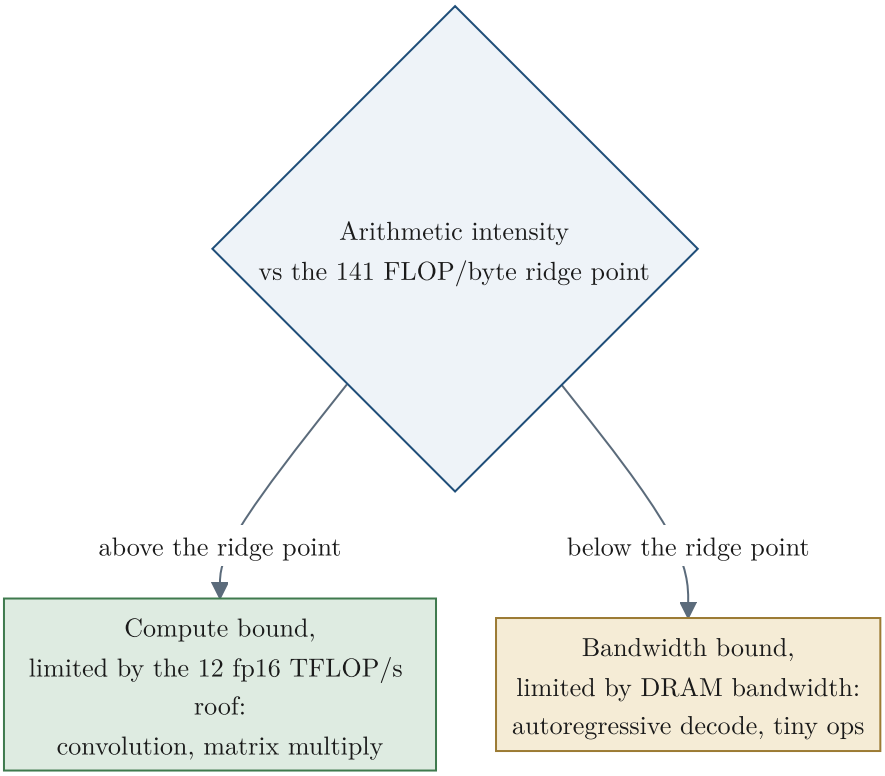

Figure 9.1. How a kernel's arithmetic intensity locates it against the roofline.

Convolutions are far to the right of the ridge point: a 3x3 conv at 256 channels reaches 466 FLOP per byte, so the engine rarely touches its DRAM ceiling on conv work.

9.2 On-chip working-set threshold

The primary design limit is not the 141 FLOP-per-byte crossing but the on-chip working set, 2 MB on the M1. While a layer's activation fits in the 2 MB on-chip memory, its intermediate values stay on chip and do not

cross DRAM. Once the activation exceeds 2 MB, every layer streams its full activation in and out of DRAM and arithmetic intensity collapses. The hardware counters show the threshold directly: at a batch where the activation is exactly 2 MB, a 96-layer matmul chain moved 426 MB of DRAM per dispatch, and arithmetic intensity fell to 60 FLOP per byte. The same matmul that reached 12 TFLOP/s with a 1 MB activation dropped to about 4.8 TFLOP/s. Crossing 2 MB moves a workload from compute-bound to bandwidth-bound on one step, so the working set is the first thing to tune. The working-set limit scales with on-chip memory: it is near 2 MB on the M1 and 4.72 MB on the M5, and chapter 12 gives the per-generation values.

9.3 Dispatch floor

Every dispatch pays a fixed minimum cost regardless of the work it holds. On the M1 that floor is about 0.23 ms per evaluation. A relu, sigmoid, average pool, and small convolution are all at 0.23 to 0.26 ms, and a 64-element linear is 0.23 ms: below the floor, neither the operation nor its size matters. Host dispatch and operand transfer set the floor, not engine compute, so a small op spends almost all of its wall time outside the multiply array. A live decomposition of a tiny model puts the full-call floor near 0.19 ms, of which about 98 percent is dispatch overhead and about 0.13 ms is the firmware round trip; chapter 2 breaks the budget down stage by stage.

The wall time of a dispatch adds the fixed floor t_0 to the time the work takes at the attainable rate R.

$$t \approx t_0 + \frac{\text{work}}{R}, \qquad t_0 \approx 0.23 \text{ ms}$$

When the work is small the second term vanishes and $t \approx t_0$, so small operations are dispatch-bound: a layer whose compute time is under 0.23 ms gains nothing from the engine being faster. The same convolution runs at 63 GFLOP/s at a 16x16 spatial size, where the floor dominates, and at 1247 GFLOP/s at 256x256, where compute amortizes the floor, a 20x span from one shape change. Batching and fusion grow the work term until it dominates the fixed t_0, amortizing the floor across more useful FLOP.

9.4 Fusion economics

Fusing a chain of operations into one program removes the per-dispatch floor from every operation but the first and removes the intermediate round-trips between them. A network run as N separate dispatches pays the 0.23 ms floor N times and copies each intermediate back to the host and forward again. The same network fused into one program pays the floor once and keeps the intermediates resident in the engine's working set. The slope measurement isolated that fixed per-dispatch cost at roughly 0.76 ms for a program with 1 MB of operand input and output, and fusion removes it from every dispatch it eliminates at close to no compute cost.

The amortization is measured directly, and depth and batch are the two controls. A stack of conv-relu layers fused into one program holds its per-call latency flat near 0.19 ms from one layer to thirty-two. A thirty-two-layer model thus pays one firmware round trip just as a one-layer model does, while the cost charged to each operation falls from about 222 microseconds at one layer to about 6.3 microseconds at thirty-two. Batching the same program to 512 samples amortizes that one round trip across the batch, dropping the per-sample cost from about 196 microseconds to about 1.5 microseconds, a 127-fold reduction. A deep model on a large batch thus pays a single round trip for all of it, which is why packing work into one fused program is the main way to cut latency below the compute roof.

9.5 Cross-device roofline

The M1 ceilings above locate a layer against one engine. Taken on three devices, the same two measurements locate a layer against the whole machine, and the difference between the three rooflines is the device map.

Each device's compute roof comes from a saturation sweep and its bandwidth roof from a streaming sweep, and the ratio of the two is the device's ridge point, the arithmetic intensity at which a layer crosses from bandwidth-bound to compute-bound.

The three ridge points are far apart, as Table 9.1 gives the compute roof, bandwidth roof, and ridge point of the engine, GPU, and CPU side by side.

Table 9.1. Per-device compute roof, bandwidth roof, and ridge point, M5/H17s measured; the engine bandwidth is its standalone-activation path.

Device	Compute roof (GFLOP/s)	Bandwidth roof (GB/s)	Ridge (FLOP/byte)
Engine	10191 (matmul) / 18771 (conv)	24.1 (standalone)	424
GPU	30862	229.7	134
CPU	1898	130.4	15

The engine ridge near 424 FLOP per byte is about three times the GPU ridge near 134 and about twenty-nine times the CPU ridge near 15. A layer must reach roughly three times the arithmetic intensity to be compute-bound on the engine that it would need on the GPU. The high engine ridge is a direct consequence of the standalone bandwidth: the engine still delivers about 10 TFLOP/s of compute while a standalone elementwise stream moves only about 24 GB/s, so the crossover is far to the right.

The engine has two distinct effective bandwidths, and the gap between them sets which regime each workload runs in. A standalone elementwise or memory-bound operation streams at roughly 24 GB/s, the activation path measured on a relu or a reduction in isolation. A compiled matmul streams its weights faster than the standalone path, though the rate depends on the measurement. A decode GEMV at arithmetic intensity near 1 reaches about 112 GFLOP/s, which implies a weight-direct-memory-access path near 112 GB/s. A direct wall-clock measurement of one large single-row matrix multiply saturates lower, at the 51 GB/s established by the M1 bandwidth ceiling above. Either way the weight path is far faster than the throttled standalone activation path.

A standalone bandwidth-bound operation, a large softmax or a large elementwise op, reaches only the 24 GB/s activation path on the engine, and the GPU reaches roughly 230 GB/s on the same stream. Such an op is thus dispatched to the GPU, not the engine: against the engine's own bandwidth roof a standalone layer_norm reaches 18 percent and a standalone softmax 63 percent, so dispatching them one at a time leaves the engine mostly idle.

9.6 Fusion as a control

Fusion moves the engine's operating point to the right of its ridge, and the move is measured rather than argued. Fusing operations into one program keeps intermediate activations on chip and streams each weight once for reuse across the whole graph, raising the program's effective arithmetic intensity above that of any single operation in it. A real three-convolution stack fused into one program reaches an effective rate of 20718 GFLOP/s. That rate is above the single-convolution saturation peak of 18771 GFLOP/s, at 110 percent of the convolution roof. Fusion lifts the block's effective arithmetic intensity from about 1076 for one standalone convolution to about 2854 for the fused stack, moving the operating point past the ridge into the compute-bound region. The over-roof GEMV at 466 percent of the standalone roof is the same effect for a single compiled weight stream. A fused convolution or attention graph runs in the compute-bound region where the engine is efficient, and the identical operations dispatched standalone do not.

9.7 Locating a layer before it is built

Two questions locate any layer. First, is its arithmetic intensity above 141 FLOP per byte, where it is compute-bound against the 12 TFLOP/s roof, or below, where it is bandwidth-bound against the 85 GB/s

roof. Second, does its activation fit under 2 MB, since above that it is forced onto the memory slope no matter its nominal intensity, and the working set must shrink before any other tuning helps. The 0.23 ms dispatch floor sets a latency a small op cannot beat, so the developer batches or fuses small ops until their compute clears the floor.

The cost estimate locates a layer against the roofline statically, with no device in hand.

```
# Locate a layer against the roofline BEFORE building it, from its shape alone,
# with no hardware in hand. Three numbers decide where it falls and which
# control moves it.

# M1 roofline constants (see the constants table below):
P            = 12e12    # compute roof, fp16 FLOP per second
B            = 85e9     # bandwidth roof, bytes per second
ridge_point  = 141      # P / B, in FLOP per byte: the compute-vs-bandwidth crossover
working_cap  = 2e6      # on-chip working set, in bytes: above this it streams from DRAM
t0           = 0.00023  # per-dispatch floor, in seconds (~0.23 ms), paid no matter the work

# Count the layer's work and the bytes it must move, from its shape.
function arithmetic_intensity(layer):
    return total_flops(layer) / dram_bytes(layer)        # FLOP per byte

# Classify the layer and name the control it needs.
function place_layer(layer):
    intensity    = arithmetic_intensity(layer)
    working_set  = activation_bytes(layer)               # live intermediate size
    work_seconds = total_flops(layer) / attainable_rate(intensity)
    latency      = t0 + work_seconds                     # t ~= t0 + work / R(I)

    # The working-set threshold dominates: above it, intensity collapses regardless.
    if working_set > working_cap:
        return ("bandwidth", "shrink the working set: reshape or split below 2 MB")

    # Below the dispatch floor the op size does not matter at all.
    if work_seconds < t0:
        return ("dispatch", "batch or fuse until the compute clears the 0.23 ms floor")

    # Otherwise the ridge point decides which roof binds.
    if intensity >= ridge_point:
        return ("compute", "near the 12 TFLOP/s roof: no tuning needed")
    else:
        return ("bandwidth", "compress the weights or fuse to raise the intensity")

# attainable_rate is the lower of the two roofs at this intensity.
function attainable_rate(intensity):
    return min(P, intensity * B)                         # R(I) = min(P, I*B)

# Worked placement of a 3x3 convolution, 256 channels:
(bound, lever) = place_layer(conv_3x3_256ch)
# intensity ~= 466 FLOP/byte  -> well right of the 141 ridge point
# working_set < 2 MB          -> not on the memory slope
# bound == "compute", lever  == "no tuning needed"

# A layer that comes back "bandwidth" or with a working set above 2 MB is
# reshaped, batched, or fused before any other tuning, because no clock-rate
# advantage helps a layer pinned to the memory slope.
```

9.8 Reference: the M1 roofline constants

Table 9.2. The M1 roofline constants, with the symbol each has in this chapter.

Constant	Symbol	M1/H13 value
Compute roof, overhead-isolated matmul slope	P	12 fp16 TFLOP/s
Compute roof, saturating large matmul		4.8 fp16 TFLOP/s
Effective peak the analytic model is fit to		3.25 fp16 TFLOP/s
Convolution end-to-end ceiling		1.8 TFLOP/s
int8 over fp16 compute rate		1.4 to 2 times
Bandwidth roof, DRAM ceiling	B	85 GB/s
Bandwidth roof, saturating weight-stream wall clock		51 GB/s
Standalone activation-stream rate		24 GB/s
Single-relu effective stream		10 GB/s
Roofline ridge point	I^*	141 FLOP/byte
On-chip working-set threshold		2 MB
Per-dispatch floor, slope method	t_0	0.23 ms
Per-dispatch floor, measured tiny model		0.19 ms
Fused per-dispatch cost, 1 MB operands		0.76 ms

10 Power and efficiency

SUMMARY

The engine draws less power than the GPU on every workload class measured, including the ones where the GPU is faster. On the M1 it delivers 2 to 14 times the GPU energy efficiency. The advantage holds across the M1, M2, and M5, so it is a property of the fixed-function fp16 datapath, not of one chip.

10.1 Headline

Across the measured workload classes the engine runs at a few watts where the GPU runs at tens of watts. On the M1 the engine delivers 2 to 14 times the GPU's energy efficiency, measured as throughput per watt of idle-subtracted total-package power. A stack of sixteen 3x3 convolutions sets the high mark: the engine reaches 2063 GFLOP/s per watt against the GPU's 142, a 14.5 times efficiency advantage, while also running about 2 times faster. The M5 holds the same shape at 2289 GFLOP/s per watt on the engine against 175 on the GPU, a 13 times advantage.

The efficiency lead survives even where the engine loses on latency. On a large square matrix multiply with inner dimension 4096 the M1 GPU is about 2 times faster in raw throughput, yet the engine still leads on energy. It computes that workload at 4.4 watts where the GPU draws 32.5 watts, a 4.0 times efficiency edge. The engine's draw stays in the single digits of watts across these classes; the GPU climbs into the tens.

10.2 Watt-complete map

The map covers the workload classes a perception or encoder deployment is built from. Convolution appears as a single 3x3 layer and as a sixteen-deep resnet-style stack. Matrix multiply is measured at three sizes: a dispatch-bound floor, bandwidth-bound mid size, and compute-bound 4096 square. Attention is measured at a short vision-transformer sequence and a longer 512 sequence. The normalization family covers layer norm, rms norm, and group norm. A set of scientific kernels covers a discrete Fourier transform as a matrix multiply, five-point stencil, and fixed-iteration linear solve.

Table 10.1 collects the efficiency standouts named above, with the engine and GPU throughput per watt and their ratio on each chip generation.

Table 10.1. Engine versus GPU energy efficiency across workloads and chip generations, in GFLOP/s/W or absolute watts.

Workload	Engine GFLOP/s/W	GPU GFLOP/s/W	Ratio	Silicon
Conv-resnet stack (16x 3x3)	2063	142	14.5x	M1/H13
Conv-resnet stack (16x 3x3)	2289	175	13x	M5/H17s
Large square GEMM (inner 4096)	4.4 W	32.5 W	4.0x	M1/H13
Five-point stencil (fused)	n/a	n/a	49x	M5/H17s
Workload-class envelope	n/a	n/a	2 to 14x	M1/H13

The large square GEMM row reports absolute package power rather than throughput per watt, because the GPU is faster on that workload on raw throughput while the engine still draws fewer watts to compute it. The

efficiency lead is largest on the convolution stack and on the fused multi-step stencil, where the engine does sustained compute at low power and the GPU spends proportionally more power to keep its units supplied. The stencil reaches a 49 times energy advantage on the M5, the widest single number in the map. On the small normalization reductions the lead narrows, with both devices finishing in well under a millisecond and the per-watt numbers close to a tie. It narrows again on the large 4096 square matrix multiply, which favors the GPU on speed and on fp16 accuracy, though the engine keeps its lower absolute draw there. Only at the trivial dispatch-bound region, the 256-inner matrix multiply, does the engine lose the efficiency comparison outright, where the FLOP count is too small to matter and the lowest call overhead is the better choice.

10.3 Two-generation result

The same sixteen-class harness was run on an M1 and on an M5, with package power taken from hardware instrumentation on both. Both silicons return the same verdict: the engine is the efficiency device on every substantial workload class. The convolution stack reads 2063 GFLOP/s per watt on the M1 and 2289 on the M5, against GPU figures of 142 and 175, so the efficiency advantage holds at 14.5 times and 13 times across the generation gap. A third run on the M2, an A14 part, fills the middle and reproduces the same shape, with the convolution stack at 2234.6 GFLOP/s per watt on the engine against 172.8 on the GPU, a 12.9 times advantage. The engine rail never exceeds about 6 watts where the GPU pulls 13 to 21 watts on the same classes.

The per-class efficiency rows on the M2 trace the same split the M1 and M5 maps draw, and Table 10.2 gives the per-class engine and GPU throughput per watt with the total-package energy ratio.

Table 10.2. Per-class engine versus GPU efficiency on the M2, an A14 part, with the total-package energy ratio.

Workload	Engine GFLOP/s/W	GPU GFLOP/s/W	Energy ratio (GPU over engine)
GEMM floor (M=64, K=256, N=256)	11.4	16.6	0.7x
GEMM bandwidth (M=128, K=1024, N=1024)	298.2	129.9	2.3x
GEMM compute (M=256, K=4096, N=4096)	773.3	213.7	3.6x
Conv single (C=64, 32x32, k=3)	25.1	31.4	0.8x
Conv resnet stack (C=256, 32x32, k=3, d=16)	2234.6	172.8	12.9x
Attention ViT (S=197)	480.8	120.2	4.0x
Attention long sequence (S=512)	567.4	150.1	3.8x
Five-point stencil (256x256, 32 steps)	37.6	0.8	47.8x

The real-model rows on the M2 are the engine's widest margin: a full twelve-layer ViT-B/16 forward runs at 67.9 millijoules per inference against the GPU's 714.3, a 10.5 times energy advantage, and a ResNet-18 forward at 2.6 against 22.3 millijoules per inference, an 8.5 times advantage. Both also run from 1.7 to 4 times faster. On the M1 the engine draws under 2.3 watts at its compute roof and spends roughly 0.5 picojoules per FLOP in its fixed-function datapath.

The advantage is wider on the older M1 because that chip is more bandwidth-limited. The M1 streams at roughly 10 GB/s effective against the M5's roughly 57 GB/s, and the smaller memory bandwidth makes the GPU spend proportionally more power to lead on raw throughput there. On the M1 the engine sustains a

256-channel 3x3 convolution at 643 GFLOP/s per watt, drawing 1.78 watts, with the rail reading about zero at idle. The 1.78 watt figure is the canonical short-run draw for this workload, and a longer 176 second run of the same convolution in chapter 31 settles slightly lower at about 1.66 watts. The two figures are thus the same workload measured over different durations rather than a disagreement.

10.4 Rail-off idle floor

The engine reads about zero milliwatts at idle, and that figure is a property of the power-state machine, not a measurement artifact. The firmware holds the engine in a fully gated state, `ANE_POWER_STATE_ALL_OFF`, with every domain off, until a job arrives. Power-up is lazy and job-triggered: the engine is at `ALL_OFF` and brings up its base domain only on the first call, transitioning through `ANE_POWER_STATE_BASE_PS_ON_WAIT` before it gates in the compute sets. There is no clock-gated but powered idle floor; the idle state is rail-off.

The M1 engine has five independently gated power domains: a base domain for the control and front-end fabric, brought up first, and four compute sets, one per ANE cluster. Dynamic power-gating, on by default, collapses the compute sets back toward `ALL_OFF` between jobs, so idle work pays no rail power. Disabling it keeps the sets powered between jobs, trading idle power for lower per-job latency. Thermal management and voltage are off-engine: the firmware image has no thermal, temperature, or throttle path, so those decisions are in the system-on-chip power manager, not in the engine. The rail-off idle is why the efficiency map subtracts a flat zero for the engine baseline while the GPU and CPU baselines subtract a nonzero idle draw.

10.5 Power scales with utilization

The idle rail reads zero and the dispatch floor draws about 0.9 watts, and from there the draw climbs with how much of the multiply array a workload keeps active. A twenty-point sweep on an M1 Max, read at the root power sampler, traces the curve: a dispatch-floor matrix multiply draws about 0.9 watts, a compute-bound 1024 by 4096 fp16 matrix multiply about 4.3 watts, and the int8 form about 5.8 watts. That is a roughly fivefold range set by utilization alone. The draw is regime-dependent as well: a bandwidth-bound matrix multiply that mostly streams a large weight draws about 1.4 watts, a convolution about 1.3 watts because its lowering leaves part of the array idle, and an elementwise operation almost nothing on the engine rail. Efficiency peaks at the fp16 compute optimum near 2.68 trillion operations per watt, about 0.37 picojoules per FLOP, and falls to about 22 picojoules per FLOP at the dispatch floor, which is a further reason to amortize work past the floor. The engine exposes only the power reading and a binary on-or-off state even to the root sampler, with no frequency or voltage telemetry, so the draw is observable but the sequence of operating points behind it is not.

10.6 Sustained load holds one clock state

The efficiency figures above are warmup measurements, so they leave open whether the engine holds its power state under minutes of continuous load. A compute-bound probe answers it directly: a chain of eight 1024-square fp16 matrix multiplies run as one program in a low-overhead zero-copy loop, at about 2.11 milliseconds per call, roughly ten times the dispatch floor, so the loop is compute-bound. Run continuously for 210 seconds and 98,092 calls on the M1, the throughput curve is flat. The whole-run median is 2109.6 microseconds, the start bucket reads 2153 microseconds and the end bucket 205 seconds later reads 2143. The per-five-second bucket medians vary by about 4 percent in a non-monotonic band that is sampling noise, not thermal decay. The engine rail, sampled read-only at five-second cadence, pins at about 5.5 watts for the whole run, with start, middle, and end thirds reading 5530, 5522, and 5391 milliwatts, a drift under 3 percent. Thermal pressure reads Nominal on every sample, and the engine never leaves its single clock state, with the steady draw set by the workload's utilization as the previous section describes. The M1 engine does not throttle on this workload over 3.5 minutes: it runs at one clock state and stays there.

The behaviors that move first-call latency are at the boundaries of a burst, not in steady state, and Table 10.3 gives the warmup, steady-state, and idle re-wake costs the flat steady state hides.

Table 10.3. Sustained-load phases on the M1, with the warmup and idle re-wake costs that the flat steady state hides.

Phase	Behavior on the M1
Warmup	About a three-call ramp: call 1 about 7.6 ms (3.6x steady), call 2 about 4.6 ms, call 3 within 13 percent of steady, then flat at about 2.15 ms.
Steady state	Flat at about 8.1 TFLOP/s and about 5.5 W for 3.5 minutes, p50 2110 microseconds, p99 only 36 percent over p50.
Sub-second idle	A modest first-call penalty of 1.2x to 1.7x, then the next call is back to steady.
Multi-second idle	The first call after a 5-second gap costs about 260 ms, roughly 123x the steady p50, then the very next call returns to about 2.1 ms.

The idle re-wake penalty is the consequence of the rail-off idle described above. Once idle crosses a few seconds the compute sets gate fully off, and the first call after the gap pays a one-time cold re-wake of tens to hundreds of milliseconds before the engine returns to its steady state on the following call. The cost falls on the first call of each active burst, not as a sustained slowdown. Latency-sensitive code that must answer immediately after an idle gap should thus keep the engine warm with a sub-second dispatch cadence or a low-cost keep-alive call.

The engine rail sampled directly on the M1 confirms that average power tracks utilization. Idle draws about 0 milliwatts. A sustained batch-of-512 hot loop draws about 1.48 watts. A single-call loop draws about 755 milliwatts, about half the hot-loop figure, because a single small call leaves the engine idle for most of the dispatch window.

10.7 Compression as an energy control

Streaming a weight in a narrower format cuts energy per inference even though instantaneous power stays roughly flat. On a bandwidth-bound 4096-inner matrix multiply on the M2, the engine draws between 2.06 and 2.59 watts across fp16, int8, int4, and sparse, with the sparse stream drawing the most watts. The energy per inference still falls with the narrower format because latency falls faster than power rises, as Table 10.4 gives the power, latency, and energy per inference across fp16, int8, int4, and sparse.

Table 10.4. Power, latency, and energy per inference across weight formats on a bandwidth-bound matrix multiply, M2.

Weight format	Engine power	Latency	Energy per inference	Versus fp16 energy
fp16	2.18 W	0.690 ms	1.50 mJ	1.00x
int8	2.06 W	0.431 ms	0.89 mJ	0.59x
int4	2.29 W	0.270 ms	0.62 mJ	0.41x
Sparse (about 32 percent dense)	2.59 W	0.330 ms	0.86 mJ	0.57x

An int4 weight stream reaches the same answer at 0.41 times the fp16 energy on this part, a 2.4 times energy reduction at equal work, with no efficiency-versus-latency tradeoff to weigh: the narrower format is faster and more efficient.

10.8 Developer takeaway

The fp16 datapath is why the lead holds even where the GPU is faster on latency: a narrow multiply and a fixed-function pipeline move and compute far fewer bits per result than a general-purpose vector unit. The engine thus spends less energy reaching the same answer wherever the math stays in its precision range.

10.9 Estimating a layer's energy before it is built

A workload's efficiency follows from where the roofline locates it. A layer that is compute-bound on the engine spends its energy in the fixed-function multiply array, which is the regime where the engine outruns the GPU on throughput per watt by the wide margins in the map above. A layer pinned to the memory slope, or below the dispatch floor, spends its energy moving bytes or paying call overhead, where the efficiency lead narrows toward a tie. The cost estimate locates the layer statically, so the energy regime can be read before any device is in hand.

The procedure estimates the layer, reads its bound, and locates it against the efficiency map: a compute-bound layer is where the engine is most efficient, a dispatch-bound one is not.

```
# Estimate the energy of one layer before any hardware is in hand.
# A 3x3 stride-1 convolution, 256 channels, on a 56 by 56 feature map.

given layer L = conv_3x3(input = [1, 256, 56, 56], stride = 1)
given target chip = H13            # M1; peak and bandwidth come from its roofline

# 1. Count the work the layer does and the bytes it must move.
flops = total multiply_adds in L           # arithmetic operations
bytes = weight_bytes(L) + input_bytes(L) + output_bytes(L)

# 2. Locate the layer on the roofline of the chosen chip.
peak      = compute_peak(chip)             # operations per second the array can sustain
bandwidth = memory_bandwidth(chip)         # bytes per second from on-chip and DRAM
floor     = dispatch_floor(chip)           # fixed per-call overhead, about 0.23 ms on H13

compute_time = flops / peak                # time if the array is the limit
memory_time  = bytes / bandwidth           # time if moving bytes is the limit
latency      = max(compute_time, memory_time) + floor

if compute_time >= memory_time:  regime = "compute"     # array busy, most efficient regime
else:                            regime = "bandwidth"    # moving bytes, efficiency lead narrows
if latency is dominated by floor:  regime = "dispatch"  # call overhead, leads converge

# 3. Energy = power drawn in that regime, times how long the layer runs.
power  = sustained_power(chip, regime)     # e.g. about 1.78 W compute-bound on H13
energy = power * latency

# 4. Amortized versus per-call: a layer called once pays the full floor,
#    a layer called many times spreads that fixed floor across the calls.
energy_per_call_standalone = power * (max(compute_time, memory_time) + floor)
energy_per_call_amortized  = power * (max(compute_time, memory_time) + floor / number_of_calls)

return regime, latency, energy_per_call_standalone, energy_per_call_amortized
```

A layer that prints `compute` runs in the region where the convolution-stack figures hold, up to 2063 GFLOP/s per watt on the M1; a layer that prints `dispatch` or `bandwidth` falls toward the tie at the small-reduction and trivial-matmul regions.

10.10 Reference: engine versus GPU efficiency constants

Table 10.5 collects the power and efficiency constants of this chapter with the silicon each was measured on.

Table 10.5. The power and efficiency constants of this chapter, with the silicon each was measured on.

Quantity	Value	Silicon
Convolution-stack efficiency, engine	2063 GFLOP/s/W	M1/H13
Convolution-stack efficiency, GPU	142 GFLOP/s/W	M1/H13
Convolution-stack efficiency ratio	14.5x	M1/H13
Convolution-stack efficiency, engine	2289 GFLOP/s/W	M5/H17s
Convolution-stack efficiency, GPU	175 GFLOP/s/W	M5/H17s
Convolution-stack efficiency ratio	13x	M5/H17s
Convolution-stack efficiency, engine	2234.6 GFLOP/s/W	M2/H14
Convolution-stack efficiency, GPU	172.8 GFLOP/s/W	M2/H14
Convolution-stack efficiency ratio	12.9x	M2/H14
Sustained 3x3 convolution efficiency	643 GFLOP/s/W	M1/H13
Engine power, sustained convolution	1.78 W	M1/H13
Engine power, sustained convolution	~3.8 W	M2/H14
Engine power, compute-bound GEMM	~5.9 W	M2/H14
Large 4096 GEMM, engine power	4.4 W	M1/H13
Large 4096 GEMM, GPU power	32.5 W	M1/H13
Large 4096 GEMM efficiency ratio	4.0x	M1/H13
Five-point stencil efficiency ratio (fused)	49x	M5/H17s
Five-point stencil efficiency ratio (fused)	47.8x	M2/H14
ViT-B/16 forward energy, engine versus GPU	67.9 against 714.3 mJ/inf, 10.5x	M2/H14
ResNet-18 forward energy, engine versus GPU	2.6 against 22.3 mJ/inf, 8.5x	M2/H14
int4 weight stream energy versus fp16	0.41x at equal work	M2/H14
Workload-class efficiency envelope	2 to 14x	M1/H13
Engine rail, upper bound on the measured classes	~6 W	M2/H14
Idle engine power	~0 W (rail-off)	M1/H13
Engine rail, sustained batch-of-512 hot loop	~1.48 W	M1/H13
Engine rail, single-call loop	~755 mW	M1/H13
Effective stream rate, M1	~10 GB/s	M1/H13
Effective stream rate, M5	~57 GB/s	M5/H17s

11 ANE, GPU, and CPU

SUMMARY

The engine is faster and more efficient on convolution, vision, short-sequence attention, and mid-size matrix multiply, running a sixteen-deep convolution stack about 4.2 times faster than the GPU at about 13 times the energy efficiency on the M5. The GPU is about 2.8 times faster on large square matrix multiply, long-sequence attention, and bandwidth-bound decode. The CPU is fastest on the trivial operations that fall below the 0.23 ms dispatch floor. Serving moves the line: the GPU overtakes the engine on a batched encoder block near a batch of 23 and on self-attention near a batch of 6.

A workload on an Apple system on chip has no single fastest processor. It has three candidates, and the right one depends on the form of the work. This chapter maps which of the engine, GPU, and CPU leads on which class of work, on raw speed and on energy per result, from a single harness across sixteen workload classes on the same silicon. The map is two-dimensional: a processor can lead on latency, on energy, on both, or on neither.

11.1 Form of the map

The engine is the efficiency processor for convolution and vision, and a low-latency processor for encoders at low to moderate batch. The GPU is the throughput processor for large square matrix multiply and for the bandwidth-bound work of autoregressive decode. The CPU collects the work too small for either, where dispatching to an accelerator costs more than the arithmetic.

11.2 Where the engine leads

The engine is faster and more efficient on convolution, convolution stacks, stencil-like fixed-iteration numerics, short-sequence attention, and mid-size matrix multiply. The convolution stack leads the set: a sixteen-deep stack of 3x3 convolutions at 256 channels runs about 4.2 times faster than the GPU and at about 13 times its energy efficiency on the M5 generation. On the M1 generation it runs about 2 times faster and at 14.5 times the GPU's efficiency, the widest power gap in the set. A five-point stencil iterated thirty-two times as one fused graph runs about 10 times faster and at about 49 times the GPU's efficiency on the M5, because the fixed-iteration graph amortizes the per-dispatch floor across every step. Short-sequence attention, a transformer block at sequence length 197, is fastest, most efficient, and most accurate in fp16 on the engine.

The real models track the primitives: a ResNet-18 forward runs about 6.1 times faster than the GPU reference and at about 11 times the energy efficiency per inference, and a twelve-layer encoder forward about 1.5 times faster and at about 18 times the energy efficiency per inference. A single-sentence encoder runs about 4.4 times faster. The energy advantage is wider than the speed advantage and persists where the speed advantage does not. On the M1 the engine draws 4.4 W on a large compute-bound matrix multiply where the GPU draws 32.5 W, a 4.0 times efficiency edge even where the GPU is about 2 times faster.

11.3 Where the GPU leads

The GPU leads on raw speed on large square matrix multiply, long-sequence attention, and bandwidth-bound autoregressive decode. A 256 by 4096 by 4096 matrix multiply runs about 2.8 times faster on the GPU than

on the engine, and the gap grows to 3.0 times at saturation, about 30.9 fp16 TFLOP/s against the engine's 10.2 TFLOP/s peak. The engine's single fused matrix multiply stalls on weight streaming once a square operand passes its on-chip working set near N of 2048. Long-sequence attention shifts to the GPU as the sequence grows: at sequence length 512 the GPU is about 2.1 times faster, though the engine keeps a 1.8 times energy edge.

On very large reductions the GPU also leads on fp16 accuracy. On the large square matrix multiply the GPU holds an fp16 relative error of 3.6×10^{-4} flat across size, while the engine's error grows with the contraction dimension, from 1.2×10^{-2} at N of 2048 to 6.2×10^{-2} at N of 8192. That workload is thus disfavored on the engine on both axes.

11.4 Where the CPU leads

The CPU is the better choice for the trivial cases, where call overhead decides the result. On the floor matrix multiply, 64 by 256 by 256, the CPU completes in about 0.026 ms, below the per-dispatch floor of either accelerator. The engine pays a fixed per-eval overhead of about 0.23 ms on the M1, so any single small operation is overhead-bound and cheaper on the CPU. On any larger class the CPU is not competitive, reaching about 1.9 fp32 TFLOP/s on a saturated matrix multiply.

11.5 M1 per-class measurement

The verdicts above are taken from one harness across sixteen workload classes, and the M1 rows make the per-class split concrete. The engine leads on energy on every class but the dispatch-bound floor, and it is fastest on the convolution-heavy and mid-bandwidth classes, as Table 11.1 gives the per-class engine and GPU speed and efficiency on idle-subtracted package power.

Table 11.1. The M1 per-class speed and efficiency comparison, engine versus GPU, idle-subtracted package power.

Workload	Engine GFLOP/s	Engine GFLOP/s/W	GPU GFLOP/s	GPU GFLOP/s/W	Engine efficiency advantage
GEMM floor (K=256)	51	17	46	27	0.6x
GEMM bandwidth (K=1024)	662	470	1109	207	2.3x
GEMM compute (K=4096)	3588	775	7286	192	4.0x
Conv single (C=64)	99	23	89	18	1.3x
Conv resnet (d=16)	9602	2063	4671	142	14.5x
Attention ViT (S=197)	1711	370	1831	187	2.0x
Attention long sequence (S=512)	1775	544	3768	217	2.5x

fp16 accuracy tracks the same boundary the M5 shows: the engine matches or beats GPU fp16 where the arithmetic is well-conditioned and degrades on the large-K reduction, reaching 2.7×10^{-2} relative error at K of 4096.

11.6 Dispatch floor and conv throughput

A single small operation is overhead-bound on the engine: it costs about 0.23 milliseconds regardless of the operation or its size. A relu, sigmoid, average pool, and small convolution are all between 0.24 and 0.26 milliseconds, with a 64-element linear at 0.23 milliseconds. That floor is host dispatch and operand transfer, not engine compute, so it sets a latency a small operation cannot beat.

A convolution amortizes the floor as its spatial size grows, and Table 11.2 traces the throughput climbing roughly twenty-fold from the floor to the saturated shape.

Table 11.2. A 3x3 convolution at 64 channels on the M1, throughput climbing as spatial size amortizes the dispatch floor.

Spatial size	Latency	GFLOP/s
16x16	0.231 ms	63
32x32	0.248 ms	267
64x64	0.395 ms	718
128x128	1.062 ms	1102
256x256	3.814 ms	1247

Raising channels rather than spatial size reaches the higher M1 end-to-end conv peak, about 2212 GFLOP/s at 256 channels. A matrix multiply with a single output row stays overhead-bound regardless of inner size, reaching only about 5.9 GFLOP/s at an inner dimension of 1024, which is why a decode-shaped projection does not supply the array.

11.7 Batch threshold for serving

The single-stream map is the device choice for one request, but serving batches requests, and the choice moves with batch size. On a true-batched encoder block the GPU overtakes the engine on throughput near a batch of 23, and on a self-attention block near a batch of 6, as the GPU scales with batch while the engine saturates near a batch of 1. The energy crossover is at a larger batch than the throughput crossover, and on three of four serving workloads it never appears. Vision convolution serving never crosses on either axis, leading throughput by 3.6 to 5.7 times and energy by 6 to 10 times at every batch from 1 to 256. Only bare large-batch matrix multiply converges, to an energy tie by a batch of 16.

Table 11.3 collects the per-workload serving crossovers.

Table 11.3. The batch at which the GPU overtakes the engine for each serving workload, on throughput and on energy.

Serving workload	Throughput crossover to GPU	Energy crossover to GPU
Encoder block, true-batched	near batch 23	none from batch 1 to 256
Self-attention block	near batch 6	none from batch 1 to 256
Vision convolution	none; engine leads 3.6 to 5.7x	none; engine leads 6 to 10x
Large-batch matrix multiply	GPU regime throughout	energy tie near batch 16

11.8 Verdict table

Figure 11.1 is the decision tree that maps a workload to the engine, GPU, or CPU.

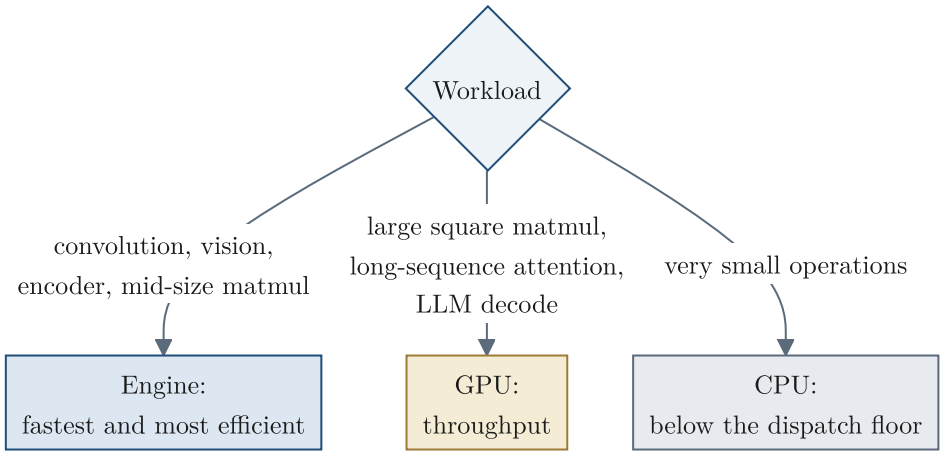

Figure 11.1. Which processor suits which workload.

The per-class verdicts collapse into one table. Table 11.4 names for each workload class the processor that is fastest, the processor that is most efficient on energy per result, and the headline figure from the sections above.

Table 11.4. The fastest and most efficient device among engine, GPU, and CPU for each workload class, with a standout figure.

Workload	Fastest	Most efficient	Standout number
Convolution and vision	Engine	Engine	Convolution stack about 4.2 times faster and about 13 times the GPU efficiency on M5; about 2 times faster and 14.5 times on M1
Encoder and embedding serving, low to moderate batch	Engine	Engine	Single-sentence encoder about 4.4 times faster; throughput crossover to GPU near a batch of 23
Mid-size matrix multiply	Engine	Engine	Engine is faster and more efficient below the square-operand working set near N of 2048
Short-sequence attention	Engine	Engine	Transformer block at sequence length 197 is fastest, most efficient, and most accurate in fp16 on the engine
Large square matrix multiply, large K	GPU	GPU	About 2.8 times faster, growing to 3.0 times at saturation: 30.9 against 10.2 fp16 TFLOP/s
Long-sequence attention	GPU	Engine	At sequence length 512 the GPU is about 2.1 times faster; the engine keeps a 1.8 times energy edge
Autoregressive decode	GPU	GPU	Bandwidth-bound, the GPU's throughput regime

Workload	Fastest	Most efficient	Standout number
Tiny operations	CPU	CPU	Floor matrix multiply 64 by 256 by 256 in about 0.026 ms, below the per-eval floor of about 0.23 ms on M1

11.9 Picking the processor for a workload

The device choice is the workload's regime read against the thresholds above. The procedure estimates the layer, then routes it: compute-bound and resident to the engine, large-square or long-sequence to the GPU, below the floor to the CPU.

```
# Pick the processor for a workload from its regime, before choosing a device.
# Example workload: a 256 by 4096 times 4096 by 4096 matrix multiply.

given workload W = matmul([256, 4096], [4096, 4096])
given target chip = H13            # M1; thresholds below are read from its roofline

# 1. Estimate the workload statically on the chip's roofline.
flops        = total multiply_adds in W
bytes        = weight_bytes(W) + input_bytes(W) + output_bytes(W)
working_set  = bytes that must stay resident at once   # in MB
compute_time = flops / compute_peak(chip)
memory_time  = bytes / memory_bandwidth(chip)
floor        = dispatch_floor(chip)        # about 0.23 ms on H13

if max(compute_time, memory_time) < floor:  bound = "dispatch"
else if compute_time >= memory_time:        bound = "compute"
else:                                        bound = "bandwidth"

# 2. Classify the workload by shape and bound.
is_engine_shape = (bound == "compute") and (working_set <= 2.0)
                  # convolution, encoder block, or mid-size GEMM that stays on chip
is_gpu_shape    = (working_set > 2.0)                  # large square GEMM, streams from DRAM
                  or (sequence_length(W) is long)      # long-sequence attention
is_cpu_shape    = (bound == "dispatch")                # below the floor, too small to dispatch

# 3. Apply the regime rules to pick the processor.
if is_cpu_shape:        choose CPU       # call overhead dominates, keep it local
else if is_gpu_shape:   choose GPU       # GPU saturates wide work and holds fp16 better here
else if is_engine_shape: choose ENGINE   # where the engine is fastest and most efficient
else:                    choose ENGINE   # default for compute-bound resident work

# The 256 by 4096 by 4096 multiply has a working set past the 2 MB on-chip limit,
# so the rule routes it to the GPU, which the verdict table records as about 2.8x faster.
return chosen_processor, bound, working_set
```

12 Across the chip family

SUMMARY

An M(n) chip has the H(n+12) ANE architecture, so M1 is H13 and M5 is H17. A network that compiles and runs on one generation compiles and runs on the others, because one compiler binary builds every target and only a per-target data table changes. The single property that varies from chip to chip is fp16 numerics, and it varies only at a unit in the last place. Target the generation a network needs by its operation set, then verify per chip only the cancellation-sensitive reductions and the width-axis slices.

12.1 Naming rule

The M-series engine and the contemporaneous A-series engine are the same architecture under two product names, offset by a fixed amount. An M(n) chip has the H(n+12) ANE architecture, the compact relation $M(n) \to H(n+12)$. M1 is H13, M2 is H14, M3 is H15, M4 is H16, and M5 is H17. The A-series anchor is one generation over: the A13 and the M1 share H13, and the A17 and the M5 share H17.

Table 12.1 is the family map of the core count and clock that scale across the generations, with the un-measured upper generations decompile-derived from the device tables.

Table 12.1. The M-series chips with their A-series anchor, engine architecture string, core count, and clock.

Chip	A-series anchor	ANE architecture	NE cores	Clock
M1	A13	H13	4 (base), 8 (Pro and Max)	~1.14 GHz
M2	A14	H14g	4 (base), 8 (Pro and Max), 32 (Max-class)	measured
M3	A15	H15	4 (base), predicted	predicted
M4	A16	H16	4 (base), predicted	predicted
M5	A17	H17s	16	~1.89 GHz

Three rows are measured on physical silicon: M1 (H13), M2 (a Pro reporting H14g), and M5 (H17s); the M3 and M4 rows are decompile-derived from the per-family device tables and not individually measured, with the A15/M3 generation the one rail that remains unmeasured. The M2 measurement closes the middle of the sequence: a seeded classifier trains to the M1 number to the digit, the four fp16 axes match the M1 bit for bit, and a watt-complete device map reproduces the M1 and M5 shape. The A14 is thus the A13's numerical twin on every axis measured. The fuller silicon-to-target table, the board-type sequence, and the full set of 28 compiler targets are the subject of a later chapter [AppleANE].

The rule reads directly off the device tables and is confirmed on the measured parts. The live M1 reports the architecture string h13g, the Pro and Max variant of H13, and the M5 compiles to the H17 target and resolves the H17s variant on disk. The system frameworks corroborate the sequence: the on-device video upscaler includes exactly the five targets H13 through H17, which are the five Mac generations M1 through M5. The target a network needs can thus be named by generation, and the mapping holds without probing the silicon.

12.2 What holds across the family

A network that compiles and runs on one generation compiles and runs on the others. The compiler is a single binary that constructs any target on demand, so the program format, operation legality, and datapath are shared, and only a per-target data table changes underneath them.

The operation limits are properties of the family, not of one chip. The operations with no hardware path on any current part, such as the product reduction, scatter family, and recurrent cells, are absent on every generation from the M1 through the M5. Gated operations arrive at a known generation and stay: the texture-engine sampler operations turn on at the A14, and native sin and cos turn on at the A15. A network that avoids them runs everywhere, and a network that uses them runs from its unlock generation forward. No operation the M1 can compute is missing on the M5, because each newer engine is an operation superset of the one before it.

Each generation adds one capability over the one before it, then stops, and Table 12.2 names the single capability each engine generation adds, read off the per-target capability bytes.

Table 12.2. The single capability each engine generation adds, read off the per-target capability bytes.

Generation	Capability added
A13 (M1)	three-dimensional convolution, the sixteen-deep kernel, native softmax, layer norm, all reductions, and fused attention
A14 (M2)	the texture-engine samplers (resize, crop-resize, resample, affine, hardware gather) and cross-die addressing
A15 (M3)	native sin and cos, the dropout and random path, and global argument-min and argument-max
A16 (M4)	the tensor dimension limit rises from 16384 to 65536, and the fp16 kernel-width ceiling rises from 13 to 15
A17 (M5)	no operation over A16; the NE-core count scales

The A17 and A18 add no operation over the A16: identical dimension limits, the same texture engine, the same legal operation set, differing only in NE-core count, which scales throughput rather than legality. The dimension limit is not a single number per chip: on the M2 the spatial and contraction extents cap at 16384 while the channel axis caps at 65536, exactly four times the spatial cap. The limit thus belongs to the axis an operation uses rather than to the tensor.

The newer parts scale the core count and the clock; they do not change the programming model. The core count runs 4 on the M1, 8 on its Pro and Max variant, and 16 on the M5, and the operating clock rises from roughly 1.14 GHz to roughly 1.89 GHz across that span. The fp16 datapath, the wide accumulator, and the form of the roofline hold across the family unchanged. The measured M5 confirms the scaling: about 19.6 fp16 TFLOP/s on the matmul slope and about 14.3 fp16 TFLOP/s on the convolution peak, both from the fused-chain probe on the same network with no source change, against the roofline saturation peak of 18.8 TFLOP/s in Chapter 9. A single large matmul above the dispatch floor runs at about 9.5 fp16 TFLOP/s, the M5 analogue of the M1's 4.8, and the engine streams weights at about 145 GB/s over two DRAM read channels, near three times the M1's 51 GB/s. Those peaks are set by the larger core count and the higher clock, not by any change to the datapath, which is the metric on which the generations compare. The working-set threshold moves with the silicon, from near 2 MB on the M1 to a measured 4.72 MB on the M5, scaling with the larger 16-core on-chip memory.

12.3 One thing that varies

The single property that differs from chip to chip is fp16 numerics. The accumulator width is uniform across every engine and the compiler text is identical, so a cross-chip value difference can only come from a data-selected codegen route that changes the order in which fp16 operations combine. That surface is limited:

most route changes are a numerical no-op, because the wide accumulator absorbs the reordering, and the rest are at a unit in the last place, set by tiling-boundary alignment.

The cross-generation measurement fixes the scale. The same seeded convolutional classifier trained on the M1, M2, and M5 reaches 0.9080, 0.9080, and 0.9070 test accuracy, each deterministic across repeated runs, a difference of one test sample in a thousand between the ends. The M2 is exactly on the M1 number, which puts the entire fp16 training drift at the A16 generation rather than spread across the family: the M1 and the M2 are numerical twins, and the gap opens only at the M5. That gap is the drift of sub-unit-in-the-last-place fp16 differences compounding over a few hundred training steps, real and negligible. The cross-silicon predictions extracted from the device tables were confirmed on the M5: all ten, covering throughput, the working-set threshold, operation limits, texture engine, and fp16 slice behavior, held on the real part. The one finite-to-infinity axis, a slice saturation that occurs on the M1, takes the non-saturating route on the M5, as predicted.

12.4 fp16 divergence axes

That data-selected codegen surface reduces to four axes. Three of them are at most a unit in the last place, and one is a finite-to-infinity saturation that occurs on the older parts. Table 12.3 names the four axes, the codegen route each selects, and the bounded magnitude of each.

Table 12.3. The four fp16 cross-chip divergence axes, the codegen route each selects, and the bounded magnitude of each.

Axis	Mechanism	Effect	Magnitude
Slice saturation	A width-axis slice with a nonzero offset routes through a fixed-point crop that multiplies by sixteen	A source value above 4094 saturates to plus or minus infinity, since 4094 times sixteen is the 65504 fp16 ceiling	finite to infinity
Reduction then square fusion	A reduction immediately followed by a square or multiply can fuse, removing one intermediate rounding step	Drops one fp16 rounding step	at most one unit in the last place
Reduction route	A reduction selects a transpose route or a reshape route by an extent threshold, 192 on the older parts and 384 from the M3	Reorders partial sums	numerical no-op, the wide accumulator absorbs it
Tiling granularity	The partial-sum tile alignment is set by the patch and core-count fields, granularity near 128	A sum off a tile boundary loses one rounding increment	at most one unit in the last place

The saturation axis is the only one that changes a finite value into an infinity, and it is magnitude-gated: it triggers only when a width-offset slice holds a value above 4094. Measured on the M1 the threshold is exact: a width-offset slice is finite at 4094 and goes to plus or minus infinity at 4100, while the zero-offset control stays finite even at 60000. The M2 saturates bit-identically to the M1, which corrects the earlier reading that the non-saturating route arrives at the A14: the saturation persists through the A14 and the M5 is the part that takes the non-saturating route. The reduction-then-square fusion never manifests on silicon: the M1, M2, and M5 all measure the unfused result, so that axis is uniform across the family. A divergence is thus predictable from a small set of fields without running every chip: a reduction or normalization denominator can differ by at most one unit in the last place where the tiling or route fields differ. A width-offset slice has saturation risk only when its values can exceed 4094.

12.5 Developer policy

Choose the generation a network requires from its operation set, then let the program run across every part at and above that generation. Verify per chip only the numerics that can move: the cancellation-sensitive reductions, the variance and normalization denominators, and any width-axis slice whose values can exceed the saturation bound. Everything else is portable by construction: the same source, the same operation legality, faster on the newer silicon by the core and clock scaling.

12.6 Compiling for a target generation

The naming rule lets a network name its target by generation rather than by probing the silicon. The compiler constructs any target from its per-chip table, so a developer compiles a program for the oldest generation it must support, then runs it unchanged on every part at and above that generation. A static estimate against a target reads back that target's core-and-clock scaling without the part in hand.

The procedure compiles for the floor generation a network requires, then estimates against the newer targets to read the core-and-clock speedup.

```
/* The target generation is held in the compiler options as a TargetArchitecture string, */
/* so the same source compiles for whatever floor the network must support.            */
e5rt_e5_compiler_options_create(&options);
e5rt_e5_compiler_options_set_custom_ane_compiler_options(options, "TargetArchitecture=h13");
e5rt_e5_compiler_compile(compiler, model_path, options, &library);  /* M1 floor, runs M1..M5 */

/* Then the same drive: retain the function, build the op, and dispatch on a stream. */
e5rt_program_library_retain_program_function(library, fn_name, &function);
e5rt_precompiled_compute_op_create_options_create_with_program_function(function, &op_opts);
e5rt_execution_stream_operation_create_precompiled_compute_operation_with_options(op_opts, &op);
e5rt_execution_stream_encode_operation(stream, op);
e5rt_execution_stream_execute_sync(stream);
```

A network compiled to `h13` runs on every generation from the M1 through the M5; estimating the same graph against `h17s` reads back the M5 scaling without the M5 in hand.

12.7 Reference: per-family scaling constants

Table 12.4 collects the per-family scaling constants and the figures that fix cross-generation behavior, with the silicon each was measured on.

Table 12.4. The per-family scaling constants and the figures that fix cross-generation behavior, with the silicon each was measured on.

Quantity	Value	Silicon
Naming rule	$M(n) \to H(n+12)$	family-wide
NE cores	4 (base), 8 (Pro and Max)	M1/H13
NE cores	16	M5/H17s
Operating clock	~1.14 GHz	M1/H13
Operating clock	~1.89 GHz	M5/H17s
Matmul-slope peak	about 19.6 fp16 TFLOP/s	M5/H17s
Single-program matmul peak	about 9.5 fp16 TFLOP/s	M5/H17s
Convolution peak	about 14.3 fp16 TFLOP/s	M5/H17s
M5 convolution peak versus M1 projected fp16 peak	near 5x	M5/H17s
Working-set threshold	near 2 MB	M1/H13
Working-set threshold	4.72 MB	M5/H17s

Quantity	Value	Silicon
DRAM weight-stream bandwidth	about 145 GB/s, two read channels	M5/H17s
Texture-engine sampler unlock	A14 generation	family-wide
Native sin and cos unlock	A15 generation	family-wide
Tensor dimension limit raised, 16384 to 65536	A16 generation	family-wide
Kernel-width ceiling raised, 13 to 15	A16 generation	family-wide
M2 spatial and contraction extent cap	16384	M2/H14
M2 channel-axis extent cap	65536	M2/H14
Cross-generation training parity	0.9080, 0.9080, 0.9070	M1/H13, M2/H14, M5/H17s
fp16 cross-chip divergence bound	one unit in the last place	family-wide
fp16 slice-saturation threshold	source above 4094 to infinity	M1/H13, M2/H14
Number of fp16 divergence axes	four	family-wide
Cross-silicon prediction pass	ten of ten	M5/H17s

Part IV turns from the engine in isolation to the workloads that run on it.

Part IV

Workloads

13 Vision, convolution, and encoders

The engine's strongest classes, fitted as one fused program.

14 LLM case study

Why decode belongs on the GPU, and the hybrid placement that remains.

15 Training on the engine

Gradients as graph inputs, resident optimizer state, and cross-generation parity.

16 Numerical and scientific computing

Stencils, Fourier transforms, and other non-network work cast as matmuls.

13 Vision, convolution, and encoders

SUMMARY

A 256-channel 3x3 convolution runs about 3.8 times faster than the GPU at about 9 times the energy efficiency, and the engine draws less absolute power on every workload class measured. Serve encoders and embeddings on the engine below a batch of about 23, a self-attention block below a batch of about 6, and vision convolution at every batch. On-engine image preprocessing arrives on the A14 generation and later; the M1 has no texture engine. On the M1 and the A14, keep slice and crop source magnitudes below 4094 on the width axis or they saturate to infinity.

Convolution and vision are the work the engine is built for. This chapter gives the measured economics of that work against the GPU, and the batch threshold below which encoder serving stays on the engine. It also covers the on-engine image preprocessing path and the family that gates it, and the model-construction rules that keep a vision or encoder model on the engine end to end.

13.1 Convolution datapath

A 3x3 convolution at 256 channels runs about 3.8 times faster on the engine than the same kernel on the GPU, at about 9 times the energy efficiency per result [AppleCoreML]. The advantage widens with depth. A sixteen-deep stack of 3x3 convolutions at 256 channels runs about 2 times faster than the GPU and at about 14.5 times its energy efficiency on the M1, the largest power difference in the M1 workload set. On the M5 it runs about 4.2 times faster at about 13 times the efficiency. A ResNet-18 forward pass runs about 6.1 times faster than the GPU reference and at about 11 times the energy per inference.

Two properties of the datapath produce this. The first is Winograd. Any dense 3x3 stride-1 convolution with enough channels lowers to the F(2x2, 3x3) transform, which replaces 36 direct multiplies per 2x2 output tile with 16, a factor of about 2.25 reduction in multiplies. The compiler selects this path automatically for dense non-unicast 3x3 stride-1 layers and falls back to direct convolution otherwise, so a model written with 3x3 stride-1 layers gets the reduction without any annotation. The conv-relevant consequence is that a layer takes Winograd only with enough channels to amortize the transform, and a float kernel requires more than a non-float one; the full eligibility test, two tile sizes, and work-threshold derivation are in chapter 20. There is no accumulator widening tied to the transform; its precision safety is that higher float work threshold, not a wider accumulator. The second property is the engine's lower power draw. On a large compute-bound matrix multiply the engine draws 4.4 W against the GPU's 32.5 W, a 4.0 times efficiency advantage that holds even on the matrix-multiply class where the GPU runs faster. Across the M1 workload set the engine runs at 2 to 14.5 times the GPU's energy efficiency, drawing less absolute power on every class measured.

The end-to-end convolution ceiling on the M1 is about 1.8 fp16 TFLOP/s at about 1.78 W, or about 643 GFLOP/s per watt under load. A 3x3 convolution at 256 channels and a 28 by 28 feature map reaches that peak: it runs in about 0.51 ms at about 1823 GFLOP/s. Throughput rises with spatial size as the fixed per-eval overhead amortizes, and the same 3x3 64-channel kernel runs at about 63 GFLOP/s at 16 by 16 and about 1102 GFLOP/s at 128 by 128.

13.2 MAC array and its tiling

The engine is a fixed-geometry multiply array supplied by DMA engines that re-base per tile. On the M1 the array is four NE cores, read from the per-chip parameter table as a core count of 4, with each core a two-dimensional multiply tile backed by an accumulator file of 8 work-units. The compiler assigns output channels across the cores by a strided round-robin: channel c is on core $c \bmod 4$, so the four cores are independent parallel slices and the scheduling granule is one output channel.

The parallelism is measured directly. Driving a single heavy convolution at each output-channel count in a back-to-back dispatch loop, where the per-dispatch overhead pins the dispatch rate constant, the sustained multiply rate rises in exact integer multiples with the number of cores active, as table 13.1 records at one through four active cores.

Table 13.1. Per-core throughput and power scaling on the M1, measured at a fixed dispatch rate.

Output channels	Cores active ($c \bmod 4$)	Engine power, net	GMAC/s	Ratio to one core
1	1	811 mW	3.8	1.00x
2	2	822 mW	7.6	2.00x
3	3	831 mW	11.4	3.00x
4	4	843 mW	15.4	4.05x

The power rail steps about 10 to 11 mW per added core over an always-on floor near 800 mW, the floor being the base plus dispatch domain and each step one of the four independently power-gated compute domains turning on. Above four output channels the array fills more lanes inside the same four cores, and the rate keeps rising linearly while the dispatch rate stays floor-bound.

The compiler tiles output channels into output-channel groups sized to the accumulator file. The group size is about the 8-accumulator budget divided by the kernel-element count $k_w k_h k_d$, rounded down to a power of two and capped by a per-element byte budget of 32, 16, or 8 bytes depending on the weight format. A 3x3 convolution thus has about 9 times the per-channel accumulator pressure of a 1x1, so its group is about 9 times smaller and it needs more passes. The measured signature is a super-linear cost step: a 1x1 fp16 convolution doubles its per-layer cost once the output channel count crosses the accumulator file between about 192 and 256 channels. A 3x3 convolution reaches that pass-doubling threshold at fewer channels for the same reason. This is the mechanism that makes Winograd worth selecting for 3x3: cutting the effective kernel-element count enlarges the group and relieves the accumulator pressure the threshold makes visible.

13.3 Convolution variants and their lowering

The convolution variants all map onto the same multiply array, and what differs is how the compiler tiles the weights and sets up the channel grouping, which table 13.2 gives variant by variant.

Table 13.2. How each convolution variant lowers onto the multiply array on the M1.

Variant	Lowering onto the datapath
Standard 2D convolution	native; weights tiled into output-channel groups, channels assigned across the cores by a strided round-robin
Transpose or deconvolution	native; runs as a fractionally-strided forward convolution on the same array, and serves the convolution data gradient
Dilated	lowered by a space-to-batch decomposition with a factor list of 2, 3, 4, or 8, folding the dilation into a strided input gather so the kernel itself stays dense
Depthwise	native; each channel is its own group with no cross-channel reduction, one input channel per output channel

Variant	Lowering onto the datapath
Grouped	native; the group count partitions the channel-to-core assignment, and runs best when the group count divides the channel count and the core count
3D convolution	capability present in the hardware parameter table but not reachable on the direct path; the compiler reports the operation as not implemented on every backend

The speed and efficiency advantage holds across the workload classes the engine is built for, which table 13.3 gives from the single convolution through ResNet to the encoder.

Table 13.3. Engine versus GPU speed and efficiency across convolution, ResNet, matrix multiply, and encoder workloads.

Workload	Engine vs GPU speed	Engine vs GPU efficiency
3x3 convolution (256 channels)	3.8x faster	9x more efficient
Convolution stack, 16 deep at 256 channels (M1)	2x faster	14.5x more efficient
Convolution stack, 16 deep at 256 channels (M5)	4.2x faster	13x more efficient
ResNet-18 forward	6.1x faster	11x more efficient
Batched matrix multiply	faster below N of 2048	more efficient at every batch size
Single-sentence encoder	4.4x faster	faster at low to moderate batch

13.4 Encoders and embeddings

An encoder forward pass favors the engine at low to moderate batch on both latency and energy. A single-sentence encoder runs about 4.4 times faster than the GPU. A twelve-layer encoder forward runs about 1.5 times faster and at about 18 times the energy per inference. Short-sequence attention, a transformer block at sequence length 197, is at once the fastest, most efficient, and most accurate in fp16 on the engine, because its partial sums stay in range and the wide accumulator holds their precision.

Batch size moves the choice, because serving batches requests. The engine saturates near a batch of 1 while the GPU scales with batch, which produces a throughput crossover. On a true-batched encoder block the GPU overtakes the engine on throughput near a batch of 23, and on a self-attention block near a batch of 6. The energy crossover is at a larger batch than the throughput crossover. On vision convolution serving the energy crossover never appears: the engine leads throughput by 3.6 to 5.7 times and energy by 6 to 10 times at every batch from 1 to 256.

Serve encoders and embeddings on the engine below a batch of about 23, and a self-attention block below a batch of about 6. Above those points the GPU is the throughput device. Vision convolution serving stays on the engine at every batch.

13.5 On-engine image preprocessing

Resize, crop-and-resize, grid sample, affine warp, and reflective or symmetric padding run on a single hardware sampling datapath, the texture engine, on the A14 generation and later [AppleVision]. The datapath is a DMA-side sampler fused onto a layer rather than a separate pass: it reads a coordinate or box tensor through a fixed interleave and applies bilinear or nearest-neighbor interpolation in line with the convolution that consumes its output. The payoff is the elimination of a host preprocessing stage. Image dequantization and resize execute on the engine in the same program as the model, so the resampled tensor never makes a round trip to the host between the camera frame and the first convolution.

One datapath backs a fixed set of front-end operations, each gated by the same parameter-table bool, which table 13.4 lists with how each lowers on the A14 generation and later.

Table 13.4. The texture-engine operation set and how each lowers on the A14 generation and later.

Operation	Bottoms	Lowering on the A14 and later
Resize, upsample	1	native texture-engine resize unit
Crop-and-resize, ROI-align	2	native; a denormalization scale-and-bias pair precedes the index input, box tensor in fp16
Resample, grid-sample, warp	2	native; coordinate tensor of 1 or 2 channels read through the index interleave
Affine transform	2	decomposes to resample with a computed coordinate grid; matrix fp16, six coefficients
Resize-as	2	decomposes to resize
Reflective or symmetric padding	1	uses the texture-engine pad mode
Hardware gather	2	uses the texture-engine index interleave path

The sampler reads its coordinate or box tensor through an interleave of 1, 2, 3, 4, or 8, set by the box-coordinate layout. A two-corner box such as `Y0X0Y1X1` selects interleave 4, an origin-and-size box selects 4, a two-coordinate point selects 2, and a full four-coordinate batched box selects the default 8. The sampling method is linear, that is bilinear, or nearest-neighbor, and the interpolation weights and the crop scale program two register arrays at the DMA-side sampler. On the M1 the whole set decomposes or rejects. Resize becomes a channelwise deconvolution for integer upsample or a transpose followed by a convolution otherwise, affine warp rejects outright with a not-supported-on-this-architecture diagnostic, and symmetric padding rejects. Gather routes through a limited software envelope that requires a gather-axis size of 3 and a batch and depth of 1.

A single bool in the per-chip parameter table gates the capability, and that bool is zero on the M1. The gate is one bit with no finer per-operation granularity: crop-and-resize, resample, affine, native resize, the hardware gather index path, and reflective or symmetric padding all turn on together at the A14.

The same gate has a precision caveat on the M1 and the A14. The crop and slice path applies a width-axis gain of 16, so a source value above 4094 on the width axis saturates to plus or minus infinity, while height, channel, and batch offsets stay free of the saturation. Chapter 3 derives the bound and chapter 19 gives the build-time guard. The preprocessing consequence is to keep slice and crop source magnitudes below 4094 on the width axis on the M1 and the A14. The A15 generation and later sampler avoids this saturation.

13.6 Keeping the model on the engine

The datapath runs best with a small set of model-construction choices.

- Prefer 3x3 stride-1 dense convolutions over 5x5 or strided forms where the model allows, so the Winograd path is taken and each layer gets the factor of about 2.25 multiply reduction.
- Keep the largest single per-layer operand at or below the 2 MB on-chip working set. A convolution whose live tiles exceed 2 MB is tiled and streamed from DRAM, which adds transfer traffic and moves the layer off the compute ceiling.
- Fuse the whole graph into one program so the about 0.23 ms per-eval dispatch floor on the M1 is paid once rather than per layer. A fixed-iteration stencil fused as one graph amortizes that floor across every step.

- Fold normalization and activation into the convolution. Per-output-channel scale and bias and a fused nonlinearity run on the convolution output at no additional cost, so a batch-normalization or a clamped activation after a convolution adds no separate dispatch.
- Serve encoders below the batch threshold of the previous section, and keep preprocessing on the engine only on the A14 and later, where the texture engine is present.

The native convolution backend operation holds its geometry in a fixed set of attributes that the compiler reads to tile the layer onto the multiply array, given in listing 13.1.

Listing 13.1. The native convolution backend operation and the geometry attributes the compiler reads to tile a layer onto the multiply array.

```
anec.convolution(%input, %weights, %bias) {
  strides         = [sh, sw],          // 1 keeps the Winograd path for a 3x3 kernel
  explicit_padding = [pt, pb, pl, pr], // or padding_style
  groups          = g,                 // 1 standard; g == Cin == Cout is depthwise
  dilation_rates  = [dh, dw],          // folded to a strided input gather, up to 8
  kernel_sizes    = [kh, kw],          // fp16 kernel width bounded at 13 on the M1
  weights_layout                        // weights are output-channel-major, [Cout, D, Cin, H, W]
}  // filter and input rank 4 or 5
```

The matrix-multiply backend operation that the encoder paths and the convolution weight gradient reduce to holds its contraction direction in two attributes rather than a fixed operand order, as listing 13.2 shows.

Listing 13.2. The matrix-multiply backend operation, which holds its contraction direction in two transpose attributes rather than a fixed operand order.

```
anec.matmul(%lhs, %rhs) { transpose_lhs, transpose_rhs }  // depth D must be 1 on both operands
```

A trainable convolution lowers its forward and its data gradient onto the native convolution and deconvolution operations, and its weight gradient through an image-to-column expansion into this matrix multiply, because the hardware cross-correlation operation is single-channel only and cannot serve the multi-channel weight gradient directly.

13.7 Fitting a convolution or encoder block on the engine

A vision or encoder block stays on the engine when its layers take the Winograd path, its working set fits on chip, and the whole graph compiles as one program. The procedure of listing 13.3 builds the convolution stack as one graph, then locates it against the roofline before any device is available.

Listing 13.3. Fitting a convolution or encoder block on the engine as one fused program.

```
# Fit a convolution or encoder block on the engine as one fused program.
# Each stage is convolution, then normalization, then activation.

build graph G:
    input x : [1, 256, 28, 28] fp16

    # Stage 1: keep the convolution 3x3 stride-1 so it takes the Winograd path.
    h = conv(x, weights = w1, kernel = [3, 3], stride = 1)
    h = normalize(h)                      # batch or layer norm, folded into the conv output
    h = activation(h)                     # e.g. relu or gelu, on the same output pass

    # Stage 2: another 3x3 stride-1 stage, same fused shape.
    h = conv(h, weights = w2, kernel = [3, 3], stride = 1)
    h = normalize(h)
    h = activation(h)

    output h

# 1. Check the working set stays on chip before committing to the engine.
working_set = max over stages of resident_bytes(stage)   # in MB
if working_set > 2.0:
    # Too large: it would tile and stream from DRAM. Shrink the operand
    # (fewer channels or smaller tile) and re-check before any other tuning.
    reshape operands until working_set <= 2.0

# 2. Fuse the whole block into ONE program so the per-call floor is paid once.
program = fuse_and_compile(G, target = H13)   # one dispatch, the 0.23 ms floor paid a single time

# 3. Run the fused program on an input image.
output = run(program, x = image)

# Note: input preprocessing can join the same graph only on A14 and later,
# where the texture engine is present.
return output
```

13.8 Reference: convolution and encoder economics on the M1

Table 13.5 collects the M1 convolution and encoder constants this chapter measures, from the GPU speed ratios through the datapath geometry to the batch thresholds.

Table 13.5. The M1 convolution and encoder constants for vision and encoder workloads.

Constant	M1/H13 value
3x3 convolution (256 channels) speed versus GPU	3.8x faster
3x3 convolution (256 channels) efficiency versus GPU	9x more efficient
Convolution-stack speed, 16 deep at 256 channels	2x faster
Convolution-stack efficiency, 16 deep at 256 channels	14.5x more efficient
Matrix-multiply power, engine versus GPU	4.4 W versus 32.5 W
Energy-efficiency range across the workload set	2x to 14.5x
End-to-end convolution ceiling	1.8 fp16 TFLOP/s at 1.78 W
Convolution efficiency under load	643 GFLOP/s per watt
3x3 256-channel 28 by 28 peak	0.51 ms at 1823 GFLOP/s
Spatial throughput, 3x3 64-channel	63 GFLOP/s at 16 by 16, 1102 GFLOP/s at 128 by 128
Per-eval dispatch floor	0.23 ms
On-chip working-set threshold	2 MB
NE-core count	4
Accumulator file per core	8 work-units

Constant	M1/H13 value
Output-channels per cycle, fp16 default / int8 fast path	4 / 8
Channel-to-core assignment	strided round-robin, channel c on core c mod 4
Output-channel-group pass-doubling threshold, 1x1 fp16	192 to 256 channels
Per-core power step over the dispatch floor	about 10 mW
Winograd work threshold, non-float / float / packed	8 / 16 / 32
Texture-engine interleave factors	1, 2, 3, 4, 8
Encoder-serving batch threshold	23
Self-attention-serving batch threshold	6
Q.4 crop-scale saturation, width axis	4094, where $4094 \times 16 = 65504$

14 LLM case study

SUMMARY

Autoregressive decode is bandwidth-bound and dispatch-bound, the two regimes the engine loses, so it belongs on the GPU at every batch size. At a batch of 16 the GPU runs decode about 2.7 times faster and about 4.6 times more energy efficient than the engine. Int8 weights halve the weight traffic but leave the hybrid decoder at about 0.99 times the fp16 rate, because the step is dispatch-bound, not weight-bandwidth-bound. Send the prefill, encoder, and vision front end to the engine, and send the autoregressive decode to the GPU.

Autoregressive decode is the one major workload class where the engine is not the fastest. The single-token decode step runs against the roofline of chapter 9, batching is the only serving control that moves throughput, and the engine's niche is on the other side of the model from the decoder.

14.1 Why decode is not a compute problem

A decoder step generates one token. It reads every weight in the model once, multiplies each against a single activation row, and produces one new row. The arithmetic per byte of weight read is thus near one multiply-add per weight, which puts the step far below the 141 FLOP-per-byte ridge point of chapter 9. Decode is bandwidth-bound: the step spends its time streaming weights from DRAM, not multiplying on the array.

A second cost adds on top of the bandwidth cost. A single decoder layer is not one operation but a chain of projections, a normalization, attention block, and feed-forward pair, and on the direct path each is a separate dispatch. A token through a transformer layer stack issues on the order of forty to fifty small dispatches, every one paying the per-eval floor of about 0.23 ms measured on the M1 in chapter 11. Decode is thus dispatch-bound as well as bandwidth-bound, and both regimes are exactly the ones the engine loses, per the verdict table of chapter 11.

The wide accumulator does not help the decoder either. A transformer down-projection loses precision in fp16 from the per-product rounding of its inputs under cancellation, the hazard named in chapter 3, not from the accumulator. The decoder thus has an accuracy penalty on the engine on top of the speed penalty.

14.2 Hybrid decoder placement

A decoder runs as a hybrid: the bandwidth-heavy, well-conditioned projections execute on the engine in fp16, and two precision-critical sites stay in wider precision off the engine. The first is the residual stream. Over twenty-two layers an fp16 residual add drops the small per-layer update, because adding a tiny delta to a large running sum loses the low bits. The following normalization rescales but cannot recover them, thus the stream must accumulate in wider precision off the engine. The second is the down-projection, the cancellation-heavy step of chapter 3, whose near-cancelled output over a contraction of 5632 takes about 3 percent error in fp16 on the engine, enough to flip the greedy argmax.

Per-operation placement follows table 14.1. The query, key, gate, up, and output-embedding projections fit the engine in fp16; the value projection, output projection, and down-projection take a wider-precision path off the engine.

Table 14.1. Per-projection placement for a hybrid decoder: the engine holds the fp16 projections, three are held off-engine in wider precision.

Projection	End-to-end fp16	Placement
Query, key	Survives	Engine, fp16
Gate, up	Survives	Engine, fp16
Output embedding	Survives	Engine, fp16
Value	Fails	Wider precision, off-engine
Output projection	Fails	Wider precision, off-engine
Down-projection	Fails	Wider precision, off-engine

Placement follows position, not per-operation error. The gate and up projections survive on the engine despite per-operation error comparable to the value and output projections. An operation survives on the engine when its result passes through a wider-precision step downstream. The gate and up projections send the feed-forward delta through the off-engine down-projection, which insulates them. The value and output projections send to the attention output and the residual directly, thus their error compounds through the key-value cache across the layer stack.

14.3 Single stream against many streams

A single decode stream is serial by construction. Token $t + 1$ cannot start until token t is produced, because it consumes that token, so there is nothing to overlap within one stream. The host issues a dispatch, waits for the result, and issues the next, and the engine is idle between the dispatch floor and the next submission.

Concurrent independent streams behave differently. Executing one stream releases the host thread, so several streams interleave their host-side work against each other's engine-side work and lift the aggregate token rate even though each individual stream stays serial. This is a serving control for many requests, not a latency reduction for one. Table 14.2 compares the single-stream, multi-stream, and GPU-batched modes, what each improves, and how its throughput compares with the GPU.

Table 14.2. Decode serving modes, what each improves, and how its throughput compares against the GPU.

Serving mode	What it improves	Throughput against the GPU
Single-stream decode	Latency for one request	Below the GPU; serial, no overlap
Multi-stream concurrent decode	Aggregate tokens per second	Higher aggregate than single stream, still below the GPU
GPU batched decode	Aggregate tokens per second	Fastest and most energy efficient at batch

The multi-stream gain is real but bounded. A fair batched comparison at a batch of 16 puts the GPU about 2.7 times faster and about 4.6 times more energy efficient than the engine on decode, because the engine path stays host-dominated while the GPU scales with batch.

14.4 What int8 weights do and do not buy

Native int8 weights stream at half the bytes of fp16 and cut the weight-read traffic of the bandwidth-bound step. That is a reduction in the primary cost of decode, and it is reachable on the direct path.

It does not make the engine faster on decode. The int8 projections come back numerically correct, but the hybrid decoder runs at about 0.99 times the fp16 version, because once the model is small or the dispatch count is high the step is dispatch-bound and host-bound, not weight-bandwidth-bound. Halving the weight

traffic of a step whose wall time is set by forty-plus dispatch floors and host marshalling does not move the wall time. An int8 gain needs a fused, engine-dominated decoder where the weight stream is the bottleneck, which the per-layer dispatch structure of a general decoder does not provide. Batching, not quantization, is the control that moves serving throughput.

14.5 Two serving controls and two hard caps

Two controls make the hybrid practical for serving. Speculative decoding drafts several tokens with an inexpensive proposer and verifies all of them in one batched forward. Because the weights are read once per forward regardless of the number of drafted tokens, the verification adds almost no cost. On a TinyLlama decoder this lifts the rate from 35 tokens per second to as high as 128 on repetitive text and 42 to 56 on factual prose, with output identical to plain greedy decoding. Batched prefill processes the prompt in chunks of several tokens at once instead of token by token, and because the engine matrix multiplies are flat in batch width, prefill latency collapses by about 2.3 to 5.9 times, bit-equal to serial prefill.

A multi-shape decoder also reaches two hard caps enforced by the device daemon. The in-flight cap is 127 outstanding requests per program, set by a dispatch semaphore of 127 inside the in-memory model. The loaded-program cap is near 128 programs per process: the next load fails with `GetANEFModel: must re-compile` and forces a recompile. For multi-shape serving, where each sequence length or batch width is a distinct compiled program, the loaded-program cap bounds how many resident shapes a process can hold at once.

14.6 Cache that stays resident

The one part of the decoder that fits the engine's execution model is the key-value cache. The cache holds the keys and values of every prior token and grows by one row per step. Re-streaming the whole cache through the host each token would add a copy proportional to the sequence length on top of the weight stream.

Instead, the cache stays resident on the engine across steps, as chapter 2 describes for resident state. A program declares the cache as both an input and an output, and after compilation the runtime aliases the output buffer onto the input buffer, so the updated cache produced by one step is the cache consumed by the next without leaving the device. A masked update writes the new key and value into the cache at the current position, with the position supplied as a small one-hot vector each step, as listing 14.1 sketches.

Listing 14.1. One resident-cache decode step, where aliasing the cache output buffer onto its input buffer keeps the key-value cache on the engine across steps.

```
# Schematic of one resident-cache decode step.
# The cache output buffer is aliased onto the cache input buffer,
# so the cache never round-trips through the host.

compile(graph with inputs [x, slot_onehot, cache_in]
                outputs [logits, cache_out])

alias_output_to_input("cache_out" -> "cache_in")   # output buffer IS input buffer

# per token, the host sends only the new row and the slot, not the cache:
cache_out = cache_in * (1 - slot_onehot) + new_kv * slot_onehot
```

With the cache resident the host sends only the new token and the position one-hot each step, a few bytes, rather than the whole cache. This removes the per-token cache copy and is the form in which a decoder is most efficiently expressed on the engine. A resident cache removes one copy from a step whose wall time is still set by weight bandwidth and dispatch count, so it makes engine decode more efficient without making it as fast as the GPU.

14.7 Two residency mechanisms

The engine has a native persistent-state type and a pair of read-state and write-state operations, the obvious mechanism for a resident cache. On the unentitled runtime path that type parses but does not reach the engine. The front end recognizes the state operations, but the backend rejects them during the conversion to the engine program, because the read-state has no state object to bind to: the runtime registers the cache operand as a plain input rather than as a persistent state, so the binding step fails before code generation. The native route thus stays closed pending a compiler option that moves the state operations off the engine subgraph.

The buffer-aliasing route is open and verified. It compiles a program whose cache is both an input and an output, then aliases the output buffer onto the input buffer after compilation, so the tensor persists on the engine across dispatches with no host re-supply. The aliasing primitive is the same one that keeps optimizer state resident for on-engine training in chapter 15. Two measurements confirm it on the M1. A resident accumulator that adds one each step, aliased output onto input with no re-supply, returns 1, 2, 3, 4 over four dispatches, accumulating in place. A resident cache of six slots, written by a masked positional update of $(t+1) \times 10$ at slot t, returns 10, 20, 30, 40, 50, 0 over five steps, each token written to its slot with the cache never leaving the device. The coordination behind residency is a hardware event signal-and-wait primitive that sequences the producer and consumer of the cache buffer, emitted by the compiler rather than requested by the user.

14.8 A resident-cache decode step

The procedure of listing 14.2 compiles the decoder once with the cache as a paired input and output, aliases the buffers, then sends only the new token and its slot per step.

Listing 14.2. The runtime calls for a resident-cache decode step, binding one cache buffer to both ports so the cache updates in place each token.

```
/* Allocate one cache buffer object and bind it to BOTH the cache_in and cache_out ports, */
/* so the engine writes the updated cache back into the same resident buffer in place.    */
e5rt_buffer_object_alloc(&cache_buf, cache_nbytes, /*type=*/0);
e5rt_execution_stream_operation_retain_input_port(op, "cache_in", &cache_in_port);
e5rt_execution_stream_operation_retain_output_port(op, "cache_out", &cache_out_port);
e5rt_io_port_bind_buffer_object(cache_in_port, cache_buf);
e5rt_io_port_bind_buffer_object(cache_out_port, cache_buf);  /* alias: cache stays resident */

for (int t = 0; t < max_tokens; t++) {      /* one encode + execute per token */
    /* host marshals only the new row and slot; the cache is NOT re-sent */
    e5rt_execution_stream_operation_prepare_op_for_encode(op);
    e5rt_execution_stream_encode_operation(stream, op);
    e5rt_execution_stream_execute_sync(stream);
    e5rt_execution_stream_reset(stream);
}
```

14.9 Reference: decode placement and the serving caps

Table 14.3 collects the decode placement figures and the two device-daemon serving caps this chapter measures.

Table 14.3. The decode placement figures and the two device-daemon serving caps.

Quantity	Value
Per-eval dispatch floor (M1)	0.23 ms
Dispatches per transformer layer stack	40 to 50
Down-projection fp16 error over contraction 5632	about 3 percent

Quantity	Value
Int8-hybrid decode rate versus fp16	0.99x
GPU versus engine decode speed at batch 16	2.7x faster
GPU versus engine decode energy at batch 16	4.6x more efficient
Speculative-decoding rate, TinyLlama	35 to 128 tokens per second
Speculative-decoding rate, factual prose	42 to 56 tokens per second
Batched-prefill latency reduction	2.3x to 5.9x
In-flight cap per program	127 outstanding requests
Loaded-program cap per process	near 128 (`GetANEFModel: must re-compile`)

14.10 Verdict: encoders, not decoders

The engine's niche is the other side of the same model. Encoders and embedding models process a whole sequence in one forward pass of compute-bound matrix multiplies and normalizations, the engine's strong regime. Chapter 11 measures a single-sentence encoder about 4.4 times faster than the GPU at low batch, with the crossover to the GPU only near a batch of 23. Vision and convolution hold the same verdict more strongly. Send the prefill, encoder, and vision front end to the engine, and send the autoregressive decode to the GPU.

15 Training on the engine

SUMMARY

The engine has no backward operation, yet a full forward, backward, and optimizer loop runs as ordinary inference-style graph operations with the optimizer state resident across steps. The registered gradient set matches the closed form to a cosine of 1.0000, so transformers, normalization-based convolutional networks, and gated linear networks all train end to end. A small convolutional network trains to a final test accuracy of 0.9080 on the M1 and 0.9070 on the M5 at a loss scale of 1024, a difference of one test sample in one thousand. The conv weight-gradient saturates above 4094 on the width axis on the M1 and M2, so a preflight check guards that one numeric edge.

There is no backward operation, no gradient layer, and no adjoint primitive in the compiler's operation catalog. A network still trains end to end on the engine, optimizer step included, because the forward pass, backward pass, and parameter update are all expressed as ordinary inference-style graph operations the engine already runs. This chapter states how that expression works, which gradients are numerically trustworthy, where one path diverges in fp16, and why the result is a capability rather than a speed claim.

15.1 A backward pass built from forward operations

A search of the compiler binary for an engine-native gradient layer returns nothing. Every gradient operation held in the shared compiler belongs to the graphics-processor dialect, is costed by the graphics-processor cost model, and executes there rather than on the engine. The convolution data-gradient and weight-gradient, the max-pool and average-pool gradients, normalization, recurrent, top-k, pad, strided-slice, and resize gradients are all graphics-processor operations. At the operation level the engine is inference-only.

The engine-native convolution family is the forward convolution and a single-channel cross-correlation. The forward convolution has a transpose mode, so the data-gradient, which is a transposed convolution, has a native operation. The weight-gradient has no native operation, because the only cross-correlation is single-channel and cannot serve the multi-channel correlation a weight-gradient requires. The weight-gradient thus lowers through a patch-extraction expansion into a matrix multiply, which is why a trainable convolution holds its weight as a graph input rather than a folded constant and why its backward cost grows with the minibatch.

The public on-device update path meets the same boundary [AppleCoreML]. When the system framework fine-tunes a model on device, the forward pass may run on the engine, but the backward pass runs on the graphics processor or the host through those gradient operations.

Training the whole loop on the engine thus means not calling any backward operation. The forward pass is a graph of engine operations. The backward pass is a second graph, built by the host from the same forward operations and their analytic gradients, that computes cotangents through the network. The optimizer update is a third graph that combines the gradients with the parameter and optimizer state. All three are inference-style graphs of operations the compiler already accepts, so all three compile and dispatch on the direct route from chapter 6.

A trainable weight is a graph input, not an embedded constant. The forward graph reads the weight from a bound buffer rather than from a folded constant tensor, so the update graph can write a new value into that

buffer between steps without recompiling. This is what lets the optimizer step run on the engine: the weight is an operand, and an operand can be both read by the forward pass and written by the update.

15.2 Gradient vocabulary and its correctness

A gradient is built by composing each forward operation with its vector-Jacobian product. The registered set was checked against closed-form derivatives by forming a linear loss $L = \sum_i (\mathrm{op}(x)_i \cdot w_i)$, whose exact gradient is $w \cdot \mathrm{op}'(x)$, and comparing the engine-computed gradient to that reference.

The core set matches to a cosine of 1.0000 against the closed form, at the fp16 error level. That set covers the activations `relu`, `sigmoid`, `tanh`, and `gelu`, the elementwise `mul`, `add`, `sub`, `square`, the linear `matmul` and batched `matmul`, `softmax`, `conv`, `avg_pool`, `max_pool`, the reductions `reduce_sum` and `reduce_mean`, and the shape operations `transpose`, `reshape`, `flatten`, `slice`, and `concat`. The four normalization layers `layer_norm`, `rms_norm`, `group_norm`, and `l2_norm`, the `silu` activation, and a set of unary math operations including `exp`, `sqrt`, `rsqrt`, `log`, `erf`, and `cos` were added later and verified against a finite-difference reference at the same cosine. The normalization gradients re-inject the per-channel scale as a supplied value-input, since the engine has no constant-tensor operation to hold it. Table 15.1 groups the registered operations by kind, from the activations through the reductions to the normalization layers.

Table 15.1. The registered gradient vocabulary, each verified to a cosine of 1.0000 against the closed form or a finite-difference reference at the fp16 error level.

Group	Operations with a registered gradient
Activations	`relu`, `sigmoid`, `tanh`, `gelu`, `silu`
Elementwise	`mul`, `add`, `sub`, `square`
Linear	`matmul`, batched `matmul`, `softmax`, `conv`, `avg_pool`, `max_pool`
Reductions	`reduce_sum`, `reduce_mean`
Shape	`transpose`, `reshape`, `flatten`, `slice`, `concat`
Normalization	`layer_norm`, `rms_norm`, `group_norm`, `l2_norm`
Unary math	`exp`, `sqrt`, `rsqrt`, `log`, `erf`, `cos`

With the normalization gradients present, a transformer block, normalization-based convolutional network, and gated linear network all train end to end on the engine, where previously only a plain multilayer network did. Some operations have a forward but no registered gradient, among them `cumsum`, `amax`, `amin`, and several parametric activations. A model that uses one of those compiles and runs its forward pass, then the backward construction raises an explicit error rather than producing a silently wrong gradient. The gap there is coverage, not correctness.

15.3 A resident-state training step

The optimizer state stays resident on the engine across steps, through the buffer-aliasing mechanism of chapter 2. The first and second moments of an adaptive optimizer, along with the weights themselves, persist as buffers in the engine working set from one dispatch to the next. The host sends only the per-step minibatch and the scalar learning rate, and reads the weight buffers back at a checkpoint. The large held tensors never cross the host boundary on every step.

One dispatch advances the network by one optimizer step, as listing 15.1 shows: forward, backward, and update in a single submitted graph.

Listing 15.1. A resident-state training step run as one engine graph, with weights and optimizer moments resident across steps.

```
# Resident buffers, allocated once and kept in the engine working set:
#   W           trainable weights        (read by forward, written by update)
#   M, V        optimizer moments        (read and written by update)
# Per-step host inputs: minibatch (x, y), learning rate lr_t

for step in range(num_steps):
    # all three stages are one engine graph, one dispatch
    logits = forward(W, x)                  # inference-style ops
    g      = backward(W, x, y, logits)      # vjp graph, no native backward op
    M      = beta1 * M + (1 - beta1) * g           # update, on engine
    V      = beta2 * V + (1 - beta2) * (g * g)
    W      = W - lr_t * M / (sqrt(V) + eps)        # writes resident W in place
    # M, V, W remain resident; host sends only (x, y, lr_t) next step
```

The adaptive update is

$$W_{t+1} = W_t - \eta_t \frac{M_t}{\sqrt{V_t} + \epsilon}, \quad M_t = \beta_1 M_{t-1} + (1-\beta_1) g_t, \quad V_t = \beta_2 V_{t-1} + (1-\beta_2) g_t^2$$

where g_t is the gradient from the backward graph. The host-supplied learning rate η_t already absorbs any loss scaling, because the scale factor cancels in the ratio $M_t/\sqrt{V_t}$ and must not be divided out a second time.

A small convolutional network trains this way to a final test accuracy of 0.9080. On the M1 generation, the seeded handwritten-digit network reaches that accuracy after 300 steps, deterministic and reproducible to the digit across runs.

15.4 Conv weight-gradient divergence

One gradient path diverges in fp16 on the M1 and the M2 generations. The convolution is built from a patch-extraction step, a set of width-offset slices, followed by a matmul. A nonzero-offset width slice on these generations saturates above 4094, the bound derived in chapter 3 and guarded in chapter 19.

The training-relevant consequence is that the weight-gradient runs the backward activations back through those same width-offset slices. When the loss-scaled backward activations exceed about 4094, a few weight-gradient elements saturate to infinity and the gradient is corrupted. The break is magnitude-gated and finite-to-infinity, not a small rounding error: at a fixed shape the path is exact at loss scale 384 and produces its first infinity at loss scale 512. A larger input magnitude crosses the same threshold at a lower loss scale.

Two independent variables cross the threshold, the input magnitude and the loss scale, as table 15.2 records across a sweep of both.

Table 15.2. Conv weight-gradient saturation at a fixed shape, finite-to-infinity and magnitude-gated.

Input scale	Loss scale	Result	Infinities
1	256	free	0
1	384	free	0
1	512	saturates	1
1	768	saturates	3
1	1024	saturates	8
4	256	saturates	36

A 1x1 convolution does not touch the slice path, because it has no width-offset patch slice. The hazard is limited. A convolution-first network sends the normalization-bounded input through the width-offset slices on the forward side, where values stay well below 4094 regardless of loss scale. Its conv input-gradient, the path that would hold the loss-scaled gradient through the slice, is discarded because no layer precedes the first convolution. The handwritten-digit network trains correctly at loss scales of 128, 1024, and 65536, with overlapping curves and final accuracies of 0.9070, 0.9080, and 0.9100. The residual risk is a network that pushes width-offset-slice values in the weight-gradient path past 4094, which a preflight check can flag on the M1 and M2 generations. The M5 generation takes a different slice route and has no such saturation; the non-saturating route arrives on the A15 generation and later.

15.5 Cross-generation parity

The same seeded network trains deterministically on both generations, with the data order and initialization fixed. The two accuracy curves track to three decimals through all 300 steps and part by a single borderline prediction at the end, which table 15.3 traces step by step from initialization to step 300.

Table 15.3. The seeded handwritten-digit network trained identically on both generations, deterministic and reproducible.

Step	M1 test accuracy	M5 test accuracy
0	0.0850	0.0850
50	0.8790	0.8810
100	0.9020	0.9020
150	0.9050	0.9030
200	0.9070	0.9070
250	0.9080	0.9070
300	0.9080	0.9070

That one-sample gap is the end-to-end signature of the per-operation cross-generation fp16 difference, at most one unit in the last place per operation, accumulating across 300 steps until it flips one near-threshold logit. The difference does not vanish, but it is far too small to affect the trained model, so training is portable across the two generations.

15.6 A resident-state training loop

A training loop on the engine builds three graphs from the same forward operations: the forward pass, a backward graph from the registered gradients, and an adaptive update. The weights and optimizer moments stay resident across steps, and the host sends only the per-step minibatch and the scalar learning rate, where the learning rate already absorbs any loss scaling.

The procedure of listing 15.2 marks the weights as trainable graph inputs, keeps the optimizer state resident, and advances the network one optimizer step per dispatch.

Listing 15.2. A resident-state training loop, keeping the weights and optimizer moments resident across all 300 steps.

```
graph G:
    input   x : [B, 1, 28, 28] fp16          # one minibatch of images
    input   y : [B]                           # one minibatch of labels
    weights W                                 # trainable, an input not an embedded constant
    logits = forward(W, x)                    # inference-style ops only
    loss   = cross_entropy(logits, y)
    output  loss, grad(loss, W)               # autodiff appends the backward pass

program P = compile(G, target = H13)   # forward + backward in one program, one dispatch
M, V := 0                                     # adaptive optimizer moments, resident on the engine
for step in 1..300:
    grads = dispatch(P, x_batch, y_batch)     # forward and backward in one submitted graph
    M = beta1 * M + (1 - beta1) * grads               # update, on-device
    V = beta2 * V + (1 - beta2) * (grads * grads)
    W = W - lr_t * M / (sqrt(V) + eps)        # optimizer step writes resident W in place
    # M, V, W stay resident; host sends only (x_batch, y_batch, lr_t) next step
    # lr_t already absorbs loss scaling: it cancels in M / sqrt(V), do not divide out again
W_final = read(W)                             # read the weights back at a checkpoint
```

All three stages submit as one engine graph per step, so the weights and moments never cross the host boundary between steps.

15.7 Reference: training correctness and the numeric edge

Table 15.4 collects the training correctness figures and the single guarded numeric edge this chapter establishes, from the gradient cosine through the final accuracies to the saturation threshold.

Table 15.4. The training correctness figures and the single guarded numeric edge.

Quantity	Value
Registered-gradient cosine versus closed form	1.0000
Final test accuracy, M1	0.9080 after 300 steps
Final test accuracy, M5	0.9070
Cross-generation accuracy gap	one test sample in one thousand
Per-operation cross-generation fp16 difference	at most one unit in the last place
Loss-scale accuracies, M1 (128, 1024, 65536)	0.9070, 0.9080, 0.9100
Conv weight-gradient saturation, width axis	4094, where $65504/16 \approx 4094$
First infinity at fixed shape	loss scale 512 (exact at loss scale 384)
Generations with the saturation	M1 and M2; non-saturating on A15 and later

15.8 Capability and its scale

Training reaches the engine for the supported operation set, with the optimizer state resident across steps and the conv weight-gradient as the one guarded numeric edge. The loop is dispatch-bound: the per-step work is small relative to the dispatch cost, so at this scale the engine is no faster than the host or the graphics processor.

16 Numerical and scientific computing

SUMMARY

Dense linear algebra and fixed-iteration numerics fit the engine, with the wide accumulator holding precision to a condition number of roughly 10^4 before fp16 input rounding sets the error floor. Dense factorizations and full spectral decompositions run by unrolling into one static graph, reaching about $n = 32$ for the factorizations and about $n = 20$ for the spectral decompositions. The discrete Fourier transform runs as a matrix multiply, fp16-clean to about $N = 2048$. The single architectural limit is a runtime-sized output, not pivoting, since a data-dependent pivot is reached by cutting the graph at the pivot selection.

The Apple Neural Engine is a dense fp16 matrix engine with a wide accumulator. That hardware structure decides which numerical and scientific kernels run on it without a precision penalty and which the architecture bars. Dense linear algebra and fixed-iteration numerics fit; the dense factorizations and full spectral decompositions run by unrolling into one static graph; the discrete Fourier transform fits as a matrix multiply while the fast-transform butterfly fits only as a static unroll. The one architectural limit is a runtime-sized output, not pivoting: a data-dependent pivot is reached by cutting the graph at the pivot selection.

16.1 What the dense engine computes well

The three levels of dense linear algebra map onto the multiply array directly [AppleAccelerate]. A level-1 operation is a vector reduction or scaling: a dot product is a single contraction, and an axpy or a norm is a scaled add or a reduction. A level-2 operation is a matrix-vector product, one matmul with a unit output dimension. A level-3 operation is a matrix-matrix product, the native shape of the array. The wide accumulator (chapter 3) holds the reduction in a register of fp32 class, so a representable sum comes back near exact and the only quantization is the fp16 rounding of the inputs and the output.

Iterative kernels inherit the same precision behavior because each step is a matmul or a reduction. Power iteration for a dominant eigenpair, conjugate-gradient sweep for a symmetric positive-definite solve, and fixed-cycle generalized-minimal-residual solve for a general system all fold the matrix as a constant weight and run as one program. Because the matrix is a fixed weight, the program size does not grow with n: a symmetric positive-definite solve runs unchanged from a small system up to $n = 512$ at the same relative error. Across these solvers the wide accumulator holds the precision to a condition number of roughly 10^4 before fp16 input rounding, not the accumulator, sets the error floor.

The measured envelope on the on-engine iterative solvers is a relative error near 10^{-3} at a condition number at or below 10^2. A dominant eigenpair from power iteration returns to about 10^{-15}, limited by the spectral gap and not by the datapath, because the iteration is a repeated matmul whose fixed point is exact in the representable range.

The accumulator that holds the precision is measured, not assumed: a reduction accumulates in a register of fp32 class and rounds to fp16 only at the output port, with the silicon-probe worked examples and the radix-4 fan-in derived in chapter 3. The consequence for these kernels is that a representable sum comes back near exact while a cancellation-heavy reduction has the input lanes round in radix-4 groups first, so the

matrix-multiply contraction path is the more accurate route, exact to fp32 then rounded once at the output. The compiler emits a precision warning steering a narrow reduction toward it.

16.2 Factorizations and spectral decompositions by static unrolling

The direct factorizations of dense linear algebra run on the engine, and the assumption that they were barred is a misreading of the static-dataflow constraint. No data-dependent control flow is not the same as no data-dependent computation. A data-dependent value, a pivot or a selection, flows through a fixed graph when a comparison, a `select`, an `argmax`, and a matmul hold it, and a convergence-gated loop becomes a fixed sweep count chosen at compile time. A factorization is thus expressible by unrolling its $O(n^3)$ recurrence into one static graph, and the closed-form recurrences that have no runtime branch unroll directly.

The explicit Gram-Schmidt QR, unpivoted Cholesky, and Doolittle LU run as one program and reach about $n = 32$ before the unrolled chain dominates compile time. The full dense spectral decompositions run as well. A fixed-sweep cyclic-Jacobi eigensolver has a compile-time-fixed sweep order and closed-form rotations with no pivot and no branch, and it converges in about six to ten sweeps for any symmetric input, so the full symmetric eigendecomposition unrolls and reaches about $n = 20$. The generalized eigenproblem runs as a Cholesky factor followed by a triangular solve and that eigendecomposition. The nonsymmetric eigenvalues run as an unshifted QR iteration assembled into one fused program, where the unshifted form avoids the data-dependent shifts and deflation that make the standard routine look unportable. The full singular-value decomposition runs as the square root of the eigendecomposition of $A^{\mathrm{T}}A$, and a top-k singular-value decomposition of a large matrix runs as a randomized sketch followed by the on-engine QR and small-block decomposition.

Pivoting is reached by cutting the graph at the pivot. A pivoted LU runs with an on-engine `argmax` selecting the pivot row, expressed as a segmented graph: the `argmax` cuts the program into segments, and the data-dependent pivot index flows between them as a value rather than as a branch. The measured relative error on the segmented pivoted LU is about 5×10^{-4} at small n.

The single architectural limit is a runtime-sized output, not pivoting and not the recurrence. A rank-revealing or adaptive-tolerance factorization emits only the values above a tolerance, and how many that is depends on the data, so a static program cannot emit a runtime-sized result and can only emit all n values plus a mask. That limit is independent of whether the arithmetic is fp16 or fp64. What fp16 limits separately is the conditioning range, the roughly 10^4 ceiling above; the Jacobi spectral decompositions hold to a condition number of about 10^1 to 10^2 because the squared system $A^{\mathrm{T}}A$ doubles the conditioning. What graph size limits is n: the explicit factorizations materialize the matrix in-graph, so they are compile-size bound at the $n = 20$ to 32 figures above, while the iterative solvers fold the matrix as a constant weight and run unchanged to $n = 512$.

16.3 Stencils, integration, and series

A computation with a static iteration count fits, because the engine unrolls a fixed loop into one fused program and pays the per-dispatch floor once for the whole sweep. A partial-differential-equation stencil iterated a fixed number of times is the clearest case. A five-point stencil run thirty-two times as one fused graph keeps every intermediate resident in the working set and amortizes the dispatch floor across all thirty-two steps, which is why it is among the engine's widest efficiency margins against the GPU (chapter 10). Explicit ordinary-differential-equation integration with a fixed step count, fixed-iteration Newton solve, and truncated power series each share this form: the trip count is known at compile time, so the whole iteration becomes one static graph.

A data-dependent trip count does not run. A loop that runs until a residual falls below a tolerance, or until a step-size controller accepts a step, asks the engine to decide at runtime how many iterations to execute, the same data-dependent control flow that bars pivoting. The fit is the fixed-budget form of each method: a convergence-gated loop becomes a fixed sweep count, chosen at compile time to cover the worst expected input.

16.4 Fourier transform as a matrix multiply

The discrete Fourier transform of a length-N signal is a linear map, so it is a matrix multiply against a fixed Fourier matrix.

$$X_k = \sum_{n=0}^{N-1} x_n \, e^{-2\pi i k n / N}, \qquad X = Wx, \qquad W_{kn} = e^{-2\pi i k n / N}$$

The matrix W is a compile-time constant, so the transform runs as a single matmul that folds W as a weight, as listing 16.1 gives in NumPy.

Listing 16.1. The discrete Fourier transform expressed as one matrix multiply against a compile-time-constant Fourier matrix.

```
import numpy as np

def dft_matrix(N):
    # W[k, n] = exp(-2j*pi*k*n/N): the compile-time-constant Fourier matrix.
    k = np.arange(N).reshape(N, 1)
    n = np.arange(N).reshape(1, N)
    return np.exp(-2j * np.pi * k * n / N)

def dft(x):
    # The transform is one matmul X = W @ x; W folds as a fixed weight.
    # fp16-clean to about N = 2048; above that the fp16 rounding of W and x
    # dominates the wide-accumulator reduction.
    return dft_matrix(len(x)) @ x
```

The arithmetic is complex, and the engine computes in real fp16, so each complex value is held as a real and an imaginary pair and the complex product expands into four real multiplies and two real adds. The wide accumulator holds the length-N reduction, so the transform is fp16-clean to about $N = 2048$, above which the fp16 rounding of the matrix entries and the input begins to dominate the result.

The fast-transform factorization does not gain over the matmul form on this engine. A fast Fourier transform replaces the $O(N^2)$ matmul with an $O(N \log N)$ chain of butterfly stages, but each stage is a small data shuffle and a complex multiply, and the shuffle is a static structure rather than a runtime loop. The butterfly fits only when fully unrolled into a static graph, and at that point it is a deep chain of tiny operations that pays more dispatch and accumulates more fp16 rounding than the single dense matmul. For the transform sizes the engine handles in range, use the matmul form.

16.5 Sparse and pruned operands

A sparse matrix gives a speed gain, not only a storage gain, when it is a constant weight. The engine has two independent sparsity mechanisms. The first is a compute-time zero-skip: the multiply array detects a zero weight and skips the multiply, driven by a scan of the weight values, so it applies to any operand with zeros regardless of its storage format and pays off in a compute-bound layer. The second is a sparse stream: the weight is stored as a one-bit keep-mask plus the packed fp16 nonzeros, fewer bytes cross DRAM, and the operand is reconstructed on chip, which pays off in a bandwidth-bound layer.

On the M1 the sparse stream is the measured gain for a pruned operand. A convolution stack at about 63 percent zeros streamed as a mask plus nonzeros runs 1.55 to 1.64 times faster than the same weights stored dense. The streamed weight file is at 0.43 times the dense size, and the effective bandwidth rises from about 29.5 to about 47.8 GB/s, the engine's weight-stream ceiling. The reconstruction is lossless apart from the fp16 rounding of the kept values, so the result matches the dense computation to a cosine of 1.0000. The

zero-skip mechanism, by contrast, returns about 1.01 times on the same bandwidth-bound stack, because skipping multiplies does not move a layer whose wall time is set by the DMA, and it needs a compute-bound layer to show. Table 16.1 sets the zero-skip and sparse-stream mechanisms side by side: what each cuts, where it pays off, and its measured result on a pruned stack.

Table 16.1. The two sparsity mechanisms on the M1 and where each one pays off, with the measured result on a pruned convolution stack at about 63 percent zeros.

Sparsity mechanism	What it cuts	Where it pays off	Measured M1 result
Compute-time zero-skip	multiply cycles	compute-bound layers	about 1.01x on a bandwidth-bound stack
Sparse weight stream	DRAM weight traffic	bandwidth-bound layers	1.55 to 1.64x, weight file 0.43x dense

16.6 Discrete Fourier transform as one matmul

The Fourier matrix is embedded as a constant weight, and the transform runs as one matmul against the real and imaginary parts of the signal, which listing 16.2 builds as a compiled engine program.

Listing 16.2. The length-N discrete Fourier transform as a compiled engine program, with the Fourier matrices folded in as constant weights.

```
# The length-N discrete Fourier transform expressed as matrix multiplies.
# X[k] = sum over n of x[n] * exp(-2*pi*i * k*n / N).
# That sum IS a matrix-vector product: X = W x, where W[k,n] = exp(-2*pi*i * k*n / N).

# 1. Form the Fourier matrix as a compile-time constant (built once, on the host).
for k in 0 .. N-1:
    for n in 0 .. N-1:
        angle  = -2 * pi * k * n / N
        Wr[k,n] = cos(angle)               # real part of the Fourier matrix
        Wi[k,n] = sin(angle)               # imaginary part

# 2. Build the transform as matmuls. The signal is held as real and
#    imaginary parts, since the hardware works in real fp16.
build graph G:
    input xr : [N] fp16                    # real part of the signal
    input xi : [N] fp16                    # imaginary part of the signal

    # Complex product (Wr + i*Wi) * (xr + i*xi) split into real arithmetic:
    Xr = matmul(Wr, xr) - matmul(Wi, xi)   # real part of the transform
    Xi = matmul(Wr, xi) + matmul(Wi, xr)   # imaginary part of the transform
    # (Equivalently one matmul over stacked real-imag pairs; same arithmetic.)

    output Xr
    output Xi

# 3. Compile once: Wr and Wi fold in as fixed constant weights, so the whole
#    transform is a single program and the per-call floor is paid one time.
program = compile(G, target = H13)         # W is a constant weight, one program
output  = run(program, xr = real(signal), xi = imag(signal))

# The wide accumulator holds the length-N reduction, so the result stays
# fp16-clean to about N = 2048.
return output
```

16.7 Reference: what fits and what is architecture-limited

Table 16.2 collects the numerical kernel classes this chapter covers, marking each as fitting the engine or architecture-limited with the binding constraint on each.

Table 16.2. Numerical kernel classes that fit the engine or are architecture-limited, with the binding constraint on each.

Kernel class	On the engine	Bound
BLAS level 1, 2, 3 (dot, axpy, gemv, gemm)	fits	fp16 conditioning, roughly 10^4
Iterative solvers (power, conjugate gradient, generalized minimal residual)	fits	fp16 conditioning, roughly 10^2 to 10^4
PDE stencil, explicit ODE, fixed-iteration Newton, truncated series	fits	static trip count required
DFT as $X = Wx$	fits	fp16-clean to about $N = 2048$
Dense factorizations (QR, Cholesky, unpivoted and pivoted LU)	fits as a static unroll	graph size, about $n = 32$; pivot via a segmented `argmax`
Full spectral decompositions (symmetric, generalized, nonsymmetric eig, SVD)	fits as a static unroll	graph size, about $n = 20$; fp16 conditioning about 10^1 to 10^2
Rank-revealing or adaptive-tolerance factorization	architecture-limited	data-dependent output size, no static-graph form
Data-dependent trip count	architecture-limited	no static-graph form
FFT butterfly	only as a static unroll	no gain over the dense matmul

Part V

Practice

17 Model-design rules

The validator limits, the working-set rule, and the alignment that keeps a layer fast.

18 Optimization and the cost model

The three-stage latency estimate and the autotuner that ranks rewrites.

19 Pitfalls and limits

The silent, target-specific failure modes and how to avoid each.

Interlude. Below the API: how the engine works

The transition from the programming surface to the engine beneath it.

17 Model-design rules

SUMMARY

A network compiles only when every operation is inside the validator limits: fp16 activations, width and height at most 16384, channel at most 65536. Convolution kernel width is capped at 13 and arg-min and arg-max reduce an axis of at most 2048. Keep a single operation's largest live operand under 2 MB or it tiles and streams from DRAM. Make channel counts a multiple of the interleave factor and group counts divide the core count, or the engine pads lanes and runs slower.

The validator limits are design rules: the shape, rank, dtype, and mode constraints the per-operation validators enforce, the working-set threshold that decides whether a tensor stays on chip or tiles, and the channel and kernel bounds that distinguish a fast layer from a padded one. Chapter 4 says which operations exist; this chapter says the shapes those operations accept.

17.1 Dtype rule

The datapath is fp16, so every operand a model presents to the engine is fp16. The backend accepts the fp32, int32, and bf16 type annotations, but does not implement them as wider arithmetic, and they do not reach the silicon. The M1 rejects a cast to int32, and bf16 is not usable as a program input or output dtype. The one widened quantity is the matrix-multiply accumulator, which is internal and not a tensor a model declares.

Weights are the exception that still resolves to fp16 at the compute step. The engine accepts an int8 weight and runs it through the quantized-convolution path, and the compressed weight forms in chapter 7 stream in their stored width. Activation tensors that flow between operations stay fp16 regardless.

17.2 Shape and rank limits

Tensor extent is bounded per axis, and the bounds are exact. A tensor may span up to 16384 along width, 16384 along height, and 65536 along channel; one element past any of these rejects at compile time.

$$W \le 16384, \quad H \le 16384, \quad C \le 65536$$

These were measured on the M1 by sweeping each axis until compile flipped from accept to reject, and the boundary matched the decompiled maximum-dimension field exactly. Width 16384 compiles and 16385 rejects, and the same one-element step holds for height and for channel.

Convolution has its own kernel bounds, narrower than the tensor bounds and set by the kernel-format field. On the M1 the fp16 convolution kernel width is bounded at 13: a kernel of width 14 rejects. Kernel height reaches 29 when the input is tall enough to hold it, and a kernel taller than its input rejects as a too-large kernel rather than as a height cap. Stride, dilation, and padding do not reject on the M1. A stride outside the native set decomposes to a large-stride path and still compiles, dilation up to 8 compiles, and padding up to 16 compiles. The pooling window is far looser than the convolution kernel and accepts sizes of 128 and beyond, with a tile-alignment quirk that rejects specific odd windows near a power-of-two boundary.

Several operations cap a reduction axis for fp16 accuracy rather than for memory. The arg-min and arg-max operations reduce a channel of at most 2048, and the spatial form caps height or width at 2048 the same way; 2049 rejects. This 2048 limit is independent of the 16384 and 65536 tensor bounds and applies to the reduction axis alone.

17.3 Mode and divisibility rules

The matrix-multiply backend operation requires the contracted depth to be one on both operands, and a depth greater than one rejects. The two operands broadcast on batch and on height when one side is one, and the inner dimensions must agree so that the left width plus padding equals the right channel count.

Grouped convolution divides both the input and the output channel count by the group count, and a count that does not divide rejects. For throughput, the group count should also divide the engine core count, which is four on the M1. A group count of one, two, or four maps one neural-engine lane to one lookup-table lane. A count that does not divide the core count loses that one-to-one mapping and runs slower.

Channel counts should be a multiple of the interleave factor. The engine stores tensors channel-interleaved, and a channel dimension that is not a multiple of the interleave factor is padded out to one, leaving lanes unused. The width axis aligns to a 16-byte direct-memory-access granule, the same factor of 16 that governs the slice-tiling path. Interleave-aligned, power-of-two channel counts avoid silent padding. Listing 17.1 contrasts an interleave-aligned tensor with a padded one and rewrites a fully-connected layer as the equivalent one-by-one convolution.

Listing 17.1. Channel-interleaved tensor layout and a fully-connected layer rewritten as an equivalent one-by-one convolution.

```
# Tensor order is [N, D, C, H, W]; the engine stores it channel-interleaved
# and aligns the last axis to the 16-byte DMA granule.

aligned = (1, 1, 64, 56, 56)   # channels first, W = 56 fills 16-byte lanes
padded  = (1, 1, 64, 56,  1)   # singleton last axis pads out to the granule,
                               # wasting every lane but the first

# Replace a fully-connected layer with an equivalent 1x1 convolution so it
# runs on the convolution datapath. A linear y = x @ W.T over C_in -> C_out
# is the same arithmetic as a 1x1 conv over a [N, 1, C_in, 1, 1] feature map.
def linear_as_1x1_conv(x, W):
    # W: [C_out, C_in] reshaped to a 1x1 kernel [C_out, 1, C_in, 1, 1]
    kernel = W.reshape(W.shape[0], 1, W.shape[1], 1, 1)
    return conv(x, kernel, strides=[1, 1], groups=1, kernel_sizes=[1, 1])
```

Several mode constraints are family-gated and reject on the M1 specifically. The symmetric and reflect padding modes need the texture engine and are unavailable on the M1, where they decompose or are refused. A square-after-reduction fused mode is absent on the M1 and arrives on the A14. The texture-engine sampling operations, resize as a hardware sampler, crop-resize, resample, and affine, are absent on the M1 and arrive on the A14, and the trigonometric sine and cosine arrive on the A15.

17.4 Working-set rule

The decisive size rule is the on-chip working set. A tensor that fits the on-chip static memory stays resident across the operation; a tensor that exceeds it splits into tiles that the engine streams one at a time. The working set is the 2 MB on-chip region, and the matrix-multiply path also bounds the output-channel footprint against a 64 KB kernel-memory budget, rejecting a matrix multiply whose output channels do not fit.

Tiling preserves the result. A reduction or transpose over an axis larger than the on-chip threshold switches to a tiled route at no change to the output, so the threshold is a performance boundary, not a correctness one. Where latency matters, keep a single operation's largest live operand under 2 MB. At fp16, 2 MB holds about $2^{21}/2 \approx 1.05 \times 10^6$ elements, so a square activation of side near 1024 is at the edge of the resident regime, and a larger one streams. This is the working-set input to the cost model in chapter 18, where the roofline relation turns the resident-versus-streaming split into a latency estimate.

17.5 Validating shapes against the design rules

The cost estimate reports the binding limit for each operation, so a shape violation or a working set above 2 MB shows before the program reaches the compiler. Listing 17.2 walks every layer of a graph against the rank, extent, kernel, group, interleave, and working-set rules and flags each violation before the build.

Listing 17.2. Walking every layer of a graph against the design rules, flagging each reject and pad warning before the build.

```
# Tensor order is [N, D, C, H, W]; the engine stores it channel-interleaved.
# Check every layer against the design rules before building, target = H13.

for each layer in graph G:
    # Rank rule: at most 5 axes
    if rank(layer.output) > 5:
        flag(layer, "rank above 5 rejects at compile")

    # Per-axis extent rules
    if width(layer)   > 16384:  flag(layer, "width above 16384 rejects")
    if height(layer)  > 16384:  flag(layer, "height above 16384 rejects")
    if channel(layer) > 65536:  flag(layer, "channel above 65536 rejects")

    # Convolution kernel and group rules
    if layer is conv:
        if kernel_width(layer) > 13: flag(layer, "kernel width above 13")
        if (in_channel(layer) mod groups(layer)) != 0: flag(layer, "in-channels % groups != 0")
        if (out_channel(layer) mod groups(layer)) != 0: flag(layer, "out-channels % groups != 0")
        if (core_count mod groups(layer)) != 0: warn(layer, "groups do not divide cores, slower")

    # Reduction-axis rule for arg-min / arg-max
    if layer is argmin or layer is argmax:
        if reduced_axis_extent(layer) > 2048:  flag(layer, "arg reduction axis above 2048 rejects")

    # Channel-interleave rule: pad warning, not a reject
    if (channel(layer) mod interleave_factor) != 0:
        warn(layer, "channel not interleave-aligned: pads out, wastes lanes")

    # Width DMA-granule rule: last axis aligned to 16 bytes
    if (width(layer) mod 16) != 0:
        warn(layer, "width not aligned to 16-byte DMA granule: pads to granule")

    # Working-set rule: largest live operand under 2 MB stays on-chip, else it tiles and streams
    bytes = element_count(largest_live_operand(layer)) * 2     # fp16 = 2 bytes per element
    if bytes > 2 * MB:
        warn(layer, "working set above 2 MB: tiles and streams from DRAM (slower, still correct)")

# Reshape any flagged layer before tuning; address pad/tile warnings where latency matters.
```

A graph that prints `validates` false, a working set above 2 MB, or nonzero channel padding is reshaped before any other tuning, because a rejected shape never reaches the silicon and a padded channel leaves lanes unused on every dispatch.

17.6 Reference: the per-operation design rules

Table 17.1 collects every validator limit a model must satisfy, with each limit and the result of exceeding it.

Table 17.1. The per-operation validator limits a model must satisfy, with each limit and the result of exceeding it.

Constraint	Limit	Consequence if exceeded
Activation dtype	fp16 only	int32 cast and bf16 input or output rejected on M1
Width axis	16384	16385 rejects at compile
Height axis	16384	16385 rejects at compile
Channel axis	65536	65537 rejects at compile
Conv kernel width (fp16)	13	14 rejects at compile
Conv kernel height	29, input permitting	kernel taller than input rejects
Arg-min/arg-max reduction axis	2048	2049 rejects for fp16 accuracy
Matrix-multiply operand depth	1	depth greater than 1 rejects
Matrix-multiply output channels	64 KB kernel-memory budget	rejects when output channels do not fit
Conv group count	divides input and output channels	non-dividing count rejects
Conv group count, for speed	divides core count (4 on M1)	one-to-one lane mapping lost, slower
Channel multiple of interleave	interleave-aligned	padded out, lanes unused
On-chip working set	2 MB	operand tiles and streams, no error

17.7 Reference: the per-operation validator envelopes

The shape rules are enforced one operation at a time by a family of per-layer validators, the `_ANECValidate<Op>Layer` checkers the compiler runs as the back-end legalizer. There are 50 per-layer validators, and the same code runs both when a high-level model is segmented and when a hand-authored layer is compiled, so a validator that accepts a shape never drifts from the real compile outcome. Table 17.2 gives the binding constraint and reject string for the operations a model presents most often, read from the `_ANECValidate<Op>Layer` family.

Table 17.2. Per-operation validator constraints and reject strings, read from the `_ANECValidate<Op>Layer` family.

Operation	Validator	Binding constraint	Reject string
Convolution	`Conv`	kernel W, H, D each within the per-chip `[min, max]`; in-C and out-C each divisible by groups; groups within the HAL group range	`"Invalid conv kernel %s = %zd, It should be in [%zd ,%zd]"`; `"input/output channels should be divisible by num group"`
Linear	`Linear`	exactly one input; input rank under 5	`"Linear layer must have only one single input."`; `"ANE cannot support Linear with input rank >= 5"`
Matrix multiply	`MatrixMult`	exactly two inputs; depth one on both operands; output channel equals the left channel; left width plus padding equals right channel; output channel bytes fit the kernel-memory budget	`"Matrix mult. layer can only have two bottoms"`; `"depth > 1 is not supported for MatMult"`; `"can not fit the Kmem"`
Pooling	`Pool`	one input; window per axis under the input extent; padding under the kernel; max-pool padding negative, min-pool positive	`"Pool layer must have only one single input."`; `"Pooling mode \"%s\" is not available on this ANE architecture."`

Operation	Validator	Binding constraint	Reject string
Reduction	`Reduction`	one input; each axis at most 4; non-reduced output extent equals input extent, reduced output extent one; square-after-reduce requires the A14 family flag	`"Reduction layer can only have one bottom"`; `"square operation after reduction is not supported"`
Arg-min/arg-max	`ArgMinMax`	channel reduction at most 2048, otherwise height or width at most 2048; padding non-negative and under the kernel; equal left and right padding; pool stride in {1, 2, 4}	`"ArgMinMax left padding value should be smaller than kernel width %d, but %d is given"`
Layer norm	`LayerNorm`	one input; output type float; channel divisible by the group count; grouped form requires depth one	`"... does not yet support depth > 1"`
Softmax	`Softmax`	one input; non-empty axis set; output type float; the general full-axis form is family-gated	`"Softmax is not supported by this ANE architecture"`
Transpose	`Transpose`	one input; each dimension appears once; extent at most 16384 on A13 through A15 and 65536 on A16; without three-dimensional support a channel transpose must factor into {2, 3, 4, 8} with height one	`"NE Input Transpose is not supported for this arch"`
Concat	`Concat`	at least two inputs; every input matches the first on every non-concatenated axis; constant positive axis; matching layout	`"ANE Concat supports only supports const positive axis"`; `"both concat inputs must have the same layout"`
Pad	`Pad`	one input; height or width axes only, no channel or depth padding; reflect and symmetric modes require the texture engine	`"Channel padding is not supported on ANE"`; `"Architecture does not support padding mode."`
Broadcast	`Broadcast`	one input; broadcast only from a length-one axis; depth-axis broadcast requires a family flag absent on the M1	`"Broadcast along depth axis is is not supported on this architecture"`
Gather	`Gather`	index channel divisible by the interleave factor; on the M1 the software envelope requires data batch one, data depth one, index channel three, index width one, index depth one, and a gather-axes count of three	`"Cannot decompose layer on this architecture"`
Reshape	`Reshape`	one input; element count preserved; rank at most 5	`"Cannot reshape a tensor of rank > 5"`

A validator that accepts a shape does not guarantee the operation compiles. A second class of operations passes the schema validator and then fails the code generator below it, the attested-is-not-reachable split of chapter 4. On the M1 the sort and dynamic-slice validators accept their inputs and the code generator rejects the lowering, and top-k compiles only outside a specific forbidden parameter band. The validator predicts schema reachability; only a compile-and-run on the target confirms an operation runs.

17.8 Reference: the channel and divisibility rules

Table 17.3 lists the layout and divisibility rules that separate a fast layer from a padded one, with the constraint and the consequence of missing each.

Table 17.3. The layout and divisibility rules that separate a fast layer from a padded one, with the constraint and the consequence of missing it.

Rule	Constraint	Consequence
Channel interleave	channel a multiple of the family interleave factor, found from the HAL table and the dimensions	a non-multiple pads out to one, leaving lanes unused
Width granule	last-axis width aligned to the 16-byte direct-memory-access granule, the times-16 quantum	misaligned width pads to the granule
Group-to-core ratio	active neural-engine count divisible by the group count, four cores on the M1 so groups in {1, 2, 4}	a non-dividing count loses the one-to-one core-to-lookup-table mapping and runs slower
Max-pool channel	input channel under the per-operation maximum-pool channel bound	a large-channel max-pool tiles
Unicast channel	channel at most the unicast input-channel maximum for the broadcast-to-all-cores path	above it the operation takes the multicast lowering

Depthwise convolution, where the group count equals the input channel count, lowers to the channelwise path and is exempt from the group-to-core preference.

18 Optimization and the cost model

SUMMARY

The compiler turns a shape into a wall-time estimate through three stages: a cycle count, roofline, and fixed dispatch floor, with only the per-chip parameters changing across the family. On the M1 the model fits the five reference convolutions it was tuned against within plus or minus 17 percent at a peak of 3.25 fp16 TFLOP/s, bandwidth of 9.0 GB/s, and 0.23 ms floor. Across a broader sweep the median error is about 31 percent, so the estimate is an ordinal placement tool rather than an absolute-latency oracle. The same model estimates latency for any of the 28 targets without the chip in hand, since bandwidth scales by core count and the floor by clock. The autotuner ranks equivalent rewrites by this estimate and preserves accuracy by default, with a 3.7 to 5.3 times gain on attention blocks at cosine similarity 1.0.

The compiler estimates how long a layer will take before it runs. The same model drives an ahead-of-dispatch latency estimate for any chip in the family and a deterministic autotuner that ranks equivalent graph rewrites.

18.1 Three stages of the model

The first stage estimates compute cycles from the operation and its dimensions. For a convolution the cycle count is the input-channel passes times the kernel volume times the output spatial extent, divided by the active compute units, and sparsity scales it down by the fraction of zero weights. Pooling and elementwise operations use their own cycle variants of the same form.

The second stage applies a roofline. Compute time is the cycle count divided by the throughput rate, memory time is the operand bytes divided by the bandwidth, and the layer latency is the larger of the two. A layer whose memory time exceeds its compute time is bandwidth-bound; the reverse is compute-bound.

The third stage converts to wall time. The model runs at a clock near 0.8 times the maximum frequency, scales the rates by a per-chip efficiency curve, and adds a fixed overhead for the per-input transfer setup plus a constant dispatch cost. The result is a latency in microseconds.

$$t \approx \max\left(\frac{\text{cycles}}{f},\ \frac{\text{bytes}}{B}\right) + t_0$$

The throughput is set by the compute-unit geometry and the efficiency curve, $f = \text{cores} \times 4 \times \text{eff}(f) \times \text{clock}$, and the bandwidth B is the DMA rate the chip sustains across the operand stream. On the M1 the model fits the measured convolution latencies with a peak of 3.25 fp16 TFLOP/s, bandwidth of 9.0 GB/s, and dispatch overhead t_0 of 0.23 ms, all five reference convolutions within plus or minus 17 percent.

18.2 Per-chip parameters

Only the parameter values change from chip to chip; the three-stage structure is identical across the family. The compute-unit count, per-cycle divisor, clock range, and efficiency curve are read from the hardware abstraction table for each target, the table chapter 24 decodes. Table 18.1 gives the source and the fitted M1 and M5 values for each parameter side by side.

Table 18.1. The cost-model parameters that change from chip to chip, with their source and the fitted values for the M1 and M5.

Quantity	Source	M1/H13	M5/H17s
Compute units	core-count field	4	16
Cycle divisor	divisor field	64	64
Clock range	DVFS table	0.40 to 1.43 GHz	0.74 to 2.36 GHz
Efficiency	frequency curve	1.0	about 0.84
Fitted peak	silicon anchor	3.25 fp16 TFLOP/s	8.9 fp16 TFLOP/s
Fitted bandwidth	silicon anchor	9.0 GB/s	57 GB/s
Dispatch floor	silicon anchor	0.23 ms	0.11 ms

The M1 model takes efficiency as 1.0, meaning it sustains peak at any clock; later generations derate to about 0.84 because the array cannot hold the peak multiply rate at the higher clock ceiling. The compute units scale 4 to 16 to 32 to 64 across the die variants, and the clock ceiling rose from 1.43 GHz to 2.36 GHz, which together put the M5 theoretical peak near 5.5 times the M1. The fitted peaks in the table scale more conservatively, about 2.7 times, from 3.25 to 8.9 fp16 TFLOP/s, since the cost model anchors to the rate the array sustains rather than the theoretical product.

Bandwidth scales with the compute-unit count rather than the clock. The M5 streams the same bandwidth-bound model at 57 GB/s against the M1's 9.0 GB/s, close to the 16-to-4 core ratio, so the per-chip anchor sets bandwidth by core count and the dispatch floor by clock. The 9.0 GB/s here is the cost model's jointly calibrated effective fit, and it is not the 51 GB/s single-row saturating weight-stream of chapter 9 nor the roughly 40 GB/s broad-shape effective rate, which measure different things. A model anchored to the M1 alone over-predicts M5 latency by a mean of 99 percent: clock-scaling the M1 bandwidth yields about 15 GB/s, which the true 57 GB/s rate exceeds by about 3.8 times. Substituting the M5 measured bandwidth, floor, and peak brings four of the five reference convolutions within 13 percent. Listing 18.1 evaluates the three-stage model against per-chip anchors, taking the larger of the compute and memory terms and adding the fixed dispatch floor.

Listing 18.1. The three-stage cost model evaluated against per-chip anchors, taking the larger of the compute and memory terms plus the dispatch floor.

```
def estimate_latency(cycles, op_bytes, chip):
    # Stage 1 input: cycles = input-channel passes * kernel volume *
    #                output spatial extent / active compute units.
    # Stage 2: roofline picks the binding term.
    compute_time = cycles / chip["peak_flops"]   # f: cores * 4 * eff * clock
    memory_time  = op_bytes / chip["bandwidth"]  # B: DMA rate the chip sustains
    # Stage 3: add the fixed per-eval dispatch floor.
    return max(compute_time, memory_time) + chip["dispatch_floor"]

# Fitted M1/H13 anchors: peak 3.25 fp16 TFLOP/s, B 9.0 GB/s, floor 0.23 ms.
m1 = {"peak_flops": 3.25e12, "bandwidth": 9.0e9, "dispatch_floor": 0.23e-3}
# A small operation collapses to the dispatch floor; a large one tracks
# whichever of compute_time and memory_time binds.
```

18.3 Reading the model before dispatch

The model returns a per-chip latency estimate for an output graph without running it, which locates a layer against the roofline of chapter 9 ahead of time. The estimate is the same three-stage number described above, evaluated for a named target. For chips that cannot be run locally it is the only available latency figure, since the structure is identical across all 28 targets and only the per-chip parameters differ.

Two readings of the estimate matter most. A small operation collapses to the dispatch floor, and the floor is the optimization target: an operation whose compute term is under 0.23 ms on the M1 gains nothing from a faster array, so the work must grow to clear the floor. A large operation tracks the binding roofline term, and the question is whether the layer is compute-bound or bandwidth-bound, which decides whether shape or streaming is the control.

Fusion removes floors and intermediate round-trips. A network run as separate dispatches pays the floor on every one and copies each intermediate back to the host and forward again; the same network fused into one program pays the floor once and keeps the intermediates resident. The model accounts for this: fusing operations removes their separate t_0 terms and the operand bytes of the eliminated round-trips, which is why a network is compiled as one program rather than a sequence of small ones.

18.4 What the optimizer does

The optimizer ranks equivalent rewrites of a graph using an op-agnostic cost estimate and a deterministic autotuner. The cost estimate is the analytic roofline-plus-floor number, applied uniformly to every operation rather than holding a hand-tuned constant per operation type. The autotuner enumerates rewrites that compute the same result, estimates each, caches the measured outcomes, and selects the lowest deterministically, so the same graph yields the same choice on every run.

The rewrites preserve accuracy by default. A route rewrite that decomposes attention into a fused form is selected only when it computes the same result to within tolerance, and a rewrite that trades precision, such as an integer-quantized weight stream, is taken only under an explicit tolerance and a speedup margin. The default never changes the numerical result of the graph.

The reported gains follow the roofline. A fused attention route rewrite is 3.7 to 5.3 times faster on attention blocks and on a full vision transformer at cosine similarity 1.0, because it removes per-dispatch floors and intermediate round-trips rather than changing the arithmetic. The optimizer's gate is the test corpus, which holds the cross-chip latency table as a regression so a change to the model cannot silently move an estimate.

18.5 Estimating latency and tuning a graph

The estimate locates a graph against any target's roofline before dispatch, and the autotuner selects the lowest-latency equivalent rewrite under an accuracy bound. The estimate returns the three-stage latency for a named target with no device in hand, and the tune step ranks accuracy-preserving rewrites by that same estimate. Listing 18.2 estimates a graph for a named target with no hardware in hand, then enumerates equivalent rewrites and keeps the lowest-latency one that holds the accuracy tolerance.

Listing 18.2. Estimating a graph's latency for a named target with no hardware, then ranking accuracy-preserving rewrites by the same estimate.

```
# Estimate latency for a named target with no hardware in hand, then tune.
# The three-stage estimate: compute_time = cycles / f, memory_time = bytes / B,
# latency = max(compute_time, memory_time) + dispatch_floor (t0).

target = H17s                                  # per-chip f, B, t0 read from the hardware table

function estimate(graph, target):
    cycles = sum over ops of compute_cycles(op)        # stage 1: per-op cycle count
    bytes  = sum over ops of operand_bytes(op)
    compute_time = cycles / f(target)                  # stage 2: roofline
    memory_time  = bytes  / B(target)
    latency = max(compute_time, memory_time) + t0(target)   # stage 3: add dispatch floor
    if   latency near t0(target):       bound = "dispatch"
    elif memory_time > compute_time:    bound = "bandwidth"
    else:                               bound = "compute"
    return latency, bound

latency, bound = estimate(G, target)
# If bound == "dispatch": batch or fuse until the compute term clears the floor;
# no faster array helps a layer pinned to the dispatch floor.

# Tune: enumerate equivalent rewrites, estimate each, keep the lowest that holds the tolerance.
best := G                                      # the unmodified graph
best_latency := latency
for each rewrite R that computes the same result as G:
    if max_abs_difference(R, G) <= tolerance:  # accuracy-preserving only (tolerance = 0 by default)
        latency_R, _ = estimate(R, target)
        if latency_R < best_latency:
            best := R
            best_latency := latency_R
# A fused-attention route rewrite is 3.7 to 5.3 times faster on attention blocks at cosine 1.0.
# Note: attention has no analytic cost form, so its estimate is absent and must be timed on device.
```

A graph whose estimate prints `dispatch` is batched or fused until the compute term clears the floor, since no faster array helps a layer pinned to the 0.23 ms floor on the M1.

18.6 Two cost models and their coefficients

There are two cost models in the toolchain, and they do different jobs. The model above is the intra-engine analytic model the compiler holds, the cycles-to-roofline-to-wall-time chain, fit to silicon for a per-chip latency estimate. A second model decides backend placement: whether each operation runs on the engine, central processor, or graphics processor, and it is in the segmenter rather than the engine compiler. The placement model costs an operation as the same roofline form, `max(flops/peak, bytes/bandwidth) + launch`, but with coarse abstract anchors rather than silicon-fit ones.

A live read of the compiler shows how the intra-engine model scores one layer. A layer's cycles are the larger of the core compute time and the direct-memory-access and L2 transfer time, plus the dependency stalls, computed in `ZinEnginePerf::ComputeRunTime`. The layer total is the execute cycles plus the overhead cycles. The tiling choice is the argument that minimizes the split cost against the unsplit cost, searched once and cached. `CalculateExeCycles` reads a precomputed double at layer offset `+0x1e0` and saturates at 65535, the width of the 16-bit task-descriptor field.

This intra-engine per-layer cost model is distinct from the placement cost model that decides engine versus central processor or graphics processor. The two answer different questions at different layers: the per-layer model sizes and tiles work already bound for the engine, while the placement model orders the three backends.

The live measurement validates the analytic roofline. The measured full-call floor of about 190 microseconds is about 16 percent below the frontend's analytic per-chip anchor of 220 microseconds, inside the model's stated plus-or-minus 17 percent fit.

The placement-model anchors are per-backend, not per-chip. Table 18.2 gives the abstract roofline peak and bandwidth each backend is charged against, read from the segmenter cost functions.

Table 18.2. The abstract per-backend roofline anchors the placement model uses to order the three backends, read from the segmenter cost functions.

Backend	Peak (GFLOP/s)	Bandwidth (GB/s)
Engine	800	50
Graphics processor	120	40
Central processor	20	10

These anchors are deliberately coarse. The 800 GFLOP/s engine peak is about four times under the M1 silicon fit of 3.25 fp16 TFLOP/s, and the 50 GB/s engine bandwidth is above the 9 GB/s effective fit, so the two errors run in opposite directions and the net placement order survives. The job of these numbers is to separate the engine from the graphics processor by 6.7 times and from the central processor by 40 times, a separation that holds under any single miscalibration. Absolute latency accuracy is not their responsibility; it is in a learned per-operation layer described next.

The placement model also charges a launch cost on every segment and a transfer cost on every backend crossing, which is why the segmenter prefers long single-backend runs, as table 18.3 gives in the model's relative units.

Table 18.3. The fixed launch and transfer penalties the placement model charges, in the model's relative units.

Penalty	To the engine	To another backend
Launch cost per segment	0.05	0.10
Transfer cost across backends	0.09	0.23

A transfer between two operations on the same backend costs zero, so a long run kept on the engine pays neither the per-segment launch nor the per-crossing transfer.

18.7 Learned per-operation leaves

Above the coarse roofline the placement model holds a learned per-operation layer: 322 regression trees, one set per platform class and compute path, that map the coarse roofline ratio onto a calibrated cost in nanoseconds. The trees split on a normalized feature built from the operation flops and bytes, and the leaf is the predicted cost. The engine trees are the small roofline-form ones, and the cost they return is monotone in operation cost: elementwise below pooling below linear below convolution below transposed convolution. Table 18.4 gives representative engine cost-tree leaves, spanning a constant-leaf elementwise operation to the largest transposed-convolution leaf.

Table 18.4. Representative learned engine cost-tree leaves, in the model's nanosecond output.

Operation (engine)	Tree form	Leaf cost
`relu`	constant leaf	50 ns
`add`	constant leaf	26.5 to 36 ns
`mul`	constant leaf	40 to 80 ns

Operation (engine)	Tree form	Leaf cost
`reduce_sum`	constant leaf	22 ns
`max_pool`	single-threshold leaf	26 to 29 ns
`conv` (main tree)	seven-split learned tree	417.7 ns to 8609.6 ns
`matmul`	three-threshold tree	116.9 ns to 66408.3 ns
`linear`	three-threshold tree	67 ns to 14.7 microseconds
`conv_transpose`	three-threshold tree	3.5 to 37.8 microseconds

Three operations have no engine tree at all, because the placement model never puts them on the engine: gather, the recurrent cell, and the argument-maximum reduction are central-processor or graphics-processor placements only.

The learned leaves and the silicon-fit analytic anchors reach calibrated accuracy by different routes. The analytic model above replaces the coarse anchors with silicon-fit ones and brings all five reference convolutions within plus or minus 17 percent. The placement model keeps the coarse anchors as a binning feature and bolts a learned per-operation tree on top. The convolution case shows both against silicon. Table 18.5 sets the coarse placement roofline, the silicon-anchored analytic estimate, and the measured M1 latency against each other for four convolutions.

Table 18.5. Coarse roofline, anchored estimate, and measured M1 latency for four convolutions.

Convolution	Coarse roofline only	Analytic, h13 anchors	Measured, M1
3x3 C256 to 256 at 28	1156 microseconds	505 microseconds	507 microseconds
1x1 C512 to 512 at 32	671 microseconds	511 microseconds	444 microseconds
1x1 C1024 to 1024 at 16	671 microseconds	569 microseconds	686 microseconds
1x1 C2048 to 2048 at 8	671 microseconds	1210 microseconds	1047 microseconds

The coarse roofline cannot even order these four: it ties the three one-by-one convolutions and misclassifies the memory-bound deep-narrow C2048 at 8 as compute-bound, predicting it faster than it runs. The anchored analytic estimate reproduces the non-obvious silicon result that the deep-narrow C2048 at 8, with about a twentieth the multiply-accumulate count, runs about twice as slowly as the compute-bound 3x3 C256 at 28. This is because the former is bandwidth-bound at arithmetic intensity 60 and the latter is compute-bound at 466.

18.8 Fidelity the estimate actually has

The plus-or-minus 17 percent fit is a property of the five reference convolutions the M1 anchors were tuned against, not of the estimate at large. A broad sweep of 68 graphs across eight operation families, each estimated by the cost model for the `h13` target and timed against the on-device per-call latency, locates the median absolute error at about 31 percent. Only 11 of the 68 shapes are inside plus or minus 17 percent. The estimate is sound as an ordinal placement tool rather than as an absolute-latency oracle: it identifies the binding roofline term and orders shapes correctly, so the backend choice and the relative ranking the optimizer needs survive the error. The estimate reads as an approximate figure and a ranking, not a calibrated wall-time prediction outside the convolution shapes it was fit on. The error is directional and concentrated in the bandwidth-bound regime, where the 9.0 GB/s anchor undershoots the roughly 40 GB/s effective rate the engine sustains and the estimate over-predicts the large-weight shapes by several times.

Attention has no estimate at all. The cost model returns no estimate for any scaled-dot-product-attention graph, because attention compiles to a segmented plan whose native sub-programs the analytic model has no cost form for. This is a coverage gap rather than an accuracy figure: the estimate is absent for attention, so such a graph must be timed on device.

19 Pitfalls and limits

SUMMARY

Five direct-path failure modes are silent or target-specific. An M1 last-axis-offset slice saturates above 4094, dynamic-weight convolution fails to compile above batch one, the saturating slice kernel is keyed to the M1 target, repeated failed compiles in quick succession can stall the shared compile service, and a true four-character-code image input is not reachable on the direct path. Each is a property of the private compiler or its data tables, so the defense is a build-time rule rather than a runtime check. Pace compiles after a failure, roughly 15 seconds apart, so a run of failed compiles does not stall the service.

A few of the engine's failure modes are silent, target-specific, or affect the shared compile service. Five of them are what a developer building on the direct path encounters, and each has a concrete trigger and a stated avoidance.

19.1 Slice saturation hazard on the M1

A last-axis slice with a nonzero start offset is not a free descriptor edit on the M1. On that generation the offset copy routes through a crop-DMA that stores values in a fixed-point format with four fractional bits, an implied scale of 16. The stored value is the input times 16, and the storage port clamps at the fp16 maximum, so any element whose magnitude exceeds the fp16 maximum divided by 16 overflows to infinity.

The threshold is exact and given by

$$V_{\max} = \frac{65504}{16} = 4094$$

A slice element at 4094 survives, an element at 4100 becomes infinity, and a control value of 60000 that never enters the offset copy stays finite. The hazard is limited and easy to miss: it occurs only on the M1 family, only on a last-axis slice, only with a nonzero start offset, and only when a value on that axis can exceed 4094. On this one axis a cross-chip route change turns a finite number into an infinity rather than moving a result by a unit in the last place. The A14 generation saturates on the same slice; the A15 generation and every part above it take a plain fp16 route and do not saturate.

The consequence shows up in training. A convolution weight gradient scaled up by a large loss-scale factor can push values past 4094 on the M1 and silently produce infinities. Keep the values on a width-offset slice under the bound, which for the training case means capping the loss-scale on M1 targets, and prefer a zero start offset where the layout allows, since a zero last-axis begin avoids the offset-DMA path entirely. Listing 19.1 checks at build time that a nonzero width-offset slice on the M1 stays under the 4094 saturation bound.

Listing 19.1. A build-time check that a nonzero width-offset slice on the M1 stays under the 4094 saturation bound of the times-16 crop datapath.

```
V_MAX = 65504 / 16   # = 4094.0; the fp16 maximum divided by the Q.4 gain of 16

def slice_offset_is_safe(x, begin_w, on_m1):
    # A nonzero width-offset slice on the M1 routes through the times-16 crop-DMA.
    # A zero begin avoids the offset-DMA path entirely; on A15 and above the
    # route is plain fp16 and never saturates.
    if not on_m1 or begin_w == 0:
        return True
    return float(abs(x).max()) <= V_MAX   # 4094 survives, 4100 -> inf
```

19.2 Dynamic-weight convolution above batch one

A convolution whose weight is a supplied runtime tensor rather than an embedded constant is an engine capability. At a batch of one it compiles and runs and matches a reference convolution to a cosine of 1.0000. At a batch of two or more the same program crashes the compiler service: the helper process dies mid-compile and the error reads as a lost helper application rather than a plain rejection of an unsupported operation.

The crash is specific to the dynamic-kernel path combined with a batch above one. A constant-weight convolution at the same batch compiles without issue, and batches of 64 are routine on that path. The reproduction is stable across fresh restarts of the compile daemon: batch one passes, batch two fails, every time.

The avoidance follows the capability boundary. Single-image dynamic-weight convolution is usable for hypernetwork and per-sample-kernel inference where the batch is one. Batched trainable convolution must take a different route: the dynamic-kernel path is closed above batch one, so a build that needs it should reject a dynamic-weight convolution at batch two or more before the program reaches the compiler. Listing 19.2 rejects a dynamic-weight convolution at batch two or more before the program reaches the compiler service that the path crashes.

Listing 19.2. A build-time guard that rejects a dynamic-weight convolution at batch two or more before it reaches the compiler service that the path crashes.

```
def guard_dynamic_weight_conv(weight_is_runtime_tensor, batch_n):
    # A const-weight conv shares one kernel across the batch; a dynamic kernel
    # must be re-split per batch element, and that path is broken on the M1.
    # Reject at build time so the program never reaches the compiler service.
    if weight_is_runtime_tensor and batch_n >= 2:
        raise ValueError("dynamic-weight convolution is closed above batch one")
```

19.3 Saturating slice kernel is target-keyed

The saturating slice kernel is keyed to the target, not to the compiler build. The same compiler binary emits a saturating slice kernel on the M1 target and a non-saturating one on the M5 target, because the slice lowering is specialized per hardware family rather than parameterized at runtime.

Inside the single binary the slice conversion is a C++ template instantiated once per family, so the same intermediate-language slice lowers through a different compiled converter for the M1 than for the M5. Per-target hardware-abstraction fields then drive the DMA source path: a width granule used as both a divisor and a stride multiplier, and a set of patch-width clamps that resolve to a power-of-two granule. On the M1 the family converter and the format selection pick the times-16 fixed-point DMA format for a last-axis-offset copy; on the M5 the same copy stays in plain fp16. The trigger is exactly the M1 family plus a last-axis slice

with a nonzero start offset, so the avoidance is the same as for the saturation hazard and the defense belongs in the per-target build path.

19.4 Pacing compiles after a failure

A failed compile is not free of side effects on the shared compile service. A compile that fails restarts the service, which takes a few seconds to come back, and failures that keep arriving faster than the service can restart between them keep it from making progress, so unrelated compiles slow down until the failures stop. The effect is a function of how fast failures arrive, not how many occur: failures spaced out past the restart interval cause no degradation at all. On detecting a failed compile, wait at least one restart interval, roughly 15 seconds, before the next compile, so a burst of failures cannot accumulate. No hard failure-count cap is needed.

19.5 A single slow compile is a separate failure mode

A slow compile is not the same failure as a run of failed compiles, and the back-off rule that handles repeated failures does not prevent it. A trainable convolution compiled over a large batch has a compile cost that scales with the batch, because the image-to-column tensors grow with the batch and the tiling and partition cost grows with them. On the M1 the same backward convolution graph compiles in about 1.9 seconds at a batch of 4, about 35 seconds at a batch of 64, about 79 seconds at a batch of 128, and never finishes at a batch of 1000. Shrinking the spatial size does not help, since the cost tracks the batch and not the feature-map size. The large-batch trainable-convolution stall is the firmware-overload throttle of chapter 29 in action: a submission that overruns the firmware command queue is throttled rather than dispatched, so the fix is to cap the batch or split the program, not to wait it out. The defense is to mini-batch real training at a modest batch per step rather than compile one full-batch graph, and the same graph compiles without issue on the M5.

A second slow-compile path is a combinatorial subgraph search. The compiler clusters a graph by a memoized search over cut points that collapses linear chains safely but is exponential in the width of parallel branches that reconverge into one consumer. The search has no iteration cap, but an internal time budget abandons it and falls back to a cheaper partition, which on the M1 plateaus the worst measured fan-in at about 9.2 seconds rather than letting it run unbounded. A single cluster with roughly twelve or more independent branches reconverging into one consumer is worth flagging at build time, since the 9-second per-cluster cost is itself worth avoiding and the budget bail is not guaranteed across compiler versions.

19.6 Direct four-character-code image input

A direct image input that declares a true four-character-code interchange format, so the engine reads a camera or video surface with no host-side conversion, is a no-go on the unentitled direct path. The capability is real and the syntax is fully reachable: the pixel-buffer input type, format enum, grammar, and type rules all parse and type-check, and the program reaches the backend. The failure is at backend lowering: the program does not compile on the direct route.

The lowering needs setup that only the entitled model-input route supplies, the four-character-code format descriptor backed by a surface, which the unentitled path lacks. The supported and terminal form on the unentitled path is the uint8 image input, which dequantizes in the graph and saves the host-side conversion at the input. It produces output byte-identical to a host conversion, saves roughly two milliseconds per frame at 1080p, and avoids the unsupported operation entirely.

19.7 A build-time pitfall check

The pitfalls share one defense: gate the graph before it reaches the compiler, keyed on the target family and the offending shape. The estimate holds the per-target hazard flags, so a saturating slice, batched dynamic-weight convolution, or four-character-code input is rejected at build time rather than at the silicon. Listing 19.3 scans a graph for all five known pitfalls, keyed on the target family and the offending shape, and stops the build before a flagged graph reaches the compiler.

Listing 19.3. Scanning a graph for the five direct-path pitfalls and stopping the build before a flagged graph reaches the compiler.

```
# Scan the graph for the known pitfalls and reject or rewrite before compiling, target = H13.
# Each hazard is keyed on the target family plus an offending shape.

target = H13
on_m1  = (family(target) is M1 or family(target) is M2)   # the saturating-slice generations
V_MAX  = 65504 / 16                                        # = 4094: fp16 max / 16 (Q.4 gain)

for each op in graph G:

    # Pitfall 1: last-axis slice saturation on the M1 (finite turns to infinity above 4094)
    if op is slice and last_axis_begin_offset(op) != 0 and on_m1:
        if max_abs_value_on_axis(op) > V_MAX:
            reject(op, "width-offset slice can saturate above 4094 on M1")
        # rewrite where possible: use a zero start offset to avoid the offset-DMA path entirely

    # Pitfall 2: dynamic-weight convolution above batch one (fails to compile)
    if op is conv and weight_is_runtime_tensor(op) and batch(op) >= 2:
        reject(op, "dynamic-weight conv needs batch one; use batch one or another route")

    # Pitfall 3: an unsupported op (validator may accept it, code generator rejects on this target)
    if not runs_on_target(op, target):
        reject(op, "op does not lower on this target; decompose or choose a later family")

    # Pitfall 4: true four-char-code image input (does not lower on the direct path, unentitled)
    if op is image_input and format(op) is fourcc:
        reject(op, "true 4CC input faults at lowering; use the uint8 in-graph image input instead")

if any op was rejected:
    stop before compiling                                  # never let a flagged graph compile

program P = compile(G, target)                             # reached only when the graph is clean

# Pitfall 5 is a rate effect, not a graph shape: after any compile FAILURE, back off at least one
# restart interval (~15 s) before the next compile, so a burst does not stall the service.
on compile_failure:
    wait(restart_interval)                                 # about 15 seconds
```

19.8 Symbolic shapes are not reachable on the direct path

The engine compiler holds a full symbolic-shape system: it accepts a dimension written as an unknown, type-checks two dynamic operands for matching symbolic expressions, and enforces affine constraints on symbolic dimensions. The silicon can thus run a program that accepts a variable sequence length on one compile in principle. The direct compile path does not reach it. A relu over a tensor with an unknown dimension parses, but the compile fails with an unsupported-operation error, because the symbolic-shape machinery is behind the entitled model-input route that lowers enumerated or range shapes onto symbolic engine programs.

The direct path thus compiles one program per concrete shape, and there is no one-compile-many-shapes form. For a variable-length workload the options are to pad to a fixed maximum length and compile once, to bucket a small set of lengths and dispatch the nearest with the compile cache making repeats free, or to recompile

per length under the per-compile cost. The same limit closes the native dynamic-slice and dynamic-offset key-value-cache primitive on the direct path, so the host manages cache state with fixed-shape windows.

19.9 Reference: the five direct-path pitfalls

Table 19.1 collects the five direct-path pitfalls, each with its symptom, its cause, and the workaround.

Table 19.1. The five direct-path pitfalls a developer encounters, each with its symptom, its cause, and the workaround.

Pitfall	Symptom	Cause	Workaround
Slice saturation (M1)	Last-axis-offset slice silently produces infinities above 4094	Offset copy routes through a Q.4 times-16 DMA that clamps at the fp16 maximum	Keep width-offset values under 4094; cap loss-scale on M1; prefer a zero start offset
Dynamic-weight conv at batch ≥ 2	Fails to compile	Dynamic-kernel path is closed above batch one	Use it only at batch one; reject batch ≥ 2 at build time; take a different batched route
Target-keyed saturating slice kernel	Same source saturates on M1, non-saturating on M5	Slice lowering is template-specialized per family; M1 picks the times-16 DMA format	Treat the slice route as M1-specific in the per-target build path
Failed compiles in quick succession	Unrelated compiles slow while failures keep arriving	Failures arriving faster than the shared service can restart between them	Pace compiles after a failure by at least one restart interval (~15 s)
Direct four-character-code image input	Does not lower on the direct path	The entitled surface descriptor the lowering needs is unavailable unentitled	Use the supported uint8 in-graph image input, the terminal unentitled form

19.10 Reference: the compile-time and capability limits

Beyond the five silent or service-level pitfalls, three further limits reject or stall a build rather than corrupt a result, which table 19.2 gives with each symptom, cause, and workaround.

Table 19.2. The compile-time and capability limits that reject or stall a build, distinct from the silent direct-path pitfalls above.

Limit	Symptom	Cause	Workaround
Large-batch trainable convolution	Single compile runs for minutes and hangs on the M1	Image-to-column tiling and partition cost scales with the batch	Mini-batch at a modest batch per step; do not compile one full-batch graph
Wide reconvergent fan-in	A single cluster compile climbs super-linearly and plateaus near 9 seconds	Memoized subgraph cut-search is exponential in parallel-branch width; an internal time budget bails	Flag a cluster with roughly twelve or more branches reconverging into one consumer
Symbolic shape on the direct path	A tensor with an unknown dimension parses but fails to compile	The symbolic-shape system is reachable only through the entitled model-input route	Compile per concrete shape; pad, bucket, or recompile per length

19.11 Reference: the firmware fault surface

When a fault does reach the silicon rather than the compiler, the firmware classifies it into one of a few responses, and the response decides whether a workload is rejected, silently dropped, or recovered through a reset. Table 19.3 gives each fault class, its detection mechanism, and whether the firmware rejects, drops, or recovers through a reset.

Table 19.3. The firmware fault surface, with each fault class, its detection mechanism, and whether the firmware rejects, drops, or recovers through a reset.

Fault class	Mechanism	Firmware response
Command integrity	Magic-word, checksum, padding, and 32-bit-address asserts on each command	Reject the command and return an error status to the host
Program section	Section bounds and overlap audit at load	Refuse to load with a sanity-check failure
Runtime mis-target	A call to a torn-down process or a wrong cache state	Drop and count, no error returned
Assertion or exception	A runtime-assert or a processor exception	Register dump, coredump, and reset
Uncorrectable cache error	The L2 controller error block and its overflow state	Coredump and reset, not a soft retry
Task-queue watchdog	A queue that will not quiesce within two seconds	Abort the queue
Host-side timeout	No completion within the deadline	Cancel outstanding commands and re-initialize the firmware

The reject-versus-drop split is explicit. Integrity and section and argument validation reject a command and return an error, so the failure is visible to the host. The firmware instead silently drops runtime mis-targeting, a call to a process that has been torn down or a trigger on a cache handle in the wrong state, and counts it in a soft-failure telemetry counter. A misbehaving client thus shows up in the drop counters well before it faults, and a host-visible restart counter increments each time the firmware re-initializes after a fault.

Interlude. Below the API: how the engine works

SUMMARY

A reader who only dispatches work to the engine can stop at the end of Part V; the back half is the mechanism beneath the programming surface, for reasoning about a numeric result, predicting an unmeasured chip, or debugging a compile failure.

Parts I through V describe the engine as a developer uses it: what it computes, how to reach it, how it performs, what it suits, and how to fit work to it. That account stops at the programming surface. The rest of the guide goes beneath that surface, to the chip and the system around it.

The back half is a reference for readers who need the mechanism rather than the interface. It is written at a lower level than the front half, and shows the structures, registers, and command protocol directly.

Table I.1 shows how the four back-half parts and the back matter divide, each with its subject and the reader it serves.

Table I.1. The back-half parts and the back matter, each with its subject and the reader it serves.

Part	Subject	For the reader who needs
VI. The Silicon	The datapath and MAC geometry, and the memory hierarchy	The mechanism behind the numerics and the roofline
VII. The Toolchain and Encoding	The compiler, the program and container format, the HAL and its gates, the compression pipeline, and direct netplist authoring	The mechanism behind compilation, the program format, capability gating, and compression
VIII. System Internals	The kernel driver and its ABI, the address-translation unit, the firmware, the host-to-firmware command protocol, power and thermal, security and isolation, and telemetry	The path from a host call to the silicon, and what is and is not observable
IX. Cross-Silicon Reference	The full target set, per-family code generation, and the upper tier	To compile for and reason about chips other than the one in hand
Back matter	The methodology, the open questions, and the reference appendices	To trust the numbers and to find a specific value

Part VI

The Silicon

20 Datapath and MAC geometry

The multiply array, the output-channel groups, and the per-core tiling.

21 Memory hierarchy

The on-chip working set, the 2 MB threshold, and the streaming boundary.

20 Datapath and MAC geometry

SUMMARY

The multiply-accumulate array is four cores on the M1, each with an eight-deep accumulator file, reading every geometry constant from a per-chip hardware-abstraction table. The lane width emits eight output channels per cycle in the int8 fast path and four in the default fp16 path, and output channels are the dimension that splits across cores and across accumulator passes. Each core reduces through a radix-4 tree of fp16-rounded tiles into one wide running accumulator of fp32 class, rounding to fp16 only at the output port. The firmware has no convolution, Winograd, or systolic code: it holds the real-time kernel and the tile, kernel, and output direct-memory-access plumbing, and nothing else.

The array is a set of cores, accumulators, and lanes whose dimensions are constants in the compiler and in a per-chip hardware-abstraction table. The compiler supplies it by re-basing direct-memory-access engines per tile. Table 20.1 gives those dimensions and the roofline that follows from them, with each constant's hardware-abstraction-table offset and the scope over which it holds.

20.1 Geometry constants

Table 20.1. The multiply-accumulate array geometry constants, with hardware-abstraction-table offset, M1 value, and the scope over which each holds.

quantity	table offset	M1 value	scope
core count	`0x238`	4	per die variant
accumulator budget per core	`0x0c`	8	all chips
accumulator-split granule	`0x230`	64	all chips
performance cycle divisor	`0x228`	64	A12 and later
output channels per cycle, fp16	accessor table	4	architectural
output channels per cycle, int8	accessor table	8	architectural
patch-width floor, log2	`0x400`	4	M1
patch-width cap, log2	`0x410`	9	M1
working-set cap	`0x1b8`	2 MB	M1

The compiler's performance model reads the array geometry from these fields in the per-chip hardware-abstraction table. The die, not the generation, keys the core count. The decoded four-core figure on the M1 counts physical compute sets, which the per-core power-rail step confirms below. Apple's published per-chip figure, the 16-core count the I/O registry also reports, is a different quantity [AppleANE]. The M1 has four cores; the M5 generation has sixteen; small reference variants have one. The accumulator budget of eight, uniform across every chip, is the depth of the accumulator file per core in work units. An operation fits single-buffered when its accumulator demand is at most eight, and double-buffered when twice its demand is at most eight, so double-buffering holds up to a demand of four.

20.2 Output channels per cycle and the lane width

Two accessors of the performance model set the lane width, listed in table 20.2 with what each returns and the input that selects the value.

Table 20.2. The two lane-width accessors of the performance model, with what each returns and the input that selects the value.

accessor	returns	selected by
`GetNumOutputChannelsPerCycle`	8 (int8 fast path), 4 (fp16 default), 2 or 1 (narrow modes)	data type and mode
`GetNumOutputChannelsPerAccumulator`	8 / 4 / 2 / 1 (table below)	source-patch size

The second accessor selects among four per-accumulator channel counts by source-patch size, which table 20.3 maps to the hardware-abstraction-table offset that holds each value.

Table 20.3. Output channels per physical accumulator slot, the hardware-abstraction-table offset holding each value, and the source-patch size that selects it.

HAL offset	output channels per accumulator	selected when
`0x3a8`	8	tiny-source mode (small-source-mode field reads 2)
`0x3b0`	4	small-source mode (small-source-mode field reads 1)
`0x3b8`	2	default, format 3 / fp16-packed
`0x3c0`	1	default, non-format-3 and not half-work-unit

The four values are not packed into one accessor argument: each is its own field in the hardware-abstraction table at offsets `0x3a8`, `0x3b0`, `0x3b8`, and `0x3c0`, and `GetNumOutputChannelsPerAccumulator` returns the one the operation's source-patch mode selects. The values are uniform across every chip in the set, so the multiplexing law is architectural, not per-die. The source-patch mode is set by `ComputeSmallSourceMode`, which classifies an operation as default, small, or tiny from the output tensor dimensions against the kernel: tiny mode requires a source of at least 2 that fits the field at `0x338`, which is 4 on the M1.

The lane width is the column dimension of the systolic tile. `GetNumOutputChannelsPerCycle` returns eight in the int8 fast path and four in the default fp16 path, degrading to two or one for narrow modes. One core streams an input patch and produces up to eight output channels per cycle in double-int8 mode and four output channels per cycle in fp16. `GetNumOutputChannelsPerAccumulator` sets how many output channels time-share one physical accumulator slot. A tiny input patch underuses the array spatially, so the compiler packs eight output channels onto one accumulator to keep the multipliers busy. A large patch lets each output channel keep its own accumulator.

20.3 Radix-4 reduction and wide accumulator

The reduction that supplies one accumulator runs in two stages. The first stage groups input lanes into tiles of four, each rounded to fp16. Those fp16-rounded tiles supply one wide running accumulator of fp32 class, and the result rounds to fp16 only at the output port.

The fan-in of four is measured, not inferred from the table. A reduction of `[+B, -B, +1]` triples, repeated sixteen times so that the true sum is sixteen, returns sixteen survivors below a partial magnitude of 4096 and saturates to exactly four survivors for every `B` at or above 4096. The saturation count of four is the radix-4 tile signature: once a tile holds a partial at or above 4096, the unit increments sharing that tile fall below half of the fp16 spacing and vanish, and the three-element triple period beats against the four-lane tile to leave four unaffected lanes. The threshold is at 4096 because that is where the fp16 spacing first reaches

four. The result is layout-independent: the `[+B, -B, +1]` and `[+B, +1, -B]` orderings return byte-identical values, so the hardware reduction order is a fixed lattice over lane index and not the source order.

The cross-tile accumulator is wider than fp16. A reduction of one value of 4096 followed by 1024 ones returns 5116, between the naive-fp16 result 4096 and the exact 5120, and sixteen unit increments all survive next to a partial of 8192 where a fp16 accumulator would drop most of them. The radix-4 first stage is consistent with the eight-work-unit accumulator file, since four input lanes plus margin fit the per-core budget.

20.4 Output-channel-group tiling

The compiler tiles output channels into output-channel groups sized to the accumulator file. `ComputeMaxOcg Size` derives the group size from the accumulator budget, the kernel-element count, and a per-format byte cap.

$$\mathrm{OCG} = \min\left(\mathrm{floor_pow2}\left(\frac{8}{k_W\, k_H\, k_D}\right),\ \mathrm{byte_cap}\right),$$

where the byte cap is 32, 16, or 8 bytes per kernel element depending on the weight format, read from the hardware-abstraction table at offset `0x388`, `0x390`, or `0x398`. In compiler terms, `ComputeMaxOcgSize` reads the accumulator budget at field `0x0c`, divides it by the kernel-element count, rounds down to a power of two, and clamps to the per-format byte cap, as listing 20.1 gives.

Listing 20.1. How the compiler computes the output-channel-group size for one pass from the accumulator budget, kernel-element count, and per-format byte cap.

```
# ComputeMaxOcgSize: output-channel-group size for one pass
acc_per_oc = GetNumOutputChannelsPerAccumulator(mode)   # 1, 2, 4, or 8
budget     = floor_pow2( HAL[0x0c] / (kW * kH * kD) )    # HAL[0x0c] = 8 accumulators
byte_cap   = HAL[ocg_cap_offset(format)] / (kW * kH * kD)   # 0x388/0x390/0x398 = 32/16/8 B/elem
OCG        = min( ComputeMaxOcg(budget, acc_per_oc), byte_cap )
```

A 1-by-1 convolution has a kernel-element count of one, so it admits a large group. A 3-by-3 convolution has a kernel-element count of nine, so its group is roughly nine times smaller and it needs more passes over the input. The compiler relieves that pressure by selecting Winograd for dense 3-by-3 stride-1 convolutions, since the transform cuts the effective kernel-element count.

Once the output-channel count exceeds what one pass holds, the array re-streams the input for a second pass. The number of passes is

$$\mathrm{OCG\ passes} = \left\lceil \frac{C_{out}}{\mathrm{OCG}} \right\rceil.$$

For a 1-by-1 fp16 convolution at a 32-by-32 spatial size, the per-layer slope shows a distinct super-linear jump in cost between an output-channel count of 192 and 256: a step from 20 microseconds to 61 microseconds per layer, a roughly threefold cost step for a 1.8-fold increase in arithmetic. That step is the pass count incrementing from one to two as the group cap is reached. A 3-by-3 convolution reaches the same threshold at fewer output channels per pass, matching the nine-fold accumulator pressure.

20.5 Winograd selection gate

The eligibility gate is `CanUseWinogradMode`, a fixed conjunction of conditions that all must hold, each given in table 20.4 with its meaning.

Table 20.4. The conditions of the Winograd eligibility gate, each of which must hold for the mode to be considered.

condition	meaning
`HAL[0x680]` bit 0 set	the per-chip Winograd-enable bit, present on the M1
kernel underlying type not 5, not 2, not unity	the format is Winograd-eligible and the weights are not 1-by-1 or identity
tensor format not 12	the input format class is eligible
one kernel axis equals 3	the transformed axis is fixed at 3, so 3-by-3 only
the orthogonal kernel axis is below 6	the input-tile width of the larger tile is 6
no third (depth) kernel extent	two-dimensional convolution
unit stride	input and output extents are consistent with stride 1
`OCG x kH x kW x kD x 2` clears the work threshold	enough work to amortize the transform

The axis-equals-3 plus orthogonal-axis-below-6 pair pins the supported tile set to two forms: $F(2 \times 2, 3 \times 3)$, with an output tile of 2 and an input tile of 4, and $F(4 \times 4, 3 \times 3)$, with an output tile of 4 and an input tile of 6. The work threshold on the term $\mathrm{OCG} \times k_H \times k_W \times k_D \times 2$ is precision-dependent: it is 32 for work-unit modes 1 and 2, 8 for a non-float kernel, and 16 for a float kernel. The higher float threshold is the compiler's guard against the precision cost of the $F(4 \times 4, 3 \times 3)$ transform, since the float convolution must clear a larger work bar before the transform is taken. There is no accumulator widening tied to Winograd: the precision safety is the eligibility and threshold gating, not a wider accumulator.

Eligibility is not selection. Even when the gate passes, Winograd is taken only when it beats both direct convolution and the accumulator-double-buffering path, decided by a cost-model comparison that reads the same accumulator and lane-width accessors. Winograd is rejected when the convolution uses unicast, when the weight is sparse (sparse weights keep their own datapath and skip zeros), or when double-buffering the accumulator file already saturates the array. The transform matrices themselves are not in the compiler: the compiler sets a single `winograd_mode` config field, and the input, filter, and output transforms run inside the engine datapath. The textbook $F(2 \times 2, 3 \times 3)$ and $F(4 \times 4, 3 \times 3)$ forms describe the behavior, but the coefficients are resident in hardware and not recoverable from the program.

20.6 Four-core granule

Output channels are the dimension the compiler splits across cores. `ZinMirNECoreAssignment` builds a strided round-robin map: core k owns output channels $\{k, k+N, k+2N, \ldots\}$ for a core count N. On the M1 with four cores, channel c is on core

$$\mathrm{core}(c) = c \bmod 4.$$

The active-core count is shape-driven, not user-selectable. The compiler sets it from the output-channel count and the group size, rounds it up to the next power of two, and writes it into the task descriptor. `GetNumNeededNEsNextPow2` sets the count, which listing 20.2 gives alongside the channel-to-core map and the pass count.

Listing 20.2. The strided round-robin channel-to-core map alongside the pass count and active-core count, both driven by the output-channel count and the group size.

```
# ZinMirNECoreAssignment: strided round-robin channel-to-core map (M1: num_nes = 4)
def core(c, num_nes):
    return c % num_nes                      # channel c -> core c mod 4 on the M1

# OCG passes and active-core count, both driven by C_out and the OCG size
def ocg_passes(c_out, ocg):
    return ceil(c_out / ocg)                # input re-streamed once per pass

def cores_needed(c_out, ocg, num_nes):      # num_nes = HAL[0x238] = 4 on the M1
    return min(num_nes, next_pow2(ceil(c_out / ocg)))
```

Table 20.5 reports the measured throughput as the active-core count rises from one to four, showing near-linear integer scaling of the single-core rate.

Table 20.5. Measured throughput as the active-core count rises from one to four.

output channels	cores lit	throughput, GMAC/s	ratio to one core
1	1	3.8	1.00
2	2	7.6	2.00
3	3	11.4	3.00
4	4	15.4	4.05

Driving a single 1-by-1 convolution at each output-channel count below the core count, in a dispatch loop pinned at a fixed rate by per-dispatch overhead, the sustained throughput rises in exact integer multiples of the single-core rate. Each added core does one more output channel's worth of multiply-accumulates in the same wall time. The power rail confirms the count independently: the engine draws a step of about 10 milliwatts per added core over an always-on floor of about 800 milliwatts, matching the firmware's one always-on base domain plus four independently power-gated compute sets, one per core.

A live capture of the compiler during a real M1 compile pins the geometry it costs against, which table 20.6 records with the active-engine count and the candidate split geometries it speculates.

Table 20.6. Costed-geometry constants captured live during an M1 compile, with the active-engine count and the candidate split geometries.

quantity	live M1 value
clusters costed	1
active engines costed (`GetTotalNumberOfActiveNEs`)	8
candidate split geometries speculated	1, 4, and 16 engines
kernel-memory budget	64 units
fixed per-layer overhead	44 cycles

The 8 active engines the compiler costs against differ from the 16 cores the I/O registry reports for the M1: the costed geometry is 8 active engines on 1 cluster, a model-internal figure rather than the registry core count or the four physical compute sets that the power rail confirms above. The model also holds a 64-unit kernel-memory budget that weight residency must fit under, and a 44-cycle fixed per-layer overhead that is added to every layer's execute cycles.

The full datapath reads top to bottom from the output-channel groups down through the four cores to each core's accumulator and its radix-4 reduction, as figure 20.1 draws it.

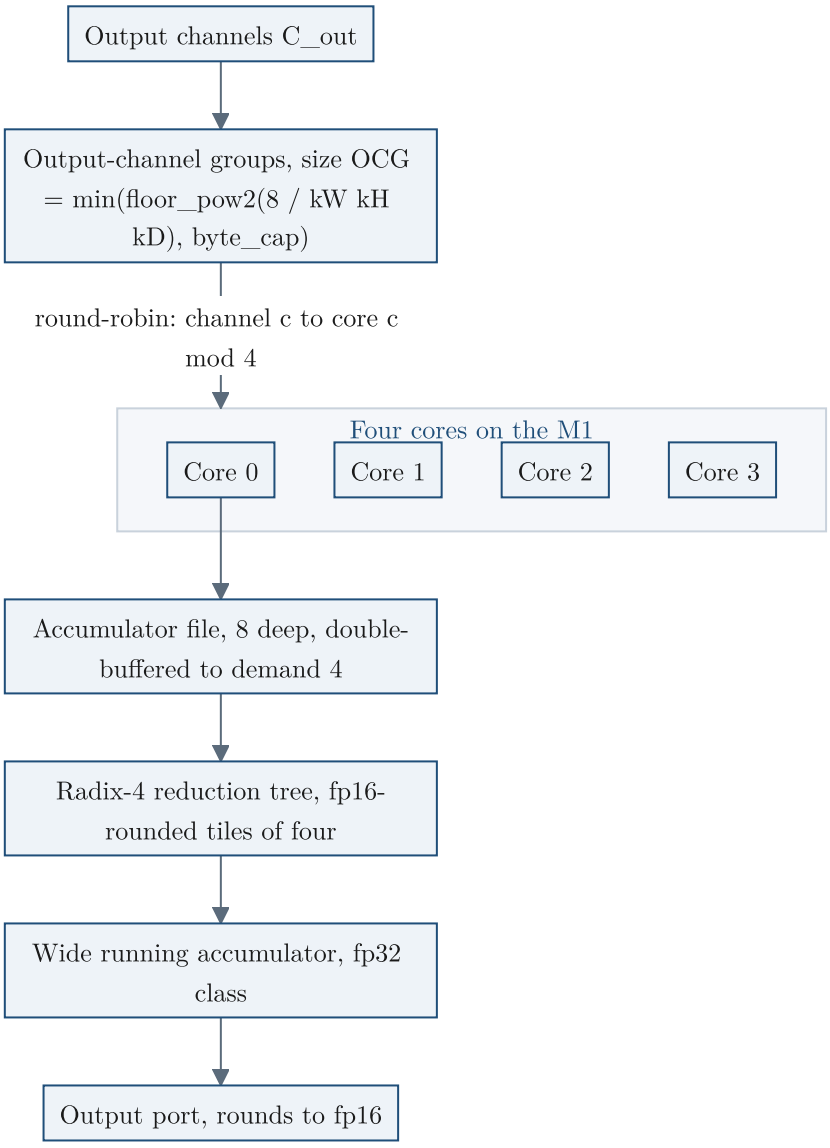

Figure 20.1. The multiply-accumulate datapath, with output-channel groups tiled across four cores, each core holding a wide accumulator fed by a radix-4 reduction tree.

20.7 Roofline from the geometry

Peak fp16 throughput is the product of the core count, lane width, and clock. Writing the lane width as output channels per cycle, the peak rate of multiply-accumulate operations is

$$P_{\mathrm{MAC}} = \mathrm{cores} \times \mathrm{lanes} \times f,$$

and the floating-point rate is twice that, since one multiply-accumulate is two floating-point operations. On the M1 the best fp16 mode is four cores times four output channels per cycle, and the int8 fast path doubles the lane width to eight. The measured all-four-cores-saturated rate for a large 1-by-1 convolution is about 3.48 trillion multiply-accumulates per second, about 7 fp16 TFLOP/s at the engine output: above the 4.8

fp16 TFLOP/s large-matmul saturating ceiling of chapter 9, because this convolution stays under the 2 MB working-set threshold and remains compute-bound, and below the overhead-subtracted roofline anchor of 12 fp16 TFLOP/s. The full-saturation power at that rate is about 4.7 watts, which is on the convolution power anchor and within 0.3 watts of the 4.4 watt large-matrix-multiply anchor, so the rail readings are calibrated.

The frontend does not reach that doubled lane. The int8 compile flag quantizes the weights and leaves the multiply-accumulate in fp16, so it halves the streamed weight bytes and never emits the eight-channel int8 compute path. The flag thus changes weight bandwidth, not compute rate. A 3-by-3 convolution at 256 input and output channels and a square matmul both run within a few percent of fp16. The int8 path reaches about 1.5 times faster only where the weight is large enough to stream from main memory, near a 4096-by-4096 weight at a batch of 256 or more.

The working-set cap closes the roofline at the memory edge. The largest single operand that stays on chip is 2 MB. Past that, the operand is tiled and streamed from main memory, which adds traffic and moves the workload onto the bandwidth side of the roofline.

20.8 Task descriptor that holds the geometry

The compiler does not address the array directly: it fills a per-partition task descriptor, a flat register image organized into seven register groups, then serializes each group into the address-value pairs the firmware writes. On the M1 the descriptor is the version-10 layout, one of fourteen versioned descriptor structs the compiler builds, selected by chip family. Table 20.7 lists the seven register groups of that layout, each with its register count, address base, and contents.

Table 20.7. The seven register groups of the version-10 task descriptor, each with its register count, address base, and contents.

group	register count	address base	contents
kernel and common	34	`0x5500`	kernel-DMA enable, format, stride, common config, task type, output transpose, network id
dimensions	19	none	input and output width, height, depth, channels, group count, broadcast, transpose, interleave, output-channel-group size
tile DMA	69	`0x4d00`	the three tile-DMA engines: enable, cache hints, base-address halves, row, channel, depth, group strides, format, wrap
elementwise, planar engine, padding	30	`0x4100`	elementwise and planar-engine config, padding mode, planar-engine index
L2 and texture	14	`0x4500`	L2 source and result config, texture mode, source dimensions
kernel format and op mode	11	`0x4900`	op mode, kernel alignment, sparse and palette flags, padding constant, bias and post-scale enables
L2 result	21	`0x5100`	L2-result base, strides, wrap, result format

The dimension group encodes the per-axis caps directly in its field widths. Input width, height, and depth are 15-bit fields masked `0x7fff`, while the channel fields are 17-bit, masked `0x1ffff`, so a channel axis reaches

131071 where a spatial axis stops at 32767. The output-channel-group size is a 3-bit field, and the active-core count is the `ActiveNE` field, a 3-bit value that records how many cores the task runs on, set from the shape by `GetNumNeededNEsNextPow2`.

A fused convolution folds its per-channel affine and its activation into four separate kernel-coefficient streams, each a distinct sub-buffer with its own base offset and a relocation slot the loader patches at load. Table 20.8 names the four streams, each with the relocation register the loader patches and the role it holds.

Table 20.8. The four kernel-coefficient streams a fused convolution emits, each with the relocation register the loader patches and the role it holds.

stream	relocation register	role
bias	`0x1554`	the per-channel additive offset
post-scale	`0x1558`	the per-channel output multiply, where a dequantize scale folds
palette lookup	`0x155c`	the palettized-weight lookup table
activation lookup	`0x1560`	the 33-segment piecewise-linear activation table

A convolution with batch normalization and a following activation thus does not become four operations: the scale and bias fold into the post-scale and bias streams, the activation runs as the activation-lookup stream, and one fused operation streams all four coefficient banks alongside its weights.

21 Memory hierarchy

SUMMARY

The engine holds its working data in one on-chip pool whose 2 MB size, at hardware-abstraction-table field `0x1b8` on the M1, is the dominant size limit: a layer whose largest single operand fits stays on-chip, and one that exceeds it is tiled and streamed from DRAM. The pool is interleaved across 64 banks at a 16-byte granule, with the bank index `floor(addr / 16) mod 64`, and a compile-time stride optimizer spreads accesses to avoid conflicts. The pool is a compiler-managed scratchpad, not a demand-filled cache: residency and stride are decided at compile time, so re-reference order does not change which operands are resident.

The numbers below come from the engine's hardware-abstraction table, indexed by field offset (for example `0x1b8`); quoted values are the M1/H13 entries.

21.1 2 MB working-set threshold

The matmul lowering shown in listing 21.1 rejects a resident right-hand operand and streams a tiled copy once that operand reaches the working-set bound.

Listing 21.1. Matmul lowering: reject the resident RHS and stream a tiled copy once the operand reaches the working-set bound.

```
/* ZinMirMatMul::LowerNEMatMulToNEConv */
if (GetTensorSizeInBytes(rhs) >= HAL[0x1b8]) {  /* HAL[0x1b8] = 2 MB on M1 */
    /* reject resident copy-cast; tile and stream the RHS from DRAM */
}
```

The largest single operand that stays in on-chip SRAM on the M1 is 2 MB, the value at field `0x1b8`, the maximum operand bytes for the SRAM working set. The same field holds 1 MB on the efficiency-class engine of the M9 part: the value is per-chip. The compiler names this field `MemCacheSize`, also `L2Size`, and exposes overrides for it through `fl2-size` and the related cache-mode and allocator options.

A layer runs entirely from the on-chip pool when its largest single operand, the larger of the input activation, weight, or output, fits within this bound. For a half-precision tensor the operand byte count is the element count times two. A `[1, 512, 32, 32]` activation is exactly 1 MB and stays resident. A `[1, 512, 64, 64]` activation is 4 MB and a `[4096, 4096]` linear weight is 32 MB, and both exceed the bound.

When the largest operand exceeds `0x1b8`, the compiler tiles it and streams the pieces from DRAM. The comparison is on the size of one tensor, not the sum of the operands. Past the bound, the streamed tile traffic adds DMA bytes that did not exist below it, so the arithmetic intensity of the layer, the ratio of multiply-accumulate work to bytes moved, falls. The design rule that follows is to keep the largest per-layer operand at or under 2 MB by tiling the batch or the spatial extent or by shrinking the channel count, so the layer stays on-chip.

The M5 softens this threshold. Its compiler tiles over the batch and the output dimensions together, so neither operand need be fully resident and the crossing is smooth in throughput rather than the sharp step the M1 shows. The working-set cost then appears in DRAM energy per operation, which bottoms near a 2 MB operand and rises beyond about 4 to 5 MB, around the M5 bound of 4.72 MB.

The measured throughput threshold is slightly above the table value, near 2.28 to 2.34 MB, because the tiler holds roughly 0.3 MB of double-buffer and alignment margin before it splits the weight. Sweeping a batched matrix multiply whose weight grows from 1.25 to 3.1 MB resolves the threshold directly, as table 21.1 shows with the smooth plateau below the bound and the sharp step just above it.

Table 21.1. Measured throughput of a batched matrix multiply as the weight crosses the 2 MB operand bound.

weight size	throughput, GFLOP/s
1.25 to 2.25 MB	smooth plateau, 700 to 850, gently rising
2.28 MB	992, the first step up
2.34 MB	1038, a sharp threshold, 185 higher in one 0.03 MB step
2.50 MB	1120
2.75 MB	1180
3.00 MB	1226
3.06 MB	1368

The throughput is higher above the threshold, not below it. Below the bound the whole weight is one resident operand that must fill before compute starts, a serialized stream into the array. Above the bound the weight is tiled and double-buffered, so the fill of tile $n+1$ overlaps compute on tile n and throughput climbs. The threshold is at the same 2.31 MB sweeping the grid up or down, so the boundary holds no memory state.

21.2 64-bank pool and bank function

The engine interleaves the on-chip pool across 64 banks at a 16-byte granule. The bank count is field `0x1c8` and equals 64; the interleave granule is field `0x1c0` and equals 16 bytes. An address maps to a bank by

$$\text{bank} = \left\lfloor \frac{\text{addr}}{16} \right\rfloor \bmod 64.$$

In compiler terms the two operands are the two table fields directly, as listing 21.2 computes the bank index from a byte address.

Listing 21.2. The on-chip bank index computed from a byte address using the interleave granule and the bank count read straight from the table.

```
/* SRAM/L2 bank index from a byte address */
unsigned bank(unsigned long addr) {
    return (addr / HAL[0x1c0]) % HAL[0x1c8];   /* (addr / 16) mod 64 */
}
```

The map is direct and modular: there is no second-level grouping above the 64 banks. The bank-conflict optimizer reads the same two table fields and chooses a row stride that spreads accesses across banks. Over candidate strides bounded by the maximum stride field `0x1f8`, it selects the stride whose per-bank cost is least, as listing 21.3 searches.

Listing 21.3. The bank-conflict optimizer searching candidate row strides and selecting the one with the least per-bank cost.

```
/* ZinMirBankConflictOptimizer::OptimizeL2StrideMinCost */
uVar1 = HAL[0x1c0];                       /* granule    = 16   */
uVar2 = HAL[0x1c8];                       /* bank count = 64   */
uVar4 = param_1 / uVar1;                  /* stride in 16-byte granules */
/* loop over candidate strides, bounded by HAL[0x1f8] (= 2 MB) */
    uVar5 = (uVar4 + uVar9) / uVar2;            /* granules / 64 */
    index = (uVar4 + uVar9) - uVar5 * uVar2;    /* granules mod 64 = bank index */
    fVar10 = cost_vector[index];                /* per-bank conflict cost */
    if (fVar10 < best) { best = fVar10; chosen = uVar8; }
```

The cost vector has exactly 64 float entries, one per bank, which fixes the modulus at 64. The modelled conflict depth for a row stride s over the 64 banks is

$$\text{requests} = \left\lceil \frac{\text{rows}}{64/g} \right\rceil, \qquad g = \gcd\!\left(\frac{s}{16},\ 64\right).$$

The conflict is worst when the granule stride is a high power of two, which drives g up, and is conflict-free when $g = 1$, which gives one request per bank. Every accepted stride is a multiple of the 16-byte granule, and the search never proposes a stride above the 2 MB ceiling in field `0x1f8`. The on-device throughput period that this structure produces is 64 bytes, which is 32 half-precision elements and four 16-byte granules; that period is the granule-quantized echo of the 64-bank interleave after the optimizer has set the stride, not the raw modulus itself.

The pool is a compiler-managed scratchpad, not a demand-filled cache. The compiler decides residency and stride at compile time and writes them into the program; there is no runtime replacement policy on the operand path, so the order in which weights are re-referenced does not change which ones are resident.

21.3 Resident vs shared output buffers

The output-buffer allocator in listing 21.4 decides whether a convolution gets a dedicated resident buffer or the shared default, gated by the residency-threshold field.

Listing 21.4. The output-buffer allocator deciding between a dedicated resident buffer and the shared default, gated by the residency threshold.

```
/* ZinIrLocalRegAlloc::AllocateOutputDMADefaultBuffer */
if (HAL[0x491] == 1 &&
    (footprint = HAL[0x1c0] * dst_buf_count,        /* 16 * count, in bytes */
     HAL[0x1f0] <= footprint && footprint != HAL[0x1f0])) {
    use_default_buffer;                  /* no dedicated conv_res_l2_buf */
} else {
    allocate "<layer>/conv_res_l2_buf";  /* dedicated resident buffer */
}
```

A convolution can be given a dedicated on-chip buffer that holds its output residency across the convolution so the result is not re-streamed. Field `0x1f0`, the resident-buffer threshold, decides whether that buffer is allocated by comparison against the convolution's output DMA footprint. On the M1 the field is 0, and the allocator skips the dedicated buffer when the footprint is strictly greater than the threshold.

With `0x1f0` equal to 0 on the M1, any buffer of at least one destination slot has a footprint greater than 0, so the condition is always true and the M1 always takes the default-buffer branch. The dedicated residency

buffer is thus never allocated on this chip. On the A16-class and M5 parts the same field is 262144, which is 256 KB, so only a convolution whose output DMA footprint exceeds 256 KB receives the dedicated buffer; on the A15 part it is 32768, which is 32 KB. The field rises with the larger on-chip pools of the newer parts.

The dedicated convolution buffer is not a separate memory. It is carved from the same on-chip pool as the 2 MB operand working set and the other DMA buffers, aligned to the same 16-byte granule, and its stride runs through the same 64-bank optimizer. Field `0x1f0` decides only whether a convolution's output residency gets its own named sub-allocation in that one pool. The kernel-coefficient store is a distinct on-chip budget: 64 KB on the M1, separate from the 2 MB operand working set.

The on-chip granules are finer than the page granule at which buffers are mapped into the engine's address space. Three alignment scales coexist on the M1: the 16-byte DMA-width granule of field `0x1c0`, a 256-byte segment alignment, and the 16 KB page at which buffers map to the engine.

21.4 Two-stage address translation

A host buffer reaches the array through two stacked translations: the kernel driver pins the pages to physical memory, then maps them through a device input-output memory-management unit to a device virtual address the engine's DMA engines read. The translation unit that serves the engine is the t6000-generation controller bound to the device node `dart-ane0`. Table 21.2 gives the page size, aperture base, and aperture size read from the live device node.

Table 21.2. The device-translation parameters for the engine, read from the live device node.

property	value	meaning
page size	`0x4000` = 16 KB	the device virtual page; sub-page buffers still consume a full 16 KB page
aperture base	`0x0`	the device-virtual window starts at zero
aperture size	`0xE0000000` = 3.5 GiB	the managed device-virtual window

The aperture is 3.5 GiB, but the firmware clamps every address it programs into a DMA engine to the bottom 4 GiB by asserting that the high 32 bits of every buffer address are zero, so the engine is confined to the bottom of its device-virtual space. Each page maps to physical memory through a single 64-bit leaf descriptor: for every live engine data page the descriptor is the physical frame address with bit 63 set, the valid bit, and no other field. The descriptor reads phys | `0x8000000000000000` for an input, weight, intermediate, or output page alike, since the physical frames are 16 KB-aligned and leave the high bits free; unmap clears the descriptor to zero, dropping the valid bit. Read-only protection of the inputs is not encoded in the leaf descriptor on the M1: a configuration bit collapses the would-be read-only class into read-write at the translation unit, so input protection is enforced upstream.

Each mapped buffer holds a usage code that records its role, decoded from the mapping call sites and listed in table 21.3.

Table 21.3. The buffer-role usage code each mapped buffer holds, recovered from the mapping call sites.

usage code	role
1	client input tensor
2	client output tensor
7	intermediate buffer
8	kernel and weights
9	program text, task descriptor, and working set
11	program constant and scratch
`0x1019`, `0x101a`	firmware shared surface and resident heap

The address-translation registers and the full fault path are the subject of chapter 28; this chapter covers the on-chip pool, its banking, and the page granule at which buffers enter the device address space.

21.5 Hierarchy on the M1

Table 21.4 gives the on-chip levels and their M1/H13 values, with the table field that holds each one and its role.

Table 21.4. The on-chip memory levels on the M1, each with its value, hardware-abstraction-table field, and role.

Level	M1/H13 value	Table field	Role
SRAM operand working set	2 MB	`0x1b8`	largest single operand that stays on-chip; over this, tile and stream
Operand threshold (measured)	2.28 to 2.34 MB	derived from `0x1b8`	observed throughput threshold, 2 MB plus tiler double-buffer margin
Bank count	64	`0x1c8`	banks the pool is interleaved across
Interleave granule	16 B	`0x1c0`	DMA-width granule and bank-stride unit
Maximum L2 destination stride	2 MB	`0x1f8`	upper bound of the bank-conflict stride search
Residency-buffer threshold	0	`0x1f0`	over this footprint, skip the dedicated buffer; 0 means M1 never allocates one
Kernel-coefficient store	64 KB	`0x200`	on-chip weight-coefficient budget at `0x200`/`0x210`, with the 64 KB-or-16 MB mode select at `0x288`, distinct from the operand working set

Table 21.5 gives the cross-generational comparison for the residency-buffer threshold, the field that varies most across parts, with its effect on whether a dedicated buffer is allocated.

Table 21.5. The residency-buffer threshold across generations and its effect on whether a dedicated convolution output buffer is allocated.

Part	`0x1f0` value	Effect
M1/H13	0	dedicated residency buffer never allocated
A15-class	32 KB	dedicated buffer when convolution output footprint exceeds 32 KB
A16-class and M5	256 KB	dedicated buffer when convolution output footprint exceeds 256 KB

The levels stack from unified DRAM at the bottom, up through the on-chip working set and its 64 banks, into the multiply-accumulate array, as figure 21.1 draws them.

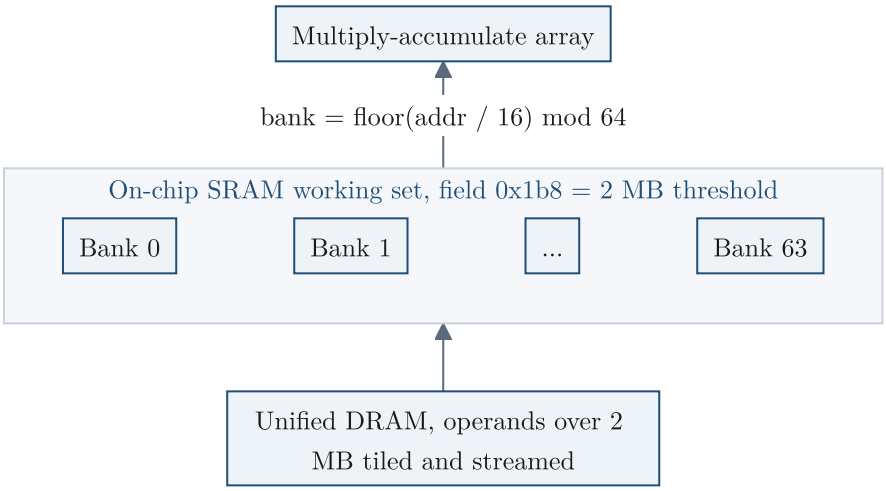

Figure 21.1. The memory hierarchy, with unified DRAM at the bottom, the on-chip SRAM working set and its 2 MB threshold, and the 64-bank interleave feeding the multiply-accumulate array.

Part VII

The Toolchain and Encoding

22 Compiler

The frontend, the anec backend dialect, lowering, and the validators.

23 Program and container format

The on-disk program, its descriptors, and the resolved layout sidecar.

24 HAL and capability gates

The per-chip parameter table and the capability bytes that gate each operation.

25 Compression internals

The reconstruction codecs and the wire layout of each compressed weight form.

26 Hidden layers and direct netplist authoring

Native layers reachable only by authoring the netplist directly.

22 Compiler

SUMMARY

The compiler turns a network description into a loadable program through four phases: fusion to one compute operation, legalization to the hardware envelope, schedule and task-descriptor partition under the on-chip working set, and memory and direct-memory-access optimization. A frontend operation lowers to one or more backend `anec.*` operations, most one to one, with a convolution absorbing its bias, activation, padding, and dequantize into a single fused operation. An exported `_ANECValidate{Op}Layer` entry point checks each operation first, one of 50 per-layer validators among 55 `_ANECValidate*` exports, and a callable validator signals that an operation is reachable on the direct path. A validator that accepts does not guarantee code generation: top-k, sort, dynamic-slice, and three-dimensional convolution pass validation and fail backend lowering on the M1.

22.1 Pipeline

The compiler runs four phases in execution order, each listed in Table 22.1 with its job and the passes that represent it.

Table 22.1. The four compiler phases in execution order, each with its job and representative passes.

phase	job	representative passes
1. Fusion	collapse the graph to one fused compute operation	transpose fusion, bias and activation hoisting into conv and matmul, elementwise-copy elimination
2. Legalization	make the fused graph hardware-legal	dimension and rank legalization (rank `\le 5`), kernel-memory split, multi-segment graph cuts
3. Schedule and task-descriptor partition	linearize the graph and carve it into on-engine partitions	parallel-execution discovery, the greedy list scheduler, partition emission
4. Memory and direct-memory-access optimization	lay out buffers and streams	bank-conflict optimization, pad optimization, width-concat, weight packing

Each phase has a fixed job, and the phase structure decides what is free to emit and what costs a separate dispatch. The figure 22.1 traces the same pipeline as a dataflow, from the network description through the intermediate forms to the loadable program on the engine.

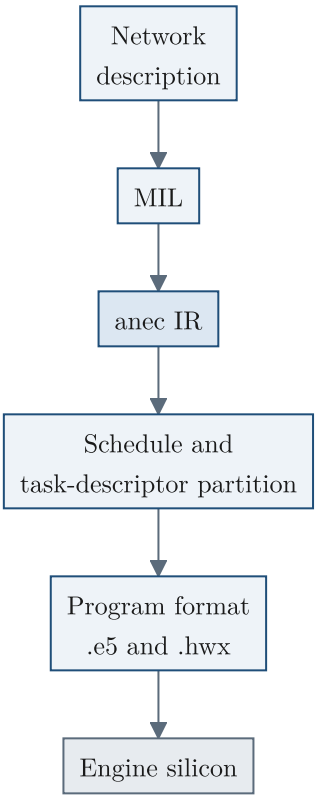

Figure 22.1. The compile pipeline from a network description to the loadable program.

Phase 1 collapses the whole graph to a single compute operation. Transposes are free: they fold into the adjacent operation or route to the dedicated transpose engine rather than becoming a separate dispatch. A bias add and an activation that follow a convolution or a matrix multiply hoist into that operation, so `conv + bias + relu` and `matmul + bias + gelu` each become one fused operation rather than three.

Phase 2 enforces the hardware envelope. Tensor rank is capped at five, per-dimension maxima apply, and an operation whose coefficients exceed the per-buffer kernel-memory cap is tiled or split. The compiler cuts a graph that cannot run as one segment into several at bridge operations.

Phase 3 turns a fused operation into real on-engine work. A greedy priority-queue list scheduler linearizes the operation-layer graph into one execution order, then chunks that order into task-descriptor partitions under the on-chip working-set budget. The scheduler pops the highest-priority ready node each step, ordered by a fixed tie-break sequence: ring-buffer membership first, then branch-sibling score, then merge-sibling score, then the working-set-aware branch test, then topological order, then raw node id. The partition boundary is a memory test, not an operation count: the scheduler tentatively schedules a layer, queries the peak L2 pressure over its live range, and closes the current partition when

$$\text{peak L2 pressure} > \text{HAL}[\texttt{0x1b8}] \times f,$$

where HAL[`0x1b8`] is the 2 MB on-chip static-memory ceiling on the M1 and f is a margin fraction below one. A partition grows until the next layer's live range would push the peak past that scaled budget. The branch decision that keeps the live set on chip reads the same 2 MB field: when a diamond's endpoints fit,

the scheduler runs the larger-footprint branch first while its peak still fits in 2 MB, draining it before the working set grows.

Phase 4 lays out the buffers the partitions read and write: it re-strides the destination buffers to avoid bank conflicts on the main engine, folds padding, and packs the weight stream.

The allocator runs alongside phases 3 and 4. It tags every root tensor with an allocation type from a nine-state set that records whether the tensor is resident on chip, streamed from dynamic memory, chained directly into the next operation in the on-chip cache, or held as a ring buffer. Table 22.2 decodes the nine allocation-type values, each with its meaning.

Table 22.2. The nine allocation-type states the allocator assigns to each root tensor, decoded from the bit-test predicates that read them.

value	meaning
0	plain on-chip resident operand
1	streamed from dynamic memory
2	L2-chained, supplies the next operation directly in L2: the double-buffer state
3	depends on an L2 producer
4	in-place rewrite in L2
5	in-place rewrite, L2-dependent
6	ring-buffer resident
7	ring-buffer resident, L2-dependent
8	in-place in dynamic memory

A single-fanout producer immediately followed by a same-engine consumer with matching tile geometry is marked chainable, allocation type 2, the structural form of double-buffering: the producer output stays on chip and supplies the consumer in place instead of round-tripping through dynamic memory. Chainability is a structural pattern, not a size threshold. The decision is a conjunction. The per-chip chaining-enable bit must be set, the producer must have exactly one outgoing layer, and both ends must be the same engine class on the same engine. The producer's emitted tile geometry must match the consumer's declared input geometry, the consumer must be scheduled immediately after the producer with no other layer in the gap, and the producer's emitted rows must cover the consumer's first tile including overlap. The ring-buffer state, allocation type 6, is the state the scheduler ranks first when it orders ready operations.

Phase 3 also discovers operations that can co-issue on the two engines. The rule is restrictive: two operations run concurrently only when one is a multiply-accumulate operation and the other is a planar-engine operation, since two operations on one engine cannot overlap. The discovery disqualifies a pair when either is unscheduled, when both are the same engine class, when they are on different engines, or when they stand in a producer-consumer relation. A pair is also disqualified when either input is chained or chainable, or when both outputs stream from dynamic memory and would serialize on the shared channel. A matrix multiply's resident weight is exempt from the last test, so it does not block its partner. The attention block is the operation this filter selects: a softmax on the planar engine co-issued with a value matrix multiply on the multiply-accumulate array. Chaining and co-issue are mutually exclusive: an operation is either pipelined into its neighbor or co-issued with an independent operation on the other engine, never both.

22.2 Lowering one operation

A frontend operation lowers to one or more backend `anec.*` operations through the operation-converter pass, most of them one to one. A matrix multiply lowers to `anec.matmul`, which holds the contraction as two attributes rather than as a fixed operand order, as listing 22.1 gives with its operands, attributes, and pre-lowering constraints.

Listing 22.1. The matrix-multiply lowering, with its operands, the transpose attributes that set the contraction direction, and the pre-lowering constraints.

```
mps.MatMulOp  ->  anec.matmul
  bottoms:  2   (A, B)
  attrs:    transpose_lhs : bool
            transpose_rhs : bool
  constraints (checked before lowering):
    depth(A) == 1 and depth(B) == 1
    out_channels == channels(A)
    width(A) + pad == channels(B)
    out-channel bytes of A must fit kernel memory
```

The contraction direction is parametric: `transpose_lhs` and `transpose_rhs` select which operands are transposed before the product, so one backend operation covers every transposed and non-transposed matrix-multiply form. The compiler rewrites a matrix multiply whose right-hand weight fits the on-chip working set into a resident convolution, a single fused operation; a weight above the 2 MB ceiling stays a tiled matrix multiply.

A convolution lowers the same way and absorbs more of its neighborhood, as listing 22.2 gives with its folded weight and bias, its attributes, and its pre-lowering constraints.

Listing 22.2. The convolution lowering, with its folded weight and bias, its attributes, and the constraints checked before lowering.

```
mps.Conv2DOp  ->  anec.convolution
  bottoms:  1   (+ weight and bias folded in)
  attrs:    strides
            dilation_rates
            groups
            explicit_padding / padding_style
            kernel_sizes
            weights_layout
  constraints (checked before lowering):
    filter and input rank 4 or 5
    kernel within the per-chip [min, max]
    in-channels and out-channels divisible by groups
    padded input >= kernel
```

The bias becomes the convolution's gain-offset control, and a following activation becomes a post-operation lookup table on the same operation. A padded, biased, activated convolution thus emits as one backend operation with the pad folded, bias as a per-channel affine, and activation as a 33-segment piecewise-linear table. A quantization scale folds into the per-channel output multiply that precedes that table, so a dequantize between the convolution and its activation does not become its own operation either.

22.3 Fusion rules

Fusion does not happen in the MIL the host hands over. The MIL is the pre-fusion handoff: a biased, activated convolution arrives as three separate operations, `conv` then `add(x=conv, y=bias_const)` then `relu`, and the collapse into one engine layer happens entirely inside ANECompiler's `ZinIr`/`ZinMir` layer-graph stage. The fused hardware unit is the gain-offset control, `anec.gain_offset_control`, a per-channel or singular affine $y = \text{gain} \cdot x + \text{offset}$ applied at engine output. The `ZinNEBypassLayer` container wraps a fused engine operation in a fixed seven-slot epilogue chain whose constructor argument order, given in listing 22.3, fixes the seven slots.

Listing 22.3. The ZinNEBypassLayer constructor, whose argument order fixes the seven epilogue slots a single engine layer can absorb.

```
ZinNEBypassLayer( engine_op,
    ZinTextureLayer,      // in-place spatial/texture remap
    ZinBroadcastLayer,    // broadcast of a fused operand
    ZinActivationLayer,   // pre-GOC activation
    ZinGOCLayer,          // the gain/offset (scale + bias) unit
    ZinActivationLayer,   // post-GOC activation
    ZinTransposeLayer,    // output transpose / layout
    ZinQuantLayer )       // output (re)quantization
```

A convolution, matrix multiply, pool, or elementwise operation absorbs, in slot order, a texture remap, broadcast, pre-GOC activation, the gain-offset affine, a post-GOC activation, output transpose, and output requantization. The bias-add fills the GOC offset, the scale or batch-norm gain fills the GOC gain, and the activation fills one of the two activation slots. The pass that forms the offset is `ScaledEWOrEWWithConstInToGOC`, and an internal guard reads `"Must have 2 inputs when convert EW to GOC"`: the elementwise-to-GOC rewrite needs exactly one live operand and one constant operand. That guard is the dividing line between a fusable epilogue and a fusion barrier. Table 22.3 gives each pattern, how it fuses, and the gating pass, alongside the barriers that keep operations as separate layers.

Table 22.3. The fusable epilogues that collapse into the producing engine layer and the barriers that keep operations as separate layers.

pattern	fuses as	gating pass
bias-add (`add` with one constant)	GOC offset	`ScaledEWOrEWWithConstInToGOC`, `ZinMirHoistGOCorActivationForConvFusion`
scale or batch-norm (`mul` with one constant)	GOC gain, or folded into the weights	`FoldScale`, `FoldWeightsWithScale`
activation (relu, leaky, clamped, swish, sigmoid, tanh, gelu)	a pre- or post-GOC activation slot; transcendentals become a palette lookup table	`MergeFusableActivationPairs`, `SimpleActivation`
transpose and reshape chains	the NEBypass transpose slot; symmetric pairs cancel	`CollapseTranspose`, `ZinMirNETransposeFusion`
dequantize	folded into the convolution weight path as `kernel_scale` or `kernel_palettized_LUT`	`CollapseQuantDequant`, `DeduplicateStandaloneDequantGOCsAcrossConcat`
matrix multiply or linear	rewritten to a convolution, then takes the convolution epilogue rules	`ReplaceMatmulWithConv`
two-live-input elementwise (`add(conv_a, conv_b)`)	does not fuse: stays a real `anec.add` engine layer (fails the one-constant guard)	barrier
concat	a fusion boundary unless `IsFusableConcat` qualifies it	barrier
scaled-dot-product attention	a hard segment cut between programs	barrier

A two-live-input elementwise operation is the one barrier that follows from the GOC definition itself. The GOC is an affine of one tensor whose gain and offset are constants or per-channel vectors, so `add(conv_a, conv_b)` cannot be a GOC and remains a real `anec.add` engine layer that separates the two convolutions. A trailing activation can still fill the add's own post-GOC activation slot, but the add itself does not vanish. Concat is a boundary unless it qualifies as fusable, scaled-dot-product attention is a hard segment cut that matches the host's own segmentation, and the `ZinMir*Split` capacity passes can re-cut a fused layer that exceeds the tile or engine limit, so a fusion formed earlier can be undone downstream.

22.4 Validator surface

Before an operation lowers, a per-layer validator checks it. The validators are a family of exported `_ANECValidate{Op}Layer` entry points, one per operation class, with an umbrella `_ANECValidateNetworkCreate` that iterates a network's layers and invokes the per-layer validator for each. The export table has 55 symbols whose name starts `_ANECValidate`, which split into 50 per-layer `...Layer` validators and 5 non-layer exports. Table 22.4 names the five non-layer exports and the role of each.

Table 22.4. The five non-layer `_ANECValidate*` exports and the role of each, alongside the 50 per-layer validators.

non-layer export	role
`_ANECValidateNetworkCreate`	the umbrella oracle: a dry-run of network creation that iterates the layers and invokes each per-layer validator
`_ANECValidate`	the top-level wrapper for a single operation or descriptor
`_ANECValidateMPSModule`, `_ANECValidateMPSModuleCreate`	validate an intermediate-language module before its conversion to the backend form
`_ANECValidateMutableProcedureInfo`	validate a weight-editing procedure descriptor

Five per-layer validators have no matching descriptor constructor, since they validate computed or umbrella operations rather than directly authored layers: argmin-max, global argmin-max, broadcast, cross-product, and elementwise. The same per-layer code runs in two places: the placement dry-run that decides whether an operation is eligible for the engine, and the backend legalizer during a real compile, so the eligibility prediction matches the compile outcome.

Each validator checks the operand count and the shape, type, and attribute envelope for its operation, and emits a fixed reject string when a constraint fails. The matrix-multiply validator requires exactly two inputs, depth one on both, the output channel count equal to the first input's channel count, and the first input's output-channel bytes to fit kernel memory. It rejects with `"Matrix mult. layer can only have two bottoms"`, `"depth > 1 is not supported for MatMult"`, and `"can not fit the Kmem"`. The convolution validator checks the kernel against the per-chip bounds and the channel-group divisibility, rejecting with `"Invalid conv kernel %s = %zd, It should be in [%zd,%zd]"` and `"input/output channels should be divisible by num group"`.

The validators that gate a feature on the chip read a per-chip support byte, so the same operation can validate on one generation and reject on another. Table 22.5 lists representative feature-gated validators, the support byte each reads, and the generation it gates on.

Table 22.5. Representative feature-gated validators, the per-chip support byte each reads, and the generation it gates on.

operation	support byte	gated on
softmax	`0x815`	rejected on the older targets
instance norm	`0x816`	per-chip
dropout, random	`0x4a9`	the A15 and later generations only
affine transform, resize, resample, padding mode	`0x81d`	the texture engine, absent on the M1
dynamic gain-offset control	`0x814`	absent only on the smallest legacy parts
three-dimensional convolution	dimension and kernel-depth fields	has the capability byte yet fails backend lowering

The reject strings name the constraint plainly: a softmax on a target without `0x815` returns `"Softmax is not supported by this ANE architecture"`, a dropout without `0x4a9` returns `"Dropout layer is not supported on this architecture."`, and an affine transform without the texture engine returns

`"affine transform is not supported on this architecture"`. The network-level umbrella adds its own checks, rejecting a module newer than the current schema, bonded network the target cannot run, unit-name collision, or missing conversion record. The reject strings include `"Bonded networks are not su pported on the target"` and `"ANE internal error: Unit name \"%s\" collision during MIL to A NEC IR conversion."`.

Every per-layer validator is an exported, user-callable symbol: an operation whose validator accepts the schema is reachable by direct network authoring. This is the attested-is-not-reachable rule from chapter 4 in its compiler form: an operation can pass its `_ANECValidate*Layer` schema check and still fail at backend lowering. On the M1 the top-k, sort, and dynamic-slice validators accept their inputs and then reject at code generation, and the unflatten validator passes yet fails lowering for every variant. Three-dimensional convolution has its capability byte yet fails backend lowering on every device mask. One opcode-surface gap exists below the validator layer: the contrast-adaptive-sharpening and reverse operations have an internal validation routine but no exported per-layer validator, so they are not reachable by direct authoring at all. The reachable surface here is confirmed against an on-device sweep.

Validation happens in three distinct layers. The exported `_ANECValidate*` catalog, the per-layer and umbrella validators, is the outward face. Behind it the in-binary `ValidateSemantics_Impl` family holds the `MatchParams` and `MatchStatus` pattern grammar that the exported validators are generated from. Below both, an in-kernel `ZinComputeProgramValidate*` family validates the compiled program again at load, across single-plane, multi-plane, uncompressed, and tiled-compressed matrix forms.

22.5 Native layers and netplist authoring

The 50 per-layer validators pair with 45 native hardware layer descriptors, the operation classes the engine silicon implements. A few of these never reach the engine through the conversion-tool front door, since that path always decomposes them, yet they are reachable by authoring the network description directly. The fused scaled-dot-product-attention layer is the clearest case: its descriptor takes four or five inputs, the query, key, value, scale, and an optional additive mask, and its validator rejects with `"4 or 5 bottoms must be present for SDPA"` and `"Mask format must be same as Q, K and V"`. Its one parsed parameter is the max-subtraction flag, which defaults to off and must be set on for a numerically correct softmax, and the causal-decode form is not a separate descriptor but the additive-mask variant.

The network description is a property-list document, the netplist, schema version 1.0.10, whose keys the compiler reads verbatim and Table 22.6 names with the role of each.

Table 22.6. The netplist dictionary keys the compiler reads, with the role of each.

key	role
`Networks`, `ProcedureList`	the network names and the callable entry points
`Units`, `Weights`	the ordered layer list and the weight-blob files
`InputList`, `OutputList`, `OperationList`	the external ports and the operation order
`Bottom`, `Type`, `Params`	per-unit wiring, layer-type tag, and typed attributes
`BatchSize`, `InputChannels`, `InputDepth`, `InputHeight`, `InputWidth`, `InputInterleave`	the port shape five-tuple and the channel-interleave packing

A layer is a unit with a `Type` tag, a `Bottom` wiring list, and a `Params` sub-dictionary of typed attributes, and the wiring is by symbol name. The compiled program that comes out is a container with the magic `0xbeefface`, whose text section holds the task-descriptor register-write stream and whose `__KERN_N` sections hold the weight coefficients, the same image the kernel loads.

23 Program and container format

SUMMARY

A compiled network reaches the engine as two layered files: a dispatch descriptor, a FlatBuffer that names the operations and binds the buffers, over a hardware container. The hardware container is a Mach-O-shaped file with the magic `0xbeefface` that holds the register-write program and the weight banks. The descriptor tracks dispatch count, not operation count: a fused graph of any depth reduces to one inference operation bracketed by input and output casts, and a bridge cut adds one inference operation per segment. Between them is the hardware task descriptor `ZinAneTdHw_v10`, a register image in seven groups, serialized sparsely as 44-byte records whose buffer bases are relocation slots resolved at load.

23.1 Two layers

A model passes through six on-disk and in-memory forms, which Table 23.1 gives with the format, identifying tag, and role of each.

Table 23.1. The six on-disk and in-memory forms a model passes through, each with its format, identifying tag, and role.

Stage	Format	Tag	Role
Intermediate-language text	text	versioned `3520.4.1`	the compiler's input
Network description	property list	version `1.0.10`	the layer and wiring graph
Hardware container	Mach-O shape	`0xbeefface`	register-write program plus weights
Dispatch descriptor	FlatBuffer	vtable `0c00 1400 0400 0800`	the operation chain the runtime submits
Loaded program image	Mach-O shape, signed	`0xbeefface`	the same container, virtualized per load
Firmware container	sectioned blob	`ANEH` / `ANEP` / `ANES`	the on-package load form

The compiler produces a dispatch descriptor and a hardware container for one network, cached together. The dispatch descriptor is above the operation level of the intermediate language: it holds the operation chain as a parametric program structure, with each operation reduced to an argument frame and a kernel attribute blob. The hardware container is below that level: it holds the register-write stream, weight coefficients laid out for the streaming datapath, and typed input and output layout, all in a Mach-O-shaped file with the magic `0xbeefface`. The hardware container and the loaded program image share one byte format: the loaded copy is the on-disk container after the address relocations are resolved.

The figure 23.1 shows the layering, with the dispatch descriptor over the hardware container over the task-descriptor register groups that drive the DMA engines and the multiply array.

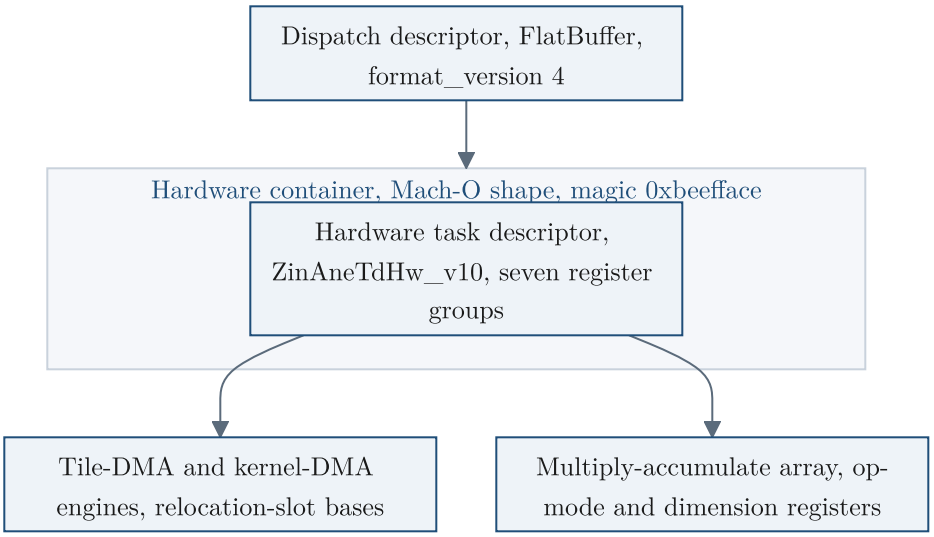

Figure 23.1. The program format, with the dispatch descriptor over the hardware container over the task-descriptor register groups that control the DMA engines and the multiply array.

23.2 Dispatch descriptor

The dispatch descriptor is a FlatBuffer whose root table has four fields. The serializer method set yields the root layout, validated byte for byte against a real descriptor and given in Table 23.2 with each field's vtable offset, contents, and type.

Table 23.2. The four fields of the dispatch-descriptor root table, with their vtable offsets, contents, and types.

Field	Voffset	Contents	Type
0	4	the symbol and section name vector	`symbol_names: [string]`
1	8	build-info key and value pairs	`build_info: BuildInfo`
2	12	the section descriptor vector	`sections: [Section]`
3	16	inline scalar, value 4	`format_version: int`

The operation set the descriptor can hold is fixed. The serializer instantiates one attribute-serialization template per operation kind, and the twelve instantiations enumerate the complete set that listing 23.1 recovers as the root table and the operation-type enumeration.

Listing 23.1. The recovered FlatBuffer schema for the dispatch-descriptor root table and the twelve-member operation-type enumeration.

```
// E5Program root table, recovered from the E5Serializer symbol family
// and validated against a real H13C dispatch descriptor (3816 B).
table E5Program {
  symbol_names: [string];     // operation, tensor, and section names
  build_info: BuildInfo;      // compiler version and source path k/v
  sections: [Section];        // per-operation arg_frame + op_attrs refs
  format_version: int;        // observed == 4
}

enum OpType : ubyte {
  Cast, AneInference, EirInference, CpuInference, BnnsCpuInference,
  MlcCpuInference, MpsGraphInference, E5MinimalCpu, Quant, Dequant,
  Barrier, JitCall            // ordinals inferred; membership measured
}
```

The schema was recovered from the serializer symbol family rather than from embedded reflection, since the runtime strips the binary schema and the schema source. The deciding evidence is the `E5Serializer` method set, one serializer method per schema table, with the attribute-serialization template instantiated once per operation kind, so the table list and the operation-type enumeration fall straight out of the symbols. Table 23.3 pairs each demangled serializer method with the schema table or operation-type union it proves.

Table 23.3. The dispatch-descriptor serializer methods, each proving one schema table or the operation-type union.

Demangled serializer method	Schema element it proves
`SerializeFunction`	`table Function`
`SerializeBlock`	`table Block`
`SerializeOperation`	`table Operation`
`SerializeOpArgFrame`	the `__arg_frame` section of an operation
`SerializeOperand`	`table Operand`
`SerializeIOPort`	`table IOPort`
`SerializeAliasSymbol(string, uint, uint)`	`table AliasSymbol{name, symbol_index, addr_offset}`
`SerializeBuildInfo`	`table BuildInfo`
`SerializeOpAttrs<CastOpT>` and 11 more	`union OpAttrs` and `enum OpType`

The tensor and surface descriptor each have a fixed field list, lifted verbatim from one runtime type-encoding string and reproduced in listing 23.2.

Listing 23.2. The tensor and surface descriptor field list, recovered verbatim from a runtime type-encoding string.

```
/* TensorDescriptor, the per-operand layout record, from the runtime type encoding.
   Q = u64, i = i32; the two leading pointers are runtime-only and not serialized. */
struct TensorDescriptor {
  void    *data, *reserved;     /* runtime pointers, not on the wire */
  uint64_t dim[4], stride[4];
  uint64_t width, height, channels, batch_number, sequence_length;
  uint64_t stride_width, stride_height, stride_channels;
  uint64_t stride_batch_number, stride_sequence_length;
  int32_t  storage_type;        /* the element-type code */
};
```

The canonical program shape is a fused compute operation bracketed by input and output format casts. A network that fuses to one engine dispatch decodes as `Op0_Cast` then `Op1_AneInference` then `Op2_Cast`.

The cast operations convert the host half-precision layout to the engine-internal interleaved layout on input and back on output; the single inference operation is the entire fused graph.

Each operation has two sections: an argument frame that names the operand binding and the per-axis tensor layout, and an attribute blob that holds the kernel parameters. The attribute blob is a nested FlatBuffer sub-descriptor, not expanded microcode: its byte diversity is 0.19, the low value of a structured and sparse record, and the descriptor size does not change with the compute shape. A contraction over an inner dimension of 64 and one over an inner dimension of 256 both produce the same descriptor size, so the compute shape is a parameter the descriptor holds, not a program it expands.

Two properties follow from this structure. The descriptor is depth-invariant: a one-operation graph and a six-operation fused graph both reduce to a single inference operation of the same size. It tracks the dispatch count, not the operation count: a bridge operation that cuts the graph into three segments produces three inference operations and a larger descriptor.

23.3 Hardware task descriptor

The hardware task descriptor is the register-level program the engine runs: each group below is a block of register writes that configures one engine unit.

Table 23.4. The seven register groups of the hardware task descriptor, each with its struct base offset, register count, addressable aperture base, and contents.

Group	Struct base	Registers	Register base	Contents
Kernel and common	`+0x2c`	34	`0x5500`	kernel-DMA enable, format, stride, task type, network id
Dimensions	`+0xfc`	19	none	input and output width, height, depth, channels, groups, transpose
Tile DMA	`+0x150`	69	`0x4d00`	the three tile-DMA engines: enable, cache hints, base address, strides
Element-wise and planar	`+0x26c`	30	`0x4100`	element-wise and planar-engine config, padding mode
L2 and texture	`+0x2ec`	14	`0x4500`	L2 source and result config, texture mode
Kernel format and op mode	`+0x32c`	11	`0x4900`	op mode, kernel alignment, sparse and palette format, bias enables
L2 result	`+0x360`	21	`0x5100`	L2-result base, strides, wrap, result format

Between the parametric attribute blob and the register-write records the engine executes is the hardware task descriptor. On the M1 generation it is the versioned descriptor `ZinAneTdHw_v10`, a flat register image the compiler fills field by field and then serializes into address and value pairs, partitioned into the seven groups Table 23.4 lists. The version family is per generation: the same descriptor is instantiated under a different version number for each silicon family, and the byte offsets, register addresses, and field widths move between versions.

The dimension fields show the hardware encoding behind the per-axis size limits. Spatial dimensions are 15-bit and channel dimensions are 17-bit, read from the register getters and confirmed against the vendor's own field symbols, as listing 23.3 gives with each field's descriptor offset and bit mask.

Listing 23.3. The dimension and direct-memory-access fields of the hardware task descriptor, with their descriptor offsets and bit masks.

```
/* ZinAneTdHw_v10 dimension and DMA fields, offsets on the descriptor base.
   Field names and widths confirmed against the vendor TD symbol table. */
Win         @0x0f4  mask 0x07fff   /* input width,  15-bit */
Hin         @0x0f6  mask 0x07fff   /* input height, 15-bit */
Cin         @0x100  mask 0x1ffff   /* input channels,  17-bit (up to 131071) */
Cout        @0x104  mask 0x1ffff   /* output channels, 17-bit */
OCGSize     @0x118  mask 0x07      /* output-channel-group size, 3-bit */
numGroups   @0x11c  mask 0x1fff    /* conv groups, 13-bit */

/* The four DMA engines. Base addresses are relocation slots, resolved
   at load; strides and formats are written in-image. */
TileDMASrc1 base   reloc reg 0x1344   /* input tile read  */
TileDMASrc2 base   reloc reg 0x134a   /* second operand   */
TileDMADst  base   reloc reg 0x1442   /* output tile write */
KernelDMASrc       group G0 @0x2c     /* weight banks, up to 16 sub-buffers */
```

The compiler does not write buffer base addresses into the image at compile time. It leaves each as a named relocation slot keyed by a register address, and the loader patches the device address in. The full v10 address-register relocation table is the input tile, second operand, output tile, and four kernel coefficient streams, which Table 23.5 maps slot by slot.

Table 23.5. The v10 relocation-register map, the address slots left symbolic at compile time and patched to device addresses at load.

Relocation register	Slot
0x1344	input tile read base
0x134a	second-operand read base
0x1442	output tile write base
0x1554	kernel bias stream
0x1558	kernel post-scale stream
0x155c	kernel palette-lookup stream
0x1560	kernel activation-lookup stream

The four weight sub-streams, bias, post-scale, palette lookup, and activation lookup, each get their own relocation slot, so a convolution with a folded batch normalization lowers to four independent weight streams. Each direct-memory-access stride is a 26-bit signed field located at bit 6 of its register word, range-checked against a per-chip bound table. A tensor's element strides become these register words: a row stride, channel stride, depth stride, and group stride per engine.

The loader patches more than buffer bases. Each relocation is a custom bar that writes a resolved value, a device address, an on-chip-memory data-set identifier, or a buffer offset, into an exact descriptor bit-field named by its register, bit offset, and bit width. Runtime-variable shapes and strides bind separately as live-in parameters, each tied to a named procedure input and range-checked against a start, stop, and step, so one compiled program serves a range of input shapes without recompiling.

The descriptor layout is versioned per silicon generation: the M1 builds the v10 image, and other generations build a different version under the same family of accessors, with the dimension block, relocation-register addresses, and field widths all moving between versions. The M1 v10 image omits the compute-cache

direct-memory-access engine entirely: every setter for the on-chip atomic, counter, and wait-event primitives asserts that it is unsupported on this architecture. This is the silicon-level reason a resident in-place state buffer cannot be expressed on the M1 and falls back to a shared buffer. Any tool that reads task-descriptor bytes must dispatch on the version number rather than assume a fixed layout.

The serializer emits the image sparsely: a register reaches the stream only when its value differs from the architectural default, which is why a small contraction yields only six to eight records. Each surviving record is a fixed 44 bytes: a count marker, the register address in one IOMMU aperture, and one or two device addresses in a second aperture that are written to the DMA base-address registers. The register addresses are per-load device addresses, virtualized by the IOMMU, so the same logical register resolves to different addresses across loads.

23.4 A decoded program

A 64-to-64 half-precision linear layer with an identity weight matrix is the smallest real program on the M1 host that includes a resolved layout sidecar. Its container is a Mach-O-shaped file with the magic `0xbeefface`, a pseudo-architecture marker, and the executable file type. The resolved frame runs `[1, 1, 1, 64]` half-precision input to `[1, 1, 1, 64]` half-precision output, all strides 128 bytes.

The container maps each external tensor and the weight bank to its own page-aligned aperture in the engine virtual address space, base `0x30000000`, which Table 23.6 gives segment by segment with each one's virtual address, size, protection, and role.

Table 23.6. The segment map of the decoded identity linear layer container, with each segment's virtual address, size, protection, and role.

Segment	Section	Virtual address	Size	Protection	Role
guard	none	`0x00000000`	`0x4000`	none	guard page
input	const	`0x30008000`	`0x80`	read	input aperture
output	data	`0x3000c000`	`0x80`	write	output aperture
text	text	`0x30010000`	`0x474`	read and execute	register-write program
weights	kernel	`0x30018000`	`0x2000`	read	weight coefficients

The segment protection encodes the hardware read and write direction: read for the input and the weights, write for the output. Two named port descriptors form the external binding table, each a 24-byte body of `{byteSize, apertureVA, nameRef}`, binding `t0` to the input aperture and `t2` to the output aperture.

The register-write program in the text section is 0x474 bytes: a 12-byte header then a sparse packed register image holding exactly two task-descriptor blocks, an input layout-convert and the fused matrix-multiply, matching the two operations the symbol table names. Table 23.7 decodes the program record by record, each with its offset, bytes, and meaning.

Table 23.7. The register-write program decoded record by record, with each record's offset, bytes, and meaning.

Offset	Bytes	Meaning
header	`00 00 00 00 00 00 9c 00 08 04`	size and version words, register-count base `0x408`
TD-0 +0x010	`28 00 00 00 ...`	record marker `0x28`, a tile-DMA base relocation slot
TD-0 +0x034	16 times `80 00 00 00`	per-group value `0x80` for the 16 output-channel-group partitions

Offset	Bytes	Meaning
TD-0 +0x0c8	`21 a0 00 50 41 20 00 00`	register `0x5021` set to `0x2041`, the kernel-common record
TD-0 +0x128	`31 20 00 01`	op-config descriptor `0x2031`, starts the operation
TD-1 +0x234	16 times `81 00 00 00`	the second descriptor's 16 partitions
TD-1 +0x274	`00 02 04 ... 1e`	the 16 output-channel-group offset indices
TD-1 +0x470	`31 20 30 01`	op-config descriptor variant, terminates the program

The 16-wide register runs are the binary form of the 16 output-channel-group partitions, one per weight sub-kernel. The kernel-common record `0x5021` set to `0x2041` is identical across both descriptor blocks and matches the firmware's own program-manager log byte for byte.

A live M1 program image has the same shape in the task-descriptor stream itself. Its program-image text section holds three chained `ZinAneTd<7u>` records linked by a next-record pointer at offset `+0x1c` and followed to a null terminator. Each operation descriptor has a 32-bit opcode word, and the three operations of this program decode as the opcode words Table 23.8 lists.

Table 23.8. The version-7 codegen opcode words of three operations decoded from a live M1 task-descriptor stream.

Operation	Opcode word
Convolution	`0x5042a063`
Reduce-mean	`0x5000a021`
Matrix multiply	`0x5000b021`

These are the version-7 H13 codegen opcodes. The high half-word, `0x5042` or `0x5000`, is shared across operations, and the low 16 bits distinguish the operation. The full operation identity is also in the lookup-table and configuration words, not in the opcode word alone. The container closes with an `LC_THREAD` trailer holding the procedure and tensor names `main_ane`, `t0_ane`, and `t5_ane@output`.

The container header is a 32-byte Mach-O-shaped block whose fields Table 23.9 gives with each one's value and meaning, and its load-command census names the binding and configuration records the loader walks.

Table 23.9. The hardware-container header fields of the decoded identity linear layer, with each field's value and meaning.

Offset	Field	Value	Meaning
`0x00`	magic	`0xbeefface`	engine compute executable
`0x04`	cputype	`0x80`	engine pseudo-architecture
`0x08`	cpusubtype	`0x4`	the H13 codegen revision
`0x0c`	filetype	`0x2`	executable
`0x10`	ncmds	15	load-command count
`0x14`	sizeofcmds	`0x32d8`	load-command bytes
`0x18`	flags	`0x200000`	

The 15 load commands of this container break down as seven segment commands, two input and output port descriptors, three operation descriptors, one build-info banner, one source-note, and one symbol table. The two-port, three-operation-descriptor signature is the form of a single-input, single-output graph: one header descriptor that lists the aperture bases plus one descriptor per engine pass. The header operation descriptor

has an aperture-address table that pre-resolves the segment bases to the engine virtual map, re-virtualized per load by the input-output memory-management unit.

The build-info banner is the compiler invocation that produced the container, recorded verbatim. It names the toolchain (`zin_ane_compiler v9.509.0`, the intermediate-language component `3520.4.1`) and the target plus the live flag set, of which the relevant entries listing 23.4 reproduces are the target selector and the streaming and cache policy.

Listing 23.4. The entries of the build-info banner, the target selector plus the streaming, cache, and fp8 policy flags.

```
-t h13g                              target H13, the M1 engine
--fl2-cache-mode=resident            keep the L2 working set resident
--fkernel-rewind=enabled             kernel-stream rewind
--split-kernel-section=true          split the weight section
--max-kernel-section-size=134217728  128 MB weight-section ceiling
--e4m3-overflow-setting=Saturate     fp8 overflow policy: clamp, not NaN
--memcache-size=4194304  --bss-limit=3221225472
--foptimize-ne-utilization=true  --enable-global-cw-optimization=true
-i  .../model.mil   -o  .../model.hwx.tmp
```

The container is self-describing. Its symbol table holds the 24-entry element-type catalog with range encodings, half-precision as type 5, the 8-bit floating form as type 16, and 4-bit integer as type 23, alongside the per-axis stride frame for each tensor as a typed array, both reproduced in listing 23.5.

Listing 23.5. The container's element-type catalog and per-axis stride frame.

```
void:t1  int8:t2=r2;0;127  uint8:t3  int16:t4  float16:t5=r1;2;0  float:t6=r1;4;0
... e4m3:t16=r1;1;0 ... int4:t23=r1;-8;7
t0:t44=ar1;0;1; s128n: s128c: s128h: s2w:5   // batch, channel, height, width strides
```

The container stores each weight tensor at a decoded symbol region with a fixed tiling. A convolution weight is in a `0xC0`-stride layout and a matrix-multiply weight in a `0x40`-stride layout, the two strides Table 23.10 gives, and a weight-value edit leaves the program descriptor unchanged, so weights are patchable in place.

Table 23.10. The decoded weight-region tiling stride per weight kind, the fixed layout that lets a host patch weight values without recompiling.

Weight kind	Tiling stride
Convolution weight	`0xC0`
Matrix-multiply weight	`0x40`

The weight bank holds 16 sub-kernels at a 512-byte stride. Decoding the bytes proves the model: only 64 of the 8192 bytes are nonzero, each the half-precision value `0x3c00` equal to 1.0, laid one per partition across the 16 sub-kernels, the diagonal of a 64-by-64 identity matrix. The container is the complete compiled form of an identity linear layer that copies its input through the multiply-accumulate array.

The dispatch descriptor and the container correspond to one network at a time. A stateful recurrent model with one input and five held states decodes as six input casts, one fused inference operation, and six output casts, with the state buffers laid at 16-kilobyte strides. The whole fused graph is still a single inference operation and a single dispatch.

23.5 On-disk container, decoded segment by segment

A second M1 sample, an `h13g` container of 64 kibibytes, confirms the same format and contributes the deltas below; its segment map, `+0x1c` next-record chain, `0xC0` and `0x40` weight strides, build-info banner, and element-type catalog all match the identity-layer decode above. The delta that proves the format is a Mach-O variant rather than a Mach-O proper is the header magic: `0xbeefface` is little-endian bytes `CE FA EF BE` where a real 64-bit Mach-O would hold `0xfeedfacf`. Patching those four bytes lets `otool` and `rabin2` parse the file, while the in-kernel loader accepts the engine magic directly.

The segment commands name `__FVMLIB`, `__TEXT`, and `__KERN_0` regions. The two `__FVMLIB` windows declare the input and output tensors with `fileoff` and `filesize` both zero: they have no on-disk bytes and exist only as virtual-memory declarations that the kernel maps into the address-translation aperture at submit time. The window sections align to `0x4000`, the 16-kibibyte page granule of the address-translation unit, while the task-descriptor and weight sections align to `0x40`, the 64-byte engine tile granule. A pair of window-binding load commands, each a `{u32 size, u32 pad, u64 vmaddr, char name[8]}` record, binds the symbolic names `t0_ane` and `t5_ane` to the input and output virtual addresses.

23.5.1 Task-descriptor linked list

This sample holds three `ZinAneTd` records linked through the `+0x1c` next-offset field at `0x000` to `0x300` to `0x500` to terminator. Each descriptor header decodes into the small set of fields Table 23.11 gives, including the next-offset field that chains the list.

Table 23.11. The decoded header fields of a `ZinAneTd` task descriptor, with the next-offset field at `+0x1c` that chains the list.

Offset	Field	Meaning
`+0x00`	index and flags	a 16-bit index plus a flags high byte, where `0x03` marks the last or barrier record
`+0x04`	size	a per-descriptor size or latency field
`+0x08`	op and engine config	an operation-kind word whose low byte distinguishes the engine pass
`+0x10`	config mask	an engine configuration mask sharing a `0x00fff8xx` prefix across records
`+0x18`	DMA word	the direct-memory-access or compute descriptor word
`+0x1c`	next offset	the byte offset of the next descriptor, zero at the tail

The weight bases each descriptor consumes are not in the header but in an inline array of relocations against `__KERN_0`. The twelve relocations on `__text` are the tile base offsets: the first weighted descriptor reads eight tiles, the third reads four, and the middle descriptor has no kernel relocation and so is a non-weighted pass such as an activation, reshape, or pool.

23.5.2 Weight tiling and the per-lane naming

The `__kern_0` section holds the half-precision weight coefficients split into 64-byte-aligned tiles, one tile per engine compute lane, named in the symbol table as `K<sha256>_ne_<i>` where the index `i` selects the lane. The first weight in this sample, `K596A4B73...`, splits into eight tiles `_ne_0` through `_ne_7` at the `0xc0`-byte convolution stride of `3 × 64`. The second weight, `KE125552B...`, splits into four tiles `_ne_0` through `_ne_3` at the `0x40`-byte matrix-multiply stride, each tile a small weight padded up to the 64-byte tile granule. The symbol table also lists tiles `_ne_4` through `_ne_15` for that second weight, all pointing at the section end as empty sentinel tiles for the unused lanes, distinguished by a `desc` field of zero against `0x0002` on the live tiles. The tiling rule this sample exhibits is that a weight splits into `min(8, lanes)` tiles, each padded to a

multiple of 64 bytes and laid out contiguously, with each descriptor referencing tile `i` by its 64-byte-aligned offset through a relocation.

23.5.3 Stabs-style shape descriptors

The input and output tensors are shaped through stabs-style type descriptors in the symbol table, the entries with symbol type `0x20`, encoded as `s<stride><axis>` byte-stride lists over the four axes `n` for batch, `c` for channel, `h` for height, and `w` for width, as listing 23.6 gives for the bound input and output.

Listing 23.6. The stabs-style shape descriptors for the bound input and output tensors, encoding the per-axis byte stride.

```
t0_ane  (input,  vm 0x30000000):  s8192n / s1024c / s64h / s2w
t5_ane  (output, vm 0x30004000):  s64n   / s64c   / s64h / s2w
```

The input strides describe an eight-channel tensor packed channel-major into the 64-byte tile and 16-kibibyte page tiling. The width stride of 2 bytes is one half-precision element, the height stride of 64 bytes is one tile row, and the channel stride of 1024 bytes is sixteen height rows. The same symbol table enumerates the full hardware element-type catalog, the wider set decoded under the element-type enumeration section below.

23.5.4 Build manifest

The build manifest matches the banner decoded above and also names the bundle `com.apple.ANECompilerFramework`. Three `LC_THREAD` commands hold the per-engine register and thread state, the context-restore blobs the kernel pushes before it launches the descriptor chain.

23.6 Element-type and operation enumerations

The serialization element-type enumeration the descriptor uses for an operand is a closed set of eleven codes, which Table 23.12 gives.

Table 23.12. The eleven serialization element-type codes the dispatch descriptor uses for an operand.

Code	Type	Code	Type
0	int4	6	uint16
1	uint8	7	int32
2	int8	8	uint32
3	float16	9	int64
4	float32	10	uint64
5	int16		

The 8-bit floating forms and the palettized index types are a separate hardware element-type space, not part of this serialization enumeration: they are gated by the hardware-abstraction table and held as typed fields rather than parsed from an attribute string. The container's symbol-table catalog runs to 24 entries because it holds that wider hardware set.

The single inference operation the descriptor holds is a fused graph of backend operations, each one of a fixed micro-operation opcode space and one operation-class selector. The operation-class enumeration, the unit the descriptor calls the operation mode, has 79 members; the compute-unit entries a developer reaches appear in Table 23.13.

Table 23.13. Selected entries of the 79-member operation-class enumeration, the selector the descriptor calls the operation mode.

Class	Code	Class	Code
Conv	1	MatrixMultiplication	18
Pooling	2	Reduction	20
Concat	3	Linear	60
ElementWise	4	NEConv	68
ScaledElementWise	5	NEMatMul	69
Neuron	6	NEPool	70
GOC	8	SDPA	77
Softmax	24	AllReduce	78

There is no transposed-convolution class: a transposed convolution lowers to a convolution, cross-correlation, or kernel rasterizer, not a separate operation mode. The full 79-member set and the 126-member micro-operation opcode space are decoded in full in Appendix C.

23.7 Submission ABI

The runtime hands the dispatch descriptor and the bound buffers to the driver through one external-method interface keyed by a selector and a structure size, and the kernel disambiguates an overloaded selector by the structure size rather than the selector alone, as Table 23.14 gives method by method.

Table 23.14. The external-method selectors of the submission interface, with the structure size that disambiguates each overloaded selector.

Selector	Method	In bytes	Out bytes
0	device open	104	104
2	program send request	2376	40 async
3	program create	32	0
3	program output set enqueue	40	0
4	program prepare	56	pointer
4	program inputs ready	3104	0
5	program memory map request	1 scalar	2080
6	program destroy	16	0
8	program create instance	32	0
8	program chaining set active procedure	32	0
9	program chaining prepare	16	pointer

The submit call is the asynchronous selector 2, whose 2376-byte argument structure holds the program token, a sequence number, the quality-of-service pair, and the array of input, output, and intermediate surface identifiers. The 40-byte asynchronous reply holds the sequence result and the echoed token, and completion arrives on a wake port rather than a shared-memory poll. The chaining-prepare and chaining-set-active selectors build the firmware chaining cache, the structure a multi-segment descriptor and the resident-state path use to wire one inference segment to the next.

24 HAL and capability gates

SUMMARY

The compiler is one binary that builds any chip in the line from a per-chip data table, the hardware abstraction layer, read at compile time. The table holds scalar fields indexed by byte offset for the numeric limits and a dense capability-byte region at offsets `0x48f` through `0x8cc`, each byte read as `hal[offset] & 1` to gate one operation or format. Operation legality is declared on the operation as a `MinimumFamily<N>` trait: native only when the target family index is `N` or greater, and decomposed below the floor, with no compute operation floored above A15. A capability in the table attests support at the layer that reads it and does not prove the operation runs: three-dimensional convolution has its kernel-depth attestation at `0x70` and fails backend lowering on every device mask.

The compiler that targets the Apple Neural Engine is one binary that builds any chip in the line on demand. What separates one target from the next is a per-chip data table, read at compile time, that records every size limit and every per-operation switch for that silicon.

24.1 HAL property table

The hardware-abstraction-layer table is the compiler's profile of a target, one packed structure that holds both the numeric limits and the feature gates for a chip. Table 24.1 gives representative scalar fields of the hardware-abstraction table, each with its offset, meaning, M1 value, and the generation at which it changes.

Table 24.1. Representative scalar fields of the hardware-abstraction table; the full decoded register map is in Appendix C.

Offset	Field	Meaning	M1 (H13)	Changes at
`0x1b8`	`max_operand_bytes`	on-chip SRAM working set	2 MB	constant across the line (1 MB on M9)
`0x1c0`	`dram_alignment`	DMA width granule in bytes	16	constant (1 only on the small profiles)
`0x1c8`	`l2_bank_align`	DMA bank-conflict modulo	64	constant
`0x1f0`	L2-resident buffer threshold	dedicated-buffer trip	0	32768 at A15, 262144 at A16
`0x200`	dense kernel-memory cap	non-streamed weight ceiling	64 KB	the fold-path budget
`0x210`	streamed kernel-memory cap	streamed weight ceiling	16 MB	the stream-path budget
`0x218`	instruction or segment alignment	record packing granule	256	16 at A14
`0x228`	`ne_perf_cycle_divisor`	cost-model per-cycle divisor	64	32 on H11, 16 on M9
`0x238`	`num_nes`	NE-core count	4 (base)	die-keyed: 4, 8, 16, 32, 64
`0x288`	extended dual-kernel-memory mode	16 MB versus 64 KB select	0	0 on all 28 targets

Offset	Field	Meaning	M1 (H13)	Changes at
0x70	max_large_conv_kernel_dim_z	3D-conv kernel depth	16	capability attested at A13 (1 below)
0x138	max_tensor_width	maximum tensor width	16384	65536 at A16
0x158	max_tensor_depth	maximum tensor depth	16384	1 below A13, 65536 at A16
0x3f0	reduction-via-transpose extent	reduction route threshold	192	384 at A15
0x400	pe_min_patch_width_log2	the $2^4 = 16$-pixel tiling floor	4	constant M1 and M5
0x580	cost-model policy name	roofline anchor string	Simple	None on older and small profiles
0x668	interchange-format map size	count of accepted image formats	3	13 at A14, 16 at A15, 14 at A16

The hardware abstraction layer is a single packed structure the compiler constructs for its target, holding two kinds of entry. The first is a block of scalar fields, indexed by byte offset, that record numeric limits: maximum kernel sizes, maximum tensor dimensions per axis, the on-chip working-set size, data-movement alignment granule, and cost-model curve. The second is a dense region of single-byte boolean flags, the capability bytes, each gating one operation or one format on or off for the target. A family of constructors, one per architecture, builds the structure per target inside the compiler. The scalar region runs from offset 0x18 to roughly 0x348 as plain data, with non-scalar members such as the format map and the cost-model curve extending past it. The capability-byte region occupies offsets 0x48f through 0x8cc and holds on the order of 165 single-byte flags, of which 24 have recovered field names and the rest are enumerated by offset and classified by the family that enables them. The compiler reads a scalar as a value at its offset and a capability byte as hal[offset] & 1, so every limit and every gate is one indexed read into this one table. The same structure holds the cost model: a policy-name string at 0x580, frequency-to-efficiency curve at 0x7a8, and per-cycle divisor at 0x228, which the roofline of chapter 18 reads from this table rather than from a separate file.

Listing 24.1 gives a partial C view of the structure, with each selected scalar field and the capability-byte region at its recovered offset.

Listing 24.1. A partial C view of the per-target hardware-abstraction structure, with selected scalar fields and the capability-byte region at their recovered offsets.

```
/* ZinIrHalParameters, selected fields at their byte offsets (M1/H13 values) */
struct ZinIrHalParameters {
    /* ... */
    uint64_t max_large_conv_kernel_dim_z;  /* 0x70:  3D-conv kernel depth   = 16  */
    /* ... */
    uint64_t max_tensor_width;             /* 0x138: max tensor width       = 16384 */
    /* ... */
    uint64_t max_operand_bytes;            /* 0x1b8: SRAM working set       = 2 MB  */
    uint64_t dram_alignment;               /* 0x1c0: DMA width granule      = 16    */
    uint64_t l2_bank_align;                /* 0x1c8: DMA/L2 bank count      = 64    */
    /* ... */
    uint64_t num_nes;                      /* 0x238: NE-core count          = 4     */
    /* ... */
    uint8_t  cap_bytes[0x8cd - 0x48f];     /* 0x48f..0x8cc: per-op capability flags */
};
```

The capability bytes are read one at a time as hal[offset] & 1, for example the texture engine at 0x81d and the kernel-streaming master at 0x48f.

24.2 Operation gate

Operation legality is declared not in the HAL table but on the operation itself, as a trait the compiler attaches to every backend operation. The trait is a minimum-family index: the operation `MinimumFamily<N>` is natively legal only inside a compilation whose family index is `N` or greater, and below that floor the compiler decomposes it into legal operations. The family index orders the generations: `A11Legacy` is 0, `A12` is 1, `A13` is 2, `A14` is 3, `A15` is 4, and so on, with the M1 at `A13` and the M5 at `A17`.

The check the compiler runs on each backend operation is the trait floor against the target family index, which listing 24.2 gives as the native-or-decompose decision.

Listing 24.2. The minimum-family gate, where an operation is emitted natively when the target family meets its floor and decomposed otherwise.

```
# mlir::OpTrait::anec::MinimumFamily<N>: native iff target family >= N
def op_is_native(op, target_family):
    return target_family >= op.minimum_family   # e.g. softmax N=2 (A13), sin N=4 (A15)

def lower_op(op, target_family):
    if op_is_native(op, target_family):
        emit_native(op)                          # one anec op
    else:
        decompose(op)                            # rewrite into ops legal below the floor
```

The M1 has family index two and the M5 has family index six, so an operation with floor four, such as sin, is native on the M5 and decomposed on the M1.

The floors fall into a small number of tiers. The base tier, family 0, holds the operations every engine runs: convolution, matrix multiply, pooling, the elementwise and activation set, reshape, transpose, and concat. At A13 come softmax, the normalizations, the reductions, fused attention, and the square-root and error functions. A14 brings the texture-engine samplers, crop-resize, and resample; A15 brings native sin and cos. No compute operation floors above A15, so the newest generations add core count and clock rather than new operations.

The two gate mechanisms work together. A capability byte read as `hal[offset] & 1` decides a route inside a single operation, for example whether the texture engine at byte `0x81d` is present, which on the M1 reads 0 and forces resize to a decomposition. The minimum-family trait decides whether the operation is native at all. When either gate is closed, the compiler either emits a decomposition into legal operations or rejects the operation with a message naming the architecture, depending on whether a legal decomposition exists.

Table 24.2 gives the minimum-family floors a developer reaches, each with the families it is native on and its representative operations.

Table 24.2. The minimum-family floors a developer reaches, with the families each is native on and representative operations.

Floor	Native on	Representative operations
F0	all families	convolution, matmul, pooling, elementwise, reshape, transpose, concat
F2 (A13+)	A13 onward	softmax, layer and instance and batch norm, reductions, attention, erf, sqrt
F3 (A14+)	A14 onward	crop-resize, resample
F4 (A15+)	A15 onward	sin, cos, global argmin and argmax

Because the floor is an attribute of the operation and the limits are a table keyed to the chip, the per-chip difference is data, not code. The compiler text that rewrites an operation is identical across the family, and the chip selects a different limit, gate, or decomposition strategy from the table beneath it.

24.3 Capability-byte gates across the line

A capability byte is a single-byte switch in that table that turns one operation or feature on or off for a target. Table 24.3 gives the named capability bytes, each with its gate and its value across the M1 and the later generations.

Table 24.3. The named capability bytes.

Byte	Gate	M1 (H13)	A14	A15	A16	A18
`0x48f`	kernel-streaming master, the 64 KB to 16 MB select	1	1	1	1	1
`0x494`	square-after-reduction fusion	0	1	1	1	1
`0x4a9`	dropout and random	0	0	1	1	1
`0x4f2`	global argmin and argmax	1	1	1	1	1
`0x529`	per-format kernel-stride enable, the palette stream	1	1	1	1	1
`0x52d`	fp8 E4M3 kernel format	0	0	0	0	1
`0x563`	FIFO-mode direct memory access	0	0	0	0	1
`0x815`	softmax, native	1	1	1	1	1
`0x816`	instance normalization, native	1	1	1	1	1
`0x81a`	local-response normalization, native	1	1	1	1	1
`0x81d`	texture engine	0	1	1	1	1

The texture engine at byte `0x81d` is the largest M1 functional gap: it reads 0 on the M1 and 1 from A14 onward. It gates resize, crop-resize, resample, affine transform, hardware gather, and symmetric padding all together, so each of those routes through a software decomposition on the M1. The fp8 byte `0x52d` is set on the A18 generation alone of the 28 targets, so the M5, an A17 part, does not have it. The streaming master at byte `0x48f` and the palette-stream byte `0x529` both read 1 on the M1, which is why the int4 palette and the sparse form stream on the M1, while int8 and blockwise fold, a mechanism chapter 25 develops. The compiler builds the table for a target by calling that target's constructor, so a single host recovers the table for every chip in the line whether or not it is the chip that is running.

24.4 Per-family scalar matrix

The scalar parameters across the generation anchors show the same pattern: a value holds for a span of generations and then steps once, as Table 24.4 gives across the generation anchors.

Table 24.4. The scalar parameters at the generation anchors.

Field (offset)	M1 (H13)	A14	A15	A16 (M4)	A17 (M5)
`num_nes` (`0x238`)	4	4	4	4	16
`max_operand_bytes` (`0x1b8`)	2 MB	2 MB	2 MB	2 MB	2 MB
`max_tensor_width` (`0x138`)	16384	16384	16384	65536	65536
`max_tensor_depth` (`0x158`)	16384	16384	16384	65536	65536
`max_large_conv_kernel_dim_z` (`0x70`)	16	16	16	16	16
L2-resident threshold (`0x1f0`)	0	0	32768	262144	262144
instruction alignment (`0x218`)	256	16	16	16	16
reduction-transpose extent (`0x3f0`)	192	192	384	384	384
interchange-format count (`0x668`)	3	13	16	14	14

The base-name M5 reads `num_nes` of 16 because the column is the 16-core Pro-class profile, while the base A17 profile has 4. The per-die sequence runs 4 for the base name, 8 for the `g` suffix, 16 for `s` and the legacy 16-core profile, 32 for `c`, and 64 for the `d` Ultra-class die.

24.5 Kernel-memory split

The streaming master byte does more than gate the compressed-weight stream: it selects which of two kernel-memory caps a layer's weights are sized against. The legalization check is two lines of logic, reading a streamable flag and the master byte to pick the offset of the cap, then comparing the demand against it, as listing 24.3 gives.

Listing 24.3. The kernel-memory split, where a streamable weight under the streaming master is sized against the 16 MB cap and a dense weight against the 64 KB cap.

```
# ExceedKmemSizeLimit: split-legalize a layer's weights when they exceed the cap
def exceeds_kmem(hal, demand, is_streamable):
    cap = hal[0x210] if (is_streamable and hal[0x48f]) else hal[0x200]
    return cap < demand   # 0x200 = 64 KB dense, 0x210 = 16 MB streamed
```

An ordinary non-streamed weight over 64 KB, or any weight over 16 MB, is thus split into multiple sub-layers on the M1, which raises the dispatch count and the compile time. A streamed compressed weight is sized against the 16 MB cap and has far more weight per layer. This is the weight path; it does not bound the activations, which stay within the maximum-tensor-dimension caps, so a layer with a tiny weight and a large activation is bounded by tiling cost in the partition passes rather than by this discrete limit.

24.6 Dead and family-gated fields

Not every per-target value in the table is a live gate. A byte-granular re-diff of all 28 target blobs leaves zero undecoded scalar fields, but several offsets that vary by family are populated by the per-chip builder and never read back through the table pointer, so their value is a write-only mirror. Five scalar offsets are dead as table fields in this fashion: the global element cap at `0x18`, kernel-depth constant at `0x80`, legacy tiling granule at `0x260`, offset at `0x320`, and die-class flag at `0x29c`. Each varies meaningfully by family, but the value a reader consumes is read off a different object that shares the byte displacement, a tensor-dimensions, compiler-parameters, or memory-pools structure, not the table. The distinguishing test is whether the base register at the access holds the table pointer, since the same displacement aliases dozens of other by-reference structures, so a raw displacement match inside a table-typed function is not proof of a table read.

The one offset that looks dead on the M1 but is not is the FIFO-mode byte at `0x563`: it reads 0 on the M1 and is read through the table pointer under a branch that is taken only when the byte is set, which happens on the A18 generation. A per-family value pattern alone does not establish a live gate; only a traced reader off the table base does.

24.7 Naming the remaining capability flags

The capability bytes and the scalar limits are both fields of one struct, `ZinIrHalParameters`, the per-family blob the compiler builds for its target. The struct has no per-field getter method, so the compiler reads a field through an inlined `ldrb` or `ldr` off the table pointer at a fixed offset, which is why a first pass recovers offsets and values but not names. The names survive in one place only. The compiler retains full mangled C++ symbols, and a reader function whose signature has `ZinIrHalParameters const&` reads each field, so the reader's name labels the field it reads. A read is attributed to the table only when the load's base register is the function's `ZinIrHalParameters const&` argument, since the same byte displacement aliases dozens of other by-reference structures.

Cross-referencing the unnamed offsets against these reader functions names 95 more of them, of which roughly 30 resolve to a precise individual meaning with the base register verified against the table argument, the recovered names Table 24.5 attributes each to its reader function.

Table 24.5. Precise capability-flag names recovered this round, each attributed to its reader function.

Offset	Field	Reader function
`0x4a8`	PE work-unit-shape supported	`PERasterization::ComputeWUShape`
`0x4ac`	small-source-mode compression supported	`ZinANELayer::AllowCompressionBasedOnSmallSourceMode`
`0x4b0`	non-power-of-2 work-unit width supported	`NERasterization::CanUseNonPowerOf2WUs`
`0x4f0`	preferred kernel layout format	`ZinIrKernel::GetPreferredKernelLayoutFormat`
`0x500`	transpose and multicast configuration	`ZinNELayer::FindValidMirInfoForTransposeCore`
`0x520`	secure-mode cache-hint DSID gate	`GetDSIDFromPriorityHalAndSecureMode`
`0x52c`	tensor-format support flag, pairs with the named `0x52d` fp8 byte	`ZinLayerValidationUtils::ValidateFormat`
`0x54c`	cache-prefetch kernel-task-interval limit	`ZinValidateTd<17>::ValidateCachePrefetchKernelTaskInterval`
`0x5a8`	cache-hint DSID value	`GetDSIDFromPriorityHalAndSecureMode`
`0x708`	reflective-padding maximum extent	`ZinValidateTd<20>::ValidateReflectivePaddingMode`
`0x748`	gather and texture-engine descriptor pointer	`ZinGatherLayer::CreateTELayer`
`0x8b4`	tile-height-errata threshold	`ZinTileHeightErrata::Workaround`
`0x8bc`	chaining enabled	`ZinIrRegAllocUtil::IsChainable`
`0x8e0`	kernel-caching enabled	`ZinIrTdValidationUtil::ValidateKernelCaching<N>`

The remaining 95-minus-30 additions are class-named: the reader identifies the subsystem the field gates without the exact semantics. Examples are the per-axis DMA range bounds read by `ZinValidateTd<N>::CheckInRangeDmaAccess` and the texture-engine plane-equation coefficients in the `0x820` to `0x8f8` block gated by the named `0x81d` texture-engine byte on A14 and later. Two candidates were rejected as table fields despite matching a displacement inside a table-typed function: `0xcf8` loads off an `adrp`-formed read-only

constant rather than the table, and `0x678` loads off a nested object two pointers deep. The same base-register test that found the five dead fields above also rules these out.

With this round the silicon-capability subset of the packed bitfield, the part the compiler reads to gate a feature per family, is fully named. The struct is `0x938` bytes, and the few entries inside it that are not capability flags are the cost-model coefficient block at offsets `0x580` through `0x7f0`. This block holds the frequency-to-efficiency curve, rate indices, performance multiplier that the roofline of chapter 18 reads, and about two soft fp64 coefficients. These are all performance coefficients rather than legality caps. A small number of capability fields are also true holdouts, read only off aliased bases. The offsets past `0x938` hold no table: an earlier reading that located an A12 operation-emulation catalog at `0xa30` through `0xe84` was a read into the adjacent zeroed memory beyond the struct, not a real field.

24.8 Attested is not reachable

A capability recorded in the HAL table attests support at the layer that reads the table. It does not by itself prove that the operation lowers to a task descriptor and runs on the silicon. These are distinct layers, and a capability present at the first can fail at the second.

The case that fixes the rule is three-dimensional convolution. The HAL scalar at offset `0x70` records a 3D-conv kernel depth of 16 on the M1, attesting that the kernel geometry is permitted, and the compiler frontend recognizes the operation. It still fails backend lowering on every device mask, returning the message that it is not supported on any backend. The capability is in the table and the operation does not run.

The gap appears in the other direction as well, where a checker accepts an operation the code generator rejects. On the M1 the top-k, sort, and dynamic-slice validators are all callable and all three are refused at code generation. A bit in the table, a frontend that recognizes an operation, or a validator that passes are each a claim about one layer; only a compile-and-run on the target confirms the operation at the layer that executes it. This is why the reachable surface of chapter 4 is smaller than the surface the table advertises, and why each native entry there was compiled and run on the M1 rather than inferred from a capability byte.

25 Compression internals

SUMMARY

Every weight blob has a format enum, an integer one through thirty-one, that keys five parallel helper tables controlling all packing; the palette range is 7 through 27 and the unity range is 28 through 30. The engine reconstructs four compression forms: per-tensor and per-channel int8 affine, int4 lookup-table, structured sparsity, and blockwise affine, each a distinct dequantization chain the converter decodes into a kernel format. Two gates decide whether a compressed weight streams in its compressed bytes or folds to a dense half-precision constant: the kernel-streaming master at offset `0x48f` and the per-format palette and affine cluster at `0x520` through `0x539`. On the M1 only the int4 lookup-table and structured-sparsity forms stream; int8 and blockwise fold to dense through the dequantize-to-dense path.

Chapter 7 stated which compressed weight forms reach the engine and which save bandwidth. The mechanism beneath that account follows: how a weight tensor becomes the compiled weight blob, what codec each form holds, and how the compiler decides whether the compressed bytes stream to the engine or fold to a dense half-precision constant first. The path runs from a high-level dequantization operation chain down to the per-output-channel-group records the firmware re-bases into on every dispatch. Bit-layout and address detail are M1/H13, table-descriptor codegen version five (family two, A13), unless another version or family is named.

25.1 Kernel-format enum and its tables

Every weight blob has a format enum that names its storage encoding, an integer in the range one through thirty-one. Five parallel tables keyed on that enum control all packing, and two structural ranges fall out of their guards and recur throughout the pipeline. Table 25.1 names the five kernel-format helper functions and what each one returns.

Table 25.1. The kernel-format helper functions keyed on the format enum and what each one returns.

helper	returns
`_ZinKernelFormatGetBitDepth(f)`	the index bit-width per element
`_ZinKernelFormatGetPaletteFormat(f)`	the palette sub-format, valid for `f` in 7 through 27
`_ZinKernelFormatGetUnderlyingType(f)`	the underlying scalar type, int8, uint8, e4m3, or half
`_ZinKernelGetPaletteLUTSize(f, n)`	the per-codebook byte stride times the codebook count
`ZinKernelFormatGetTypeno(f)`	the type number written into the descriptor

The palette formats are the enum range 7 through 27, selected by the guard `f - 7 < 0x15`. The unity formats, the identity and scale-only weights that skip the multiply array, are the enum range 28 through 30, selected by `f - 0x1c < 3`.

A compiled weight unit is a fixed set of sections, and the descriptor that names them is the authoritative field list. The descriptor holds an activation lookup table, palette lookup table, output-channel-group table, per-output-channel scale vector, per-output-channel bias vector, and the packed weight or index blob, the six-section field list listing 25.1 reproduces.

Listing 25.1. The per-output-channel-group kernel-unit descriptor, the authoritative six-section field list emitted into the compiled weight section.

```
t%u = s%lu
  lut:     ; activation LUT     (typeno, bitdepth, lut_bytes)
  pal:     ; palette LUT        (palette_format, bitdepth, num_luts, lut_bytes)
  ocgs:    ; output-channel-group table  (the per-OCG records)
  scale:   ; per-output-channel scale    (half or single precision)
  bias:    ; per-output-channel bias
  weights: ; the packed weight or index blob
```

The palette table tuple holds the palette format, index bit-depth, codebook count, and codebook byte size; a codebook count above one marks the vector-palettized case. The activation table and the palette table are peers in one descriptor and share a single on-chip-memory budget.

The element types the descriptor writes for a weight section are a subset of the 24-entry catalog. They are int8 as type 2, uint8 as type 3, half-precision as type 5, the lookup-table tag as type 8, 4-bit unsigned as type 9, the 8-bit floating form as type 16, and 4-bit signed as type 23. The half-precision type holds the codebook entries, scale, and bias, so a palette codebook supplies the multiply array with no further conversion.

25.2 Four codecs

Table 25.2 gives the four weight-compression codecs, each with its dequantization relation, on-device representation, the family it streams on natively, and the compiler gate that selects the stream.

Table 25.2. The four weight-compression codecs, each with its dequantization relation, native-stream family, and selecting compiler gate.

form	dequant relation	representation	streams natively on	gate
int8 per-tensor / per-channel affine	$w = s(q - z)$, $z = 0$ on M1	int8 byte stream plus per-channel half-precision scale	A14 and later	folds on M1 via the dequantize-to-dense path
int4 lookup-table	$w = \mathrm{LUT}[g/v][k][c \bmod v]$	four-bit index stream plus sixteen-entry half-precision codebook	M1/H13 and later	streaming master plus the kernel-stride enable
structured sparsity	scatter of packed values into the mask	one-bit mask plus packed half-precision nonzeros	M1/H13 and later	streaming master, as a separate mask-and-values operand
blockwise affine	$w = s_b\, q$ per block	int8 byte stream plus per-block half-precision scale	A15 and later	folds on M1 and M2 via the dequantize-to-dense path

Per-tensor and per-channel int8 use the affine form. The stored byte is dequantized as

$$w = s\,(q - z)$$

with quantized byte q, half-precision scale s, and zero point z, the encode being $q = \mathrm{round}(x/s) + z$. The scale is scalar or one value per output channel. The M1 forces the zero point to zero, so the form is symmetric and reduces to $w = s\,q$: the asymmetric setter hard-asserts on the version-five table descriptor, and the firmware invariant requires the asymmetric-quantization configuration bit clear.

The per-output-channel scale and bias do not cost a runtime operation: the compiler folds them into the weight coefficients at compile time and holds them in the descriptor's scale and bias arrays as per-output-channel vectors. The engine applies them at the gain-offset-control stage as $y = s_{\text{out}} \left(\sum_k w'_k x_k\right) + b_{\text{out}}$ where the coefficient already absorbs the dequantization scale. This is why a symmetric quantized convolution on the M1 costs nothing extra for the affine: the compiler folds the scale and the zero point into the stored weights and the per-output-channel arrays before the dispatch.

The int4 lookup-table form stores a four-bit index per element into a sixteen-entry half-precision codebook. Reconstruction is a table lookup with no arithmetic, since the codebook entries are already half-precision and supply the multiply array directly. The reconstructed weight is

$$w = \text{LUT}\big[g/v\big]\big[\,k\,\big]\big[\,c \bmod v\,\big]$$

with output-channel-group index g, palette vector size v, the four-bit index k for the element, and channel position c. For the common per-tensor case the vector size is one and a single codebook holds all sixteen entries, so the relation flattens to one table lookup. Four-bit weights have no affine path at all: there is no four-bit underlying scalar type in the type table, so a four-bit weight can only be a palette index. That is why the lookup-table form is structurally a palette and is the one form that streams on the M1.

The palette index width is 2, 4, 6, or 8 bits, but the M1-practical widths are 4-bit, a 16-entry codebook, and 8-bit, a 256-entry codebook. The 1-bit and 2-bit widths are rejected on the general path, and the 3-bit and 6-bit widths are version-gated to a later table descriptor, while the quantized-palette combination is capped below 256 entries. A legality mask constrains the vector size, the count of consecutive output channels that share one codebook, to the set $\{1, 2, 4, 5\}$. The format selector tests 2^v against the value `0x36`, which is `0b110110`, so vector sizes 0, 3, and any value of 6 or more are illegal. A 4-bit codebook is 32 bytes and an 8-bit codebook is 512 bytes per lookup table, and the palette lookup table and the activation lookup table share one on-chip-memory budget bounded by the per-target palette-lookup-table size field.

Structured sparsity stores a one-bit nonzero mask plus the packed half-precision values of the surviving nonzeros. The mask costs one bit per element and the values cost two bytes per survivor, so a weight that is half zeros or more stores well below its dense size. Reconstruction walks the mask, consuming one packed value for each set bit and emitting a zero for each clear bit, exact apart from the half-precision rounding of the kept values. Blockwise affine assigns a separate scale to each contiguous block of elements, finer than a per-channel scale and so lower in quantization error.

Listing 25.2 gives the three reconstruction codecs in pseudocode, each the exact relation the engine applies for the affine, lookup-table, and structured-sparsity forms.

Listing 25.2. The three reconstruction codecs in pseudocode, the affine dequantization, lookup-table reconstruction, and structured-sparsity scatter.

```
# int8 / uint8 affine dequant (constexpr_affine_dequantize)
def dequant_affine(q, scale, zero_point):     # zero_point forced to 0 on the M1
    return scale * (q - zero_point)           # symmetric M1 form: scale * q

# int4 lookup-table reconstruction (constexpr_lut_to_dense)
def dequant_lut(index, codebook, g, vector_size):
    return codebook[g // vector_size][index][g % vector_size]   # already fp16, no arithmetic
    # per-tensor case: vector_size == 1, one 16-entry codebook -> codebook[0][index][0]

# structured-sparsity scatter (constexpr_sparse_to_dense): 1-bit mask + packed fp16 nonzeros
def densify_sparse(mask_bits, nonzeros, n):
    out = [0.0] * n                           # mask bit 1 = keep (nonzero)
    j = 0
    for i in range(n):
        if mask_bits[i]:                      # LSB-first packed, one bit per element
            out[i] = nonzeros[j]              # consume the next packed fp16 value
            j += 1
    return out
```

The sparse layout on the wire is the bitmask blob followed by the packed values blob, the operand layout listing 25.3 gives.

Listing 25.3. The on-the-wire operand layout of the structured-sparsity form, the bitmask blob followed by the packed half-precision nonzeros.

```
constexpr_sparse_to_dense operand layout
  mask     : ceil(n / 8) bytes   # 1 bit per element, LSB-first, dtype code UINT1 (9)
  nonzeros : 2 * popcount(mask) bytes   # fp16 survivors in scan order, dtype code FP16 (1)
  # streamed size ~ (1/16 + density) x the dense fp16 size
```

25.3 Stream-versus-fold decision

Two cooperating gates decide whether a compressed weight reaches the engine in its compressed bytes or is materialized to a dense half-precision constant before the dispatch. The master gate is the kernel-streaming check. It returns false unless the hardware-abstraction-layer streaming master bit at offset `0x48f` is set, which holds from the A13 generation onward. It then requires the format be primary or convertible to half-precision by the direct-memory-access engine, plus unit stride on all axes, no dilation, no tile overlap, and an immutable weight: a mutable weight, a trained-in-place parameter, cannot stream.

A second gate guards the palette path. The native lookup-table form streams only when the weight is vector-palettized and the per-format kernel-stride-enable bit at offset `0x529` is set. When both hold, the compiler sets the palette-enable and stream flags and zeroes the fold sub-fields. On the M1 the offset `0x529` bit is set, so the palette stream is live while the other formats fold.

Structured sparsity reaches the engine through the master gate as a separate operand rather than through the palette path. The sparse weight lowers to a mask producer and a nonzero-values producer held as their own blobs. Those stream under the master at offset `0x48f`, which is set on the M1, not under the per-format palette and affine cluster at offsets `0x520` through `0x539`, which is clear on the M1. A live byte comparison confirms the form: the sparse weight blob stores the one-bit mask and the half-precision nonzeros at about 0.43 times the dense size, with no dense half-precision constant present, whereas a fold would show a single full-width half-precision blob.

The int8 and blockwise forms have no streamed encoding on the M1. The converter builds the native quantized kernel, but the lowering routes it through the dequantize-to-dense path, which reconstructs the weight to a dense half-precision constant before the direct-memory-access transfer. The stored bytes are then plain half-precision and move at full width, so a weight-streaming-bound layer gets no bandwidth gain from these forms on the M1.

A budget relaxation accompanies the stream decision. A streamed weight is sized against the relaxed on-chip-memory cap at offset `0x210`, while a folded or dense weight is sized against the dense cap at offset `0x200`, which is sixty-four kilobytes on the M1. The compressed path thus has far more weight per layer.

25.4 Per-output-channel-group packing

The weight section is a sequence of per-output-channel-group records, and each record is the fixed 14-word block Table 25.3 details word by word with its computed offset and size and its ten unrelocated address slots.

Table 25.3. The per-output-channel-group record, the 14-word block with its computed offset and size and its ten unrelocated address slots.

Word	Field	Initial value
`[0]`	output-channel-group byte offset into the section	computed
`[1]`	output-channel-group size, the channel count for this engine	computed
`[2]` through `[0xb]`	ten address relocation slots	`-1`
`[0xc]`, `[0xd]`	reserved flags	0

The ten slots left at `-1` are the per-record device addresses the loader patches in: the input and output tile bases, per-buffer coefficient sub-buffer bases, and four coefficient-stream bases for bias, post-scale, palette lookup, and activation lookup. Those four streams map one-to-one to four kernel direct-memory-access sub-channels in the kernel-and-common register group, each with the enable, base-offset, and relocation register Table 25.4 gives.

Table 25.4. The four kernel direct-memory-access coefficient streams, each with its enable, base-offset, and relocation register.

Descriptor slot	Enable register	Base-offset register	Relocation register
bias	`0x5548`	`0x554c`	`0x1554`
post-scale	`0x5558`	`0x555c`	`0x1558`
palette lookup	`0x5568`	`0x556c`	`0x155c`
activation lookup	`0x5578`	`0x557c`	`0x1560`

There is no zero-point stream, since the M1 symmetric form folds the zero point to zero; the per-output-channel scale and bias streams supply the dequantization scale instead. The M1 replicates the kernel coefficients per engine core, one copy per core, because it lowers to the per-core layout rather than the shared kernel-memory layout that the A14 generation and later use. A four-core M1 thus has four per-core records each with its own output-channel-group vector.

The activation lookup table shares the descriptor and the budget with the palette lookup table. It is a 43-entry half-precision record: two input-clamp values, a mode flag, a scale, the 33 knot values uniform in the input domain, four tail extrapolation coefficients, and a packed mode word. At runtime the input maps affinely onto the 32 segments between the 33 knots, the integer part selects a segment, and the fraction controls a half-precision linear interpolation, with the output clamping to the end-knot beyond the input-clamp domain.

The activation lookup table and the palette lookup table are co-located in the weight section and located by one offset cursor, which is why they draw on one shared budget.

25.5 Sparsity datapath

The engine has two independent sparsity mechanisms. The first is compute-time zero-skip. The table descriptor has a detect-zeros bit that is implemented on the M1, and when it is set the multiply array skips a multiply-accumulate whose weight is zero. The cost model scans the weight values for their zero density once, caches the ratio, and reduces the convolution cycle estimate by it, adding the implicit zeros that strided and padded convolutions contribute. This mechanism is format-independent: it fires on any kernel with zeros regardless of storage encoding, including weights that fold to dense for storage. On a bandwidth-bound stack the zero-skip alone moves the limit by about one percent, because skipping multiply-accumulates does not help a layer limited by the weight stream rather than by the array.

The second is the sparse-binary store with on-chip decompress, which is the source of the bandwidth gain. The mask and the packed nonzeros cross the direct-memory-access stream, and the engine decompresses them on chip into the dense tile. The on-device kernel-configuration register holds a single packed sparse-format field beside the palette-enable and palette-bits fields in the same word. A six-by convolution stack at sixty-three percent zeros streams the sparse form at about 0.43 times the dense weight bytes and runs 1.55 to 1.64 times faster than the same weights stored dense. Effective bandwidth rises from about 29 to about 48 gigabytes per second, which is the M1 weight-stream ceiling. The compiled program is byte-identical between the dense and sparse bundles at 2416 bytes, so the difference is entirely in the streamed weight payload and its descriptor, which is the signature of a native stream rather than a fold or a changed program.

The version-five table descriptor asserts that sparse-binary mode is not supported, and this assert reconciles with the live stream because it governs a different path. That assert bounds the sparse packing of the palette-index plane inside the compiled weight blob, reachable only through the vector-palette flag, so a non-palettized weight never reaches it. The streamed sparse weight is the separate mask-and-values operand described above, which does not pass through that table-descriptor bit.

25.6 fp8 datapath

The compiler has a complete fp8 datapath that no M1 generation can reach. The 8-bit floating form is two distinct things in the binary: the E4M3 form, a gated hardware weight and activation format with element code `0xc`, and the E5M2 form, a conversion format with element code `0xd`. E4M3 is the gated one: its kernel-format validity admits it as a native weight element type, but the encoder that writes its element code exists only from the A17 generation. The runtime capability byte at `0x52d` that enables the format is set on the A18 generation alone. On the M1 the format register is two bits wide with no E4M3 codepoint at all, so passing the format triggers a compile-time assert rather than a runtime refusal. The accumulator stays at the wide half-precision-class width on every family, so the 8-bit form is an input and storage width supplying the same multiply array, not an accumulate width. Its throughput runs on the same double-multiply path the int8 form uses. The full fp8 datapath, format-register delta, E5M2-to-half fold asymmetry, and conversion functions are decoded in chapter 36.

25.7 Codegen version matrix

The same kernel format produces different weight bytes per table-descriptor version, which is why the M1 and the M5 differ at the bit level rather than only in scalar limits, as Table 25.5 records capability by capability.

Table 25.5. Codegen capability per table-descriptor version: the M1 version-five descriptor lacks the sparse-binary, multi-codebook, and asymmetric encoders.

Capability	M1 version-five descriptor	Later descriptor
sparse-binary index packing	asserts unsupported	a packed flag at descriptor offset `0x424`
multi-codebook palette	asserts unsupported	implemented
asymmetric quantization	asserts unsupported	implemented
zero-skip detect	implemented	implemented
palette enable and bit width	implemented, 4 and 8 bit	implemented, plus 3 and 6 bit

The M1 legal kernel-format space is thus the intersection of the format enumeration and the version-five descriptor: dense half-precision, palettized 4-bit and 8-bit including the single-codebook vector form, symmetric int8 that folds to half-precision, and the unity forms. The sparse-binary index packing, multi-codebook palette, asymmetric quantization, fp8 form, and 3-bit and 6-bit palette widths are absent from the version-five codegen entirely, a missing encoder rather than a runtime refusal.

25.8 Winograd and the compressed-weight interaction

The full Winograd eligibility gate is derived in chapter 20: the enable bit, kernel-axis and tile-size conditions, and the OCG $\times K_y \times K_x \times K_d \times 2$ work threshold against its non-float, float, and packed floors. The compression-relevant point is the weight-format interaction: eligibility requires a non-unity, non-sparse weight, so a sparse or unity-format weight is excluded from the Winograd path and keeps its own datapath. The transform matrices G, $B^\top$, and $A^\top$ are resident in the array rather than stored in any shippable weight blob, so no codec holds them.

26 Hidden layers and direct netplist authoring

SUMMARY

The compiler validates and lowers a native catalog of 45 hardware layer descriptors, and the model converter surfaces only the subset its public operation set covers. Authoring the network description directly reaches the rest, including fused attention, ranking, spatial rearrangement, and geometry layers, each cutting its own dispatch segment. Authoring does not bypass a family gate: the texture-engine samplers and whole-tensor arg-min and arg-max are accepted from the A14 and A15 and rejected on the M1, and a layer is confirmed only by a compile-and-run on the target.

The engine implements more layer kinds than the high-level conversion path emits. Authoring the network description directly reaches the rest; this chapter gives the method, names the classes it reaches, and marks the ones a later chip accepts and the M1 rejects.

The machinery this rests on is decoded in full in the back half: the `.espresso.net` program format in chapter 23, compiler's layer parsers and validators in chapter 22, and per-family capability gates in chapter 24. Here it is enough that the format is hand-authorable and that each layer has a validator and a family gate; Appendix B is the full per-layer schema.

26.1 Framework and native catalogs

The compiler has two parallel layer catalogs. The framework-level set is the engine-agnostic kernel abstraction, about 190 layer types, of which some run only on the host or the GPU. Below it is the native hardware set: 45 descriptor structs, each with a constructor named `_ANEC<Name>LayerDescInitialize` and a checker named `_ANECValidate<Name>Layer`, and this set is the list of operations the engine silicon runs.

The conversion path from a trained model walks the documented intermediate-language operation set [AppleCoreMLTools] and emits only the native layers that set maps onto. Several native descriptors have no operation in that set, so the converter never produces them, and a model passed through it decomposes the work into the operations it does know. Fused attention is the worked case: the converter always splits scaled dot-product attention into a matrix multiply, scale, softmax, and second matrix multiply, because its operation set has no atom that lowers to the single fused descriptor. The descriptor and its validator are present in the compiler the whole time. Reaching them means handing the compiler a network description that names the layer directly, rather than one produced by decomposing a model.

26.2 Method

A network expressed for the compiler is a property list, the `.espresso.net` representation the compiler accepts alongside the intermediate language. Its layers are dictionary entries the compiler calls Units, each holding a `Type` tag, a `Bottom` wiring list, an output type, and a `Params` sub-dictionary of typed attributes, all under a `ProcedureList` of callable entry points that name the `InputList`, `OperationList`, and `OutputList`. The representation is hand-authorable, which is what makes the hidden layers reachable.

The reusable method has four steps. Author one Unit whose `Type` is the native layer name and whose `Params` hold the descriptor attributes that layer's parser expects. Supply the wiring and any constant weight blobs the layer needs. Compile the description through the runtime, which lowers the Unit to its `ANEC<Name>LayerDesc`, runs the matching validator, and assigns it to an engine. Then load, bind buffers to the named ports, and dispatch, exactly as chapter 6 gives for any compiled program. A raw native descriptor enters the network without re-expressing it as a framework kernel through the tunneled-unit path: the path passes the descriptor through the framework layer untouched, with its float16 and integer operands intact, so a `Type` of `SDPA` or `CostVolume` reaches the silicon directly.

The descriptor attributes are read out of the compiler's per-layer parsers, the `ZinParse<Name>Unit` routines, and cross-referenced constant-string tables. A required key the parser does not find raises a parse error such as `InvalidParamSyntax`, so an authored layer either has the exact attribute set the parser expects or fails at compile time rather than at dispatch.

26.3 Classes reached this way

The reachable native layer kinds fall into several groups. Fused attention is the scaled dot-product attention descriptor, four or five operands of query, key, value, and constant scale, with an optional additive mask as the fifth that holds the causal and decode cases as data. Ranking and selection cover the top-k, sort, and argument-minimum-and-maximum descriptors, including the whole-tensor argument form, with their integer index outputs returned float16-encoded and exact for the index ranges these layers produce. Spatial rearrangement holds the pixel-shuffle and pixel-unshuffle pair, channel-and-space pair, and space-and-batch pair, each parameterized by per-axis integer factors and distinguished by channel-ordering convention rather than being aliases. Three further normalizations appear: the range normalization that maps a tensor to its minimum-to-maximum span, local response normalization, and per-channel affine gain-offset control in static and runtime-tensor forms. Geometry and point-cloud work draws on the template cross-correlation, the three-vector cross product, furthest-point sampling, radius neighborhood search, and the stereo cost volume. Data movement completes the set with the re-strided input view, runtime-offset dynamic slice, and tile and concatenate descriptors. The full catalog of these layers, with each layer's `Type` tag, its descriptor, and its `Params` schema, is Appendix B.

26.4 A native layer descriptor

The fused-attention descriptor is the clearest illustration, because the high-level path never emits it and the validator pins its shape exactly. Listing 26.1 names the four-or-five operand contract and the one attribute its parser reads.

Listing 26.1. A hand-authored fused-attention Unit naming the native scaled dot-product attention descriptor in the network description.

```
Unit "attn" {
  Type    = "SDPA"
  Bottom  = [ "q", "k", "v", "scale" ]   # optional 5th: additive mask
  Params  = { SubtractMax = true }        # ANECSDPALayerDesc byte 0x00
  OutputType = "Float16"
}
```

The validator enforces the operand count with `4 or 5 bottoms must be present for SDPA`, requires the key and value to share a shape, and checks that the query times the transposed key contracts against the value. The `SubtractMax` attribute is the single key the attention parser reads, and it defaults to false in the descriptor constructor, which is numerically wrong for softmax, so an authored attention Unit must set it true. The optional fifth operand is an additive float16 mask broadcast over heads, zero on and under the diagonal and a large negative bias above it for the causal case.

At the backend-dialect level the same fused operation is one atom with a parametric contraction, the same form the matrix multiply uses, as listing 26.2 gives.

Listing 26.2. The fused attention atom at the backend-dialect level, a single parametric contraction that covers the matrix-multiply, softmax, and transpose path.

```
anec.sdpa(%q, %k, %v, %scale)   // 4 or 5 bottoms; covers matmul + softmax + transpose
anec.matmul(%lhs, %rhs) { transpose_lhs, transpose_rhs }   // depth D must be 1 on both operands
```

The attention atom covers the matrix-multiply, softmax, and transpose path and is not gated behind the texture engine, which is why it runs on every family from the M1 onward.

26.5 Authoring a hidden layer

A native layer reaches the silicon by naming its descriptor in the graph directly, then compiling and dispatching it as any other program, with the target chosen as the first family that runs the layer. The graph names the native layer kind directly, and the compile step runs the layer's validator and assigns it to an engine, failing at compile time where the family gate rejects it, as listing 26.3 walks step by step.

Listing 26.3. Authoring a hidden layer through the bridge route, then compiling and dispatching to confirm it lowers and runs.

```
# Author a hidden layer via the bridge route: name the native layer kind and its parameters,
# cut it into the graph as a bridge node, then compile and confirm it lowers.

graph G:
    input q : [1, 8, 197, 64] fp16
    input k : [1, 8, 197, 64] fp16
    input v : [1, 8, 197, 64] fp16
    const scale

    # Describe the native layer directly as a bridge node, passed through untouched to the silicon.
    bridge_node attn:
        kind    = "SDPA"                      # the native descriptor name, not a decomposition
        inputs  = [ q, k, v, scale ]          # 4 operands; optional 5th mask is the causal case
        params  = { subtract_max = true }     # required: the default is false, wrong for softmax

    output attn

program P = compile(G, target = H13)   # runs the validator and assigns it to an engine
# Fused attention runs from the M1 onward (not texture-gated). A family-gated layer would fail here,
# at compile time, below its minimum family.
output = dispatch(P, q_data, k_data, v_data)  # confirm it lowers and runs on the target
```

An authored layer is confirmed only by this compile-and-run on the target, not by the presence of its descriptor, since a gated layer fails the code generator below its minimum family.

26.6 Arch-gated negatives

The same family gates as chapter 12 accept some authored layers on a later chip and reject them on the M1. A native descriptor exists in the compiler binary on every target, but its validator has a minimum-family trait, and below that family the code generator rejects it.

The texture-engine samplers are the largest group: resize as a hardware sampler, crop-and-resize, grid resample, and the affine spatial transform are accepted from the A14 generation and rejected on the M1,

where the compiler reports that the affine transform is not supported on this architecture. The whole-tensor argument-minimum-and-maximum layer is gated to the A15 generation and rejected on the M1. Range normalization is arch-gated and rejected on the M1. The native circular state layers behind an on-device key-value cache are hard-gated on the M1 and reached there only through a shared resident buffer. Authoring the Unit does not bypass the gate: the layer compiles where its family allows and fails at compile time where it does not, so the same network description targets the generation that first runs the layer and every generation above it.

A second class of rejection is not about family but about a layer that passes an earlier check and fails the code generator, the attested-is-not-reachable rule from chapter 4. On the M1 the top-k, sort, and dynamic-slice validators are all callable, yet the code generator rejects sort and dynamic-slice and accepts top-k only outside a small forbidden parameter band.

26.7 Two compile routes and their gates

The same operation can have different availability on the same chip depending on which route reaches it. The conversion route from a high-level model is gated by a minimum-family trait on each operation, the floor the public conversion path checks. The direct-authoring route is gated instead by the per-chip hardware-abstraction feature bytes the layer validators read. The whole-tensor argument-minimum-and-maximum layer is the clearest example: the conversion route floors it above the M1, yet the direct-authoring route gates it on a feature byte that is set from the A13 onward, so it is rejected through conversion on the M1 and runs through direct authoring on the same chip. Trigonometric sine and cosine have no direct-authoring bridge, so they stay conversion-only and reject on the M1; sort and top-k have a bridge but the code generator rejects it on the M1.

26.8 Reference: the native layer classes reached by direct authoring

The native catalog is the set of layer kinds the engine silicon runs, each with a constructor and a validator in the compiler. The classes Table 26.1 gives are the ones the conversion path does not emit and direct authoring reaches, each with the binding validator gate and the first family that runs it.

Table 26.1. The native layer classes direct authoring reaches; the full per-layer descriptor and parameter schema is in Appendix B.

Class	Native layers	Binding gate	First family
Fused attention	scaled dot-product attention	four or five operands; key and value share a shape; the subtract-maximum attribute must be set	M1, not texture-gated
Ranking and selection	top-k, sort, argument-minimum-and-maximum, whole-tensor argument form	index outputs returned float16-encoded; sort and top-k pass the validator and fail the M1 code generator	whole-tensor form A15; top-k and sort code-generated above the M1
Spatial rearrangement	pixel-shuffle, pixel-unshuffle, space-and-channel, space-and-batch	per-axis integer factors that factor into {2, 3, 4, 8}; depth factor one	M1
Normalization	range normalization, local response normalization, per-channel gain-offset control	range normalization arch-gated; gain-offset in static and runtime forms	range normalization above the M1; gain-offset M1
Geometry and point cloud	template cross-correlation, three-vector cross product, furthest-point sampling, radius neighborhood, stereo cost volume	cross product requires interleave one and float16 operands; cross-correlation template depth one	M1 for the cross and correlation forms

Class	Native layers	Binding gate	First family
Texture samplers	resize as a hardware sampler, crop-and-resize, grid resample, affine transform	the texture-engine feature byte	A14; rejected on the M1
Data movement	re-strided input view, runtime-offset dynamic slice, tile, concatenate	input view must follow a reshape; dynamic slice passes the validator and fails the M1 code generator	M1 for the static movers
Streaming state	live state, ring-buffer reader and writer, tensor-to-buffer movers	the ring-buffer writer must connect to a live-state buffer; circular mode arch-gated	above the M1; reached on the M1 only through a shared resident buffer

26.9 Reference: the descriptor lowering of a fused operation

Table 26.2 lists the descriptor lowering of four representative atoms, each with its operand contract and the binding constraint its validator enforces.

Table 26.2. The descriptor lowering of four representative atoms, with the operand contract and the binding constraint each validator enforces.

Atom	Operands	Binding constraint
`sdpa`	four or five: query, key, value, scale, optional mask	key and value share a shape; query times the transposed key contracts against the value; softmax uses the family path
`matmul`	two	depth one on both operands; output channel equals the left channel; contraction parametric through the transpose flags
`gain_offset_control`	one, plus scale and bias	per-channel affine, height one; the fold target for a bias or activation
`layer_norm`	one, with gamma and beta folded	channel divisible by the group count; output type float; the grouped form is the same atom with a group count above one

26.10 Programmable activation LUT

The named activations are not distinct hardware. The compiler synthesizes each one, sigmoid, tanh, gelu, swish, and the rest, into a 33-knot piecewise-linear table over a fixed domain, and the engine evaluates that table. The format is recovered byte for byte, 33 knot values at a fixed step and 32 inter-knot deltas behind a short header, and the operation set includes a custom-table opcode, `kZinIrNonLinearCustomLUT`, that runs an arbitrary pointwise function the same way.

The raw custom-table path is not reachable from a netplist. The unit parser requires a saturation set and a version-specific set together, and a consistency check in the same routine then rejects their coexistence, so no authored table satisfies both and every attempt fails to compile. The capability is real but sits below the user artifact: the firmware synthesizes the table from the program at load.

An arbitrary pointwise function still runs on the engine by composition. A piecewise-linear curve over chosen knots is a `linear`, `relu`, `linear` chain, the same form the hardware table evaluates, so a small rectifier basis reproduces any knot table to fp16 exactly, demonstrated on a Gaussian bump that no named activation provides. The engine's exposed model is thus a linear map, a pointwise nonlinearity, and a linear map: custom weights and any scalar function, but not a new arithmetic primitive, a custom reduction, or data-dependent control flow.

Part VIII

System Internals

27 Kernel driver and IOKit ABI

The user clients, the selectors, and the IOKit ABI of the coprocessor endpoint.

28 Address translation and the DART

The engine IOMMU, the leaf page-table entry, and the firmware rebase.

29 Firmware

The real-time controller, its task model, and the dispatch loop.

30 Host-to-firmware command protocol

The ninety-three-command mailbox protocol across the host boundary.

31 Power and thermal

Idle and active draw, the credit sequence, and sustained thermal behavior.

32 Security and isolation

The signed-load chain, secure mode, and the exclave path on the M1.

33 Telemetry and hardware counters

The counter block, the stats-mask gate, and the free-running timestamp.

27 Kernel driver and IOKit ABI

SUMMARY

The engine is reached through a single kernel driver and a flat user-client selector ABI, with 17 control-client selectors and 9 direct-path selectors, each pinned to one exact size tuple. Opening either user client is gated on the kernel entitlement `com.apple.ane.iokit-user-access`, which exactly two system binaries hold, so every application reaches the engine through a privileged broker daemon. Selector 2 alone reaches a hardware doorbell from user space, through a four-layer call path ending in an MMIO mailbox write. The size tuples match across the M1 and M2-class kernel cache, so the ABI at this layer is family-invariant.

This chapter covers the driver class hierarchy, the two user-client selector spaces with their exact struct sizes, the dispatch-array record format the kernel validates against, the path from a user-space call down to the hardware doorbell, and the broker model that fronts the device with one privileged daemon.

27.1 Driver stack and class hierarchy

The three cooperating kexts, their bundles, principal classes, and provider matches appear in Table 27.1.

Table 27.1. Bundle, principal class, and provider match for each of the three kernel extensions.

Kext role	Bundle	Principal class	Provider match
interface and hardware driver	`AppleH11ANEInterface`	`ANEHWDevice` (registered as `H11ANEIn`)	`RTBuddyService`, role `ANE`
per-die abstraction layer	`AppleT8132ANEHAL`	`AppleT8132ANEHAL`	`IOResources`
multi-engine arbiter	`AppleANELoadBalancer`	`ANEDriver` (`H1xANELoadBalancer`)	`IOResources`, `IOKit`

Three version-locked kexts cooperate, all at build `9.511.3` on the M1 generation. The interface and hardware driver registers the device, vends the user clients, and rings the firmware. A per-die hardware abstraction layer supplies the clock, power, and topology constants. The load balancer owns the program-to-engine residency map and arbitrates across physical engines on parts that have more than one. The engine attaches as an Apple RTKit coprocessor endpoint, so the interface driver is the host side of a real-time-operating-system mailbox client. Three kernel families handle the work: the address-translation family that drives the device IOMMU, surface family that backs zero-copy tensor buffers, and real-time mailbox family that holds firmware commands.

`ANEHWDevice::newUserClient(task*, void*, uint type, IOUserClient**)` vends the two distinct user clients plus a hint-only client of Listing 27.1.

Listing 27.1. The three user-client types, by requested type.

```
/* ANEHWDevice::newUserClient vends, by requested type: */
H11ANEInUserClient        /* control client: program lifecycle + the inference hot path */
H11ANEInDirectPathClient  /* direct path: enqueue, memory-map, session-hint            */
ANEClientHints            /* scheduling-hint client (setClientHint)                    */
```

The two functional clients hold separate connections and separate selector spaces that both start at zero, so one small selector integer names different methods on each. Each stores its connection at object offset `+0x40`, and every selector call loads that offset before issuing the kernel call.

On the user-space side an `IOServiceOpen` whose return type is `0xe00002c5` opens the device, and the connection object is at offset `0x40` in the device handle. Two front-ends open that type: `ANEServicesDevic e` for inference and `ANEHWDevice` for administration. The user-space selector immediates read on the M1 are selector 0 for the device open, selector 2 for the send-request, selector 3 for create, and selector 4 for prepare. The rest are selector 6 for destroy, selector 10 for the version query, and selector 16 for the firmware load. The lifecycle indices differ from the kernel-registered table because the kernel has two user-client classes with independent selector tables. The two selectors agree across both sides, selector 0 opening the device at 104 bytes and selector 2 sending a request with a 2376-byte input and a 40-byte output.

Every selector shim reads its arguments out of the framework `IOExternalMethodArguments` block at the fixed set of offsets in Listing 27.2, identical across all 26 selectors.

Listing 27.2. The IOExternalMethodArguments field offsets every selector shim reads.

```
/* IOExternalMethodArguments field offsets, used by every selector shim: */
+0x08  asyncWakePort       /* mach completion port (async selectors only)        */
+0x10  scalarInput[]       /* also holds the 0x20-byte asyncReference block       */
+0x20  scalarInputCount
+0x30  structureInput      /* pointer to the typed argument struct                */
+0x38  structureInputSize
+0x48  scalarOutput[]
+0x50  scalarOutputCount
+0x58  structureOutput
+0x60  structureOutputSize
```

Three return constants recur across the shims: `0xe00002c2` is `kIOReturnBadArgument`, returned on a size or null-pointer check failure, `0xe00002c7` is `kIOReturnUnsupported`, returned on a disabled or stub path, and `0xe00002c5` is the closed-or-closing client state.

27.2 User-client dispatch-array format

Each user client routes a selector through the standard IOKit 2022 dispatch pattern. `H11ANEInUserClient ::externalMethod` loads the dispatch-array pointer and the method count, then tail-calls the framework dispatcher, which validates the declared scalar and structure sizes against the array entry before it calls the named handler. Both dispatch arrays were read byte for byte out of the read-only data section of the kernel cache, and Listing 27.3 gives the disassembled selector dispatch for each user client with its array pointer and selector count.

Listing 27.3. The disassembled selector dispatch for each user client, with its dispatch-array pointer and selector count.

```
/* H11ANEInUserClient::externalMethod, disassembled (M1, T6000): */
add x3, x3, #0xc08    /* x3 = &_sANEDriverClientMethods           */
mov w4, #0x11         /* count = 17 selectors (0..16)             */
bl  IOUserClient2022::dispatchExternalMethod

/* H11ANEInDirectPathClient::externalMethod: */
add x3, x3, #0xeb0    /* x3 = &_sANEDriverDirectPathClientMethods */
mov w4, #0x9          /* count = 9 selectors (0..8)               */
bl  IOUserClient2022::dispatchExternalMethod
```

Each array element is one `IOExternalMethodDispatch2022` record at a 40-byte stride, packing the authenticated handler pointer and the four size checks the dispatcher enforces, shown in Listing 27.4.

Listing 27.4. The dispatch-array element layout, packing the handler pointer and the four size checks the dispatcher enforces.

```
/* IOExternalMethodDispatch2022 element, 40-byte stride, field offsets: */
struct IOExternalMethodDispatch2022 {
  void *function;                  /* +0x00  pointer-authenticated handler   */
  uint32_t checkScalarInputCount;  /* +0x08  exact scalar-input count        */
  uint32_t checkStructureInputSize;/* +0x0c  exact struct-input bytes        */
  uint32_t checkScalarOutputCount; /* +0x10  exact scalar-output count       */
  uint32_t checkStructureOutputSize;/* +0x14 exact struct-output bytes       */
  uint8_t  reserved[0x10];         /* +0x18  reserved (debug-WP group flag)  */
};
```

No entry on either client uses the sentinel `0xffffffff` that means “do not check”. Every selector pins an exact scalar count and an exact struct size, and the dispatcher rejects any other size with `kIOReturnBadArgument`. Size-based overloading does not exist here: each selector index has exactly one record with one fixed size tuple.

27.3 Control-client selector table

The control client has 17 selectors covering the open handshake, the program lifecycle, status and version reads, and the firmware-driven debug work-processor channel. Table 27.2 lists every control-client selector with its handler and kernel-authoritative sizes read from the dispatch array.

Table 27.2. The seventeen control-client selectors, with handler name and kernel-authoritative scalar and structure sizes.

Sel	Handler	Scalar in	Struct in	Scalar out	Struct out
0	`ANE_DeviceOpen`	0	104	0	104
1	`ANE_DeviceClose`	0	0	0	0
2	`ANE_ProgramSendRequest`	1	2376	0	40
3	`ANE_ProgramCreate`	0	32	0	0
4	`ANE_ProgramPrepare`	0	56	0	56
5	`ANE_ProgramUnprepare`	0	56	0	0
6	`ANE_ProgramDestroy`	0	16	0	0
7	`ANE_GetStatus`	0	0	0	32
8	`ANE_ProgramCreateInstance`	0	32	0	0

Sel	Handler	Scalar in	Struct in	Scalar out	Struct out
9	ANE_ProgramChainingPrepare	0	16	0	24
10	ANE_GetVersion	0	0	1	0
11	ANE_RegisterDebugWorkProcessor	0	24	0	0
12	ANE_UnregisterDebugWorkProcessor	0	0	0	0
13	ANE_GetDebugWorkProcessorItem	2	0	0	0
14	ANE_CompleteDebugWorkProcessorItem	2	0	0	0
15	ANE_ReleaseDebugWorkProcessorBuffers	0	0	0	0
16	ANE_LoadFirmware	3	0	0	0

Selector 0 is the open handshake. It passes a 104-byte device-info structure as both the input and the output buffer, echoes the caller header back, fills the output half with the device descriptor, and returns the session token at offset +0x00 that every later request holds, with the handshake fields given in Table 27.3.

Table 27.3. The ANEDeviceInfo handshake structure passed in and echoed back by selector 0.

Offset	Input (client to kernel)	Output (kernel to client)
+0x00	usage type byte (1 standard, 2 unsupported)	program / session token (u64)
+0x08	callback function pointer; board id 0x1111222233334444	echoed
+0x10	receiver context pointer	echoed
+0x18	timeout 0x2710 = 10000	echoed
+0x48	(output only)	ANE version 0x20 = 32, 256
+0x50	(output only)	number of engines = 1
+0x60	(output only)	CPU subtype = 4

The usage-type byte selects the standard client profile: usage 1 opens, usage 2 returns the unsupported code 24. Selector 16 is inactive on this build: its shim returns kIOReturnUnsupported unconditionally, and the real firmware load runs internally at driver start.

A compiled program reaches the kernel in one of two representations: ANEProgramLegacyResource, a loader for the program-image executable, and ANEProgramRTResource, a runtime op-graph variant.

27.4 Direct-path selector table

The direct-path client has 9 selectors, listed in Table 27.4 with their handlers and sizes. Selectors 0, 1, and 2 reuse the control client's handler functions; the remaining six are the enqueue, memory-map, and session-hint methods of the low-latency submission model.

Table 27.4. The nine direct-path selectors with their handler names and scalar and structure sizes; the full reference is in Appendix C.

Sel	Handler	Scalar in	Struct in	Scalar out	Struct out
0	ANE_DeviceOpen	0	104	0	104
1	ANE_DeviceClose	0	0	0	0
2	ANE_ProgramSendRequest	1	2376	0	40
3	ANE_ProgramOutputSetEnqueue	0	40	0	0
4	ANE_ProgramInputsReady	0	3104	0	0
5	ANE_MemoryMapRequest	1	2080	1	0
6	ANE_MemoryUnMapRequest	0	2080	0	0

Sel	Handler	Scalar in	Struct in	Scalar out	Struct out
7	ANE_SessionHintRequest	0	16	0	24
8	ANE_ProgramChainingSetActiveProcedure	0	32	0	0

Selector 5 is the device-IOMMU map. Its 2080-byte parameter structure describes a host buffer, and on success the handler writes the resulting engine-visible device address back into the single scalar output slot. Selectors 3 and 4 are the pre-post and trigger of the resident submission model: an output buffer set is enqueued, the inputs-ready signal fires, and the same doorbell path as selector 2 rings the engine.

27.5 Register and exclave method catalog

Beyond the nine kernel selectors, the direct-path client exports a wider register, power, firmware, and secure-world method surface. These are not distinct kernel selector indices: each routes through one of the nine kernel selectors or through a separate entry point, and Table 27.5 names the surface by role.

Table 27.5. The register, power, firmware, and exclave method surface exported by the direct-path client beyond its nine kernel selectors.

Method	Role
ANE_PowerOn / ANE_PowerOff / ANE_IsPowered	power-domain control
ANE_LoadFirmware / ANE_ForgetFirmware	firmware image lifecycle
ANE_SendCommand	raw firmware command injection
ANE_SetPowerManagement / ANE_SetDynamicPowerGating / ANE_SetPowerGatingHysteresisTime	power policy
ANE_SetThrottlingPercentage	thermal throttle
ANE_SetDARTCacheTTL / ANE_FlushInactiveDARTMappings / ANE_UnmapDartBuffers	address-translation controls
ANE_ReadANERegister / ANE_WriteANERegister	raw memory-mapped register read and write
ANE_FWSharedEventDoorbellRing	ring the firmware shared-event doorbell
ANE_AddPersistentClient / ANE_RemovePersistentClient	keep the device resident
ANE_MPMMemoryMapRequest / ANE_MPMMemoryUnmapRequest	the multi-process managed-memory region
ANE_ExclaveCycle / ANE_ExclaveLoad / ANE_ExclaveEvaluate / ANE_ExclaveUnload	secure-world load and evaluate
ANE_ExclaveReadPropertyValue / ANE_ExclaveWritePropertyValue	secure-world property access
ANE_GetClientsInfo / ANE_ShowSharedMemoryAllocations / ANE_ShowModelMemoryStatus	diagnostics

A second access check beyond the kernel entitlement gates the raw register read and write, command injection, and exclave methods: a privileged-virtual-machine-access property probed at client open, distinct from the device-open entitlement.

27.6 From a user-space call to the doorbell

A submit on selector 2 crosses four layers from the user-space call down to the hardware doorbell write, traced in Listing 27.5.

Listing 27.5. The four-layer call path from a user-space submit selector down to the hardware doorbell write.

```
/* The submit path for selector 2 / direct-path selector 4: */
H11ANEInUserClient::externalMethod(sel=2, args)
  -> dispatchExternalMethod                /* validates 2376-in / 40-out */
  -> ANE_ProgramSendRequest(client, ref, args)            /* arg shim     */
  -> ANEClientDevice::programSendRequest(ANEProgramRequestArgs*, ...)
  -> ANEDriver::ANE_ProgramSendRequest(...)               /* gated        */
  -> ANEHWDevice::doorBellRing(db)
  -> ANERegisterControl::write32(reg, 1 << idx)           /* MMIO mailbox */
```

Below the dispatcher, the thin shim re-checks the argument sizes and unmarshals the typed argument structure, the client object method builds the memory descriptors and retains the shared-event fences, and the gated driver method runs on the command-gate workloop. The 2376-byte request structure holds the program handle minted at create time, a sequence number, the quality-of-service and execution-priority pair, and the array of surface identifiers for the input, output, and intermediate buffers, with the measured field layout in Table 27.6.

Table 27.6. The measured field layout of the 2376-byte request structure submitted on selector 2.

Offset	Field	Observed
+0x000	program / instance token (u64)	the handle minted by program-create
+0x008	sequence number	0, then 1 on the next submit
+0x010	priority / quality-of-service pair	(5, 21), qos class and execution priority
+0x01c	io category	2
+0x020	surface identifier array	input, output, intermediate surfaces

The 40-byte output returns the sequence and the echoed token: +0x00 is the sequence and result, +0x08 is the echoed token, and +0x20 is a status flag. Selector 2 alone uses the asynchronous machinery, and it is the only path that reaches a hardware doorbell from user space. Completion arrives at a mach wake port as a callback, not by a shared-memory poll. The doorbell write itself reads the doorbell index from the request, requires it below 32, computes the mask `1 << index`, and stores that mask into the engine register aperture, the mailbox signal that triggers the firmware.

The reverse signal, the engine telling the host a job is complete, travels the same windowed store mechanism in firmware. The engine rings a host interrupt with an interrupt-atomic memory-mapped store to the host-supplied target register, bracketed by clearing and then setting bit 39 of the implementation-defined AArch64 system register S3_3_C15_C8_0. Clearing bit 39 opens the posted-write window, the firmware stores the doorbell value into the host aperture, and a barrier separates the store from the status sample. The firmware then reads back the uncorrectable-cache-error bit (bit 1) and the transaction-reject bit (bit 7) to confirm the store has committed before setting bit 39 to close the window. The whole sequence runs with interrupts disabled so a nested handler cannot corrupt the status read.

27.7 Entitlement gate and broker model

The client open checks the two kernel entitlements of Listing 27.6, the hard device-open gate and the resident data-chaining gate.

Listing 27.6. The two kernel entitlements checked when a user client is opened.

```
/* checked at H11ANEInUserClient::init via copyClientEntitlement: */
"com.apple.ane.iokit-user-access"           /* the hard device-open gate   */
"com.apple.ane.allow-dataChaining-access"   /* resident data-chaining gate */
```

A single kernel entitlement gates opening either user client. The check runs once at client construction and is a boolean on the client object, not re-checked per selector. Across the whole system, exactly two binaries hold `com.apple.ane.iokit-user-access`: the system broker daemon and its per-user sibling. No application process opens the device. Every other consumer reaches the engine through the broker over a cross-process call, proving itself with an entitlement from the broker's own private family rather than the kernel gate.

The driver stamps the capability at client creation in `ANEClientInfo::create`, which reads each entitlement through `copyClientEntitlement` and records `isPrivileged` and `allowDataChaining` as bits on the client. Beyond the two open gates, the driver enforces six further `com.apple.ane` and `com.apple.private.ane` entitlements covering scheduling priority, memory and data access, and client and coalition hints, as Table 27.7 lists.

Table 27.7. The kernel-driver entitlements beyond the two open gates, with the capability each grants.

Entitlement	Capability
`com.apple.ane.realtime-priority-client`	the real-time-priority client grant
`com.apple.ane.allow-system-reserved-priorities`	use of the system-reserved scheduling priorities
`com.apple.ane.memory`	a memory-access grant
`com.apple.ane.allow-data`	a data-access grant
`com.apple.private.ane.allow-set-client-hints`	set per-client hints
`com.apple.private.ane.allow-share-coalition-hints`	share hints across a coalition

The broker keys are a private entitlement family, checked per connection and per method over a cross-process call, with each key, its capability, and its holder count given in Table 27.8.

Table 27.8. The entitlement family that gates the engine, from a static scan of the M1 system binaries, independent of boot-security state.

Entitlement	Capability	Holders
`com.apple.ane.iokit-user-access`	the hard kernel gate: open the user client, privileged device open	2: the broker and its per-user sibling
`com.apple.ane.allow-dataChaining-access`	resident data-chaining on the direct-path client	kernel-checked at client init
`com.apple.aned.private.allow`	baseline: compile, load, instantiate through the broker	18
`com.apple.aned.private.ANEAccess.allow`	inference-client access-grant variant	14
`com.apple.aned.private.adapterWeight.allow`	stream adapter weights onto a shared resident base model	5
`com.apple.aned.private.processModelShare.allow`	share one resident model across processes	4
`com.apple.aned.private.secondaryANECompilerServiceAccess.allow`	the longer-duration compiler service for large models	1
`com.apple.aned.private.aggressivePowerSaving`	the aggressive-power-saving execution mode	gate helper only
`com.apple.aned.private.modelPurgeInAllPartitions`	purge models across all cache partitions	gate helper only

Entitlement	Capability	Holders
`com.apple.security.temporary-exception.iokit-user-client-class`	open the direct-path user client for own submission	27
`com.apple.security.ts.ane-client`	trust-cache blessed-client slot for latency-critical consumers	5

The broker is a listener that enforces per-connection and per-method entitlement checks. It sorts admitted clients into a restricted tier, unrestricted tier, and per-user tier, and it threads a quality-of-service argument through every compile, load, and instantiate method. The restricted tier admits the adapter-weight, model-share, and aggressive-power-saving requests through a per-method admission helper, and the per-user tier serves the per-user broker.

The adapter-weight path is the mechanism behind swappable model weights without recompilation. A base model is loaded once, and each adapter is a new instance bound to a named base-model identifier holding only its per-adapter weight files, through a create-instance-with-weights method that names the base-model identifier and the weight-file count. Residency and power are explicit per-instance arguments on the create-instance method. They are an enable-power-saving flag, more-aggressive variant gated by the restricted tier, opt-out-of-model-memory-unwiring flag that keeps a hot client's weights resident at the cost of footprint, and queue-depth that the broker down-adjusts under contention. A queue-index function and a program-priority function map the quality-of-service argument to hardware scheduling. A privileged subset of system daemons also holds a sandbox exception that opens the direct-path user client and drives per-inference submission on its own connection, skipping the per-call round trip through the broker. That exception is a latency optimization, not a capability grant: the privileged device open still happens in the broker, which hands the client a program handle and an intermediate-buffer handle.

The kernel binds residency to code-signing identity. A second client may attach to an already-resident program only when its team identifier and code-directory hash match the owner, so a shared resident model or key-value cache cannot leak across tenants. The kernel resolves the caller's team identifier and code-directory hash, and the attach path tests them against the resident owner before a sibling instance reuses the shared intermediate-buffer handle. With one physical engine on this generation, cross-client arbitration is time-division multiplexing on a single gated request queue, biased by the per-stream quality of service the clients declare.

27.8 Live device properties

The driver publishes its topology and version constants into the registry, read live on the M1 host and decoded in Table 27.9.

Table 27.9. The device properties the driver publishes into the registry on the M1 host.

Property	Value	Meaning
architecture type string	`h13g`	microarchitecture family, drives per-family codegen
version	96 = `0x60`	major hardware version
minor version	17	minor revision
board type	96	board and system-on-chip type id
board subtype	0	board sub-variant
number of cores	16	compute cores in this engine
number of engines	1	distinct engine units, load balancer is a pass-through
CPU subtype	4	program-ABI gate
internal build	No	release build, gates the debug surfaces

The number-of-cores value is a topology count and not the throughput-relevant multiply-array width, so a floating-point rate is taken from the measured cost-model anchor rather than inferred from the core count. At rest the registry shows the device, load-balancer instance, and standing hints client present, with zero control clients and zero direct-path clients open, confirming the brokered, lazily-opened model. The driver opens the user clients on demand per active client and tears them down when idle.

28 Address translation and the DART

SUMMARY

The engine reads and writes DRAM through its own IOMMU, the DART, which translates a device address into a physical page at a 16 KB granule across a 3.5 GiB aperture. A normal engine page is one 64-bit leaf word, the physical frame located unshifted with bit 63 set: leaf = phys|0x8000000000000000. Program-create re-bases each host-mapped buffer into a firmware aperture with high half 0x1bc4, a second translation with no constant offset. The fault-capture registers cannot be read on the M1, because every function on the fault path ends in a kernel panic.

Chapter 21 covered the on-chip working set: the pool the engine reads and writes once data is resident. The off-chip step that puts data there is the subject here. The engine reads its inputs and writes its outputs directly from DRAM through an input-output memory management unit, the DART, which translates a device-visible address into a physical DRAM page before any DMA engine touches memory.

28.1 DART and its address space

Table 28.1 gives the page granule, aperture base and size, and active stream set read from the live device tree.

Table 28.1. The page granule, aperture base and size, and active stream set, read from the live device tree.

Property	Value	Meaning
page granule	0x4000 = 16384 = 16 KB	the IOVA page; every buffer maps at this granularity
aperture base	0x0	IOVA window starts at zero
aperture size	0xE0000000 = 3.5 GiB	the managed IOVA span, 0x0 to 0xE0000000
active streams	{0, 1, 2}	the engine streams sharing one translation-table base; client isolation is separate, per-client contexts mapper-ane0-iso1 through iso7

The DART is the engine's IOMMU. A host tensor buffer never reaches the engine as a host virtual address. The DART maps it into the engine's own address space, the device virtual address or IOVA, and the DMA engines issue reads and writes against IOVAs. The DART holds the page tables that translate each IOVA back to a physical DRAM page, so a buffer physically scattered across DRAM appears IOVA-contiguous to the engine. The DART instance serving the engine on the M1 is a single controller bound to the engine stream set, and its managed address window and granule come from the live device tree. The controller is the device-tree node dart-ane0 at physical base 0x85800000, bound to the t6000-generation driver class, not the t8020 class. It has four 16 KB register windows at 0x85800000, 0x85810000, 0x85820000, and 0x85804000. Table 28.2 gives the live dart-ane0 device-tree properties read from the controller node.

Table 28.2. The live `dart-ane0` device-tree properties read from the controller node.

Property	Raw	Decoded
compatible	`dart,t6000`	the t6000-generation controller
page size	`0x00004000`	16 KB IOVA page
stream-ID enable bitmap	`0x0000a001`	bits 0, 13, 15
bypass bitmap	`0x0000a000`	bits 13, 15
stream count	`0x10`	16 stream slots
options	`0x25`	low byte of the live config word `0x80000025`

The host-side translation object is a three-level mapper chain: the controller driver owns a mapper nub, which owns the translation mapper the engine driver is handed. That mapper holds the highest retain count of any mapper on the system, consistent with its role as the live tensor-mapping object. The page granule is 16 KB, not the 4 KB of the host page tables. The firmware validates the same value independently and rejects a wrong one with `MMU invalid page size: %x`, so the host maps and wires every engine DMA buffer at 16 KB granularity. A sub-page buffer still consumes a full 16 KB IOVA page. This 16 KB page is the coarsest of the three alignment scales on the M1: the 16-byte DMA-width granule of chapter 21 is below a 256-byte segment alignment, which is below the 16 KB DART page.

A two-level page-table walk, the L2 and L3 tables under the per-stream translation-table base, over the 16 KB page covers the 3.5 GiB span. The per-stream translation-table base is in the controller's translation-table base register, measured live at `0x90022320` for the active streams: bit 31 marks the base valid, and the remaining field shifts left by 12 to the physical table base `0x10022320000`. Streams 0, 1, and 2 share one table base, a single table for the active engine streams. That base is in the same DRAM band as the measured leaf physical frames, so it points at the controller's own page-table memory.

Client isolation is separate from this engine-stream table. The DART gives each client its own isolation context, the eight address-translation mappers `mapper-ane0-iso1` through `iso7` plus a base in the live IORegistry, so each client's buffers map into its own translation domain. The secure exclave receives these contexts as the capabilities `ANEIsoID1` through `ID7`, confining each client's DMA to its own domain, the address-translation half of the exclave capability model in chapter 32.

The controller register layout is recovered from its capture routine, as the byte offsets into a stream's 16 KB register window in Table 28.3.

Table 28.3. The controller register-offset map, recovered from the register-capture routine.

Offset	Register
`+0x40`	error status: fault flag at bit 31, plus stream id and fault code
`+0x50`, `+0x54`	error address low and high, the faulting device address
`+0xfc`	enabled-streams global bitmap
`+0x100` + 4·sid	per-stream translation-control array
`+0x200` + 4·idx	per-stream translation-table base array
`+0x1000`, `+0x100c`	translation-buffer control and status
`+0x1020`, `+0x1028`	translation-buffer error registers

28.2 Leaf page-table entry

The leaf entry that the DART stores per page is one 64-bit word. For a normal engine data page it is the physical frame with the valid bit set, given by

$$\text{leaf} = \text{phys} \mid \texttt{0x8000000000000000}.$$

Table 28.4 gives the bit-field layout of that 64-bit leaf word.

Table 28.4. The bit-field layout of the 64-bit leaf page-table entry the DART stores per page.

bits	field	value	note
63	valid / active	1 on map, 0 on unmap	the only flag set for an engine page
62	aux protection class	0	set only when prot bit 3 is set; never reached on the engine path
60	aux protection class	0	set only when prot bit 4 is set; never reached
59	aux protection class	0	set only when prot bit 5 is set; never reached
46:14	physical frame	full physical address, unshifted	low 14 bits zero at the 16 KB granule
13:0	within-page offset	0	always zero at page granularity

Bit 63 is the valid bit, set on map and cleared to the all-zero template on unmap. The physical frame is the full 16 KB-aligned physical address located unshifted, so its low 14 bits are zero and the frame and the valid bit do not overlap.

The word holds the same shape for every live engine usage type. The mapping software holds two software protection classes, read-write for inputs, weights, and intermediates, and a device-write class for outputs, yet both collapse to the identical leaf template of bit 63 alone. The DART encodes access permission in the per-stream translation-control configuration, not in these high page-table bits. Bits 62, 60, and 59 are aux-protection classes that the engine driver never sets, because its direction-to-protection mapper produces only the values $\{1, 2, 3\}$, whose bits 3, 4, and 5 are always zero. Across 26178 measured leaf-map events none of those three bits was ever set.

The mapping path that produces this word stacks two translations. The host pins the physical pages, then fills the leaf table by accumulating per-page segments and flushing them through the page-protection-layer write. That write holds the 40-byte per-page segment structure of Listing 28.1, from which the leaf word is assembled per page.

Listing 28.1. The per-page segment structure passed into the page-protection-layer write and how the leaf word is assembled from it.

```
struct ppl_iommu_seg {   /* 0x28 bytes, measured layout */
    uint64_t iova;       /* +0x00  device virtual address      */
    uint64_t phys;       /* +0x08  physical DRAM page           */
    uint64_t size;       /* +0x10  0x4000 (16 KB granule)       */
    uint64_t prot;       /* +0x18  3 = RW, 1 = device-write     */
    uint64_t reserved;   /* +0x20  0                            */
};
/* assembled leaf word, per page i:  leaf[i] = seg[i].phys | template */
/* template = 0x8000000000000000 (bit 63) on map, 0x0 on unmap        */
```

The leaf table itself is page-protection-layer memory. The word above is the value the kernel hands to that layer to store, captured at the store register on a live serialized dispatch, not a read-back of the stored page.

The host maps each buffer under a usage code that names its role, held in the segment structure and recovered from the map call sites, with the codes and their protection classes in Table 28.5.

Table 28.5. The buffer-role usage codes and the protection class each maps under.

Code	Role	Protection class
1	client input tensor	read-write (3) on the M1
2	client output tensor	device-write (1)
7	intermediate buffer	read-write (3)
8	kernel and weights	read-write (3)
9	program text, task descriptors, working set	read-write (3)
11	program constants and scratch	read-write (3)
`0x1019` = 4121	firmware power-on shared surface	firmware class
`0x101a` = 4122	firmware resident heap	firmware class

A direction-to-protection mapper produces only the values 1, 2, and 3: a device-write output is class 1, a read-only input would be class 2, and read-write or no-direction is class 3. On the M1 a configuration bit in the controller collapses the would-be read-only class 2 into read-write 3, so the controller maps read-only inputs read-write and enforces input protection upstream rather than in the leaf word. The high `0x1000` bit on the firmware codes flags the firmware-owned shared class, distinct from the per-program client and kernel buffers.

28.3 Host-to-firmware rebase boundary

The host-side translation resolves to a physical address for every buffer. For a single matmul-with-activation load, every host-programmed buffer resolves to its named buffer role: input tensor, output tensor, weights, program text, constants, intermediate, working set, and two firmware shared surfaces. Table 28.6 shows the nine buffers of that load, each with its usage code.

Table 28.6. The nine buffers of one matmul-with-activation load, each shown with its usage code.

Named buffer	Usage
input tensor	1
output tensor	2
weights	8
program text	9
constants	11
intermediate	7
working set	9
firmware shared surface	4121
firmware resident heap	4122

An in-place operation maps two distinct device addresses onto one physical page, a single shared memory descriptor that the runtime reads as input and overwrites as output. The physical pages are scattered across the DRAM band and are not physically contiguous: the page table is what makes each buffer appear contiguous to the engine.

One further translation is past the host boundary. The values the engine reads from its instruction stream are not host IOVAs. At program-create the firmware re-bases each host-mapped buffer into its own resident aperture and patches that rebased address into the engine registers, so the engine-register operand values are in a firmware window with high half `0x1bc4`, not in the host IOVA band, shown in Listing 28.2.

Listing 28.2. The firmware rebase from the host device-address band into the resident firmware aperture.

```
host IOVA band     -->    firmware DRAM-tile aperture     (no constant host-side offset)
```

The host IOVA and the firmware-aperture address are unequal and have no constant offset between them, so the rebase is a second translation rather than a fixed displacement. The bridge that establishes it runs once at program-create and is cached. On the cached dispatch path the host never re-emits the host-to-firmware pair, so the firmware rebase is the only translation below the host boundary there.

Three address spaces coexist for one load, recovered from the loaded program text and the page-table fill and named in Table 28.7.

Table 28.7. The three coexisting address spaces for one load: the host device addresses, engine-register aperture, and firmware DRAM-tile aperture.

Space	Address space	What it holds
A	Host IOVA	host page table, fully resolved to physical
B	Engine-register	where the firmware writes the DMA-engine bases
C	DRAM-tile aperture	what streams: the weight and data tiles

The program text is a list of 44-byte register-write records, each pairing an engine-register address in space B with two DRAM-tile operand values in space C. The engine-register address selects which data-movement engine to re-base: the weight-streaming sources, input-tile reader, and output-tile writer. The operand values hold the high half `0x1bc4` of space C, the firmware-resident rebase of the host buffers, which is why a naive comparison of raw register addresses across loads fails: the aperture base moves with each program load.

28.3.1 Firmware rebase arithmetic

The rebase that produces space C from a host IOVA is a pure linear translation recovered from the firmware itself. The firmware validated-translate routine reads the descriptor IOVA, range-checks it against the mapped region, then computes the runtime address as

$$\text{runtime} = \text{mappedBase} + (\text{IOVA} - \text{dvaBase}).$$

There is one subtract of the IOVA-region base and one add of the firmware aperture base. There is no shift, no mask, and no page rounding, which is why the host IOVA and the firmware-aperture address have no constant offset: the offset is the difference of two independent region bases that each move per load.

The three runtime quantities the formula needs are in the firmware aperture-config object at `engine+0xa000`, whose fields Table 28.8 gives.

Table 28.8. The firmware aperture-config fields at `engine+0xa000` that drive the rebase, populated at buffer-map time and absent from the firmware image.

Offset	Field	Meaning
`+0xb98`	`dvaBase`	host / DART IOVA region base
`+0xba0`	`regionSize`	mapped region size
`+0xba8`	`mappedBase`	firmware runtime aperture base, the rebase target base

These three are firmware-runtime state set at buffer-map time, not constants in the binary; the image holds only the arithmetic and the field offsets. The firmware then writes the rebased value into the engine DMA bar registers at `0x285c25020 + engine*0x148 + barId*4`, where `engine*0x148` is the per-engine MMIO stride and `barId*4` selects the word-indexed bar register. Only the low 32 bits of the rebased address reach this MMIO path; a per-bar config flag gates wider bars onto a separate software descriptor table. Both the rebase tail and the register-target prologue were emulated under unicorn and reproduced the formula and the `0x285c25020 + engine*0x148` register address exactly.

The firmware also clamps every address it programs into the DMA engines to a 32-bit ceiling, below the 3.5 GiB aperture. Each device address the engine touches, text, weights, descriptors, intermediate, output, and chained buffers, must satisfy `addr >> 32 == 0`, asserted per buffer in the firmware. The host allocator thus hands out only sub-4-GiB IOVAs, and the engine operates in the bottom 3.5 GiB of its address space.

Three distinct alignment scales govern an engine buffer, each coarser than the last, collected in Table 28.9.

Table 28.9. The three alignment scales on the M1, from the finest DMA width granule to the coarsest page granule.

Scale	Value	What it governs
DMA width granule	16 B	the data-movement quantum
segment alignment	256 B	program-text and segment packing
page granule	16 KB	the device-address page, allocation and wiring unit

The host may pre-map a buffer before the inference that uses it, through a pre-map command that establishes its device-address mapping ahead of the hot path. The firmware keeps explicit buffer pools, tagging each mapped buffer with a pool identifier, and runs a buffer-recycle state machine that reuses output buffers across chained calls. A per-process pool tracks outstanding requests with an in-flight count capped at 127 per request.

Under device-address pressure the kernel applies a least-recently-used eviction policy over its mappings, scoring each with `getDartBufferFreeUpScore` and freeing through the `FreeUpDart*` family. A long-running process that maps more than the address window holds thus has mappings reclaimed rather than failed.

28.4 Fault-capture registers

The DART captures a translation or protection fault into the single-shot register block of Table 28.10, an error-status word, the faulting device address split low and high, and a translation-buffer status word.

Table 28.10. The DART fault-capture register block, giving each register device offset and the field it holds.

register	device offset	field
error / status	`+0x40`	fault flag at bit 31, plus stream id and fault code
error address low	`+0x50`	faulting device address, low half
error address high	`+0x54`	faulting device address, high half
translation-buffer status	`+0x100c`	busy / error status

These registers cannot be read safely on the M1. The capture routine writes the block into a DRAM snapshot only on the fault path, and on this controller that path is unconditionally a kernel panic. Every function on the fault path ends in a direct call to the kernel panic routine, and control flow reaches it: the driver implements a DART fault as a `REQUIRE(...)` assertion, which panics the machine rather than returning the captured registers. A read-only probe that reads the snapshot at the function boundary is built and validated

against the disassembly, but it cannot fire without a fault, and any fault panics the box. The faulting-address and status values are thus stated here as the structure of the block, not as measured values, since obtaining them on this hardware would require panicking the machine. The register layout above is recovered from the capture routine; the field decode of the words a contained fault would return follows the published controller field layout.

28.4.1 `IODARTErrorInfo` fault descriptor

Above the hardware DART-side capture is a software fault descriptor, `IODARTErrorInfo`, that the kernel `t6000dart` core constructs from the DART fault MMIO and hands to each registered consumer. This descriptor is the structure the driver fault callback reads and logs, and it is a kernel-wide ABI: the sibling DART consumers `AppleAVD` and `AVE_DART` read the identical layout at the identical offsets, which confirms it is not an engine-private struct. Most fields are object pointers whose stringifier the callback invokes; Table 28.11 gives the byte offsets into the descriptor.

Table 28.11. The `IODARTErrorInfo` software fault descriptor, the shared kernel ABI the `t6000dart` core hands to its registered fault consumers.

Offset	Field	Type	Meaning
`+0x00`	`Type`	string	fault type, also the event-id header
`+0x08`	`HwClass`	string	hardware fault class
`+0x10`	`HwError`	string	hardware error code
`+0x18`	`HwStatus`	`u32`	raw DART fault-status word
`+0x20`	`IsWrite`	bool	`0` read fault, `1` write fault
`+0x28`	`SID`	`u32`	DART stream id of the faulting agent
`+0x30`	`Address`	`u32`	faulting IOVA
`+0x40`	`TTBRIndex`	`u32`	translation-table base index
`+0x50`	`L2Index`	`u32`	page-table walk L2 index
`+0x58`	`L3Index`	`u32`	page-table walk L3 index
`+0x90`	`AXI_ID[0..3]`	4 × 8-byte slot	AXI master / transaction ids, each printed as a 32-bit value

The four slots at `+0x90` are the `AXI_ID[0..3]` array, the four fault descriptors a prior decode reported. The per-fault metadata is the scalar set above: `Address` at `+0x30` localizes the faulting page, `SID` at `+0x28` names the stream. `Type`, `HwClass`, and `HwError` at `+0x00`, `+0x08`, and `+0x10` give the fault taxonomy, and `IsWrite` at `+0x20` holds the read-write bit. The walk indices `TTBRIndex`, `L2Index`, and `L3Index` at `+0x40`, `+0x50`, and `+0x58` localize the failing page-table entry.

One register the callback reads is not part of the descriptor: it reads an engine status word directly off its own device object at `[engine+0xe028]`. The callback treats the fault as benign and returns early when `(status | 0x80) == 0xa0`, that is when the low seven bits equal `0x20` and bit 7 is a don't-care. Any other value is a real fault. The same word gates the engine clock elsewhere in the driver, where `0xa0` is the powered-idle state.

The ANE kext does not raise the panic. The kext fault path returns and, on a real fault, sets a sticky latch, dumps the shared-memory allocation table and firmware debug state, then parks for 250 ms awaiting external recovery. The machine-halting panic is in the kernel `t6000dart` core, where the mapping and page-table-walk fault detection is a `REQUIRE` assertion that calls `panic()` directly. That substrate panic fires before, or instead of, the kext recovery park.

29 Firmware

SUMMARY

The engine controller runs a C++ application over a real-time kernel, distributed as an unencrypted preload executable. Every subsystem is a long-lived message-pumped task, and all kernel objects sit in fixed build-time pools. The execution loop accepts four command classes on the shared channel, three procedure-call variants plus the cache-request trigger, and rejects anything else as unsupported. The scheduler is fixed-priority with eight levels, no hardware watchdog, and a software deadline of about 2 seconds that aborts a stuck queue. Recovery is serialized behind a single command gate and rate-limited, so crashes arriving faster than the recovery cycle keep the channel not-ready.

The engine's controller runs a small real-time operating system, not bare metal. A real-time kernel hosts an application that loads compiled programs, schedules their task descriptors, drives the multiply array and the data-movement engines, and reports faults back to the host. That controller is CHINOOK, an embedded ARM core with eleven hardware thread contexts (`CHINOOK_CPU_IMPL_THID0` through `THID10`), its own level-two cache, and pipeline error-capture and power-down-save registers, so the firmware runs on a real multithreaded processor rather than a sequencer.

29.1 Substrate

The firmware is a C++ application over a real-time kernel, distributed on the package as an unencrypted preload Mach-O wrapped in an Image4 container with payload tag `anef`. The kernel layer provides tasks, a priority scheduler, semaphores, mutexes, message queues, two heap arenas, a fault handler, and a generic finite-state-machine framework. The application layer above it provides the program loader, execution loop, task-descriptor driver, tensor-mover driver, address-translation manager, and power-control service.

Every subsystem is a long-lived task with its own stack, supplied by a message queue. A task blocks on its queue, wakes on a posted message, runs that message to completion, and loops. Each task has two queues: a task-context input queue and a separate interrupt-context queue. An interrupt handler never runs subsystem logic in interrupt context. It pushes a record into an interrupt buffer, posts the target task's interrupt queue, and returns, deferring the work to task context. The named tasks include an idle task, task terminator, execution-loop pump, address-translation manager, tunable-register loader, load monitor, interrupt manager, call manager, server, and host remote-procedure-call daemon.

Each task is in one of the five scheduler states of Table 29.1, recovered from the task-list dumper.

Table 29.1. The five scheduler task states.

State	Meaning
`RUNNABLE`	ready or running
`SUSPENDED`	explicitly parked
`WAITING`	blocked on a generic wait, such as a message queue
`SEMWAIT`	blocked specifically on a semaphore
`BLOCKED`	blocked

The distinct semaphore-wait state, separate from the generic wait, is the heritage of the real-time executive model the primitives follow. The message queue each task blocks on is a bounded-buffer control block holding a maximum message width, a ring depth, read and write pointers, a lock, and two counting semaphores. The send side blocks when full and the receive side blocks when empty. The kernel holds all its objects in fixed pools sized at build time: tasks, semaphores, mailboxes, queues, signals, timers, and mutexes each occupy a slot in a pre-sized table. There is no unbounded dynamic task creation at run time, and the capacity of each pool is fixed and reportable as a total-and-available count. A hard boot invariant ties the heap to the system reservation: the initial heap must exceed twice the reserved framework allocation, and the kernel refuses to boot if the arena is too small.

29.2 Bring-up

The three-level kernel bring-up call chain of Listing 29.1 separates device attach from firmware bring-up.

Listing 29.1. The three-level kernel bring-up call chain.

```
ANEHWDevice::start()              attach + provider + RTBuddy client + 7 MMIO banks + clocks
  ANEHWDevice::power_on_hardware()    clocks/power + MPM + resume
    ANEHWDevice::ANE_Init()             RTBuddy endpoints + scratch handshakes + first commands
```

Bring-up splits into two phases on distinct entry points. `ANEHWDevice::start` attaches the device, wires the RTBuddy client, maps the register apertures, and enables clocks and power. `ANEHWDevice::ANE_Init`, reached from `ANEHWDevice::power_on_hardware` on first power-on and on every wake, drives the firmware handshake. This separation keeps device attach independent of the firmware boot that runs again after each sleep.

The kernel maps seven named register apertures, not six. The six-bank count belongs to a different system-on-chip family and does not hold on the M1. The apertures are constructed in order, each with a device-tree `reg` aperture index, and Table 29.2 names the seven by role.

Table 29.2. The seven memory-mapped register apertures the kernel maps during `ANEHWDevice::start` on the M1.

Order	Bank	Role
1	`ANE`	main control aperture, holding the scratch registers and the reset-vector register
2	`PS`	power-state
3	`PWGATE`	power-gate
4	`PTD`	page-table and translation-domain control
5	`ANEHAL1`	hardware-abstraction register aperture
6	`ANEHAL2`	hardware-abstraction register aperture
7	`ANEHAL3`	hardware-abstraction register aperture

The first four apertures take a contiguous index base and three successive offsets. The three `ANEHAL` aperture indices are read from per-family device fields, which is why the aperture count and indices differ across chips.

The kernel allocates the RTBuddy client during attach and stores it on the device object, with a name object retrieved from the `ANE` property and a separate mailbox object. The endpoint itself comes up inside `ANE_Init` through `EnableRTBuddyEndpoints`, which holds the endpoint identifier literal `0xe400`. The path first tries `ANE1Endpoint1`, and on failure cleans up and retries `ANEEndpoint1`. The endpoint lookup resolves

the named service, retrieves the endpoint object, and registers the inbound message handler, with the doorbell paths wired alongside.

`ANE_Init` then runs the firmware handshake in program order. It quiesces pending work, brings up the `0xe400` endpoint, initializes the scratch mailbox registers, and selects a warm or cold boot by writing `1` or `0` to scratch register 7. It reads the firmware boot address and writes it to the reset-vector register `rANE_H11_CHINOOK_IO_RVBAR`, then releases reset and polls. Three scratch handshakes follow, observed by the kernel in the order Table 29.3 gives.

Table 29.3. The three scratch-register handshakes the kernel waits on during firmware bring-up, in order.

Handshake	Register	Meaning
1	scratch 7, first wake	the engine controller is alive after the reset-vector handoff
2	scratch 7, second wake	the channel-description table the firmware published is ready
3	scratch 3, plus a magic-number-1 check	final acknowledgement before interrupts are enabled

Between the second and third handshakes the kernel walks the channel-description table the firmware published and resolves the named inter-process channels: `SHAREDMALLOC`, `TERMINAL`, `BUF_H2T`, `BUF_T2H`, `IO`, `IO_T2H`, `DEBUG`, and `DATA_CHAIN_H2T`. If the second wake never arrives, the kernel powers the hardware off and on in a bounded retry loop and increments a failure count.

On the firmware side the reset-vector entry brings up the real-time kernel: it sets up the memory-management unit and the exception and interrupt stacks, then the kernel heap, subject to the boot invariant that the initial heap exceeds twice the framework reservation. The firmware then checks its boot arguments and chip revision against the values the kernel passed, exchanges an inter-process-communication protocol version, and validates each ring buffer in the channel table against a ring-buffer version constant. The firmware marks each control channel ready in turn and rejects commands addressed to a channel before it is ready.

Bring-up is ready to accept the first command when all of the following hold.

- The kernel has enabled the `0xe400` endpoint, mapped the seven apertures, and enabled clocks and power.
- The firmware has booted from the reset vector, brought up its heap, passed the chip-revision and protocol-version checks, published the channel table, and marked every control channel ready.
- The three scratch handshakes have completed in order.
- The kernel has logged that the engine controller is ready, enabled interrupts, and round-tripped the start command.

Only then does the kernel issue its first commands over the channel, in order: print-enable, start, the performance-monitoring-unit base set, a host-to-engine time synchronization, channel-property write, resource-information query, default-setting write, memory-cache power-on, and the shared-event-information initialization. The resource-information query returns the engine count, cache-request limits, and maximum procedure count, and the default-setting write holds the context-switch latency threshold.

29.3 Execution loop

The execution loop accepts the four command classes of Listing 29.2 on the shared channel.

Listing 29.2. The four execution-loop command classes accepted on the channel.

```
union uCExeLoopSupportedCmd          ; insize <= sizeof(union uCExeLoopSupportedCmd)

[PROC CALL]              ProgId=%d  ProcId=%d  Proc=%d  Pri=%d
[PROC CALL WITH BARS]    + nbrOfCustomBars <= 32        ; custom buffer-access-register overrides
[PROC CALL WITH EVENTS]  + nbrOfSignalEvents in 1..16   ; wait/signal events
                         + (nbrOfWaitEvents + nbrOfSignalEvents) > 0
[CR TRIGGER]             cacheHandler 0x%llx            ; data-chaining cache-request trigger
```

The execution loop is the hot path that turns a host procedure-call command into work for the task-descriptor driver. A host writes a command into the shared ring buffer. The loop receives that command holding a program identifier, procedure identifier, procedure index, and priority. It builds the request list and pushes the program's task-descriptor partitions onto a task queue; the task-descriptor driver runs them on the cores while the tensor-mover streams the operands. The command is a fixed-layout record on the shared channel, and the loop logs each class as it accepts it. A bounds check rejects any record larger than the supported-command union before the loop reads its fields. The bar field is a buffer-access register, the buffer base and size binding a task descriptor references by index, and it logs as `bar[%d]: type=%d cfg=%d barId=%d value=0x%llx bufSize=%lld`. The loop rejects an unknown opcode with `Cmd 0x%x is not supported through ExeLoop cmd channel` and drops the record.

The main command classes on this channel are a plain procedure call, procedure call holding buffer-base overrides, procedure call holding wait-and-signal events, and data-chaining cache-request trigger. The dispatcher checks each command identifier on every event against a fixed accepted set and drops anything else as unsupported: the procedure-call family (`0x204`, `0x20c`, `0x211`, and `0x212`), the cache-request trigger (`0x209` with its path variant `0x20a`), the channel data-file load `0x2d`, and the back-channel and process-id controls (`0x404`, `0xff00`). The complete decoded command set is in chapter 30. The firmware drops and counts a procedure call addressed to a process that is not running rather than faulting it.

The procedure-call command structures hold the buffer-access registers and optional event and execute-order arrays under the fixed field limits of Table 29.4, recovered from the firmware assertion strings.

Table 29.4. The field limits on a procedure-call command, recovered from the firmware assertion strings.

Field	Limit
custom buffer-access registers	at most 32
signal events	greater than 0 and at most 16
wait events plus signal events	greater than 0
task-descriptor partitions	at least 1 and below the per-procedure maximum
custom execute-order entries	at most 128
output buffer sets	exactly 1
event masks	exactly 1

Each buffer-access register binds a buffer base and size that a task descriptor references by index, with at most 32 hardware register slots addressing the distinct buffer regions for one operation, and at most 16 input buffers per trigger.

The data-chaining trigger is the firmware mechanism behind resident state across dispatches. A cache request chains one execution's output set onto the next execution's input set, so a value produced by one dispatch is available to the next without a copy back to the host. The resident key-and-value cache and the resident optimizer state described in chapter 2 use this native chaining path. The firmware can chain procedures, running several in sequence on the engine without returning to the host between them.

29.4 Doorbell emit

Listing 29.3 gives the guarded doorbell-emit sequence that brackets the host-notify store with the window-gate bit clear and set.

Listing 29.3. The guarded doorbell-emit sequence, bracketing the host-notify store with the window-gate bit clear and set.

```
ring_doorbell(DoorBellReg, DoorBellBit):     ; host-notify site @0x4c890
  w0 = disable_irq()                         ; DAIFSet #0x2, make the quad atomic
  x8 = mrs S3_3_C15_C8_0
  x8 = x8 & ~(1 << 39)                        ; CLEAR bit 39: arm the doorbell window
       msr S3_3_C15_C8_0, x8
  str DoorBellBit -> [DoorBellReg]            ; *** the doorbell store (the interrupt) ***
  dsb sy ; isb                                ; force the store + status update to commit
  x8 = mrs S3_3_C15_C8_0
  if x8 & (1 << 1): fatal "Uncorrectable L2C error overflow"
  if x8 & (1 << 7): rc = 1, clear sticky {1,7} ; transaction rejected, retry
  x8 = mrs S3_3_C15_C8_0
  x8 = x8 | (1 << 39)                         ; SET bit 39: close the window
       msr S3_3_C15_C8_0, x8
  restore_irq(w0)
  return rc
```

After staging a task-descriptor list the driver notifies the host by ringing a doorbell. The engine-to-host interrupt is a single 32-bit memory-mapped store to a host-supplied register, bracketed by the windowed-store gate of chapter 27. Bit 39 of the implementation-defined system register `S3_3_C15_C8_0` is cleared to arm the doorbell window, and the store is forwarded as a fabric-coherent transaction. Bit 39 is set again to close the window. The firmware masks interrupts across the four steps so the arm, store, sample, and disarm cannot be interrupted, and the two status bits sampled after the store hold the fabric response. The same guarded sequence drives the engine-to-graphics-processor synchronization doorbell, which writes the value `1` to the fixed memory-mapped target `0x2_0646_8000` between the identical bit-39 clear and set.

The system register is not a single-bit gate. At least six bit positions hold distinct meaning, recovered from the read, write, and sample patterns at the 28 access sites and decoded in Table 29.5.

Table 29.5. The decoded bit map of the implementation-defined fabric system register `S3_3_C15_C8_0`.

Bit	Role
0	fabric-path enable, cleared with bits 2 and 4 on the power-gate path
1	uncorrectable last-level-cache error, read-only and sticky
2	fabric-link status, must be 1 for the power-gate condition
4	fabric-link secondary status, must be 0 for the power-gate condition
7	transaction-reject status, clearable on a rejected doorbell
39	doorbell-window arm and disarm: clear opens the store window, set closes it

Clearing bit 39 opens a window in which a store to a fabric doorbell aperture is forwarded as a coherent doorbell transaction, and bits 1 and 7 are the fabric response status latched by the barrier after the store. The power-gate path reads the register to sample bits 2 and 4 as fabric-link status before it clock-gates the engine, requiring bit 2 set and bit 4 clear, then clears bits 0, 2, and 4.

A completion event can fan out to up to three host doorbells. The host doorbell target is a register-and-bit pair the host supplies in an endpoint descriptor and the engine stores resident, in three slots, so the shared-event path can notify the inference client, a cross-agent waiter, and a telemetry sink from one completion.

29.5 Scheduler

Table 29.6 collects the firmware scheduler properties, covering its priority model, bands, preemption rules, watchdog, and software deadline.

Table 29.6. The firmware scheduler properties.

Scheduler property	Value
Priority model	fixed-priority, eight levels (0 through 7)
System band	levels 0 through 1
Application band	levels 2 through 7
Preemption within a task	none, run to completion
Preemption between tasks	by kernel thread priority
Hardware watchdog	none
Software deadline	about 2 seconds, queue-idle wait then abort
Task-queue identifiers	1 through 255, identifier 0 reserved

The scheduler is fixed-priority with eight levels, numbered zero through seven. The levels split into two bands: levels zero and one are reserved for the system, and levels two through seven are application priorities. Preemption within a task does not occur: each message pump runs to completion, and preemption happens only between pump threads by kernel priority. Two priority axes coexist. The kernel thread priority orders the worker tasks, and the application job priority orders queued inference jobs onto the per-priority task queues. They are distinct numbers on distinct objects. No hardware watchdog panics the engine. The firmware self-polices with a software deadline: a task queue that will not quiesce within about two seconds is force-aborted rather than left hung.

The submit path pushes work onto numbered task queues keyed by a network identifier in the range 1 through 255, with identifier 0 reserved. The submit path guards the queue identifier below 8, polls the queue for vacancy, enables the queue, copies a 35-word task-descriptor register block into the queue's register file, and rings a submit doorbell. Each queue occupies a fixed stride of `0x148` bytes in the engine register aperture, with the per-queue registers the submit path writes given in Table 29.7.

Table 29.7. The per-queue register map, at the queue base plus the queue identifier times `0x148`, that the submit path writes to start hardware work.

Offset within queue	Register
`+0x24000`	submit doorbell
`+0x24004`	size and count
`+0x24008`	priority and network identifier
`+0x25000`	queue enable
`+0x2500c`	queue status and vacancy
`+0x250a0`	buffer-access-register file

The eight application priorities map to a fixed in-firmware table of queue weights, `{1, 2, 3, 4, 5, 6, 30, 31}`, and scheduling is strict fixed-priority with per-priority credit and no preemption. The scheduler bounds how many requests are in flight on the firmware at once and applies backpressure when it saturates, through `ANEScheduler::wakePendingRequestsQueueWithFWOverload` and `ANEScheduler::pendingReque`

`stsWithFirmwareCount`, with a dedicated unwire thread that releases resources behind completed work. The firmware-overload throttle is the mechanism behind the large-batch trainable-convolution stall described in chapter 19: a submission that overruns the firmware queue is throttled rather than dispatched.

29.6 Control state machine

A finite-state machine drives the execution loop through the states of Table 29.8, each described by what it does.

Table 29.8. The execution-loop control state machine, with what each state does.

State	What it does
`INIT` (0)	the start node, after the loop is constructed but before it is armed; no hardware work dispatches
`RUN` (1)	non-secure and runnable, the engine is powered and owned but nothing is in flight
`EXEC` (2)	a task-descriptor partition is in flight, non-secure; the state in which a handler is permitted to touch hardware
`EXEC-SECURE` (3)	in flight while the secure phase is asserted, the engine owned by the secure tenant
`PAUSE`	fully quiesced for a secure-mode boundary, rejects all events

The framework stores no name for each state, so the meaning of each is reconstructed from the handlers that read the current state and from the events that drive it. Four states have framework identifiers zero through three, and a fifth is a named transient crossed during a secure-mode switch. Posted events drive the machine rather than direct state pokes. The event alphabet is four entries: a command-dispatch event when a procedure call or trigger is accepted, drain event toward idle, enter-secure event, and exit-secure event. A command accepted on the channel posts the dispatch event, which moves the machine from `RUN` to `EXEC` and pushes the task-descriptor list. A request finishing returns the machine from `EXEC` to `RUN`.

Roughly seven independent handlers gate hardware work on the same test: the current state must be `EXEC` or `EXEC-SECURE` before the handler writes the memory-mapped registers or rings a doorbell. A switch into or out of the secure phase quiesces the engine first, with the task queues disabled, and the machine passes through `PAUSE` between the two phases. Any event delivered while in `PAUSE` is a firmware invariant violation and asserts.

Three smaller state machines are alongside the execution loop, each with explicit named states. A two-state process machine marks each process slot idle or running. A three-state cache-request machine moves a resident chain through available, in-use, and invalidating. A two-state output-set machine marks a chained output buffer available or in-execution.

29.7 Per-run statistics buffer

Listing 29.4 traces the per-run statistics-buffer path, from the host size query and map through the firmware completion write.

Listing 29.4. The per-run statistics-buffer path, from the host size query and map through the firmware completion write.

```
host: stats-buffer size-get            -> {hdr 0x28, evDesc 0x14, dbg 0x10, perEngine 0x20}
host: PreMap stats buffer {addr,size}  -> DART-map into the firmware aperture
        if addr == 0 or size == 0:  cbz skips the map; later writes have no destination
host: procedure call (inference)
firmware completion (CAneProgramManagerH11):
        per engine: copy the 16-byte counter quad from engine_block+0x30 into a descriptor
        header: magic 0x0101, per-call timing, per-call counter, descriptor-area size
host: reads validCount, logEvents, statEvents back out of the mapped buffer
```

The firmware reports per-run timing and event counts into a statistics buffer the host supplies. `CAneProgramManagerH11` packs the records on inference completion, but it never allocates the destination. The host first queries the layout sizes, then maps a buffer through the address-translation manager, and the firmware writes into that mapping at completion.

The record begins with the `0x28`-byte header of Table 29.9. The first two bytes are the magic value `0x0101`, a version stamp.

Table 29.9. The `0x28`-byte statistics-buffer header `CAneProgramManagerH11` writes on inference completion.

Offset	Width	Field
+0x00	2	magic `0x0101`
+0x04	8	per-call timing value, from the real-time-kernel timebase
+0x0c	4	word from the call object
+0x10	8	reserved, cleared
+0x18	4	word from the call object
+0x1c	8	per-call counter value
+0x24	4	descriptor-area byte length, the total size minus `0x28`

After the header the firmware copies one descriptor per active engine, looping over up to 32 engines at a stride of `0x48` bytes in the engine table. Each descriptor holds the engine identifier and type, call index, and engine index, followed by a 16-byte counter quad read from `engine_block+0x30`. The size query returns four constants the host uses to size the buffer: a `0x28`-byte header, a `0x14`-byte per-event descriptor, a `0x10`-byte per-debug-event record, and a `0x20`-byte per-engine descriptor. The per-engine values are software-tracked latency and event accumulators maintained by the firmware profiler, not raw register reads. The buffer also holds running totals the host reads back: a valid-record count, log-event total, and stat-event total.

The read is gated by a host-side null check. The map handler loads the host buffer address and size from the command, and a `cbz` on each skips the address-translation map when either is null, as Listing 29.5 shows.

Listing 29.5. The null check that skips the statistics-buffer map when the host supplies no address or size.

```
PreMap stats buffer:
  x22 = host stats-buffer address     ; from the command
  cbz x22, skip                       ; address == 0 -> no map
  x23 = host stats-buffer size
  cbz x23, skip                       ; size == 0    -> no map
  ... DART-map {x22, x23} into the firmware aperture ...
```

When the host never maps a buffer, the mapped pointer stays null, the completion writer has no destination, and every counter value the host observes is null. A separate mechanism reads the performance-monitoring-unit aperture at the fixed base `0x2_8e08_c000`. Those reads are 32-bit power-status registers at a stride of eight bytes, up to five entries, whose low byte is a per-engine power state. They drive power and frequency-scaling decisions and are never packed into the statistics buffer. The kernel hardware performance-counter block is a third mechanism again, armed and drained by the kernel rather than the firmware, and the firmware does not touch it.

29.8 Faults and recovery

A failed invariant routes into the kernel fault handler, which produces a post-mortem dump in a fixed order. The dump records the exception class, fault-address and exception-syndrome registers, full general-purpose register file, cache-controller error registers, a restart count that persists across restarts, and a replay of the recent-command ring newest first. The firmware keeps that command ring so a crash dump shows what the engine was running when it died. The platform exception handler stringifies four exception classes for the dump: non-maskable interrupt, interrupt, fast interrupt, and synchronous abort. The dump names the faulting task by name, marks whether the fault occurred in interrupt context rather than a task, gives the processor identifier, and prints a call stack.

The fault model below a true exception is abort-and-recover per task queue. The firmware aborts a stuck queue through the engine driver's hardware-abort path, and an abort that does not clear within the two-second deadline escalates to a panic. A dedicated timeout interrupt fires on a global stuck flag, logs an event record, and drives the abort. When a per-queue abort cannot clear the fault, the firmware escalates to a host-coordinated reset: it takes a reset mutex and sends a reset notification holding the task-queue range the reset spans, telling the host to tear down and re-initialize.

The engine cannot write a filesystem, so it depends on the host to drain the dump. After staging the dump the firmware blocks while the host pulls it section by section, then the host resets the channel and re-initializes the firmware to the ready handshake.

Firmware status is not a flat numeric enum. A namespace of notification commands sent to the host holds it, alongside inline return values at the point of failure. A generic per-channel error notification holds an opaque error index, a reset notification holds the task-queue range the reset spans, and a tile-sync error notification reports a data-movement error.

Faults divide into four dispositions. A version-minor mismatch warns and continues. A command addressed to a torn-down process or a wrong-state cache handle is dropped and counted. A command failing integrity, section, or argument validation is rejected without running. A failed invariant, a processor exception, or an uncorrectable cache error faults into the dump-and-reset path.

The recovery sequence is serialized behind a single command gate. A fault stages a firmware dump, cancels outstanding commands, and re-boots the firmware to the ready handshake before the gate frees, so a fault has a recovery cost rather than completing instantly.

30 Host-to-firmware command protocol

SUMMARY

The host drives the engine with one message protocol over a shared mailbox: a fixed `sCSneControllerCmdHdr` header naming a program, process, and procedure, followed by a command body. The dispatched command space on the M1 is 93 identifiers, `0x00` through `0x5c`, indexed by `eCSneCmdId`, with `0xFFFFFFFF` reserved as the invalid sentinel. A submission posts to a bound ring endpoint, the controller validates and queues it by priority, runs it, and returns completion as a firmware-to-host notification in the same identifier space.

The in-package controller is a real-time-kernel firmware image, and the host communicates with it through a ring-buffer endpoint with a single command vocabulary, `CSNE_CMD_*`, the controller-side neural-engine command set. This chapter covers the wire protocol between the host-side execution model of chapter 2 and the program file layers of chapter 23.

30.1 Command header

`sCSneControllerCmdHdr` prefixes every ring message with the six fields of Table 30.1 that name the command and bind it to a loaded program.

Table 30.1. The six fields of the command header.

Field	Offset	Width	Meaning
`id`	`0x00`	`u32`	the `eCSneCmdId` selector, logged as `%#04x`
`size`	`0x04`	`u32`	byte length of the command body that follows
`priority`	`0x08`	`u32`	scheduling band, `0..7`
`programId`	`0x0c`	`i32`	loaded-program slot, `-1` when invalid
`processId`	`0x10`	`i32`	per-program process instance, `-1` when none
`procedureId`	`0x14`	`u32`	index into the program's procedure table

The dispatcher validates the full message length before reading the body: `maxCmdBufSize >= sizeof(*pMsg) + pCmdHdr->size`. The byte offsets follow the field order under a packed 32-bit layout, and the field presence and widths are read from the firmware's assert and log format strings. The `priority` field holds two scheduling classes: values `0` and `1` are the privileged realtime band and values `2` through `7` are the normal queue. These are encoded by the firmware checks `(priority >= 0 && priority <= 1) || (priority >= 2 && priority <= 7)` and the normal-only `(priority >= 2 && priority <= 7)`, under the full-range bound `fwPriority < 8`. The signed `programId` and `processId` hold `-1`, which reads as `0xFFFFFFFF`, as the unbound-slot marker. The firmware resolves the pair against `progProcInfo[builtinProgramId]` with

the fields `.valid`, `.progId`, and `.procId`, under `builtinProgramId < maxProgramServerSupported` and `id < CAneBuiltInNetworkId::ANE_NET_TOT`. The resident cache-request commands prepend one further field, an opaque 64-bit `cacheHandler` logged as `cacheHandler: 0x%llx`, naming a long-lived request object rather than a single submission.

The scheduler maps the eight priority levels onto the two admitted classes of Table 30.2.

Table 30.2. The eight priority levels and the two scheduling classes they map onto.

Level	Class	Firmware gate
0, 1	high, realtime	`(priority >= 0 && priority <= 1)`, separately admitted
2 through 7	normal queue	`(priority >= 2 && priority <= 7)`

The full-range bound is `fwPriority < 8`, with `maxPriorityNbr <= 8`, and a submission is queued under the scheduler key `(userId, jobId, fwPriority, hwqId, tqId, queueId)`, where `queueId < priorityQ Count` selects the schedule-info slot.

30.2 Command set

The dispatched space is 93 entries, identifiers `0x00` through `0x5c`. The firmware enumerates them in one contiguous, ordered string table, `CAneControllerStringsH11`, which begins at firmware file offset `0xa83b4` and runs sequentially. The command logger indexes that table by identifier to render each `CMD = %#04x [%s]` log line. The dispatched identifiers group into subsystems rather than ninety-three unrelated commands: lifecycle and power bring the controller and engine up and down, property reads and writes config and registers, stats drives printing, profiling, and tracing, ipc binds endpoints, buffer configures and recycles the channel pool, and program, execution, cache, and secure carry the inference path itself. The complete set in numeric-identifier order, with each command's direction, subsystem, and recovered request struct, is in Table C.15.

Most identifiers are host-to-firmware requests; fourteen in the same space are firmware-to-host notifications, and one, `BACK_CHANNEL_RPC` at `0x5b`, flows in both directions. The sentinel `ECSneCmdId_Invalid`, value `0xFFFFFFFF`, is the no-command marker and has no table entry. Two further strings the firmware contains are not dispatched identifiers: `CSNE_CMD_START` is a standalone log string with no table slot, and `CSNE_CMD_IPC_ENDPOINT_TYPE_DATA_CHAINING` is an endpoint-type enum value rather than a command.

30.3 Command lifecycle

A submission moves through the four stages of Figure 30.1: the host posts, the controller picks it up, executes it, and notifies completion.

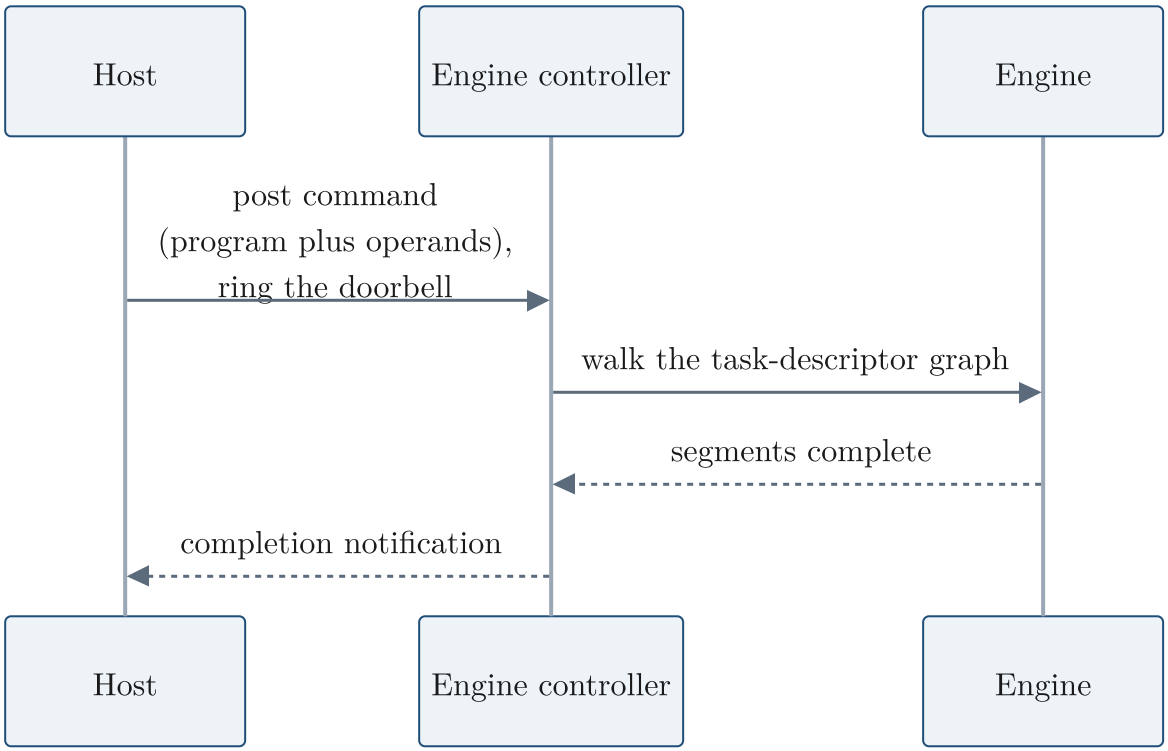

Figure 30.1. A submission moving between the host, controller, and engine.

The host posts by writing the header and body into the ring-buffer slot of a bound endpoint and signalling the controller. The host must bind the endpoint first with `CSNE_CMD_IPC_ENDPOINT_SET`, and the execution-loop endpoint is required to be the data-chaining type: the firmware asserts `endPointId == CSNE_CMD_IPC_ENDPOINT_TYPE_DATA_CHAINING`. The flag `ipcEndpointSetDone` guards endpoint state, false before the bind and true after, so the firmware rejects a submission on an unbound endpoint before any work begins.

The controller picks the message off the ring and dispatches on `id`. For a procedure call it validates the named procedure against the loaded program, `procedureId < pProg->pProgram->procNbr`, and resolves the content type through `getProcedureCallType(programId, procedureId)`, which must fall in the admitted set or the firmware rejects the call with `getProcedureCallType(): invalid procedureId=%d, tot=%d`. The validated call enters the priority queue keyed by `(userId, jobId, fwPriority, hwqId, tqId, queueId)`, and a full queue is reported by `[isHWReady] priority queue is full!`. The controller drops a submission whose target program is not running, logged for the inference path as `CSNE_CMD_INFERENCE_CALL dropped. prog %d process %d not running.`

Completion and asynchronous status return as firmware-to-host notifications that reuse the command identifier space: a per-program event as `CSNE_CMD_PROGRAM_EVENT`, a data-chaining stage completion as `CSNE_CMD_DATA_CHAINING_EVENT`, a prefetch completion as `CSNE_CMD_PREFETCH_DSID_EVENT` logged `Dsid (%d) Event (0x%x)`, and an error as `CSNE_CMD_CH_ERROR_NOTIFICATION`. Performance signposts captured during a run originate in the firmware as `CSNE_CMD_CH_SIGNPOST*` notifications, not as a host-side instrumentation artifact.

30.4 Procedure-call body

The body of a procedure call is `sCSneCmdProcedureCall`, a shared container whose trailing variable-length arrays spill into an over-allocated `sCSneCmdProcedureCallCopyContainer`. The counted fields after the header describe the call's shape, each bounded by a firmware assert, as Table 30.3 gives.

Table 30.3. The counted fields of the procedure-call body.

Field	Type	Bound
`nbrOfOutputBufferSets`	u32	`== 1` on the M1 generation
`nbrOfWaitEvents`	u32	`nbrOfWaitEvents + nbrOfSignalEvents > 0`
`nbrOfSignalEvents`	u32	`<= 16`
`nbrOfCustomBars`	u32	`<= 32` on the wire
`stats.buffer`	u64	`>= pProc->pStatArrayBaseOrig`
`stats.size`	u64	covers the three stats records

The firmware logs the decoded shape on one line: `Bufs = %d Priority = %d CustomBars = %d WaitEvents = %d SignalEvents = %d`. A wait or signal event is a fixed 24-byte record, `{ uint32 type; uint32 targetMask; uint64 id; uint64 value; }`, recovered from the event log strings `waitEvent[%d]: type=%d mask=%x id=%lld value=%lld` and the matching signal line.

The procedure-call type gate admits only certain content types: `getProcedureCallType(programId, procedureId)` must return `ContentType_0`, `_3`, or `_4` for a plain `PROCEDURE_CALL`, and `_0` or `_3` on the cache-request path, under `procedureId < pProg->pProgram->procNbr`. The `WITH_CUSTOM_BARS` variant adds a trailing custom execute-order array, whose offset and length are bounded by `callSize <= customExecuteOrderArrayOffset`, `0 < nbrOfCustomExecuteOrder`, `nbrOfCustomExecuteOrder <= 128`, and `(customExecuteOrderArrayOffset + nbrOfCustomExecuteOrder * sizeof(uint32_t)) <= sizeof(sCSneCmdProcedureCallCopyContainer)`. Each custom bar holds `type == ANE_CUSTOM_BAR_GENERIC` and `config == ANE_CUSTOM_BAR_HW_32BIT`, the on-wire count is bounded `nbrOfCustomBars <= 32`, and a blob in the compiled program may hold up to 128. The `WITH_SIGNAL_EVENTS` variant adds the `procCallWithSignalEvents` sub-struct, bounded by `nbrOfSignalEvents > 0 && nbrOfSignalEvents <= 16`, with the sub-network custom execute-order disallowed (`0 == bSubNetworkCustomExecuteOrder`).

Table 30.4 collects the per-call and per-request numeric limits so a host can size a submission ahead of dispatch.

Table 30.4. The per-call and per-request numeric limits on the M1 generation.

Quantity	Limit	Firmware gate
trigger input buffers	16	`nbrOfInputBuffers <= (16)`
active shared events	fewer than 2	`sharedEventsActiveNbr < 2`
task-descriptor partitions on the M1 single path	1	`1 == nbrOfTdPartition`
ANE requests in list on the M1 single path	1	`1 == pCacheReq->nbrOfAneRequestInList`

The resident cache-request family is the firmware substrate behind keeping a buffer set on the engine across calls. An install command at `0x45` returns the 64-bit `cacheHandler` and creates a long-lived request indexed internally by `cacheReqIdx` in the range `0 <= cacheReqIdx < maxCacheRequest`, allocated from a `CIndexPool`. A trigger command at `0x46` names that handle and fires it, holding an `execTimestamp` that must be strictly monotone against `pCacheReq->lastTriggerExecTimestamp`, and checking that the handle is live first. The firmware drops a trigger on a handle not in the `ANE_DATA_CHAINING_CACHE_REQUEST_IN_USE` state with

`cacheHandler (0x%llx) in wrong state (%d) for trigger. trigger dropped`. The recycle command at `0x47` returns a consumed output buffer to a resident request, the invalidate command at `0x48` tears it down, and the group select at `0x4f` picks the active member of a buffer-sharing group. The force-disable at `0x4d` is the global disable, which the firmware admits only when set true. The firmware constrains chained buffers to the low address window: a data-chaining buffer descriptor must hold `buffer.type == eCSneBuffDescriptorType_OutputBuffer`, `outputBufferSetId < nbrOfOutputBufferSets`, and a device address whose high 32 bits are zero (`(buffer >> 32) == 0`).

The inference call at `0x5a` is the higher-level submission path, which the firmware expands into one or more `CANE_SUB_PACKET_CMD_PROCEDURE_CALL` sub-packets, holding a pre-mapped property buffer and validating the buffer count against the program descriptor: `pMsg->bufNbr == 1 + pAneProgramDesc2->procedures[procedureId].numIoBuffers`. The `PREMAP_BUFFER` command at `0x4a` pre-maps that property buffer, hard-validated for exactly one operation and exactly one buffer (`pPreMapOp->nbrOfBuffers == 1`) with nonzero address and size. The procedure call at `0x41` is the lower-level direct form addressing the program, process, and procedure triple; both share the `sCSneCmdProcedureCall` body and the same priority scheduler.

The property channel exposes a register-poke surface that the host uses to read and write firmware registers directly. The read command at `0x1f` and the write command at `0x1e` each hold a two-field body, recovered from the `IPC Writer regAddr 0x%lx regValue 0x%x` log line as `{ uintptr regAddr; uint32 regValue; }`, and a write checks `regValue != 0`. The back-channel RPC at `0x5b` is a firmware-initiated call into the host driver: the firmware client posts directly into the direct-proc-call event pool, gated on `GetDirectProcCallEventPoolAvailNum() >= 2`, tracks outstanding calls in `RPCEventMap[%d]`, and reports a stuck slot with `RPCEventMap[%d] is dirty. (No ack from driver)`.

30.5 Recovered command-id enum

The dispatched table above is the contiguous logger ordering, but the firmware also holds the canonical protocol enum as a packed `{char* name; u64 id}` array at vaddr `0xf5ba8` (file offset `0xf9ba8`, in `__DATA.__const`). This is the `ECSneCmdId` enum, 94 named entries plus a trailing zero-id terminator, and the id is the literal 16-bit value held on the wire at message `+0x4`. The array is reached through a fixed-up pointer with no `adrp`+`add` cross-reference, consistent with a shared name-lookup helper. The protocol enum and the dispatched logger ordering are not the same numbering in the `0x2xx` region, so this enum is the wire vocabulary while the table in the preceding section is the firmware's own logger index.

The named ids partition by high byte into the families of Table 30.5, each a contiguous range.

Table 30.5. The six command-id families of the `ECSneCmdId` enum at `0xf5ba8`.

Family range	Count	Subsystem	What the family covers
`0x01`–`0x34`	52	control, config, power, debug	`START`, `STOP`, `RESET`, `CONFIG_GET`/`CONFIG_GET_EXT`, `PRINT_ENABLE`, `BUILDINFO`, `BOOT`, `PING`, the `TIMEPROFILE_*` trio, `POWER_DOWN`, `POWER_DEVICE_ON`/`OFF`, the PMU-base and dynamic-powergate setters, `IPC_ENDPOINT_*`, the `CH_BUFFER_*` pool and recycle set, `CH_PROPERTY_READ`/`WRITE`, `TRACE_ENABLE`, `RESOURCE_INFO_GET`, `STATS_BUFFER_SIZE_GET`, `SUSPEND`, `DSID_SET`, `MCACHE_SIZE_GET`, `SECURE_MODE_START`/`STOP`, `EXCLAVE_MODE_START`/`STOP`, `QUIESCE_STATE`, `CPU_LOAD_GET`
`0x100`–`0x108`	9	channel notifications	firmware-to-host async messages: `CH_ERROR_NOTIFICATION`, `CH_POWER_CONTROL`, the `CH_SIGNPOST*` and `CH_SIGNPOST64*` single and grouped variants, `CH_RESET_NOTIFICATION`, `CPU_LOAD_NOTIFICATION`, and the secure-mode resume transition
`0x200`–`0x212`	19	program, process, procedure call	the inference path: `LOAD_PROGRAM`/`UNLOAD_PROGRAM`, `CREATE_PROCESS`/`TERMINATE_PROCESS`, `PROCEDURE_CALL`, `LOAD_AFPP`/`UNLOAD_AFPP`, `PROGRAM_INTERFACE_VERSION_CHECK`, the `PROCEDURE_CALL_*` cache-request and custom-bar and signal-event variants, `PREMAP_BUFFER`, `FORCE_DISABLE_CACHE_REQUESTS`, and the time-management sync error
`0x300`–`0x305`	6	events	`SET_ACTIVE_CACHE_REQUEST_IN_GROUP`, `PROGRAM_EVENT`, `USER_EVENT`, `DBG_EVENT`, `DATA_CHAINING_EVENT`, `PREFETCH_DSID_EVENT`
`0x400`–`0x404`	5	id handshakes	`SECURE_MODE_EVENT`, the `REQUEST_PROGRAM_ID`/`RETURN_PROGRAM_ID` and `REQUEST_PROCESS_ID`/`RETURN_PROCESS_ID` allocate-and-free handshakes
`0x7000`, `0xff00`, `0x0000`	3	outliers	`INFERENCE_CALL` at `0x7000`, `BACK_CHANNEL_RPC` at `0xff00`, and `DEBUG_COMMAND_DATA_CHECK` at `0x0000`

30.6 Firmware fast path

The live on-chip dispatcher is `CAneEngineExeLoop::dispatch` at vaddr `0x4a42c`, which reads the 16-bit id with `ldrh w8,[x21,#4]` and decodes it through a compare tree rather than a jump table. The tree implements only about ten ids, all on the procedure-call and inference fast path, and every other id falls through to a default arm at `0x4aa4c` that logs `Cmd 0x%x is not supported through ExeLoop cmd channel` and then asserts, panicking the firmware. Sending a non-fast-path id on this channel is thus a hard fault on the live device, consistent with the unrecoverable behavior of a DART or fence fault.

The implemented arm of the compare tree dispatches each id to a PAC-signed virtual method on the engine object, reached through the `ldr/autda/blraa` pattern at a fixed vtable offset. Each such slot is an arm64e auth-rebase chained pointer in `__DATA.__const` whose low thirty-two bits hold the raw target address, matched to its call site by the `movk` discriminator, so the seventy-six-slot table resolves statically. The procedure-call slot at `+0x200` targets `0x7374c` and the inference slot at `+0x190` targets `0x74510`. Table 30.6 gives the ids the ExeLoop dispatcher implements, with each id's vtable offset and behavior.

Table 30.6. The ids the ExeLoop dispatcher implements.

ExeLoop id	Meaning	Vtable offset	Behaviour
`0x2d`	`CH_DATA_FILE_LOAD2`	`+0x250`	guarded by the engine flag at `[x19,#0x2bb]`
`0x204`	procedure call	`+0x200`	validate `(programId, procId)` then dispatch, the inference trigger
`0x209`	cache-request trigger	`+0x218`	logs `cacheHandler 0x%llx`
`0x20a`	cache-request submit	cache path	cache-request submission
`0x20c`	procedure call with bars	`+0x208` region	custom buffer-address-register call
`0x211`	procedure call with events	events path	event-signalled call, force-disable-cache arm
`0x212`	procedure-call variant	`+0x210`	shared reply and verify path
`0x404`	`RETURN_PROCESS_ID`	`+0x190`	process-id handshake reply
`0xff00`	`BACK_CHANNEL_RPC`	func `0x47818`	length-and-magic guarded host-to-firmware payload

The procedure-call arm at id `0x204` decodes a fixed message layout, recovered from the debug line `[EXELOOP] CMD=%#04x [PROC CALL] at %lld : ProgId=%d ProcId=%d Proc=%d Pri=%d`: a `u16` id at `+0x04`, then `u32 programId`, `procId`, `proc`, and `priority` at `+0x08` through `+0x14`. Its argument validator at `0x48644` rejects a `programId` of `144` or more and a `procId` of `288` or more, then indexes a per-program validity byte at `engine + programId * 0x20 + 0x9948`, so a 144-entry program-slot table of stride `0x20` is at engine offset `0x9948`. The back-channel RPC handler at `0x47818` caps its payload length at `0x800000` and requires the magic `0x55aa55aa` before processing, the same back-channel surface as id `0x5b` in the dispatched table.

The `0x00xx` config, power, trace, and profile commands are present in this image only as enum names and as the per-command struct and assert strings, for example `sCSneCmdReset`, `sCSneCmdConfigGet`, `sCSneCmdTraceEnable`, and `sCSneCmdPowerSupplyControl`. Exhaustive `adrp`+`add` and constant-pointer scans find zero references to any of those handler strings anywhere in `__text` or `__DATA.__const`, so the strings are linked in from a shared protocol header while the handlers are not compiled into this M1 and H13 build. `AppleH11ANEInterface` and `AppleANELoadBalancer`, the kext that owns power, IPC-endpoint, and buffer management, service those config-command handlers host-side, so their status is opaque by absence rather than undecoded. The `0x01xx` notification family is firmware-to-host output emitted on the channel rather than dispatcher input, so it does not appear in the ExeLoop compare tree either.

31 Power and thermal

SUMMARY

The engine runs at one fixed clock with no voltage or frequency scaling, and manages power through a peak-power regulator armed once at power-up by five loop-free stores with the control word `0x11`. Idle reads 0 W with rails off, a sustained convolution holds flat at about 1.66 W over 176 seconds, and the densest peak reaches about 4.7 W at 6.96 TFLOP/s. The firmware has no thermal logic and reads no temperature; thermal protection is off-engine in the system power manager.

The engine runs at one fixed operating point and manages power through a peak-power regulator, not a frequency sequence. The system power manager handles voltage and temperature off-engine.

31.1 Boot and device tree

The engine appears in the device tree as a single node `ane0@84000000`, class `AppleARMIODevice`, `compatible = "ane,t8020"`, `device_type = "ane"`. The match key `ane,t8020` is the kext bind key even on a later die, since the engine block has the t8020-generation identity string across the whole M1 family. Table 31.1 gives the properties of that `ane0` node, with the decoded value and meaning of each.

Table 31.1. The properties of the `ane0` device-tree node, with decoded value and meaning.

Property	Decoded value	Meaning
`compatible`	`"ane,t8020"`	the kext match key
`ane-type`	`0x60`	engine variant identifier
`reg` bank 0	`0x2_8400_0000`, length `0x200_0000`	the 32 MB engine control aperture
`reg` bank 1	`0x2_8E08_0000`, length `0xC02C`	the power-manager slice
`interrupts`	`0x302`	the engine interrupt number
`clock-ids`	`0x13e`, `0x13f`, `0x140`, `0x141`	four clock identifiers, one per compute cluster
`clock-gates`, `power-gates`	`0x1cf`	the single clock and power-gate index for the whole block
`asc-dram-mask`	`0x1F0_0000_0000`	the high-address window the coprocessor may reach in DRAM
`segment-names`	`__TEXT;__DATA`	the firmware segments the loader places
`pre-loaded`	`1`	the firmware is resident in DRAM before the kernel attaches

Bank 0 is the engine control aperture at absolute physical `0x2_8400_0000`, 32 MB, holding the coprocessor registers, mailbox doorbells, and engine register file. Bank 1 is a small power-manager slice at `0x2_8E08_0000`, length `0xC02C`, whose base matches the base of the system power-manager node `pmgr@8E080000`, tying the engine's power-gate `0x1cf` directly to the power-management register neighborhood. The four clock identifiers `0x13e` through `0x141` are one per compute cluster, matching the four-cluster geometry of the power model.

The firmware is not read from disk at first boot. The boot loader verifies the signed firmware image and pre-loads its segments into the carve-out named by the node's `segment-ranges`, marked by the property `pre-loaded = 1`. The kernel driver then attaches to `ane0`, hands the already-trusted image to the engine's

coprocessor over the mailbox, and the coprocessor boots. The image itself is a bare uncompressed and unencrypted preload Mach-O wrapped in an Image4 container with the payload tag `anef`, and it has no embedded manifest, keybag, or code-signature load command of its own. The per-device personalized boot ticket enforces its authenticity externally, listing the `anef` object digest alongside the kernel, boot-loader, and device-tree digests and binding to the device by its unique chip identifier. A firmware image cannot be transplanted to a different machine.

The power and peak-power tunables the firmware consumes are not on `ane0`. They are on `pmgr@8E080000`: the master enable `ane-dpe = 1`, calibration vector `dpe-ane-data` (ten 32-bit words), and per-operating-point current ceiling table `ifane-max`. The ten calibration words decode to `[9384, 661, 1323, 2641, 2395, 844 89, 163813, 51491, 178500, 51491]`, the per-platform coefficients that convert dynamic-power-estimation activity counts to energy, with the repeated value `51491` as a shared scale factor. The `ifane-max` table is rows of an index key and three current ceilings: the head rows are index `0x1f8` with `{0x18000, 0x17333, 0x17333}`, index `0x2e8` with `{0x1fd70, 0x1fd70, 0x1fd70}`, index `0x498` with `{0x2deb8, 0x2deb8, 0x2deb8}`, index `0x6a8` with `{0x428f5, 0x428f5, 0x428f5}`, and index `0x858` with `{0x628c5, 0x65666, 0x65666}`. The first column is a monotone activity-keyed operating point, and the three trailing columns are the peak-current ceilings, one per energy-accumulator partition, that the peak-power regulator clamps against. A parameter-push call at init delivers these, and the firmware arms the power blocks from them.

31.2 Power model

Table 31.2 collects the decoded boot, power, and clocking facts with their values and sources.

Table 31.2. The decoded boot, power, and clocking facts; the register-init constants are decoded in Appendix C.

Fact	Value	Source
Engine node	`ane0@84000000`, `compatible "ane,t8020"`	device tree
Engine aperture	`0x2_8400_0000`, 32 MB	device tree
Power-manager slice	`0x2_8E08_0000`, length `0xC02C`	device tree
Power-block base	`0x2_6b8f_0000`	firmware disassembly
Voltage base (opaque)	`0x2_3b70_c008`	firmware disassembly
Power-block arm	5 fixed stores, no loop	firmware disassembly
Peak-power control word	`0x11`	firmware disassembly
Compute clusters	4, gated independently	device tree and firmware
Idle power	0 W, rails off	M1/H13 measured
Sustained convolution	≈ 1.66 W flat over 176 s	M1/H13 measured
Densest peak	≈ 4.7 W, 6.96 TFLOP/s	M1/H13 measured

The engine runs at a single fixed clock with no local voltage or frequency scaling of its own; any frequency change is the system power manager's to make externally. The only clock string in the firmware is the boot-time timebase report; the image holds no frequency table, no enumerated operating points, and no voltage string. Voltage is the system power manager's concern, reached only through an opaque power-management base address (`0x2_3b70_c008`) that the firmware never interprets.

Two controls modulate performance, neither of them frequency. The first is how many of the four compute clusters are powered. The second is a peak-power regulator that limits activity under a fixed budget. That regulator is the engine's substitute for a frequency and voltage sequence, and it has three parts the firmware arms once at power-up. Dynamic power estimation counts switching activity to estimate instantaneous dynamic power. Peak-power tracking watches that estimate against the budget and applies back-pressure on issue when the work would exceed it, throttling activity within the fixed clock rather than dropping frequency. Leakage and energy estimation wires the result into the system power manager. Three accumulator partitions hold the running energy total that the energy-model telemetry channel reports.

Arming this model is loop-free and idempotent, the five memory-mapped stores of Listing 31.1 at the fixed base `0x2_6b8f_0000`, gated on two per-instance enable flags at offsets `+0x95` and `+0x96`, with no ramp and no per-frequency programming.

Listing 31.1. The five-store arm sequence for the power-estimation and peak-power blocks, with the fixed control words at their decoded memory-mapped offsets.

```
this+0x95 (byte)  : dynamic-power-estimation mode enabled?   (tested at entry)
this+0x96 (byte)  : peak-power-tracking mode enabled?

store  0x11             -> [0x2_6b8f_0000]   ; peak-power-tracking control word
store  (this+0x96 ? 1 : 0) -> [0x2_6b8f_0004] ; dynamic-power-estimation control word
reg = [0x2_6b8f_4000]; reg |= 1; store -> [0x2_6b8f_4000]  ; leakage-and-energy enable bit
store  0x3fff           -> [0x2_6b8e_c42c]   ; trailing control word
store  0xf              -> [0x2_6b8e_c5dc]   ; trailing control word (= 0x2_6b8e_c42c + 0x1b0)
```

The peak-power control word is `0x11`, the leakage-and-energy block is enabled by setting bit 0 of the register at `0x2_6b8f_4000`, and the sequence finishes with two fixed trailing control words, `0x3fff` at `0x2_6b8e_c42c` and `0xf` at `0x2_6b8e_c5dc`. This estimation aperture at `0x2_6b8f_0000` is distinct from the power-gating base `0x2_3b70_c008`, so power gating and power estimation are separate hardware blocks. The system-on-chip leakage-and-energy enable bit folds the result into the system power manager. The rails drop fully when the engine goes idle, so the estimation block loses state and the firmware re-applies the retained calibration on every wake.

The four compute clusters gate independently through the stride-eight status array of Table 31.3, polled by the host driver until each domain's low byte reads `0xff`, meaning all eight power straps have settled.

Table 31.3. The power-domain status registers, a stride-eight array at `0xc000 + domain*8` polled until each low byte reads `0xff`.

Aperture-relative offset	Poll	Domain
`0x2c8`	`expect 0xff, mask 0xff, retries 5000`	top-level ready and fabric gate, validated first
`0xc000`	`expect 0xff, mask 0xff, retries 5000`	base domain, the always-on control fabric
`0xc008`	`expect 0xff, mask 0xff, retries 5000`	compute cluster 1
`0xc010`	`expect 0xff, mask 0xff, retries 5000`	compute cluster 2
`0xc018`	`expect 0xff, mask 0xff, retries 5000`	compute cluster 3
`0xc020`	`expect 0xff, mask 0xff, retries 5000`	compute cluster 4

The five domains are one always-on base domain and four independently gated compute clusters; there is no fifth cluster. The firmware brings the base domain up first and adds a compute cluster only when work needs it, which is the mechanism behind the measured idle of 0 W. The engine is in the all-off state until a procedure call arrives, then brings the base domain up to a wait state and gates in the clusters, so the idle floor is rail-off rather than clock-gated. A free-running firmware timestamp pair is in the same aperture at `0x1170000` for the low word and `0x1170004` for the high word, distinct from the engine-internal timebase, and the firmware zeroes four scratch registers at `0x1840048` through `0x1840054` at init.

For a workload held at the operating point for time t, the energy is

$$E = P\,t$$

with measured constants from a sustained saturating loop on the M1: a dense convolution at $P \approx 1.66$ W held flat over $t = 176$ s, and a densest-packed peak of $P \approx 4.7$ W. At that peak the engine delivers 6.96 TFLOP/s, which gives an efficiency of about 1.5 TFLOP/s per watt in fp16, rising to about 2.6 TFLOP/s per watt on weight-reuse convolutions. The power scales with cluster engagement and lane density, not with any clock change. Relative to a single cluster, throughput rises 1.99×, 3.02×, and 4.00× at the second, third, and fourth clusters. The rail then climbs smoothly from 779 mW to 1429 mW as channels pack the four clusters, with no discrete steps.

31.3 Thermal behavior and the operating point

A scan of the image for temperature, throttle, junction, and similar terms returns nothing. The engine does not read temperature, does not throttle on temperature, and emits no thermal event of its own. Thermal protection is entirely off-engine: the peak-power regulator bounds power, which indirectly bounds heat, and package-level thermal control is in the system power manager, which can delay or refuse dispatch but does not reach into any engine counter.

A 176-second saturating loop sampled thermal pressure every two seconds and read nominal on every one of the 89 samples, with power and throughput flat to within half a percent and a slightly negative drift. A frequency-scaling engine would show a ramp or multiple power modes as it settled; a thermally limited engine would decay. The engine does neither; power and throughput hold flat. The single outbound power signal the firmware emits is a normalized margin level for the host to react to, holding no thermal field.

The first dispatch after idle pays a fixed power-up cost because the engine reaches a true rails-off state between jobs. That cost grows from near zero at back-to-back dispatch to about 0.5 ms once the idle gap reaches roughly 100 ms, then plateaus, which sets the residency window for keeping the engine warm.

31.3.1 Operating points and the absence of local DVFS

The engine has no frequency-or-voltage sequence of its own. The kext delegates operating-point selection to the system power manager: the SoC CLPC against ApplePMGR. The kext holds no clock register addresses. A disassembly of `enableAneSysClock` shows it performs no memory-mapped writes itself; it loads an ApplePMGR service object, authenticates the vtable pointer, and calls a PMGR method with the enable argument set, with the only literal in the path being the string `"ApplePMGR"`. The `ane0` device-tree node publishes the PMGR clock and power-gate identifiers but has no `dvfm-states`, `voltage-states`, or `perf-states` property, so the per-state frequency in Hz and voltage in mV stay held privately by ApplePMGR behind the power-manager base and are not recoverable from the engine binaries.

What is on the engine side is the firmware `H13TunableManager` register-init table, where `H13` is the M1 family. A `TunableManager` descriptor at base `0x2_6b8f_4000` (`aneDpePpt_soc_dpe_lee`, the SoC dynamic-power-estimation control block) holds the 7-step monotonic credit sequence of Table 31.4, written once at firmware bring-up and gated by chip revision.

Table 31.4. The 7-step monotonic DPE/PPT credit sequence in the `aneDpePpt_soc_dpe_lee` block, with the decoded value of each 9-bit field.

Offset	Mask	Value	Decoded
+0x18	0x1ff	0x19	25
+0x1c	0x1ff	0x32	50
+0x20	0x1ff	0x46	70
+0x24	0x1ff	0x55	85
+0x28	0x1ff	0x5f	95

Offset	Mask	Value	Decoded
+0x2c	0x1ff	0x69	105
+0x30	0x1ff	0x73	115

These nine-bit fields are the per-operating-point power-credit and peak-power-throttle thresholds the on-die estimator caps activity against, indexed by the perf state the CLPC selects. Their strictly increasing sequence is the signature of a seven-state perf sequence. The engine has seven operating points whose registers hold throttle credits, not a clock frequency and not a voltage. A parallel eight-step sequence for the ASC block holds the values `4, 7, 13, 20, 27, 36, 46, 59`.

Live read-only dtrace confirms the engine holds one power state through sustained work. A six-layer matmul-and-relu program dispatched in a continuous loop for 16 seconds drove 56,527 dispatches, traced entry-only with no destructive flag. `ChangePowerState`, `populateCLPCPerfInfo`, `power_on_hardware`, and `setPowerStateGated` each fired exactly once at warm-up and never again across the 56,527 dispatches. Every dispatch instead walked a fixed path: `EnableANEClocksAndPower` to un-gate, one `submitWorkToPerfController` that hands an `ANEPerfRequest` and a modeled-performance hint to the CLPC, a `notifyPerfController` start-and-end pair, then `enableDynPowerGating_gated` to re-arm the idle gate. The kext does not walk a frequency sequence per submit; the engine holds a single power-domain state for the whole run while the CLPC drives any frequency change internally and invisibly to the kext. This is the runtime counterpart to the flat power and throughput measured under sustained load: the firmware tracks a single PerfMode, and setting it twice is a no-op.

32 Security and isolation

SUMMARY

The trust boundary for a submitted program is the kernel driver, not the firmware: the driver checks the program signature, on-disk trustcache, and client's code-signing identity before any work reaches the engine, which then runs only structural bounds checks. Secure mode is a working transition that gives one tenant the quiesced, power-cycled engine; exclave mode is inert `mov w0, #0; ret` selector stubs on the M1 but the live, capability-scoped execution substrate on the M5. The engine is a shared timing and occupancy oracle: a confirmed cross-process side channel leaks a co-tenant's presence and a coarse duty cycle at roughly 20 to 50 bit/s, but never a value, weight, or input.

The kernel driver enforces the engine's security model almost entirely above the firmware. It vets every program before the work reaches the engine, secure mode isolates one tenant in time rather than in hardware, and the one confidentiality gap is a timing side channel rather than a data leak.

32.1 Secure and exclave mode

Table 32.1 gives the secure-mode and exclave mechanisms with their status on the M1 and the supporting evidence.

Table 32.1. The secure-mode and exclave mechanisms, with M1 and M5 status and supporting evidence. The M5 was measured with System Integrity Protection enabled.

Mechanism	M1 status	M5 status	Evidence
Program signature check	live, kernel-side		corecrypto link, signature symbol
Vnode trustcache check	live, kernel-side		vnode trust symbol
Firmware program check	structural only, no crypto		bounds and overlap asserts
Secure-mode FSM	live, working		non-secure to secure transition
Secure isolation	temporal, power-cycled	temporal plus exclave capability partition	quiesce plus power-cycle plus pause
Exclave firmware switch	dormant, stubbed	live	`SwitchExclaveMode not supported` on the M1
Exclave host binding	present, real bodies, unbound	bound, proxy active	proxy IOService, recovery FSM
Exclave selector ABI	inert stubs on M1	live implementations	`mov w0, #0; ret` on the M1

The firmware has no cryptography of its own, and the secure-boot chain authenticates it externally. The trust boundary for user-submitted programs is the kernel driver, which checks the program signature and validates the backing file against the platform trustcache before the work ever reaches the firmware; the firmware then performs only structural bounds checks.

The kernel driver enforces three independent checks on a submitted program before it reaches the firmware. The first is a cryptographic signature over the compiled program bytes, run by `AneMachoSignatureCheck` against the platform code-signing trust root, for which the driver links the system cryptography library. The second is

a vnode trust check, `aneVnodeTrustVerification`, that validates the on-disk model file against the platform trustcache and ties a mapped buffer back to its backing file to defeat a map-then-swap race. The third binds a resident program to the submitting client's code-signing identity through `GetTeamIdAndCodeSigningId` and `hasSameCodeSigningId`, so one client cannot attach to another client's resident program or cache. Once the kernel clears the program and loads its sections, the firmware checker runs only bounds, overlap, and type checks under a verification banner, with no hashes, and trusts that the kernel has already vetted the signature and provenance.

Secure mode is a real, working transition that gives one tenant exclusive ownership of the quiesced engine. The state machine moves between a non-secure and a secure phase under the commands `CSNE_CMD_SECURE_MODE_START`, `STOP`, and `RESUME_TRANSITION`, with a firmware-to-host `CSNE_CMD_SECURE_MODE_EVENT` and a host acknowledgement. A boolean `aneSecurePhase` records which side the engine is on, and the transition runs in four steps. The engine first reaches readiness with no pending work. A quiesce command drains in-flight work and disables the task queues. The firmware then power-cycles the engine block across the boundary. The execution loop finally enters a paused state, with the reverse path returning it to a running state. While secure, the firmware silently drops and counts non-secure cache-request triggers. The boundary power-cycles the block and drains every task queue and DMA channel, so a secure tenant starts from a reset engine and neither side observes the other's residue. This is temporal single-engine isolation, not a hardware partition, since the device exposes one physical engine.

The engine firmware has no digital-rights-management or content-decryption code of its own. A full string sweep finds no FairPlay, Widevine, PlayReady, or content-decryption module; the only `CDM` token in the image is `CDMediaBusManager`, a Common-DMA media-bus endpoint manager over the inter-processor ring buffers, not a Content Decryption Module. Secure-mode exclusivity and the host trust chain handle protected media rather than an in-engine content module. The kernel admits a protected workload only after it verifies the program signature and trustcache and puts the engine in secure mode so no other tenant shares it while protected buffers are resident.

Exclave mode is compiled into the loaded binary but dormant on the M1. The firmware commands `CSNE_CMD_EXCLAVE_MODE_START` and `STOP` exist, but the M1 handler returns `SwitchExclaveMode not supported`, and the support is a runtime capability bit rather than a compiled-out feature. The host binding runs ahead of the firmware side. The proxy IOService, secure-to-exclave plumbing, secure-processor handoff, and firmware-recovery state machine all exist as real function bodies in the loaded driver, fifty-one exclave symbols in all. What is dormant splits cleanly, as Table 32.2 classifies each method group: the externally callable selector ABI, the `ANE_Exclave*` methods, is compiled as inert stubs (`mov w0, #0; ret`), while the lower-case infrastructure methods and the proxy service are real.

Table 32.2. The exclave method set, classified by status on the M1 and the M5: real implementations and inert stubs on the M1, all operative on the M5.

Method group	Status on the M1	Status on the M5	Examples
enablement and binding	real bodies	operative	`checkExclaveEnablementStatus`, `setupANEExclaveProxyService`, `ANEExclaveInit`
interrupt and worker plumbing	real bodies	operative	`aneExclaveInterruptHandler`, `aneExclaveUpcallEventHandler`, `aneExclaveWorkerThreadEntry`
query, handoff, and recovery	real bodies	operative	`aneExclaveQuery`, `aneSEPToExclaveHandoff`, `aneExclaveStartFWRecoveryProcess`
operational selector ABI	inert stubs	operative	`ANE_ExclaveLoad`, `ANE_ExclaveEvaluate`, `ANE_ExclaveModeCycle`, `ANE_ExclaveSaveState`

The host-side gate is a two-tier check. A platform call reports whether exclaves are available, and a driver flag reports whether the feature is enabled. The driver-internal proxy service `ANEExclaveProxy` is a fully implemented IOService subclass with real lifecycle bodies, but it binds only when an exclave device-tree node exists, and that node is declared only by a personality the later-generation kext has and the M1 kext lacks. On the M1 the platform call reports unavailable: there is no exclave device-tree node, no exclave proxy binds, and no matching exclave core is present in the firmware bundle, so the proxy is never set up and the live behavior cannot be exercised.

On the M5 this dormant path is the live execution substrate, which confirms the later-generation prediction above. The exclave device-tree node is present, the proxy binds, and the model program loads and runs inside a capability-scoped Swift secure component, `com.apple.aneexclave`, reached from the kernel only through a typed Tightbeam channel. The component holds explicit segment-access capabilities and nothing more: the seven per-client address-translation contexts the IORegistry exposes as `mapper-ane0-iso1` through `iso7`, plus thirty-two exclave memory regions. Because the program executes behind this boundary, the per-task-descriptor performance counters are produced secure-side and withheld from an unentitled host, which is why their live values stay gated. The M5 ran with System Integrity Protection enabled, so this is the enforced posture rather than an artifact of lowered security, and it adds a structural capability and address-translation layer to the temporal single-engine isolation the M1 provides.

32.2 Cross-process isolation and the timing oracle

The engine is single-in-flight per die and shares firmware and DART state across clients, so two processes that both submit work contend for one queue. That contention produces a cross-process timing side-channel, confirmed on the M1, that leaks engine occupancy while leaving data confidentiality intact.

A measuring process running a single tiny matmul in a continuous zero-copy loop is near the dispatch floor, where it is sensitive to anything that delays its turn at the engine. Alone, that process reads a median per-call latency of 153 microseconds. The moment a second process holds the engine with a heavy compute-bound load, the measuring process jumps to a median of about 355 microseconds, a 2.3x increase, because the shared single-in-flight queue serializes its calls behind the contender's. Toggling the contender on and off produces a distinct square wave in the victim's latency that tracks the schedule at the exact toggle edges with no false transitions across a 10-second run. Table 32.3 gives the victim per-call latency as the contender toggles between idle and heavy.

Table 32.3. Victim per-call latency tracking a contender toggled on and off on a shared M1 engine, bucketed at 250 ms.

Contender state	Victim latency p50	samples
off (idle)	152 microseconds	32 499
on (heavy)	355 microseconds	10 943
delta	+202 microseconds (2.33x)	

The channel reads occupancy, not workload magnitude. A near-fixed serialization step of about +178 microseconds appears the instant the contender holds the queue at all, so even its tiniest workload pays almost the full penalty. Beyond that step there is only a weak monotonic trend, from 333 microseconds to 395 microseconds as the contender's work grows by a factor of roughly 64. The victim thus reliably learns that a co-tenant is running and a rough busy fraction, but cannot finely size individual operations from latency alone. The signal is engine-queue serialization rather than unified-memory bandwidth: a concurrent 1 GB-per-loop CPU memory-copy hog moves the victim's latency by only 4 microseconds (a factor of 1.03), so only engine contention drives it.

As a covert or signature channel the resolution is about 20 to 50 ms. Single contender pulses down to 20 ms were each detected. A random 16-bit pattern sent at 0.4 s per symbol was recovered at 14 of 16 bits with a

median-threshold detector, with the two tail errors attributable to end-of-run clock drift rather than channel capacity. This puts the practical channel at roughly 20 to 50 bit/s, enough to read a co-tenant's presence, the timing of an inference, and a coarse duty cycle.

Data isolation holds under the same contention. Three processes ran distinct programs concurrently, each with distinct seeds and weights and each self-checking its output against its own fp32 reference for 3000 rounds. All 9000 of 9000 checks were correct with zero mismatches and a maximum relative error of 0.0003, which is pure fp16 rounding. No output ever reflected another process's inputs or weights. A separate run held 60 distinct program handles across three processes (20 each) and re-executed every one after all were loaded, with 60 of 60 compiling and re-running and no cross-process eviction at that scale.

The root cause is single-in-flight-per-die scheduling, and there is no per-call isolation control exposed; masking a sensitive workload's duty cycle requires coarse-grain time-slicing, batching, or constant-rate padding above the queue rather than any engine setting.

33 Telemetry and hardware counters

SUMMARY

The engine has a hardware performance-counter block of twenty-four per-task-descriptor counters, master enable at `ANEProgramCreateArgs+0x6c`, and free-running firmware timestamp at MMIO `0x2_6b17_8000`. The block geometry and the timestamp are readable, but the per-task-descriptor counter values are not, because one kernel gate blocks them on the unentitled path. Forcing the stats mask non-zero turns a successful load into a rejected one, since the compiled program has no stats-descriptor section for the kernel to size. What remains readable is the whole-engine telemetry outside that gate: DRAM read and write bytes, engine energy in millijoules, and clock-state residency.

33.1 Counter block geometry

Table 33.1 gives the per-task-descriptor counter groups, the number of counters in each, and a representative counter name.

Table 33.1. The per-task-descriptor counter groups, with the count in each and a representative counter name.

Group	Counters in group	Representative counter
Neural engine cycles	6	`kANE_NE_COMPUTE_CYCLES`, `kANE_NE_INPUT_STALL_CYCLES`
L2 (on-chip 2 MB) cycles	4	`kANE_L2_NOMINAL_CYCLES`, `kANE_L2_READ_STALL_CYCLES`
L2 processing-element cycles	3	`kANE_L2PE_COMPUTE_CYCLES`
Precision compute cycles	2	`kANE_FP16_CYCLES`, `kANE_INT8_CYCLES`
Kernel-manager stall	1	`kANE_KM_STALL_CYCLES`
Data-movement bytes	7	`kANE_DMA_READ_BYTES`, `kANE_L2_TO_NE_DATA`
Per-descriptor energy	1	`kANE_DPE_ENERGY`

The per-task-descriptor counter namespace is twenty-four named counters, every one of them per task descriptor, grouped by engine block. Table 33.2 gives the complete namespace, recovered from the counter-name accessor and grouped by engine block.

Table 33.2. The complete twenty-four-counter per-task-descriptor namespace, recovered from the counter-name accessor, grouped by engine block.

Group	Counters
Neural engine cycles	`kANE_NE_NOMINAL_CYCLES`, `kANE_NE_COMPUTE_CYCLES`, `kANE_NE_THROTTLE_CYCLES`, `kANE_NE_INPUT_STALL_CYCLES`, `kANE_NE_OUTPUT_STALL_CYCLES`, `kANE_NE_KERNEL_STALL_CYCLES`
On-chip 2 MB memory cycles	`kANE_L2_NOMINAL_CYCLES`, `kANE_L2_THROTTLE_CYCLES`, `kANE_L2_READ_STALL_CYCLES`, `kANE_L2_WRITE_STALL_CYCLES`
L2 processing-element cycles	`kANE_L2PE_COMPUTE_CYCLES`, `kANE_L2PE_INPUT_STALL_CYCLES`, `kANE_L2PE_OUTPUT_STALL_CYCLES`

Group	Counters
Precision compute cycles	`kANE_FP16_CYCLES`, `kANE_INT8_CYCLES`
Kernel-manager stall	`kANE_KM_STALL_CYCLES`
Data-movement bytes	`kANE_DMA_READ_BYTES`, `kANE_DMA_READWRITE_BYTES`, `kANE_AF_TO_KM_DATA`, `kANE_AF_TO_L2_DATA`, `kANE_L2_TO_AF_DATA`, `kANE_L2_TO_NE_DATA`, `kANE_NE_TO_L2_DATA`
Per-descriptor energy	`kANE_DPE_ENERGY`

The byte counters track data movement along the activation-function to kernel-manager to L2 to neural-engine path, and one per-descriptor counter estimates energy from the digital-power estimator. With this set a host can attribute, per scheduling unit, whether a task descriptor is compute-bound, input-stalled, output-blocked, or throttled, directly in hardware counters. The master enable is a single field in the program-create argument struct. It is the per-task-descriptor stats mask at `ANEProgramCreateArgs+0x6c`, a `u32` that is after the quality-of-service field and the packed boolean flags and before the memory-pool identifier, located in the argument fields of Listing 33.1.

Listing 33.1. The program-create argument fields, with the per-task-descriptor stats-mask master enable at offset +0x6c.

```
ANEProgramCreateArgs (offsets):
  +0x18..+0x57 : two SHA256-class hashes (model hash + key)
  +0x5c (u32)  : count (number of procedures)
  +0x64 (u32)  : qos                  = 21 (0x15)
  +0x68 (u32)  : packed bool flags    = 0x10
  +0x6c (u32)  : statsMask            <- the per-TD counter master enable
  +0x70 (u32)  : memoryPoolID         = 0
  +0x80        : program name "main_main__0p0_AneInference"
```

The driver remaps a client-facing mask to a driver mask before it reaches this field. The remap keeps only the low nibble: a mask of `0xffffffff` translates to a driver mask of `0x0`, that is, no collection, and `0xf` is the only fully-enabled translation.

$$\text{driverMask}(m) = \begin{cases} \text{remap}(m \bmod 16) & m < 16 \\ 0 & m \geq 16 \end{cases}$$

When the mask is non-zero the firmware writes each task descriptor's counter block into a shared stats buffer in DRAM. The host decoder is built against the `sCAneStatsData` ABI version `0x0201`, distinct from the on-wire header magic `0x0101` the firmware writes into the buffer (chapter 29). The buffer's required size is the sum of the stats header, event descriptors, and per-event records. The host decodes it through a parser that walks a Group to Layer to task-descriptor hierarchy, where the leaf task-descriptor node holds the counter block.

33.2 Free-running timestamp

A monotonic firmware timestamp underlies the whole telemetry surface. The firmware reads it from a single free-running memory-mapped counter through the one-line helper of Listing 33.2, then stamps each trace-event record from that read.

Listing 33.2. The firmware helper that reads the free-running engine timebase counter.

```
/* free-running engine timebase counter, read by the firmware helper @0x30988 */
#define ANE_TIMEBASE_COUNTER  0x26b178000ULL   /* MMIO 0x2_6b17_8000 */

static inline uint64_t ane_read_timebase(void) {
    return *(volatile uint64_t *)ANE_TIMEBASE_COUNTER;  /* ldr x0, [x8] ; ret */
}
```

Each firmware trace-event record has a `timeStamp` field alongside its task-descriptor identifier, network identifier, program identifier, process identifier, and task-queue, as Listing 33.3 shows.

Listing 33.3. The fields of each firmware trace-event record.

```
[ANE_TM_EVENT_START]:          tid, nid, progId, procId, currTQ, timeStamp
[ANE_TM_EVENT_FINISH]:         tid, nid, progId, procId, currTQ, timeStamp
[ANE_EVENT_CONTEXT_SWITCH_IN]: tid, nid, prevTQ, progId, procId, currTQ, timeStamp
```

The host-visible clock residency runs in 24 MHz ticks, one tick every 41.67 ns, read out of the system-on-chip state-residency channels. The timestamp is monotonic and survives a power-gate, since the firmware re-anchors it from the same free-running source rather than resetting it across a clock-state transition. It is the basis for the per-dispatch wall-clock intervals that the signpost stream exposes, and it is readable with no entitlement beyond root.

33.3 What the host can read and what it cannot

The block geometry and the timestamp are observable; the counter values are not. The split follows the stats mask: the timestamp and the whole-engine channels do not route through it, and the per-task-descriptor counters do, as Listing 33.4 contrasts.

Listing 33.4. The readable timestamp and whole-engine channels contrasted with the blocked per-task-descriptor counter path.

```
/* READABLE: no stats mask in the path */
uint64_t  t  = ane_read_timebase();                /* free-running firmware timestamp */
int64_t   rd = ioreport_delta("AMC Stats|Perf Counters|ANE0 RD");  /* DRAM read bytes  */
int64_t   mj = ioreport_delta("Energy Model|-|ANE0");              /* engine energy, mJ */

/* BLOCKED: gated by the per-task-descriptor stats mask */
args.statsMask = 0xf;                  /* master enable, ANEProgramCreateArgs+0x6c     */
create = ANE_ProgramCreate(&args);     /* create -> 1                                  */
load   = ANE_ProgramLoad(create);      /* load   -> 0: initStatsBufferSection bails     */
perf   = read_perf_iosurface();        /* every byte 0: per-run output buffer is null  */
```

The values are blocked because the master enable never takes effect on the host path. On the unentitled runtime path the runtime sets the stats mask to `0` below the model layer, so firmware collection is never armed and the per-run output buffer comes back null. Forcing the mask non-zero through an in-process hook does not help: the program-create call then reaches the kernel routine of Listing 33.5, which looks up the program's stats-descriptor section by name, reads its size field, and bails on a zero size. The `aned` daemon is what zeroes the mask: it sets `statsMask=0` for a `coreAnalyticsClientType` of `ThirdPartyAppUsingANE`, so the mask is cleared by client type, and the host-side null check is the string `perfStatsIOSurface is NULL!`.

Listing 33.5. The kernel routine that bails when the stats-descriptor section size is zero, failing the create call.

```
initStatsBufferSection(ANEProgramCreateArgsOutput*, task*):
  ldr  w8, [x8, #0x28]     ; size of the stats-descriptor section
  str  w8, [x28]
  cbz  w8, bail            ; size 0 => return 0 (create fails)
  ...                      ; non-zero => kalloc + map the stats buffer
```

The compiled program has no stats-descriptor section, so the size is always zero and the kernel returns failure. Forcing a non-zero mask thus turns a successful load into a rejected one (`create -> 1; load -> 0`), and no host-side primitive synthesizes the missing section. One kernel gate thus blocks both the per-task-descriptor counters and the per-run output buffer: each depends on a stats-descriptor section that only an internal profiling-compile emits.

What remains readable is the whole-engine telemetry outside that gate, the channels of Table 33.3 with their format, unit, and what each reads out.

Table 33.3. The whole-engine telemetry channels readable on the unentitled path, with their format, unit, and what each reads out.

Channel	Format	Unit	Reads out
`AMC Stats | Perf Counters | ANE0 RD`	fmt=1 delta integer	B	DRAM read bytes
`AMC Stats | Perf Counters | ANE0 WR`	fmt=1 delta integer	B	DRAM write bytes
`AMC Stats | Perf Counters | ANE0 DCS RD`	fmt=1 delta integer	B	DRAM read bytes via the compression-subsystem path
`AMC Stats | Perf Counters | ANE0 DCS WR`	fmt=1 delta integer	B	DRAM write bytes via the compression-subsystem path
`Energy Model | - | ANE0`	fmt=1 delta integer	mJ	engine energy
`PMP | AF BW | ANE0 RD+WR`	fmt=2 residency	events	aggregate read-plus-write bandwidth events
`SoC Stats | Cluster Power States | ANE0`	fmt=2 residency	24Mticks	clock-state residency
`SoC Stats | Events | SOC0_ANE_F1, SOC0_ANE_F2`	fmt=2 residency	24Mticks	clock-domain frequency-point residency
`SoC Stats | Events | ANE0_ADCLK_TRIG, ANE0_DITHR_TRIG`	fmt=2 residency	24Mticks	adaptive-clock and dither trigger ticks
`Interrupt Statistics | ane0 0`	fmt=1 counters	counts, MATUs	first and second-level interrupt-handler count and time

The memory-controller per-agent byte counters report DRAM read and write bytes for the engine, both on the raw path and separately on the compression-subsystem path, and the energy model reports engine energy in millijoules. The system-on-chip state channels report clock residency, frequency-point residency, and the adaptive-clock and dither trigger counts, and the per-map fabric arbiter reports the engine's bandwidth and clock-floor votes. The interrupt statistics report the host kernel's cost of servicing engine completions, in counts and Mach Absolute Time Units, split into first-level and second-level handlers and isolated separately for the engine and its address-translation unit. A storage coprocessor that an earlier reading mistook for the engine, exposing vector-lane read and write counters, is unrelated to engine telemetry; the bandwidth counter is the memory-controller `ANE0` channel and the power channel is the energy model.

The aggregate bandwidth channel resolves into per-channel histograms, the fabric channel `PMP0 / DCS BW / ANE L0` and `L1` carrying read and write distributions for the two DRAM read channels, read as State-residency

histograms rather than plain integers. The `SoC Stats / Events` channel also exposes the throttle-trigger family `ANE_THROTTLE_{SW,HW,PPT,DITHER,EXT}_TRIG` and `VDD_DRAM_VOLTAGE_CHANGE`, per-trigger software, hardware, peak-power, and dither throttle counts, all readable without entitlement.

The per-dispatch op lifecycle is also readable as the timestamped signpost stream of Listing 33.6, captured on the engine subsystem with no entitlement beyond root, which confirms the three-request structure of a dispatch without exposing any counter value.

Listing 33.6. The client-side signpost intervals of one dispatch, with the three driver requests that correspond to the Cast, AneInference, and Cast program operations.

```
_ANEF_MODEL_EVALUATE                    one per host execute call
  _ANEF_MODEL_EVAL
  _ANEF_MODEL_EVAL_DRIVER_REQUEST       request 1 of the dispatch (Cast)
  _ANEF_MODEL_EVAL_DRIVER_REQUEST       request 2 of the dispatch (AneInference)
  _ANEF_MODEL_EVAL_DRIVER_REQUEST       request 3 of the dispatch (Cast)
  _ANEF_MODEL_EVAL_PERFCOUNTER_SAMPLE   the per-descriptor counter sample is reported here when armed
_ANEF_MODEL_COMPILE   _ANEF_MODEL_LOAD   _ANEF_MODEL_UNLOAD
_ANEF_IOSURFACES_MAP  _ANEF_INPUT_BUFFERS_READY  _ANEF_ENQUEUE_OUTPUT_SET
```

Each interval has a Mach-time begin stamp, and the three driver requests are the three program operations of a dispatch, one driver-to-firmware request each. The firmware reports the blocked per-descriptor counter sample on the evaluate interval only when the stats mask is armed, so the signpost stream confirms the dispatch structure on live silicon while the counter values stay behind the same gate.

33.4 Roofline and power figures from the readable channels

The measured roofline elsewhere in this guide rests on the readable whole-engine channels, not on the blocked per-descriptor counters. Bandwidth comes from a delta on the memory-controller byte counters across a real workload. A 96-layer matmul chain at batch 8192 moved 17.2 GB of reads and 16.9 GB of writes over 432 ms, which is between 79 and 90 GB/s, the engine's share of the unified memory. A delta on the energy-model millijoule channel over the same run gives about 0.48 pJ/FLOP and about 2.3 W under sustained compute.

The compute roof and the dispatch floor are wall-clock measurements, timed against the firmware timestamp and the host clock rather than read from a counter. The 2 MB working-set threshold is confirmed directly from the byte counters. At batch 8192 the activation is exactly 2 MB, the counters show 426 MB of DRAM moved per dispatch, arithmetic intensity falls to 60 FLOP/byte, and throughput drops from 12 TFLOP/s to about 4.8 TFLOP/s. The per-descriptor counters would attribute that drop to specific stall classes, the input-stall and L2-read-stall cycles, but the whole-engine byte and energy channels already locate the workload on the bandwidth slope, which is what the roofline needs.

Part IX

Cross-Silicon Reference

34 Cross-silicon targets

The full target set and the rule mapping each M-series part to its H-series identity.

35 Per-family code generation

How one compiler binary builds every chip from a per-family data table.

36 Predicted upper tier

fp8 and the double-rate path the newest generations add.

34 Cross-silicon targets

SUMMARY

The compiler builds 28 architecture targets, one per silicon profile, under the fixed relation $M(n) \to H(n+12)$. A suffix letter selects the NE-core count, and the operation surface stops expanding at A15, so the surface measured on the M5 is the surface for everything above it. A device's runtime architecture string is a separate identifier from the compiler target name. A resolver-derived board-type sequence maps every shipping chip onto its generation.

The compiler that builds for the Apple Neural Engine has 28 architecture targets, one per silicon profile it knows how to construct. Each target is a named hardware-abstraction-layer table the compiler builds by calling one per-architecture constructor, `ZinIrHal<T>::GetParams()`, and calling every constructor on a single host recovers the full set regardless of which chip runs it.

34.1 Full set

Table 34.1 gives all 28 targets, each with its silicon class and decoded NE-core count.

Table 34.1. The 28 compiler targets, each with its silicon class and decoded NE-core count.

Target	Silicon and class	NE cores
`H11`, `H12`, `M9`, `T0`	pre-A13 legacy	1 to 4
`H13`	A13, M1 base	4
`H13g`	M1 Pro, Max, Ultra	8
`T1`	A13 reference	4
`H14`	A14, M2 base	4
`H14g`	M2 Pro, Max	8
`H14c`	A14 Max-class	32
`H15`	A15, M3 base	4
`H15g`	M3 Pro, Max	8
`H15c`	A15 Max-class	32
`H16`	A16, M4 base	4
`H16g`	M4 Pro, Max	8
`H16s`	A16 Pro-class	16
`H16c`	A16 Max-class	32
`H17`	A17, M5 base	4
`H17a`	A17 variant	4
`H17g`	M5 Pro, Max	8
`H17s`	A17 Pro-class, the M5	16
`H17c`	A17 Max-class	32
`H17d`	A17 Ultra-class	64
`H18`	A18 base	4
`M11`	small embedded ANE	1
`U1`, `U2`, `U3`	reference, not silicon	4

The names fall into four groups: the H-architecture targets that stand for shipping A-series and M-series silicon, the pre-A13 legacy targets, a single small embedded profile, and three reference targets that are not silicon at all. A suffix letter selects the NE-core count within a generation, which the compiler decodes from the core-count field at hardware-abstraction-layer offset `0x238`. The base name is 4 cores, the suffix `g` is 8, `s` is 16, `c` is 32, and `d` is 64, while `M9` and `M11` are single-core. `H17s` is thus the 16-core Pro-class part that is the M5, and `H17d` is the 64-core Ultra-class die, the largest in the table. These decoded `num_nes` values are the compiler's per-die core field, not Apple's marketing Neural Engine count; on the base M1 the decoded four stands against the published sixteen [AppleANE].

The reference targets hold placeholder limits that no part has: a maximum tensor depth of 1, a kernel-width limit of 1023, and no interchange-format support. They are unconstrained validation profiles the compiler builds for its own checking, not addressable silicon. The small embedded profile `M11` is addressable silicon. It is an efficiency-class engine that has the A16-class feature flags but the A13-class 16384-dimension limit, a single NE core, and the odd kernel-width ceiling of 15 that is between the A13 value of 13 and the A14 value of 16.

34.2 Capability tiers

Table 34.2 groups the targets into capability tiers, giving each tier its dimension limit and the four gated capabilities that separate the generations.

Table 34.2. The capability tier of each target, with the dimension limit and the four gated capabilities that separate the generations.

Tier	Targets	Max dimension	3D conv	Texture engine	`sin`, `cos`	Dropout
pre-A13	`H11`, `H12`, `M9`, `T0`	16384, depth 1	no	no	no	no
A13	`H13`, `H13g`, `T1`	16384	yes	no	no	no
A14	`H14`, `H14g`, `H14c`	16384	yes	yes	no	no
A15	`H15`, `H15g`, `H15c`	16384	yes	yes	yes	yes
A16	`H16`, `H16g`, `H16s`, `H16c`	65536	yes	yes	yes	yes
A17	`H17`, `H17a`, `H17g`, `H17s`, `H17c`, `H17d`	65536	yes	yes	yes	yes
A18	`H18`	65536	yes	yes	yes	yes
small	`M11`	16384	yes	yes	yes	yes
reference	`U1`, `U2`, `U3`	65535 placeholder	no	no	no	no

A17 and A18 add no operation over A16: identical dimension limits, identical kernel-width and kernel-depth ceilings, the same texture engine, same dropout and global-argmax flags, and same legal operation set. They differ from A16 only in NE-core count, which scales throughput rather than legality. The operation behavior measured on the M5, an H17 part, is thus the operation behavior of every target at or above A16, since the decoded capability tables are identical; the cross-silicon performance measurements of chapter 12 are predicted to carry to the unshipped generations on the same basis, with the per-chip rates confirmed only on the two measured silicon points.

34.3 Silicon to target

The map from a shipping chip to its architecture is a resolver Apple distributes that decompiles cleanly. The method `aneArchitectureType` on the private device-info class builds the architecture string from a

board-type value read from the platform configuration store, switching on a strictly increasing board-type sequence. The live anchor on an M1 Max reads board type 96, which resolves to `h13g` with a 16-core count, matching the registry exactly. Table 34.3 gives the resolver-derived map from system-on-chip to runtime architecture and compiler target across the M1 through M5 generations.

Table 34.3. The resolver-derived map from system-on-chip to runtime architecture and compiler target, M1 through M5.

Chip	Product	Runtime arch	Compiler target
T8103	M1 base	`h13`	`H13`
T600x	M1 Pro, Max, Ultra	`h13g`	`H13G`
T8112	M2 base	`h14`	`H14`
T602x	M2 Pro, Max	`h14g`	`H14G`
T8122	M3 base	`h15`	`H15`
T603x	M3 Pro, Max	`h15g`	`H15G`
T8132	M4 base	`h16`	`H16`
T604x	M4 Pro, Max	`h16g`	`H16G`
T8142	M5 base	`h17`	`H17`
T605x	M5 Pro, Max	`h17s`	`H17s`

The map follows the fixed $M(n) \to H(n+12)$ relation of chapter 12. The sequence is anchored at both ends, the live M1 Max at `h13g` and the measured M5 Pro at `h17s`, and the intervening steps are corroborated independently. A single shipping vision filter has exactly the five tables `H13`, `H14`, `H15`, `H16`, `H17`, the five Mac engine generations. The board-type kext for an absent chip cannot be read on a different host, since only the running chip's table is resident, so each middle step rests on the anchored monotone sequence.

34.4 Runtime string and compiler target

The architecture name a device reports at runtime is not the compiler target name. The runtime string is the coarse form, `h1N` for a base part and `h1Ng` for a Pro, Max, or Ultra part, the only two variants the runtime emits on the desktop platform. The compiler target is the finer set, the full `H17`, `H17s`, `H17c`, `H17d`, `H17g` family, of which the runtime collapses several onto one string. A developer names the target by its compiler form and treats the runtime string as a separate identifier.

The direct compile entry point accepts any of the 28 target names and rejects an unknown name. The dispatch library, in contrast, falls back silently when handed an unknown architecture, so a developer gates a cross-target compile against the known-name set before dispatching it.

34.5 Interchange formats across the set

Each target has a per-chip table of accepted image-input formats, the interchange-format map at hardware-abstraction-layer offset `0x658`, keyed by a four-byte ASCII format tag. Table 34.4 gives the accepted image-input format count by generation tier, with the format set each tier adds.

Table 34.4. The accepted image-input format count by generation tier, with the format set each tier adds.

Tier	Chips	Format count	Set
older, reference	`H11`, `H12`, `M9`, `T0`, `U1`, `U2`, `U3`	0	none
A13, M1	`H13`, `H13g`, `T1`	3	`&BGA`, `&L0h`, `&L16`
A14	`H14`, `H14g`, `H14c`	13	A13 set, RGBA-half, three compression variants

Tier	Chips	Format count	Set
A15 and small	`H15`, `H15g`, `H15c`, `M11`	16	A14 set, YUV 4:2:0, luma-half
A16, A17, A18	`H16`, `H17`, `H18` families	14	A15 set minus YUV 4:2:0

The tag is a one-byte compression-variant prefix on a three-byte base pixel format. The compiler does not parse the prefix character by character: it validates the whole four-byte tag against a 34-entry allow-list, and the prefix's meaning is the third byte of the format's packed-integer value, a packing-mode index on a uniform stride. Table 34.5 gives the compression-variant prefix on an interchange tag and the packing-mode index it selects.

Table 34.5. The compression-variant prefix on an interchange tag and the packing-mode index it selects.

Prefix	Mode index	Meaning
`&`	0	uncompressed, default raster surface
`-`	1	lossless compression, 32 by 32 macroblock
`/`	2	lossless compression, 16 by 16 macroblock
`|`	3	lossless compression, mode 3
`*`	0	compound prefix that sets the dynamic-channel flag

The packed integer that names each format is three bytes: a pixel class, a base-format code, and the packing-mode index. The base-format codes are BGRA8 (`BGA`, code `0x11`), RGBA-half (`RhA`, code `0x13`), 8-bit luma (`L0h`, code `0x07`), 16-bit luma (`L16`, code `0x08`), and YUV 4:2:0 (`8f0` and `8v0`, code `0x09`). A base format routes to a vector of 20-byte plane descriptors, each a tuple of width divisor, height divisor, element type, channel count, and depth. BGRA8 is thus one four-channel uint8 plane, and YUV 4:2:0 is a luma plane with a half-resolution two-channel chroma plane. The binary string that reads "Architecture only supports lossless compression" confirms that `&` is the uncompressed variant and that `-`, `/`, and `|` are the lossless-compressed packing families.

The A15 generation and the small embedded profile are the only targets in the set that accept YUV 4:2:0 input, in both full-range (`8f0`) and video-range (`8v0`) form. The M5 and every A16-and-later part keep luma-half but drop the two YUV 4:2:0 formats. The full per-target format records, the wider 10-bit and packed YUV family, and the plane-layout structures are appendix material.

35 Per-family code generation

SUMMARY

One compiler binary lowers a network for every generation, and the per-chip difference is data rather than code. Native operation support is declared by a `MinimumFamily<N>` trait in four floors, F0 through F4, and nothing floors above F4. The only operations the M1 decomposes that the M5 runs natively are crop-resize, resample, `sin`, `cos`, and the global arg-reductions.

The compiler that targets every Neural Engine generation is one binary. A single build lowers a network for the M1 and for the M5, and the divergence is driven by the family enum and the per-chip hardware-abstraction parameter blob from chapter 24.

35.1 One binary, eight families

The compiler enumerates the eight generations as the `mlir::anec::Family` values that listing 35.1 gives.

Listing 35.1. The eight-value family enumeration the compiler keys per-generation operation legality from.

```
mlir::anec::Family = {
  A11Legacy = 0, A12 = 1, A13 = 2, A14 = 3,
  A15 = 4, A16 = 5, A17 = 6, A18 = 7
}
```

The M1 compiles as `A13` (index 2) and the M5 compiles as `A17` (index 6), and `A17` is a strict superset of `A13` in operation support. Two string parsers build the per-target objects from the target name: `CreateTargetFromString` constructs the `ZinIrTarget` instance, and `CreateHalFromString` constructs the matching `ZinIrHalParameters` blob. Both branch on string length and the first characters compared as little-endian shorts, `0x3168` for “h1”, `0x3074` for “t0”, `0x396d` for “m9”, and `0x316d` for “m1”.

The numeric hardware version maps to the family enum through a fixed table, the `unordered_map<int, Family>` named `ANEFamilyToMLIR`, whose data table table 35.1 decodes key to value along with the targets that resolve to each family.

Table 35.1. The hardware-version-to-family map read from the `ANEFamilyToMLIR` data table, with the targets that resolve to each family.

Hardware version	Family index	Targets
1	A11Legacy (0)	`h11`
3	A12 (1)	`h12`
4	A13 (2)	`h13`, `h13g`, `t1`
5	A14 (3)	`h14`, `h14g`, `h14c`
6	A15 (4)	`h15`, `h15g`, `h15c`
8	A15 (4)	`h16`, `h16g`, `h16c`, `h16s`
7	A16 (5)	`h17`, `h17a`, `h17g`, `h17c`, `h17d`

Hardware version	Family index	Targets
9	A17 (6)	`h17s`
10	A17 (6)	`h18`

This table holds two collisions that the compiler must guard. Hardware versions 6 and 8 both resolve to family A15, so `h16` and the A16-tier parts compile with A15-family operation legality. Hardware versions 9 and 10 both resolve to family A17, so `h18` uses A17-family operation legality. The family enum drives the `MinimumFamily<N>` operation legality only. The per-target hardware-abstraction blob drives code generation, the numeric limits, and the cost model. The suffixed targets such as `h17s` thus select a distinct `ZinIrTarget` object with its own descriptor generation, core count, and cost curves, even when several family-keyed paths collapse onto the same enum value. The hardware-version-to-family table is read directly from binary data; the per-target hardware-version value has some inference, since the per-target hardware-version getter is virtual and not present in the readable decompilation. The two anchors `h13` to A13 and `h17s` to A17 are confirmed by the live measurement campaign.

The lowering code is shared, so the per-family split is in two locations. An operation trait declares a minimum family for native support. The per-chip hardware-abstraction blob is a fixed-size per-target structure whose field values, not whose code, select the route a shared lowering pattern takes.

35.2 Operation floors: the minimum-family trait

Table 35.2 gives the four minimum-family floors, the families each is native on, and representative operations at each floor.

Table 35.2. The four minimum-family floors, the families each is native on, and representative operations at each floor.

Floor	Native on	Representative operations
F0	all 8 families	convolution, deconvolution, matmul, pooling, all elementwise, reshape, transpose, concat, sigmoid, tanh, relu, gelu, swish, quantize, dequantize
F2 (A13+)	A13 through A18	softmax, layer-norm, instance-norm, batch-norm, all reductions, resize, scaled dot-product attention, `erf`, `exp2`, `rsqrt`, `sqrt`, tile, space-to-channel
F3 (A14+)	A14 through A18	crop-resize, resample
F4 (A15+)	A15 through A18	`sin`, `cos`, global arg-min and arg-max

Native operation support is declared by the `MinimumFamily<N>` trait held on each backend operation, the operation-side member of the two capability gates chapter 24 defines. An operation is natively legal only inside a region whose family index is at least `N`; below that floor the compiler decomposes it into a sequence of supported operations. The trait is the mechanism, not the dynamic-legality registration: `addDynamicallyLegalOp` wraps only the eight region operations and four texture-gated operations, while every backend operation has the `MinimumFamily<N>` impl. 88 compute operations have the trait, 52 at F0, 31 at F2, two at F3, and three at F4. These operation floors are measured on physical silicon for the M1 and M5; the M3, M4, and upper-tier figures are decompile-derived, predicted from the per-chip tables rather than measured.

No operation in the decoded compute-op floor table floors above F4, so almost nothing is exclusive to `A16`, `A17`, or `A18`: the upper-tier generations add Neural Engine cores rather than a usable operation set. The one exception is a small set of `MinimumFamily<N>` template instantiations at N=5, N=6, and N=7, including the

fp8-bearing operations, which stay inert below H18 (chapter 36). Softmax, layer-norm, all reductions, resize, and scaled dot-product attention floor at `A13`, so they run natively on the M1.

The two oldest families behave as a different, more constrained instruction set. On `A11Legacy` and `A12` there is no native reshape, broadcast, or divide. Reshape, squeeze, expand-dims, and broadcast all share `ConvertToReshape<Op, Family>` and `ConvertBroadcast<Family>`. On A13 and above they emit a plain native `anec::Reshape` or `anec::Broadcast`, while on the two legacy families reshape lowers to `anec::Flatten` and the compile aborts through `verifyCompatibilityWithFlatten` when the layout does not collapse to a pure flatten. On the two legacy families squeeze or expand-dims emit the string "cannot be lowered as Flatten on ANE". Broadcast on the legacy families asserts fp16 and switches engine by axis: a channel-group axis broadcasts as a matrix multiply by ones, and other axes broadcast as an elementwise add by zero. Divide on the legacy families matches only a constant fp16 divisor and lowers as a reciprocal multiply with the reciprocal pre-rounded to fp16, with a non-constant or non-fp16 divisor producing a match failure. The floor-divide form is $\lfloor a \cdot \mathrm{f16}(1/b) \rfloor$. The double rounding can give the wrong integer at a boundary, for example $\lfloor 6 \cdot \mathrm{f16}(1/3) \rfloor = \lfloor 1.999 \rfloor = 1$. On A13 and above divide is a native `anec::ElementwiseDiv` over arbitrary dynamic divisors.

35.3 Same operation, a different kernel

Table 35.3 gives the five categories where the shared lowering emits different machine code or different numbers per chip, with the per-chip parameter that drives each.

Table 35.3. The five categories where the shared lowering emits different machine code or different numbers per chip, with the per-chip parameter that drives each.

Category	Divergence	Driven by
width slice	M1 saturates at the Q.4 crop route, M5 stays in fp16	patch-width clamps, width granule
resize	three strategies round differently by chip	texture-engine-present flag
matmul	lowers to a resident convolution, splits along height	per-family copy-cast and max-tensor-size thresholds
transpose	native within a height-width bound, else multi-pass	per-chip maximum height and width
strided convolution	factorizes stride into a sub-convolution sequence	per-chip allowed-factor list

The lowering pattern is byte-identical across families in each case, and the divergence is in the hardware-abstraction parameters or the task-descriptor emitter. Take first the slice that saturates on the M1. The trigger is `ZinSliceLayer::SliceNeedsCropMode`, which is family-independent. A slice with a nonzero offset on the height axis (axis 3) or the width axis (axis 4) routes through crop mode to `ZinTECropModeLayer`, the transpose-engine crop direct-memory-access path, while a begin of zero and the non-height-width axes skip it. The height axis decompiles to the same path but was measured unaffected on silicon, and a channel or batch offset stays unaffected as well: only the width offset triggers the saturation. The conversion rewrites `ConvertSlice<Family>` and `ConvertStridedSlice<Family>` are byte-identical across all eight families, and the crop scale is still a float at the layer-configuration stage where `Create` builds `1.0/(extent - 1)`. The multiply by sixteen appears only when that geometry is written into the task descriptor. The patch width is a four-bit field that `SetPatchWidth` writes with the instruction `bfxil w9, w8, #0, #4`, and `IsFormatDMAConvertibleToFP16` and `L2Allocate::ConvertFmt` make the fixed-point-versus-fp16 choice, keyed to the format rather than to the chip. On the M1 the compiler sends that copy through a fixed-point storage format with four fractional bits, an implied multiply by sixteen. The stored value is the source times sixteen and the storage clamps at the fp16 maximum, so any source magnitude above the threshold overflows to $\pm\infty$. The threshold is exact:

$$V_{\max} = \frac{65504}{16} = 4094$$

`ZinMirL2Config::CalculateDMASrcBufferSizeAndStrides` reads the width granule and clamps, taking a `ZinIrHalParameters const&` and reading `HAL+0x1c0` as both a divisor that recovers an element count and a multiplier on the final byte stride. It also reads the patch-width clamps at `HAL+0x3f8`, `HAL+0x400`, and `HAL+0x410` through `ComputeMaxPatchWidth`, which returns a `CeilLog2(width)` bounded by those clamps. A sixteen-wide granule gives a log-base-two of four, the shift-left-by-four that is the multiply by sixteen. On the M5 the route avoids the fixed-point format and the slice stays in plain fp16, so the same slice that saturates on the M1 is unaffected on the M5. The lowering pattern is the same template on both chips; the route and format selection at the descriptor and hardware-abstraction layer differs, driven by the patch-width clamps and the width granule in the per-chip blob. The saturating path is present on every patch-capable descriptor generation including the A17 generation. The M1-versus-M5 difference is the route and format selection upstream, not a per-family rewrite. Any height-width-axis slice with a nonzero offset whose fp16-convertible source can exceed 4094 is thus hazardous on the whole patch-capable generation set, unless the route is known to avoid crop mode.

The remaining four cases are decompositions gated by the per-chip blob rather than the family enum. Resize is family-uniform at the conversion level and diverges in `ZinResizeLayerUtils`: `DecomposeResize` branches on the texture-engine-present flag at `HAL[0x81d]`, and if it is absent and the no-texture decomposition fails the compiler asserts “failed to map resize layer on this arch”. The no-texture path picks a deconvolution upsample against a transpose-plus-convolution path through `QualifiesforUpsampleDecomposition`, gated on the one-by-one fast-path flag at `HAL[0x812]`, and `GetMaxSmallKernelWidth`; the three strategies round differently, so resize results differ by chip. Matmul lowers to a resident convolution through `LowerNEMatMulToNEConv` when the right-hand weight bytes meet the per-family copy-cast threshold at `HAL[0x1b8]`, and `SliceMatMulAlongHeight` splits the operation by the per-family maximum tensor size. Transpose runs natively only when `IsValidNETransposeConfiguration` passes the maximum height and width at `HAL+0x468` and `HAL+0x470` plus the divisibility checks, and otherwise decomposes into multiple passes on the more constrained families. A convolution with a large stride factorizes the stride through `DecomposeConvWithLargeStride` against the per-family allowed-factor list at `HAL[0x730]` through `HAL[0x750]`, at most two factors per axis, so one convolution becomes a different sub-convolution sequence per chip.

35.4 What the M1 blocks and which chip enables it

Table 35.4 gives the operations and behaviors the M1 blocks, the first Apple family that enables each, and the cause.

Table 35.4. The operations and behaviors the M1 blocks, first Apple family that enables each, and cause; the mapping of family to silicon is the one given in chapter 34.

Operation or behavior	M1 / H13	Enabling family	Cause
`sin`, `cos` native	decomposed	A15 (M3+)	F4 trig floor; M1 uses a polynomial decomposition
crop-resize, resample, affine, height-width gather	decomposed	A14 (M2+)	texture engine absent (HAL flag `0x81d`)
top-k, sort, dynamic-slice	rejected at code generation	A14 (M2+)	rank and sort bridge; passes the validator, fails lowering
dropout, random	unavailable	A15 (M3+)	random-number block off (HAL `0x4a9`)
global arg-min, arg-max	decomposed	A15 (M3+)	HAL `0x4f2`
reduce-then-square fusion	unfused form emitted	A14 (M2+)	a fusion, not a capability; result within one unit in the last place

Operation or behavior	M1 / H13	Enabling family	Cause
width slice with offset over 4094	saturates to ±inf	A15 (M3+)	M1 routes through the Q.4 crop direct-memory-access; A14 saturates too, the clean route arrives on A15+
int8-affine weight streaming	folds to fp16	A14 (M2+)	streaming gate off on A13; the M1 streams int4-LUT and sparse only, the M2 adds int8
blockwise weight streaming	folds to fp16	A15 (M3+)	folds on A13 and A14; broad streaming arrives on A15

Some operations have no native form on any family, including tan, asin, acos, atan2, sinh, cosh, atanh, asinh, acosh, the logical and-or-xor, the recurrent cells, scatter, one-hot, non-zero, mod, band-part, and reverse-sequence. These are decomposed on the host or in the graph on every chip and are not a per-family difference. The slice saturation is the only axis that turns a finite value into an infinity across chips; the wide accumulator and the reduction route are uniform across families and are not a block.

35.5 Kernel-data path: per-core replication against shared kernel memory

The weight layout splits into two code-generation functions that the kernel-memory register at HAL[0x48f] selects. ZinMirBuildNEKernelData is the per-core replicated path: on the M1 the kernel coefficients are built per Neural-Engine core, walked through the per-core byte stride that CalculateNEOffsetJump computes, and must fit the 64 KB kernel-memory cap. ZinMirBuildNEKernelDataSharedKmem is the A14-and-above shared path, looped over core groups, selected by the kernel-memory register-select byte and the ExceedKmem SizeLimit logic that listing 35.2 gives.

Listing 35.2. The kernel-memory offset selection between the shared 16 MB path and the 64 KB per-core path.

```
off = (useShared && HAL[0x48f]) ? 0x210 : 0x200
```

On the M1 the register-select byte at HAL[0x48f] is set, but the extended-kernel-memory-mode scalar at 0x288 reads zero on every target in this compiler. The shared 16 MB path at offset 0x210 is thus taken only when an operation explicitly requests shared and the chip enables it. Otherwise the M1 falls to the 64 KB cap at offset 0x200 and tiles. The A14 and later chips route the large kernels through the shared builder instead of replicating per core, which is the concrete kernel-memory architecture difference between the M1 and the later families.

35.6 Task-descriptor layer: per-generation field offsets

Below the family enum is the descriptor emitter ZinAneTd<Nu>, instantiated for the generations $\{1, 4, 5, 6, 7, 8, 10, 11, 17, 19, 20\}$. Only the generations $\{7, 8, 10, 11, 17, 19, 20\}$ have the patch-width path; the legacy generations $\{1, 4, 5, 6\}$ use HandleCcdmaLayer with no patch settings. The same logical field is at a different byte offset per generation, so a field written at the wrong offset for the target corrupts an unrelated descriptor word. Table 35.5 gives the patch-width descriptor offset per task-descriptor generation.

Table 35.5. The patch-width descriptor offset per task-descriptor generation.

Descriptor generation	Family	Patch-width offset
7	A13, M1	`desc+0x200`
8	A13, A14	`desc+0x224`
10	A14	`desc+0x12c`
11	A14, A15	`desc+0x21c`
17	A15, A16	`desc+0x23c`
19	A16, A17	`desc+0x248`
20	A17, M5	`desc+0x26c`

The primary and secondary source direct-memory-access fields and the result direct-memory-access field follow the same per-generation-offset pattern. A single compiler defect is in this layer. `SetPatchSettings<7u>` targets `ZinAneTd<7u>` on every setter except the last, which calls `ZinAneTd<8u>::SetTileOverlapPadReflect` on a generation-7 pointer, writing the reflect flag at the generation-8 offset; a shipping target that uses generation 7 locates its tile-overlap-pad-reflect bit in the wrong word.

35.7 Hardware-abstraction parameter field map

`ZinIrHalParameters` is a 0x348-byte per-chip blob, copied by a 0x348-byte copy constructor. Table 35.6 gives the fields the shared lowering reads to drive the per-chip route, format, and limit selection.

Table 35.6. The fields of the 0x348-byte hardware-abstraction parameter blob and the route each drives.

Offset	Meaning
`0x1b8`	matmul right-hand copy-cast byte threshold
`0x1c0`	large-stride-conv enable and direct-memory-access width granule
`0x328`	conv stride and unicast input-channel limit
`0x30`, `0x38`, `0x48`, `0x50`	maximum small and large kernel width by tensor format
`0x88`, `0x90`, `0x98`, `0x138` through `0x1a8`	maximum spatial and per-dimension tensor sizes
`0x468`, `0x470`	maximum transpose height and width
`0x730` through `0x750`	allowed kernel-width and stride factor list
`0x812`	one-by-one input upsample fast-path flag
`0x81d`	resize and texture-engine-present flag
`0x3f8`, `0x400`, `0x410`	patch-width clamps for max-pool, floor, and non-pool
`0x228`, `0x238`, `0x240`, `0x248`	cost-model frequency, cycles, and core count

The concrete numeric values at these offsets are in a constant data section behind a packed selector the target constructors pass: `H13` and `H17s` pass the identical selector, so both share a base hardware-abstraction descriptor and diverge only in the literals. The per-target literals are deferred; the offsets, consumers, and mechanism are pinned here.

35.8 Per-family cost model

`getDeviceInfo` fills a device-information struct from the family enum and the operation size. It writes a constant `0.0008` at offset `0x30`, a size-scaled pair at offsets `0x14` and `0x18`, a bandwidth clamp at offset `0x20`, and a setup-latency and throughput pair at offsets `0x8` and `0xc` from a data block. The dispatch reads family 5 and 6 as the A16 and A17 tier, family 4 as the M1, and family 3 as A12, and buckets the size at thresholds of 6, 10, 20, and 40. Table 35.7 gives the bandwidth-clamp sequence and setup ramp per family in the analytic cost model.

Table 35.7. The bandwidth-clamp sequence and setup ramp per family in the analytic cost model.

Family	Bandwidth clamp sequence by size bucket	Setup ramp
A12	50, 100, 200, 400, 800	1.84 to 14.72
A13, M1	50, 100, 200, 400, 800	1.84 to 7.36
A16, A17, M5	62.5, 125, 250, 500	1.84 to 7.36
default (A11, A14, A15)	34.1, 68.2, and on	1.84 to 13.57

The M5 is modeled with a higher per-bucket throughput but a clamp ceiling of 500 against the M1 and A12 ceiling of 800.

35.9 A15 codegen branch

The A15 family is its own code-generation tier, not a relabel of A14, and the difference is visible in three independent locations in the compiler. The following is decode-derived from the static compiler image and is not measured on A15 silicon.

`getDeviceInfo` has a dedicated `anec::A15` branch. The dispatch range-tests the family value and special-cases `A14` and `A15` separately, so the `A15` arm fills the device-information struct from its own cost-table block rather than falling to the A14 arm or the shared default. A15 thus has distinct cost-model constants, which is the data-side proof that the family is a distinct capability tier.

The operation legality matches. Forty-five `MinimumFamily<N>` template instantiations have the F4 floor, the A15-and-above floor, against twenty-nine at the A14 floor, so A15 adds a concrete operation set over A14 rather than reusing the A14 set. The distinct compute operations at that floor are `sin`, `cos`, and the global arg-reductions.

Beyond the H15 constructors in the chapter 34 census, the compiler carries recognized target strings for a wider H15 die family, `h15g`, `h15m`, `h15p`, `h15s`, `h15c`, and `h15d`. The `h15m` suffix is an extra M-class target that the h13 and h14 families do not list. A15 is also the only tier whose per-chip interchange-format table accepts YUV420 image input, a 4CC route that the surrounding families drop, which is consistent with A15 being a distinct capability tier rather than a renamed A14. The A15 cost-table constants and the YUV420 acceptance are decoded from the compiler; the numeric behavior of the A15 operation set, such as any accumulator or rounding change against A13 and A14, is an A15-silicon measurement that this guide does not have.

35.10 Named validity gate

Only the M1 family has a named per-family validity gate. `IsValidForH13` is the gather gate: it asserts the gather-axes size is 3, the data batch and depth are 1, the index channel is 3 with width and depth 1, plus an axis-pattern check. `IsValidForH13GatherNCH` holds the channel-height-width-layout gather restriction. No equivalent named gate exists for any other family; the rest gate numerically through the hardware-abstraction clamps, and the compression gate `IsValidForCompression` is hardware-abstraction-driven with the universal rule that only kernels of at least eight bits are compressible.

35.11 Apple documents

Per-family code generation has no public counterpart, and this chapter reports it as reverse-engineered across the target set. The public conversion tools document the frontend operation set and the palettization, quantization, and pruning passes that produce a model, and the compute-unit selector that requests the engine [AppleCoreMLTools]. They do not document the family enum, minimum-family operation trait,

per-chip hardware-abstraction parameters, or route and format selection that makes one slice saturate on the M1 and run clean on the M5.

36 Predicted upper tier

SUMMARY

Two datapaths are above the M5 in the compiler that no part measured here can run: an fp8 weight and activation format, and a multi-die collective communication layer. Both are present as decoded structure, gated off on every family this guide reached, and stated as predicted, not measured. The fp8 gate is the capability byte at offset `0x52d`, set on H18 alone; the collective-enable byte at offset `0x48b` reads zero on all 28 targets.

36.1 64-core ceiling

The largest die this guide measured is the 16-core H17s, and the line ends at the 64-core H17d that no part here could run. The decode shows that nothing extra is family-gated for the 64-core case beyond the count itself, and the following is read from the compiler rather than measured on a 64-core part. Core count is a runtime parameter, the `ne_core_count` value sourced from the hardware-abstraction blob, not a hard-coded per-family constant. `getDeviceInfo` takes that count as its second argument and buckets it against the thresholds 7, 10, 20, and 40, which straddle the four die sizes 8, 16, 32, and 64 held by the `g`, `s`, `c`, and `d` target suffixes. Several device-information fields then scale linearly with the count, and the top bucket reached on the A14-and-above branch stores the 800-unit peak. The compiler thus emits a 64-core-scaled cost model for H17d purely from the count of 64, and the geometry is decoded while the realized 64-core throughput stays an on-silicon measurement.

The two datapaths above the M5 are the fp8 weight format and the multi-die collective layer, which table 36.1 gives field by field, with each gate and its state on the M5.

Table 36.1. The decoded fields of the two fp8 formats and the collective layer, with their gates and their state on the M5.

field	E4M3	E5M2	collective layer
sign bits	1	1	not applicable
exponent bits	4	5	not applicable
mantissa bits	3	2	not applicable
exponent bias	7	15	not applicable
max finite magnitude	448	57344	not applicable
role	gated weight and activation	fp16 conversion	mesh reduce, gather, slice
family gate	`0x52d` set on H18 only	folds to fp16 broadly	`0x48b` zero on all 28
register encoding	present from A17 encoder	output slot on modern encoders	every setter a stub
state on M5	off	conversion available	inert

36.2 fp8 datapath

The compiler has two eight-bit floating-point formats, and they are not symmetric in role. They are in the internal `ZinTensorFormat` code-generation and direct-memory-access enumeration, not in the serialization data-type enumeration that the operation attributes use. The serialization enumeration runs codes 0 through

10 with fp16 at 3, fp32 at 4, and int8 at 2, and has no fp8 codepoint at all; the fp8 formats are recovered from the byte-size map `_ZinTensorFormatGetSizeInBytes`. Table 36.2 gives the fp8 codepoints in the internal tensor-format enumeration, with the integer and half-precision neighbors that bound them.

Table 36.2. The fp8 codepoints in the internal tensor-format enumeration, with the integer and half-precision neighbors that bound them.

`ZinTensorFormat` code	Format	Bytes
1	int8	1
2	uint8	1
3	fp16	2
`0xb` (11)	fp32	4
`0xc` (12)	E4M3	1
`0xd` (13)	E5M2	1
`0xe` (14)	two-byte live-input format	2
`0x10` (16)	int4	under 1

The compiler has the full set of fp8 variant types as C++ symbols, `e4m3_t` with 1273 references and `e5m2_t` with 176, alongside the finite-and-NaN and unsigned-zero variant type classes. The shipping format is the E4M3FN variant, proven by the ±448 saturation bound that `ZinPadLayer::ValidateBackground PaddingValue` enforces with the string “in [%u, %u] for e4m3 format” and the NaN-only special case in `ZinE4M3Expand`.

E4M3 is the gated weight and activation format. A value decodes from a sign bit, four-bit exponent, and three-bit mantissa as

$$x = (-1)^s \, 2^{e-7} \left(1 + \frac{m}{2^3}\right)$$

for a normal value, with an exponent bias of 7 and a maximum finite magnitude of 448. The shipping variant is E4M3FN, the finite-and-NaN form: it has no infinity, the all-ones exponent with a full mantissa encodes NaN, and a value past ±448 either maps to NaN or clamps to the maximum normal. A one-bit mode held per direct-memory-access source controls the overflow behavior, 0 for NaN and 1 for saturate, so a narrowing cast picks the bit pattern `0x7e` for the saturated maximum or `0x7f` for NaN. The narrower exponent and the ±448 ceiling cap the representable range below half precision.

E5M2 is a conversion format, not a gated weight format. It decodes as

$$x = (-1)^s \, 2^{e-15} \left(1 + \frac{m}{2^2}\right)$$

with five exponent bits, a two-bit mantissa, and the same exponent bias of 15 as half precision. That shared exponent is why E5M2 is the high byte of an fp16 value: the conversion to fp16 is a left shift of eight bits, and the inverse is the top byte with round-to-nearest and an infinity clamp. E5M2 folds to fp16 by a direct-memory-access conversion, so it is broadly available wherever that conversion path exists, and the upscaler layer accepts it as input. The fold asymmetry is exact in the conversion functions. `ZinE5M2ToF16` is a left shift of eight bits, the literal high byte, and `ZinF16ToE5M2` is the top byte with round-to-nearest and an infinity clamp at `0x7c00`. E4M3 is not so trivial. `ZinE4M3Expand` is a bit-surgery decode with sign at bit 7, the four-bit exponent at bits 6 through 3, and the three-bit mantissa at bits 2 through 0, handling the subnormal and NaN encodings. `ZinF32ToE4M3` narrows and selects the overflow byte, `0x7e` for the saturated maximum and `0x7f` for NaN. The fold predicate `IsFormatDMAConvertibleToFP16` returns `(fmt < 0xe) & (0x2ff0 >> fmt)`, and the mask `0x2ff0` includes code 13 and excludes code 12. E5M2 thus

folds by direct-memory-access conversion and E4M3 cannot, which is the binary-level reason E4M3 requires the native datapath that is gated.

The family gate is a single hardware-abstraction-layer capability byte at offset `0x52d`, the E4M3 direct-memory-access and kernel-format capability. It is clear on the M1 generation, clear on the M5 generation, and clear on the intermediate families, and it reads set on the H18 family alone of the twenty-eight compiler targets. The master per-format direct-memory-access validity function `CheckValidDMAFormat` takes the four capability bytes `HAL[0x52c]`, `HAL[0x52d]`, `HAL[0x52e]`, and `HAL[0x685]`. It validates fp32 against `0x52c`, E4M3 against `0x52d`, E5M2 against `0x52e`, and the two-byte live formats against `0x685`, with every code below `0xb` always valid. Three operation-semantics sites re-check the byte and abort with "E4M3 is not supported on this architecture": the dequantize validator at line 129078, the quantize validator at line 227466, and a third semantics validator at line 3407368. Below that runtime gate is a harder compile-time limit on the M1 generation. The M1 task-descriptor encoder packs the source element format into a two-bit field, which holds only the integer and half-precision codes and has no bit space for the E4M3 codepoint. Feeding fp8 to the M1 encoder thus aborts the compile rather than refusing the operation at runtime. The encoder that does have the E4M3 codepoint widens that field to three bits and adds a distinct output slot, and that wider encoder is present from the A17 generation onward even though the runtime capability at `0x52d` fires only on H18.

The operation-side gate matches the encoder, decoded from the compiler and not measured on an H18 part. The `MinimumFamily<N>` trait that chapter 35 reads as the operation-legality floor has a high-N set, a few operations at N=5, N=6, and N=7, the A16, A17, and A18 floors, and the fp8-bearing operations are in that set. The fp8 converters and the quant-unit storage are compiled into the one byte-identical image, so an fp8 operation parses and type-checks on any target. Native execution depends on both that high-N family floor and the `0x52d` capability byte, which is why the datapath is present yet inert below H18.

The format-register delta between the M1 and the H18 encoder is exact. The M1 generation builds the generation-5 and generation-7 descriptors, whose `SetCommonInFmt` writes a two-bit field at `this+0x48` holding only int8, uint8, and fp16, and the code `0xc` aborts with "Error: Invalid Common InFmt E4M3". The generation-20 descriptor that the A17 generation builds packs all three format fields into a 32-bit word at `this+0x228`, widening each lane to three bits. `SetCommonInFmt` gives E4M3 the source-one code 4 at bits 2 through 0, `SetCommonSrc2InFmt` gives it the source-two code `0x20` at bits 5 through 3, and `SetCommonOutFmt` gives it a distinct output slot `0x100` at bits 8 through 6. Table 36.3 gives the task-descriptor format-register delta, showing that the M1 encoder has no bit space for an E4M3 codepoint while the generation-20 encoder widens each lane to three bits.

Table 36.3. The task-descriptor format-register delta across the upper-tier generations.

Field	M1 generation-5 at `this+0x48`	H18 generation-20 at `this+0x228`
source-one format	two-bit lane, int8, uint8, fp16 only	three-bit lane, plus E4M3 as 4
source-two format	folded into the two-bit space, no distinct E4M3 code	three-bit lane, E4M3 as `0x20`
output format	two-bit lane, E5M2 reuses the fp16 slot `0x20`	three-bit lane, E4M3 as `0x100`
E4M3 codepoint	absent, a compile-time assert	present as 4, `0x20`, `0x100`
E4M3 overflow register	stub assert on the generation-8 setter	`this+0x2fc` and `this+0x300`, bit 24

The overflow mode is a per-source register field on the generation-20 encoder, `SetTileDmaSrc1E4M3Overflow` writing bit 24 of `this+0x2fc` and `SetTileDmaSrc2E4M3Overflow` writing bit 24 of `this+0x300`, where 1 selects saturate and 0 selects NaN. On the older encoders the same setter is a stub that asserts "E4M3Overflow is not supported" only when the overflow option is engaged.

In the multiply array fp8 is an input width, not an accumulator width. An E4M3 weight expands to fp16 going into the multiply, the products accumulate in the same wide register every family uses, and the

output port rounds the result to fp16. No fp8 accumulator type, register field, or symbol exists in the compiler: the generation-20 format word encodes only the source-one, source-two, and output element formats, with no accumulator-format field. E4M3 is a native kernel format, code 6 in the kernel-format set that `ZinSetFormat` admits through the mask `0x3b`, alongside int8, uint8, fp16, and fp32; E5M2 is absent from that set, so it is a conversion and upscaler-input format rather than a kernel format. Throughput runs on the same double-rate path int8 uses, gated by `ZinDoubleMacMode::CanUseDoubleMacModeBasedOnFormats`. A one-byte activation against a one-byte kernel of the same numeric class is eligible for the double-multiply mode, the eligibility being the exclusive-or term that is true only when the activation-float bit equals the kernel-float bit. An E4M3 activation against an E4M3 kernel is both one-byte and both float, so it runs at twice the per-element rate into the fp16 accumulator, while a mixed int8-against-E4M3 pair is not eligible. E4M3 is symmetric-only, with the zero point forced to zero, the same constraint the M1 imposes on int8. The quantize and dequantize validators reject a zero point with “Zero point is not supported for quant with E4M3 output format”, and the same rule reaches the palette layer. E4M3 weights stream as a palette through `ZinIrWeight::DePalettizeWeightData<e4m3_t>`, a codebook of E4M3 values indexed by a packed stream at bit-widths 1, 2, 3, 4, 6, and 8. This is the identical machinery the int4, int8, and fp16 palettes use under template specialization, dequantized as scale times E4M3 with no zero point.

36.3 Multi-die collective layer

Table 36.4 gives the collective operations of the multi-die dialect, their backend operation classes, and the unit-type code of each.

Table 36.4. The collective operations of the multi-die dialect, their backend operation classes, and the unit-type code of each.

silc mnemonic	C++ operation class	role	unit-type code
`silc.all_reduce`	`SilcAllReduceOp`	reduce a tensor in place across a mesh axis	78
`silc.all_gather`	`SilcAllGatherOp`	concatenate shards into a replicated tensor	76
`silc.all_slice`	`SilcAllSliceOp`	scatter a replicated tensor into shards	75
`silc.mesh`	`SilcMeshOp`	declare the device mesh	none
`silc.call`	`SilcCallOp`	per-die single-program call	79

The collective layer is a cross-die data path, not the independent per-job steering a multi-die part otherwise does. The kernel load-balancer steers whole independent submissions to the least-busy engine die and never exchanges tensor data between them, and the driver supports up to four dies. The M1 and the M1 Max each register a single engine die, so this steering engages only on a multi-die part such as the Ultra. The compiler’s collective instead splits one tensor across dies and exchanges partials, threaded through an `optional<ZinIrDeviceMesh>` that `GetAndValidateSpmdDeviceMesh` and `ZinParseDeviceMeshAttributes` build, so a program compiled with a device-mesh attribute gets a real all-reduce, all-gather, or all-slice across the mesh rather than per-die placement. The cross-die move itself is the `HandleCcdmaLayer` direct-memory-access primitive below, and `ValidateMeshAxesInTensorFamily` is the family gate on whether a given die admits the mesh. This contrast is decode-derived; whether a two-die Ultra in any reached family accepts the mesh path is a multi-die measurement this guide does not have.

It is a distinct intermediate-language dialect, `mlir::silc`, whose closed operation set is the three collectives above over a device mesh, plus the mesh declaration and the per-die call. The operation-class list is exactly these, with no reduce-scatter and no broadcast, the all-slice operation standing in for the scatter. The collective operations have the attributes `mesh`, `mesh_axes`, `sharding`, the `members` membership list, and `reduce_op`.

The reduction kind is an enumerated attribute decoded directly from the packed-string compare in `symboliz eReductionKind`, which table 36.5 gives token by token.

Table 36.5. The reduction-kind enumeration the all-reduce reduce-operation attribute holds.

Token	Integer
`sum`	1
`max`	2
`min`	3
`product`	4
`mean`	5
miss	0, invalid

The mesh is an N-dimensional grid of dies by engines-per-die, a vector of per-axis extents held in `Zi nIrDeviceMesh`, which exposes the die count, total engine count, and engines-per-die count. A device identifier maps to a mesh coordinate and an engine index by a mixed-radix split at the die axis, which `ZinSPMDUtils::AneIndexFromDeviceId` computes:

$$\text{ane} = \sum_{i \geq d} \text{id}[i] \prod_{j > i,\, j \geq d} \text{ext}[j] \; + \; \left(\sum_{i < d} \text{id}[i] \prod_{j < i} \text{ext}[j] \right) \cdot \text{anesPerDie}$$

where d is the die-axis split index: the axes above the die axis fold into the within-die engine offset, and the axes below fold into the die index, multiplied by engines-per-die. The inverse decode is `DeviceIdFromAneIndex`. A layer runs on a die through `ZinEngineLayer::RunsOnDeviceId`. A layer not assigned to the single-program path runs on all dies. Otherwise it emits on a die only when the engine index for that die is a key in the layer's engine-set map, which is how the `members` attribute becomes a per-die emit decision.

A sharding attribute maps each tensor axis to a mesh axis, splitting that axis into one shard per device along the axis. The split requires even divisibility, which the strings "Input tensor dimension must be divisible by the number of shards along the tensor dimension" and "Kernel dimension must be divisible by number of shards" enforce. The compiler rejects sharding the same mesh dimension twice and allows a replicated section only on a multi-die network.

The reduce-across-the-mesh operation lowers to a collective direct-memory-access that holds a hardware atomic read-modify-write: the reduction happens in memory as the direct-memory-access writes into a shared buffer, which is why the operation requires in-memory-reduction support that the single-die families lack. The reduction-to-atomic map is decoded from the `ZinIrReductionTypeToZinAtomicOpType` switch, which table 36.6 gives along with the reductions that have no hardware atomic and are rejected.

Table 36.6. The reduction-type-to-atomic-operation register map for the all-reduce collective, with the reductions that have no hardware atomic and are rejected.

Reduction type	Atomic-op register value
1	4
2	3
4	1
5	2
8	5
9	6
10	7
3, 6, 7, 11	assert, no hardware atomic

Reduction type	Atomic-op register value
other	0

The atomic configuration packs as `(atomicOp & 0xff) | (atomicDataType << 8)`, and the input supplied to the collective must arrive through a bypass pass-through, asserted by "Input to inter-die AllReduce should be NEBypass". The all-reduce lowering itself further asserts "AllReduce is currently not supported for architectures that do not support in-memory reductions".

`HandleCcdmaLayer<Nu>` programs the collective direct-memory-access into the task descriptor, instantiated for the generations $\{1, 4, 5, 6, 7, 8, 10, 11, 17, 19, 20\}$. Each opens with the collective-enable gate and the per-die predicate before driving the setters in a fixed order. That order is the source mode, the counter mode, the data size, five shape words, four destination strides, the destination base address, the optional constant, four source strides, the source base address, the wait-event address from the mesh symbol, the counter address, the atomic data type, the atomic operation, the counter amount, and the wait-event value. The base addresses are all emitted through `ZinSPMDUtils::GetSymbolOffsetToBaseAddr`, the extended-addressing path that needs the cross-die address-reach byte.

The layer is decoded but inert in the compiler this guide reads. Every collective direct-memory-access register setter is a stub that asserts "CCDMA is not supported for this arch", in every task-descriptor generation present including generation 20, the would-be Ultra path: the body calls the full setter sequence, but the setters abort. The collective-enable capability byte at offset `0x48b` reads zero on all twenty-eight targets including the M-class and Ultra-reference targets. Table 36.7 gives the collective and cross-die capability bytes decoded across the twenty-eight targets, with the collective-enable byte clear everywhere.

Table 36.7. The collective and cross-die capability bytes decoded across the twenty-eight targets, with the collective-enable byte clear everywhere.

Byte	Capability	Floor	M1 (H13)	A14	A15	M5 (H17s)	H18	M-class
`0x48b`	collective-enable	none in this build	0	0	0	0	0	0
`0x55c`	cross-die extended addressing	A14	0	1	1	1	1	mixed, M11 and U2, U3 set, U1 clear
`0x564`	multi-die hazard tracking	A14	0	1	1	1	0	0
`0x687`	multi-die remote dependency	A15	0	0	1	1	0	0
`0x4a4`	M-class secondary cap	M-class only	0	0	0	0	0	1

The compiler thus has the front end of the collective in full: the operation set, mesh and sharding math, reduction-to-atomic map, and per-die dispatch. No family in this compiler arms the register encoding that would drive the engine.

A second-layer gate is in the source-direct-memory-access emitters, where the single-program flag drives an assert "This target does not allow sharding or SPMD functions". A third gate in the tasklet emitter asserts "No tasklet for given architecture" when the per-section tasklet bit and the single-program predicate are not both set. On the M1 all three gates fail, so the M1 is single-die.

The pieces the single-die M1 silicon does touch are the on-die ordering bits the same machinery shares. Those bits are the layer-two barrier, event masks, and remote-dependency bookkeeping, none of which is the cross-die reduction itself. On the M1 generation-10 descriptor the layer-two barrier sets bit 23 of `TD+0x134` and the forward barrier sets bit 30. The event setter writes a 26-bit signal mask into `TD+0x10` and a 26-bit wait mask into `TD+0x18`, and a direct-memory-backed event is rejected with “DRAM Events not supported for architecture”. The distributed unit is the tasklet, the multi-die variant of the descriptor instruction, one tasklet per participating die.

36.4 What requires newer parts

The fp8 datapath requires an H18-class part to set the capability byte and exercise the native E4M3 multiply. The collective layer requires a multi-die part and a newer compiler that arms the collective-enable byte and replaces the register stubs with a real encoding. Until that hardware and that compiler are in hand, the encodings here stand as the predicted upper tier of the line.

Back Matter

Methodology

The measured silicon, the tools, and what each technique can and cannot observe.

Open questions

The findings that remain unconfirmed and what would settle each.

Statements

Provenance, reproduction, and the work's declarations.

Methodology

SUMMARY

Every finding rests on one of four techniques: a direct private-runtime path, static decompilation of the stack, live read-only instrumentation, and compile-and-run probing. The results rest on two measured silicon points, M1/H13 as the primary host and M5/H17s as the second, with claims marked as measured on a named generation or as predicted.

Every quantitative claim in this guide is either measured on running silicon, read out of a binary or table, or marked as predicted from those artifacts. This chapter states how the engine was reached and how it was characterized. The four converging lines of evidence behind the findings are the direct private-runtime path, static decompilation, live instrumentation, and cross-silicon measurement, which figure 1 shows against the engine.

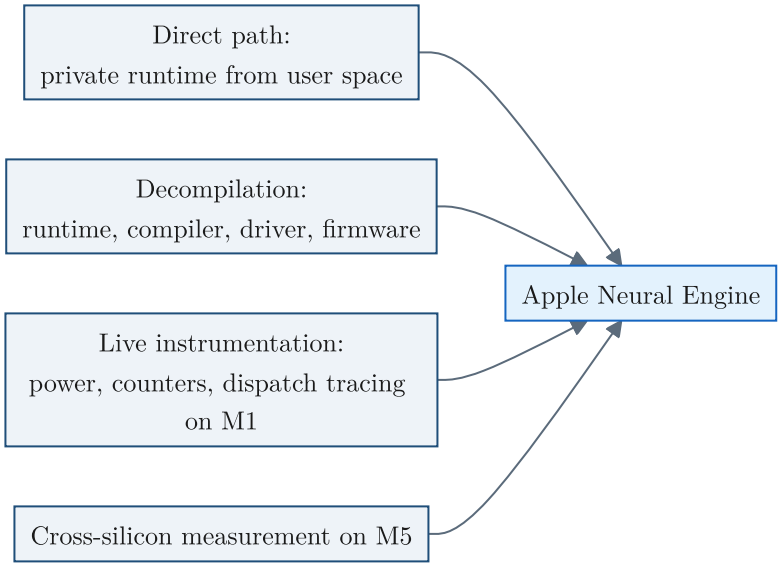

Figure 1. The four lines of evidence behind the guide findings.

Reaching the engine

The engine was reached without the public model framework. The same private Espresso and dispatch runtime that the system's own dispatchers use is callable from ordinary user space. No high-level framework is in the path, and the operations the compiler accepts need no special entitlement. The compiler lowers a graph to the engine's program format, and the runtime loads it and drives it through an execution stream directly. This route bypasses the public compute-unit selector [AppleCoreML] and holds every measured result in the guide.

In-process instrumentation mapped the dispatch path: a library inserted into the dispatching process interposed on the kernel-driver calls. The mapping showed the dispatch as input and output surfaces passed to an asynchronous driver selector, and it located the boundary below which user space cannot see.

Static analysis of the stack

Decompilation and static analysis of four artifacts read out the structure of the stack, which table 1 pairs with what static analysis recovered from each.

Table 1. The four decompiled artifacts of the stack and what static analysis read out of each.

Artifact	What was read out
the dispatch and compiler runtime	the operation vocabulary, the compiler passes, and the program bundle format
the kernel driver	the user-to-kernel boundary and the expanded program binary it lowers
the engine firmware	the on-engine execution model and the host-to-firmware command protocol
the intermediate-language operation set	the lowering from the public operation set to the engine's own

Apple distributes the firmware on the M1 unencrypted, an ARM64 real-time-kernel image behind an image wrapper. Its execution loop, its driver, and its ninety-three-command host protocol were decoded statically rather than inferred. The compiler lowers a graph to a ninety-seven-operation internal vocabulary that is a superset of the public set. Static reading of those passes separated the hidden internal operations from the user-reachable ones.

Live instrumentation

Three escalating instruments supplied the running values that static reading cannot recover, each read-only or recoverable on a dedicated host. A user-space counter interface exposed live hardware counters: DRAM bytes moved, energy in millijoules, and clock frequency. Sampling those counters around a hot loop gave the roofline directly. That fixed the M1 at about 12 fp16 TFLOP/s of compute, about 85 GB/s of DRAM bandwidth, and near 0.5 pJ per FLOP sustained, 0.37 at the compute optimum. It also fixed a 2 MB on-chip working-set threshold, confirmed by the counters falling off exactly where the working set crosses 2 MB. A signpost trace on the engine subsystem gave the op-level event sequence and independently confirmed the bundle finding that each dispatch wraps three driver requests.

The lowest-level instrument was kernel tracing with boot security lowered on a wipeable development machine. Read-only function-boundary tracing instrumented over a hundred thousand kernel probes, about eighteen hundred of them inside the engine driver, with no driver extension and no crash class. That trace captured the expanded program binary that does not exist on disk. The binary is a sectioned executable lowered below user space whose program section is a list of forty-four-byte records, each one a register write that wires a buffer address into a direct-memory-access engine.

Attestation versus reachability

A capability listed in a hardware table or accepted by the frontend attests existence, not that the engine will run it. Only a compile-and-run on the target confirms a capability. One case forced the rule: the hardware abstraction layer advertises three-dimensional convolution and the intermediate language recognizes it, yet it fails backend lowering on every device mask and never reaches the engine. Two-dimensional convolution, fused attention path, normalization family, and activation and reduction set appear here because they survived compile-and-run probing, not because a capability bit promised them. The same rule, read off the counters

and the device mask, separates operations that compile but route to the CPU or GPU from operations that run on the engine.

What each technique cannot see

The boundaries do not overlap. Table 2 pairs each characterization technique with what it observes and the boundary it cannot cross.

Table 2. Each characterization technique paired with what it observes and the boundary it cannot cross.

Technique	What it observes	What it cannot observe
decompilation and static analysis	code structure, formats, vocabularies, command and register tables	runtime values, and any data the firmware computes rather than stores
user-space hardware counters	aggregate power, energy, bandwidth, and frequency around a loop	per-operation register-level timing, which is gated below the dispatch layer
signpost and dispatch tracing	the op-level event order and the three-request dispatch shape	the expanded program, which is lowered below user space
kernel function-boundary tracing	the expanded program binary and the section layout it lowers	the semantic identity of an individual register, virtualized per load by the memory-management unit
compile-and-run on the target	what the engine accepts and runs, and its numeric output	another generation's behavior, which requires that chip to measure

The last unmapped item is the name of each individual silicon register. Its address is a per-load device address that the memory-management unit remaps on every load, the firmware writes it through a generic writer with no address-to-name table, and the per-offset naming is undocumented. What remains is a labeling gap behind address virtualization and undocumented silicon, not a deeper layer left uncaptured.

Two measured silicon points

Two generations were measured directly, and they anchor the cross-generation claims. The M1, internally H13, is the primary measurement host. It fixed the roofline, unencrypted firmware decode, kernel capture of the expanded program down to the register-write records, operation-conformance set, and all four axes of the fp16 divergence model. It is also where the system-wide compile service was characterized. A rapid burst of compile crashes spaced faster than the daemon's ten-second relaunch drives an unrelated control compile from 120 ms to a multi-minute hang. This is a rate condition rather than a per-crash leak, recoverable only by terminating the daemon.

The M5, internally H17s, is the second measured point and confirmed the cross-generation scaling. It established that the numeric divergence accumulates to at most a fraction of a percent over a full training run, that the family-wide capability limits are not specific to one chip, and that the operation set is portable across the generations. A capability table can be cross-compiled statically for an unmeasured generation, but only the chip itself confirms its numerics.

The M5 ran with System Integrity Protection enabled, the shipping configuration, while the lowest-level M1 instrumentation lowered boot security. Security and isolation claims therefore take the M5 as the authoritative point, since it reports the entitlement gates, exclave boundary, and counter access a normal client meets under the enforced configuration rather than what a lowered-security system exposes.

Reproducing the measurements

Every headline number maps to one command and one committed result file, taken in a fixed environment. The measurements were taken with ANECompiler 9.509.0, whose intermediate-language component is versioned

3520.4.1, on macOS 14 or later (verified on macOS 26.5), with Python 3.10 or later (developed and verified on Python 3.14), and with numpy as the only core numeric dependency. Python 3.10 is the floor because several core modules use PEP 604 union syntax in runtime-evaluated signatures.

The reproduction has two layers. A single top-level driver script runs the full sequence and is fail-soft: it reports and skips a device or power step that cannot run, so the deterministic claims still reproduce. Beneath it are the individual measurement harnesses, each producing a committed result file, so a paper claim resolves to a command and to a stored result. A capability smoke test and an operation smoke test confirm that the closed capability set and every released operation compile and run on the engine. A corpus runner is the correctness gate that the optimizer is held to. A device-comparison harness records latency and speed-per-watt across the engine, GPU, and CPU; a roofline harness records the saturation and bandwidth ceilings. Both harnesses write JSON result files that the committed roofline analysis and figure are built from.

Several limits bound how the numbers should be read.

- The per-rail power figures come from the system power estimator, `powermetrics`, which reports a modeled estimate rather than an independent wall-meter measurement. There is no separate validation of that estimator: the reported total-package active power with idle subtracted is a calibrated estimate, not a metered reading.
- The work runs on Apple silicon only. It installs on any platform but cannot run without an Apple silicon Mac and the built dispatch library, and there is no CPU, GPU, or other fallback for the engine path.
- The path calls private Apple framework symbols tied to ANECompiler 9.509.0. A macOS update can break the dispatch-library build or the dispatch path, and none of it is an Apple API contract.
- Determinism splits by claim type. The capability census, operation conformance check, operation smoke test, and correctness corpus are deterministic pass-or-fail gates; the device-comparison, serving, and roofline numbers are measurements and vary run to run.

Open questions

SUMMARY

Open questions fall into three kinds: questions another chip or an on-silicon probe would answer, questions the sanctioned entitlement would answer, and questions that are impossible to answer because the data does not exist or the silicon leaves no observable trace. Each item gives why it is open and what would close it.

The engine is decoded from the host call down to the register writes, and everything that exists as bytes or strings has been resolved into human-readable form. Two silicon points are measured directly: M1/H13 and M5/H17s. M5/H17s confirmed all ten cross-silicon predictions on device. The M3/H15 generation and the upper tier above H17s are decompile-derived, not measured. M5/H17s closes none of the upper-tier items, because it is a 16-core H17s part rather than the H17d 64-core ceiling, H18-gated fp8 runtime, or Ultra multi-die collective.

Why an item is open

Each open item falls into one of three kinds, which set the three tables below. A chip-measurement item has its structure decoded but needs another generation, the upper tier, or an on-silicon probe to read the realized values. An entitlement item is attested in the hardware-abstraction layer or the intermediate language but is reachable only through the sanctioned model path. An impossible item cannot be resolved at all: either the data does not exist, or the behavior is irreducible silicon that leaves no trace in any result and no artifact to decode.

Ledger

Three tables sort the open items by how each could be resolved. Table 3 gives the items another chip or an on-silicon probe would resolve, each with its decoded structure and the measurement that would close it.

Table 3. Open items a further on-silicon measurement would resolve: the structure is decoded, the realized values need the part or the probe.

Open item	Why it is open	What would close it
Live values of the single-shot fault and translation-fault capture registers	The registers are located (four fault descriptors and a status word at engine+0xe028) and the ANE-side handler has a benign branch, but the IODART substrate panics on a real fault, so the live values are uncaptured	A fault path that recovers instead of panicking the IODART substrate
Realized A15 / M3 / H15 silicon behavior	The dedicated A15 compiler branch is decoded (its cost table, its 45-operation family set, the full H15 targets, the YUV420 input it alone accepts), only the realized numbers need the part	On-chip measurement of an H15 part

Open item	Why it is open	What would close it
Realized upper-tier behavior: 64-core H17d, H18-gated fp8 datapath, Ultra multi-die collective	The encodings are decoded (the core-count parameter, the fp8 e4m3 and e5m2 convert-and-quantize path, the Ultra device-mesh collective) and the upper tier adds only cores over A16, no new operations, but none of it materializes without the part	An H17d, H18, or Ultra part
Set behavior of the family-gated abstraction-layer fields	The fields are named and the silicon-capability subset is complete, their set behavior needs later silicon	A part where the gate is on, for example the A18 or H17-plus FIFO-mode field
Per-state frequency and voltage of the operating-point sequence	The engine has no local DVFS and is driven by the SoC power controller, the firmware seven-step credit sequence is recovered, the per-state frequency and voltage stay behind the opaque power-management base	A power-management-side probe

Table 4 gives the items the sanctioned entitlement would resolve, attested in the binaries but gated off the direct path.

Table 4. Open items the sanctioned entitlement would resolve, attested but gated off the direct path.

Open item	Why it is open	What would close it
Live per-run firmware performance-counter values	The read path and its gate are decoded (the `aned` daemon clears the stats mask for a third-party client, and a host-side null check skips the buffer on the unentitled path), only the live values need the entitled output buffer	The entitled path that supplies the output buffer
Entitled-only features: 3-D convolution, native state and ring buffer, bf16 program input and output, flexible shapes	Attested in the hardware-abstraction layer but unreachable on the direct path	The sanctioned model path that can reach them

Table 5 gives the items that cannot be resolved: the data does not exist, or the behavior is irreducible silicon that leaves no trace in any result.

Table 5. Open items that cannot be resolved externally: irreducible silicon behavior, or data that does not exist.

Open item	Why it cannot be resolved
Gate-level MAC and adder-tree wavefront skew	The output is order-independent, so no result reveals the per-cycle wiring, which sits below the timing floor; only Apple's register-transfer netlist would show it
Exact fp16 partial-sum rounding order within one MAC reduction	The wide accumulator makes the output bit-identical for any summation order, so the internal sequence leaves no trace; only the register-transfer netlist would show it
No flat numeric error or status enumeration	Status is held by notification names and positional indices, not a stored enum; the flat enumeration does not exist
The execution-loop state labels	The five states are reconstructed from handler behavior; no name table exists in firmware

Form of the residual

The impossible items are few: the gate-level reduction wavefront and the internal fp16 rounding order are irreducible silicon, and the flat status enumeration and the execution-loop state labels do not exist as stored

data. Everything else is decoded in structure and waits only on a part, a probe, or the entitled path: the chip-measurement items the tables above list, and the entitlement items attested in the binaries but gated off the direct path. Static analysis closed the firmware address rebase that once sat at the host and firmware boundary, and the M5 measurement confirmed the cross-silicon model without reaching any of the parts the hardware items need.

Statements

Author

Spencer Bryngelson, Georgia Institute of Technology. Correspondence: shb@gatech.edu.

Data and code availability

ANEForge, the open-source code artifact accompanying this guide, is at https://github.com/comp-physics/ANEForge, described in arXiv:2606.17090. Appendix E records the provenance of every substantive claim, measured or decompile-derived, and the methodology summarizes it.

Ethics and responsible disclosure

The work reported here is reverse engineering conducted on the author's own Apple hardware, by static decompilation and on-device measurement, for interoperability and research. The guide redistributes no Apple source code or proprietary binaries; it documents facts about the hardware and its interfaces. The direct route it describes is undocumented, unsupported, and version-fragile, and is intended for measurement, research, and on-device work, not for shipping software, where Core ML remains the supported path.

Funding

This was independent work and received no external funding.

Competing interests

The author declares no competing interests. This is an independent work and is not affiliated with, authorized by, or endorsed by Apple Inc.

Appendices

A Operation-by-device matrix

Every operation against every device, native or decomposed.

B Hidden-layer catalog

The native layers and their descriptors, with parameters and symbols.

C Decoded reference tables

The enum tokens, structs, status codes, register map, and program schema.

D Glossary

The terms, the family and silicon map, and the core facts to read first.

E Provenance

The evidentiary basis of every claim, Part by Part and chapter by chapter.

Appendix A Operation-by-device matrix

SUMMARY

This appendix is the full per-operation, per-family status reference behind chapter 4. Read down a family column to see what compiles and runs on that chip, and read the status marks and note for the gate and the route.

Each row is one intermediate-language operation, grouped by operation class, with its status on each Mac engine family from the M1 through the M5. Chapter 4 summarizes this table; the cells here are the reference.

The status marks are fixed:

- Native: the operation compiles and runs on that family on the direct engine path.
- Family-gated: no path on the listed family, native from the family named in the note.
- Bridge: reachable only through a decompose, software fallback, or compiler-internal route, never as a standalone code-generated operation.
- No path: rejected on every family from the M1 through the M5, computed off-engine.

The family columns are M1 (H13, A13), M2 (H14, A14), M3 (H15, A15), and M4 and M5 (H16 and H17s, A16 and A17). The A11 and A12 engines are below the floor that runs any of this vocabulary and are out of scope for the table.

The M1, M2, and M5 columns are measured on physical silicon. The M3 column and the M4 part of the merged M4 and M5 column are decompile-derived predictions from the per-chip tables, so a per-cell status there is a predicted capability rather than a measured one.

The table covers the 187 intermediate-language operations the compiler exposes. Of these, about 108 are native on the M1: the full elementwise, compare, activation, convolution, pooling, structural, and quantization vocabulary, plus the reduction, normalization, softmax, square-root family, fused attention, tile, and space-channel set. Nine need the M2 or later: the texture-engine operations (crop-resize, resample, affine, hardware gather) and the rank and sort bridge (top-k, sort, dynamic slice). Four need the M3 or later: native `sin` and `cos`, the hardware random generator, and the whole-tensor argument reductions on the intermediate-language route. Thirty-seven are rejected on every family and decompose on the host. About twenty-four are compiler-internal: mapped but with no observed standalone code generation, reachable only inside a wrapping construct.

A.1 Per-chip numeric limits

The status of an operation is one axis; the numeric envelope it runs in is the other. Table A.1 gives that envelope across the five capability tiers, measured from the live compiler by calling every per-architecture parameter constructor on a single M1, and a dash marks an unsupported value. The `older` column is the

pre-A13 legacy targets the compiler still has parameter tables for, below the floor of the operation-status table above.

Table A.1. The per-chip numeric limits of the engine, measured from the live compiler across the five capability tiers.

Limit	older	M1, A13	A14	A15	A16, M5
max kernel W (default format, large)	29	29	32	32	32
max kernel W (fp16, large)	13	13	16	16	16
max kernel W (default, small)	15	15	16	16	16
max kernel W (fp16, small)	7	7	8	8	8
min kernel W (default / fp16, large)	16 / 8	16 / 8	1 / 1	1 / 1	1 / 1
max kernel H (large / small)	29 / 15	29 / 15	32 / 16	32 / 16	32 / 16
max kernel D (large / small)	1 / 1, no 3D	16 / 8	16 / 8	16 / 8	16 / 8
max patch W / H / D	15 / 15 / 0	28 / 28 / 15	31 / 31 / 15	31 / 31 / 15	31 / 31 / 15
max tensor W / H	16384	16384	16384	16384	65536
max tensor D	1	16384	16384	16384	65536
max tensor C	65536	65536	65536	65536	65536
max tensor N (batch)	4096	65536	65536	65536	65536
max transpose W / H	0, always split	16384	16384	16384	65536
reduction-to-transpose threshold	none	192	192	384	384
group-conv decompose limit (Cin·kW·kH)	64	2048	2048	2048	2048
stride factor list	[2,3,4,8]	[2,3,4,8]	[2,3,4,8]	[2,3,4,8]	[2,3,4,8]
matmul SRAM working set	2 MB, M9 1 MB	2 MB	2 MB	2 MB	2 MB
DMA width granule	16 B	16 B	16 B	16 B	16 B
patch-width floor / max	none	16 / 512 px	16 / 512	16 / 512	16 / 512
instruction alignment	256 B	256 B	16 B	16 B	16 B
has texture engine	no	no	yes	yes	yes
kernel-memory budget	64 KB	64 KB	64 KB	64 KB	64 KB
activation-LUT budget	150 B	86 B	86 B	86 B	86 B
context-switch live-tensor limit	2	2	∞	∞	∞

The four generational dividing lines are visible in this table. The M1 adds the depth and three-dimensional axis and every reduction-class operation. The A14 adds the texture engine. The A15 raises the reduction-to-transpose threshold from 192 to 384 and adds native trigonometry. The A16 quadruples the maximum tensor and transpose dimensions from 16384 to 65536.

A.2 Convolution, matrix multiply, and pooling

Table A.2 lists the convolution, matrix-multiply, and pooling operations with their per-family status and the lowering note for each.

Table A.2. Convolution, matrix-multiply, and pooling operations by device family.

Operation	M1 (A13)	M2 (A14)	M3 (A15)	M4, M5 (A16, A17)	Note
`conv`	Native	Native	Native	Native	M1 kernels up to 29x29, M5 up to 32x32; Winograd auto-selected for eligible 3x3 stride-1 convs
`conv_transpose`	Native	Native	Native	Native	Deconvolution; strided axes use the small-kernel caps
`linear`	Native	Native	Native	Native	Folds to convolution when the right operand fits the on-chip working set
`linear_activation`	Native	Native	Native	Native	Fused linear and activation
`matmul`	Native	Native	Native	Native	Engine lane or convolution fold; same tensor caps as convolution
`ne_matmul`	Native	Native	Native	Native	Private engine-lane matrix-multiply unit
`einsum`	Native	Native	Native	Native	Lowers to a matmul and transpose chain
`ne_conv`	Native	Native	Native	Native	Private engine-lane convolution unit
`avg_pool`	Native	Native	Native	Native	Window up to 29 on the M1, up to 31 from the M2
`max_pool`	Native	Native	Native	Native	
`l2_pool`	Native	Native	Native	Native	Lookup-table pool
`ne_pool`	Native	Native	Native	Native	Private engine-lane pooling unit
`pe_pool`	Native	Native	Native	Native	Private planar-engine pooling unit
`pe_elementwise`	Native	Native	Native	Native	Private planar-engine elementwise unit
`pe_goc`	Bridge	Bridge	Bridge	Bridge	Private planar-engine gain-offset unit, compiler-internal
`ne_bypass`	Bridge	Bridge	Bridge	Bridge	Private engine-lane bypass unit, compiler-internal
`scaled_dot_product_attention`	Native	Native	Native	Native	Runs on the matmul and softmax path, not texture-gated

The `ne_` and `pe_` rows are private engine-lane and planar-engine unit selections of the same convolution, matrix-multiply, pooling, and elementwise atoms, not separate operations.

A.3 Normalization

Table A.3 gives the normalization operations, native on every family from the M1.

Table A.3. Normalization operations by device family.

Operation	M1 (A13)	M2 (A14)	M3 (A15)	M4, M5 (A16, A17)	Note
`batch_norm`	Native	Native	Native	Native	Inference fold-to-affine; native statistics form from the M1
`layer_norm`	Native	Native	Native	Native	

Operation	M1 (A13)	M2 (A14)	M3 (A15)	M4, M5 (A16, A17)	Note
instance_norm	Native	Native	Native	Native	
l2_norm	Native	Native	Native	Native	
local_response_norm	Native	Native	Native	Native	Measured on the M1

A.4 Elementwise arithmetic

Table A.4 gives the elementwise arithmetic operations, where only mod takes no engine path.

Table A.4. Elementwise arithmetic operations by device family.

Operation	M1 (A13)	M2 (A14)	M3 (A15)	M4, M5 (A16, A17)	Note
abs	Native	Native	Native	Native	
add	Native	Native	Native	Native	Constant and tensor forms
sub	Native	Native	Native	Native	Lowered to add of a negated constant
mul	Native	Native	Native	Native	Constant and tensor forms
real_div	Native	Native	Native	Native	General divide
floor_div	Native	Native	Native	Native	Lookup-table assisted
pow	Native	Native	Native	Native	
square	Native	Native	Native	Native	
sqrt	Native	Native	Native	Native	Lookup-table activation
rsqrt	Native	Native	Native	Native	Lookup-table
inverse	Native	Native	Native	Native	Reciprocal lookup-table
maximum	Native	Native	Native	Native	
minimum	Native	Native	Native	Native	
mod	No path	No path	No path	No path	Decompose on host
cumsum	Native	Native	Native	Native	Native through a curated runtime path, not the standard compile path; M1 measured

A.5 Comparison and logical

Table A.5 gives the comparison and logical operations, the bitwise-logical ones decomposing on the host.

Table A.5. Comparison and logical operations by device family.

Operation	M1 (A13)	M2 (A14)	M3 (A15)	M4, M5 (A16, A17)	Note
equal	Native	Native	Native	Native	
not_equal	Native	Native	Native	Native	
greater	Native	Native	Native	Native	
greater_equal	Native	Native	Native	Native	
less	Native	Native	Native	Native	
less_equal	Native	Native	Native	Native	
logical_not	Native	Native	Native	Native	
select	Native	Native	Native	Native	The where operation

Operation	M1 (A13)	M2 (A14)	M3 (A15)	M4, M5 (A16, A17)	Note
`logical_and`	No path	No path	No path	No path	Decompose through minimum or multiply on host
`logical_or`	No path	No path	No path	No path	Decompose through maximum on host
`logical_xor`	No path	No path	No path	No path	Decompose through not-equal on host

A.6 Activations

Table A.6 gives the activation operations, native on every family and most lookup-table backed.

Table A.6. Activation operations by device family.

Operation	M1 (A13)	M2 (A14)	M3 (A15)	M4, M5 (A16, A17)	Note
`relu`	Native	Native	Native	Native	
`relu6`	Native	Native	Native	Native	Lookup-table
`leaky_relu`	Native	Native	Native	Native	Lookup-table
`prelu`	Native	Native	Native	Native	Per-channel slope; native at rank 3 or above
`clamped_relu`	Native	Native	Native	Native	Lookup-table
`thresholded_relu`	Native	Native	Native	Native	Lookup-table
`threshold`	Native	Native	Native	Native	Lookup-table
`clip`	Native	Native	Native	Native	The clamp operation
`elu`	Native	Native	Native	Native	Lookup-table
`sigmoid`	Native	Native	Native	Native	Includes the hard variant
`sigmoid_hard`	Native	Native	Native	Native	Lookup-table
`tanh`	Native	Native	Native	Native	Lookup-table
`scaled_tanh`	Native	Native	Native	Native	Lookup-table
`gelu`	Native	Native	Native	Native	Lookup-table approximation
`silu`	Native	Native	Native	Native	Also named swish; lookup-table
`softmax`	Native	Native	Native	Native	Lookup-table
`softplus`	Native	Native	Native	Native	Lookup-table
`softplus_parametric`	Native	Native	Native	Native	Lookup-table
`softsign`	Native	Native	Native	Native	Lookup-table
`erf`	Native	Native	Native	Native	Lookup-table
`exp`	Native	Native	Native	Native	Lookup-table
`exp2`	Native	Native	Native	Native	Lookup-table
`log`	Native	Native	Native	Native	Lookup-table
`sign`	Native	Native	Native	Native	Lookup-table
`ceil`	Native	Native	Native	Native	Lookup-table
`floor`	Native	Native	Native	Native	Lookup-table
`round`	Native	Native	Native	Native	Round-to-nearest lookup-table

A.7 Reduction

Table A.7 gives the reduction operations, where `reduce_argmin` is gated and `reduce_prod` takes no path.

Table A.7. Reduction operations by device family.

Operation	M1 (A13)	M2 (A14)	M3 (A15)	M4, M5 (A16, A17)	Note
`reduce_sum`	Native	Native	Native	Native	Reduced axis at or above 192 takes the transpose route, at or above 384 from the M3
`reduce_mean`	Native	Native	Native	Native	
`reduce_max`	Native	Native	Native	Native	
`reduce_min`	Native	Native	Native	Native	
`reduce_sum_square`	Native	Native	Native	Native	The reduce-then-square fusion is M2 onward; the M1 emits an extra fp16 round
`reduce_l1_norm`	Native	Native	Native	Native	
`reduce_l2_norm`	Native	Native	Native	Native	
`reduce_log_sum`	Native	Native	Native	Native	Lookup-table assisted
`reduce_log_sum_exp`	Native	Native	Native	Native	Lookup-table assisted
`reduce_argmax`	Native	Native	Native	Native	Per-axis argmax on all families
`reduce_argmin`	Bridge	Bridge	Native	Native	Per-axis argmin; the intermediate-language route is gated to the M3, the bridge route works on the M1 and M2
`reduce_prod`	No path	No path	No path	No path	Decompose through log-sum-exp on host

The whole-tensor argument reductions `global_argmax` and `global_argmin` follow the same gate as `reduce_argmin`: native on the intermediate-language route from the M3, reachable through the bridge on the M1.

A.8 Data movement and structural

Table A.8 gives the data-movement and structural operations, the largest class, spanning reshape, slice, gather, scatter, and the space-channel set.

Table A.8. Data-movement and structural operations by device family.

Operation	M1 (A13)	M2 (A14)	M3 (A15)	M4, M5 (A16, A17)	Note
`reshape`	Native	Native	Native	Native	Metadata edit
`reshape_like`	Native	Native	Native	Native	
`expand_dims`	Native	Native	Native	Native	
`squeeze`	Native	Native	Native	Native	
`flatten2d`	Native	Native	Native	Native	
`transpose`	Native	Native	Native	Native	Capped by the maximum transpose extent, 16384 through the M3, 65536 on the M5
`concat`	Native	Native	Native	Native	DMA
`split`	Native	Native	Native	Native	
`stack`	Native	Native	Native	Native	
`pad`	Native	Native	Native	Native	Constant pad is native everywhere; symmetric and reflect pad are texture-gated, software on the M1 and native from the M2

Operation	M1 (A13)	M2 (A14)	M3 (A15)	M4, M5 (A16, A17)	Note
`slice_by_size`	Native	Native	Native	Native	M1 and M2 nonzero width-offset routes through a fixed-point crop-DMA that saturates a magnitude above 4094 to infinity; clean from the M3
`slice_by_index`	Bridge	Bridge	Bridge	Bridge	Static-offset slice folds into the descriptor inside a graph
`slice_update`	Native	Native	Native	Native	
`reverse`	Native	Native	Native	Native	Measured on the M1
`reverse_sequence`	No path	No path	No path	No path	Decompose on host
`tile`	Native	Native	Native	Native	Factors of 2, 3, 4, and 8
`gather`	Native	Native	Native	Native	M1 software path valid only for a batch of one and a depth of one; the hardware path is M2 onward
`gather_along_axis`	Native	Native	Native	Native	Same M1 envelope caveat
`gather_nd`	Bridge	Native	Native	Native	M1 software envelope only (batch one, depth one, three-element index channel); native texture path from the M2
`scatter`	No path	No path	No path	No path	Decompose on host
`scatter_along_axis`	No path	No path	No path	No path	Decompose on host
`scatter_nd`	No path	No path	No path	No path	Decompose on host
`depth_to_space`	Native	Native	Native	Native	The pixel-shuffle operation
`space_to_depth`	Native	Native	Native	Native	The pixel-unshuffle operation
`pixel_shuffle`	Native	Native	Native	Native	Engine-lane reorganization, factors of 2, 3, 4, and 8; z-factor must be 1
`pixel_unshuffle`	Native	Native	Native	Native	Engine-lane reorganization; input dimension divisible by the factor
`space_to_batch`	Native	Native	Native	Native	Factor in 2, 3, 4, 8; batch cap 4096 on older families, 65536 on the newer
`batch_to_space`	Native	Native	Native	Native	Inverse of the above
`identity`	Native	Native	Native	Native	Aliases a cast or no-op
`fill`	Native	Native	Native	Native	Constant tensor producer
`fill_like`	Native	Native	Native	Native	Constant tensor producer
`range_1d`	Bridge	Bridge	Bridge	Bridge	M1 code generation rejects it; host-precompute the constant
`crop`	Native	Native	Native	Native	Slice and crop, distinct from the texture crop-resize
`band_part`	No path	No path	No path	No path	Mask on host
`non_zero`	No path	No path	No path	No path	Data-dependent shape
`one_hot`	No path	No path	No path	No path	Decompose through an identity gather on host
`shape`	No path	No path	No path	No path	Static-shape graphs only
`sliding_windows`	No path	No path	No path	No path	Decompose on host

A.9 Image, resize, and texture

Table A.9 gives the image, resize, and texture operations, gated to the texture engine from the A14 with software fallbacks on the M1.

Table A.9. Image, resize, and texture operations by device family.

Operation	M1 (A13)	M2 (A14)	M3 (A15)	M4, M5 (A16, A17)	Note
`resize`	Bridge	Native	Native	Native	Texture-gated; M1 takes a software transpose fallback with different rounding, native from the M2
`resize_bilinear`	Bridge	Native	Native	Native	Software fallback on the M1
`resize_nearest_neighbor`	Bridge	Native	Native	Native	Software fallback on the M1
`upsample_bilinear`	Bridge	Native	Native	Native	Software fallback on the M1
`upsample_nearest_neighbor`	Bridge	Native	Native	Native	Software fallback on the M1
`crop_resize`	Family-gated	Native	Native	Native	Texture engine, M2 onward; no host substitution wired
`resample`	Family-gated	Native	Native	Native	Texture engine, M2 onward
`affine`	Family-gated	Native	Native	Native	Texture engine, M2 onward
`pixel_buffer_to_tensor`	Bridge	Bridge	Bridge	Bridge	Four-character-code image input; an entitlement gate, not a chip gate
`tensor_to_pixel_buffer`	Bridge	Bridge	Bridge	Bridge	Compiler-internal
`gamma`	Bridge	Bridge	Bridge	Bridge	Image-signal operation, compiler-internal
`degamma`	Bridge	Bridge	Bridge	Bridge	Image-signal operation, compiler-internal

A.10 Quantization and dtype

Table A.10 gives the quantization and dtype operations, with the per-family streaming gates carried in the note column.

Table A.10. Quantization and dtype operations by device family.

Operation	M1 (A13)	M2 (A14)	M3 (A15)	M4, M5 (A16, A17)	Note
`cast`	Native	Native	Native	Native	fp16 to fp32 and bool native on the M1; cast to int32 is rejected on the M1
`quantize`	Native	Native	Native	Native	Not texture-gated
`dequantize`	Native	Native	Native	Native	
`const`	Bridge	Bridge	Bridge	Bridge	Folded at compile, not a standalone code-generated operation
`constexpr_affine_dequantize`	Bridge	Bridge	Bridge	Bridge	int4 lookup-table streams from the M1; int8 and affine fold to fp16 below the M2, and stream from the A14 and M2
`constexpr_lut_to_dense`	Native	Native	Native	Native	Palette and lookup-table stream; int4 lookup-table streams natively from the M1
`constexpr_lut_to_sparse`	Bridge	Bridge	Bridge	Bridge	Folded constant; sparse stream from the M3
`constexpr_blockwise_shift_scale`	Bridge	Bridge	Native	Native	Blockwise stream from the M3; folds to fp16 on the M1 and M2
`constexpr_sparse_blockwise_shift_scale`	Bridge	Bridge	Native	Native	Sparse and blockwise stream from the M3

Operation	M1 (A13)	M2 (A14)	M3 (A15)	M4, M5 (A16, A17)	Note
`constexpr_sparse_to_dense`	Native	Native	Native	Native	Sparse streams natively from the M1
`constexpr_cast`	No path	No path	No path	No path	Rejected on every family

A.11 Attention, control flow, and state

Table A.11 gives the attention, control-flow, and state operations, where the state pair is native and the control-flow operations are compiler-internal.

Table A.11. Attention, control-flow, and state operations by device family.

Operation	M1 (A13)	M2 (A14)	M3 (A15)	M4, M5 (A16, A17)	Note
`read_state`	Native	Native	Native	Native	Stateful; needs the inout tensor-descriptor plumbing for a key-value cache
`write_state`	Native	Native	Native	Native	Stateful
`tensor_buffer_to_tensor`	Bridge	Bridge	Bridge	Bridge	Ring and streaming buffer mover, reachable inside a stateful graph
`tensor_to_tensor_buffer`	Bridge	Bridge	Bridge	Bridge	Compiler-internal
`circular_buffer_to_tensor`	Bridge	Bridge	Bridge	Bridge	Ring-buffer reader
`tensor_to_circular_buffer`	Bridge	Bridge	Bridge	Bridge	Ring-buffer writer
`cond`	Bridge	Bridge	Bridge	Bridge	No standalone code generation; flatten on host
`while_loop`	Bridge	Bridge	Bridge	Bridge	No standalone code generation; unroll on host
`call`	Bridge	Bridge	Bridge	Bridge	Inlined

A.12 Recurrent cells

Table A.12 gives the recurrent-cell operations, none of which take an engine path; each unrolls on the host.

Table A.12. Recurrent-cell operations by device family.

Operation	M1 (A13)	M2 (A14)	M3 (A15)	M4, M5 (A16, A17)	Note
`gru`	No path	No path	No path	No path	Unroll to a convolution, matmul, and activation graph on host
`lstm`	No path	No path	No path	No path	Unroll on host
`rnn`	No path	No path	No path	No path	Unroll on host

A.13 Trigonometric, special, and math

Table A.13 gives the trigonometric, special, and math operations, where `sin` and `cos` go native from the M3 and `atan` is the one M1-native primitive.

Table A.13. Trigonometric, special, and math operations by device family.

Operation	M1 (A13)	M2 (A14)	M3 (A15)	M4, M5 (A16, A17)	Note
`sin`	Family-gated	Family-gated	Native	Native	Native from the M3; the M1 and M2 use a host polynomial
`cos`	Family-gated	Family-gated	Native	Native	Native from the M3; the M1 and M2 use a host polynomial
`atan`	Native	Native	Native	Native	The one trigonometric primitive native on the M1
`tan`	No path	No path	No path	No path	Decompose through a sin and cos identity on host
`asin`	No path	No path	No path	No path	Host decomposition
`acos`	No path	No path	No path	No path	Host decomposition
`atanh`	No path	No path	No path	No path	Host decomposition
`asinh`	No path	No path	No path	No path	Host decomposition
`acosh`	No path	No path	No path	No path	Host decomposition
`sinh`	No path	No path	No path	No path	Host decomposition
`cosh`	No path	No path	No path	No path	Host decomposition
`cross_product`	Bridge	Bridge	Bridge	Bridge	Reachable through the bridge route, measured on the M1
`cost_volume`	Bridge	Bridge	Bridge	Bridge	Reachable through the bridge route, measured on the M1
`matrix_decomposition`	Bridge	Bridge	Bridge	Bridge	No observed code generation

A.14 Detection and sampling

Table A.14 gives the detection and sampling operations, the rank and sort bridge gated to the M2 and the random and tensor-list operations off-engine.

Table A.14. Detection and sampling operations by device family.

Operation	M1 (A13)	M2 (A14)	M3 (A15)	M4, M5 (A16, A17)	Note
`non_maximum_suppression`	Bridge	Bridge	Bridge	Bridge	Reachable only with a CPU or GPU backend in the mask; the engine-only mask reports not supported on any backend, so it offloads to the CPU or GPU rather than the engine
`topk`	Family-gated	Native	Native	Native	Rank and sort bridge, M2 onward; the validator is callable on the M1 but code generation rejects it
`argsort`	Family-gated	Native	Native	Native	Sort family, M2 onward; code-generation-rejected on the M1
`random_uniform`	Bridge	Bridge	Native	Native	Hardware generator from the M3; host random below it
`random_bernoulli`	No path	No path	No path	No path	Host random
`random_categorical`	No path	No path	No path	No path	Host random
`random_normal`	No path	No path	No path	No path	Host random
`list_gather`	No path	No path	No path	No path	Tensor-list operation
`list_length`	No path	No path	No path	No path	Tensor-list operation
`list_read`	No path	No path	No path	No path	Tensor-list operation
`list_scatter`	No path	No path	No path	No path	Tensor-list operation

Operation	M1 (A13)	M2 (A14)	M3 (A15)	M4, M5 (A16, A17)	Note
`list_write`	No path	No path	No path	No path	Tensor-list operation
`make_list`	No path	No path	No path	No path	Tensor-list operation

Appendix B Hidden-layer catalog

SUMMARY

This appendix is the reference catalog of the hardware-native layer kinds reached by authoring the network description directly, behind chapter 26.

Every row is a native descriptor the conversion path never emits: the descriptor and its `_ANECValidate<Name>Layer` checker are present in the compiler on every target, and the layer is reached by handing the compiler a Unit whose `Type` is the native name and whose `Params` hold the attributes the matching `ZinParse<Name>Unit` parser reads.

B.1 Catalog

Table B.1 lists each native layer kind with what it computes, its netplist `Type` and compiler symbol, and its family gate.

Table B.1. The hidden-layer catalog.

Layer kind	What it computes	Netplist `Type` and compiler symbol	Family gate
Fused attention	$\mathrm{softmax}(QK^\top \cdot s + M)\,V$ from four operands (Q, K, V, scale) plus an optional fifth additive mask M, with the channel axis holding the sequence	`SDPA`; `ANECSDPALayerDesc`, `ZinParseSDPAUnit`, `anec.sdpa`. The one parsed key is `SubtractMax`, defaulting false in `_ANECSDPALayerDescInitialize` and set true for a correct softmax	All families from the M1 onward; runs on the matmul, softmax, and transpose path, not the texture engine
Sort	Full sort along a chosen axis, ascending or descending, values or argsort indices	`Sort`; `ZinParseSortUnit`. Keys: `Direction`, `SortDimension`, `VectorDimension`, `SortIndices`, `Indices`	Validator callable on the M1, but the code generator rejects `Sort` there; runs on later families
Top-k	The k largest or smallest along an axis, values or indices, index outputs returned float16-encoded and exact below 2048	`TopK`; `ZinParseTopKUnit`. Keys: `Type` (Max or Min), `K`, `SortDimension`, `VectorDimension`, `SortIndices`, `Indices`	Runs on the M1 outside a forbidden band: `K` in $\{3, 4\}$ fails the compiler at every width
Argument min and max	Spatial or channel argmin and argmax over a kernel window	`ArgMinMax`; `_ANECValidateArgMinMaxLayer`. Keys: `Mode` (SpatialArgMax, ChannelArgMax, SpatialArgMin, ChannelArgMin), `KernelWidth`, `KernelHeight`, `Pad*`	All families from the M1 onward
Whole-tensor argument min and max	Argmin or argmax over an entire tensor dimension	`GlobalArgMinMax`. Keys: `Type` (Max or Min), `Dimension`	Gated to the A15 generation; rejected on the M1

Layer kind	What it computes	Netplist `Type` and compiler symbol	Family gate
Spatial rearrange	Depth-to-space and space-to-depth in two channel-ordering conventions, plus space-and-batch reshuffles, parameterized by per-axis integer factors	`PixelShuffle`, `PixelUnshuffle`, `ChannelToSpace`, `SpaceToChannel`, `SpaceToBatch`, `BatchToSpace`; `ZinParse<Name>Unit`. Three int32 keys: `FactorX`, `FactorY`, `FactorZ`	All families from the M1 onward; `BatchToSpace` requires batch N divisible by `FactorX` times `FactorY`
Range normalization	Maps a tensor to its minimum-to-maximum span per row or per column	`MinMaxNormalization`; `_ANECValidateMinMaxNormLayer`, `_ANECMinMaxNormLayerDescInitialize`. Keys: `Dimension` (Width or Height), `Epsilon` as a float16 bit pattern	Width and Height run; `Dimension` of Channel is arch-gated and rejected on the M1
Local response normalization	Cross-channel response normalization over a channel window	`LocalResponseNormalization`; `_ANECValidateLRNLayer`. `Alpha` is a float16 bit pattern divided internally by `KernelChannel`; only the first `KernelChannel` channels are normalized	All families from the M1 onward
Scaled elementwise	A binary elementwise op fused with a scalar scale, $y = s\,(x \operatorname{op} z)$	`ScaledElementWise`. Keys: `Type` (Add, Mult, Sub, and the elementwise vocabulary), `Scale` as a float16 bit pattern	All families from the M1 onward
Template cross-correlation	Valid cross-correlation of a single-channel map with an unflipped template, $y[i,j] = \sum_{u,v} x[i+u, j+v]\, t[u,v]$	`CrossCorrelation`. The MIL frontend rejects the op; the netplist `Type` reaches it directly	All families from the M1 onward
Three-vector cross product	The cross product of two length-3 vectors held in the channel axis, $\operatorname{cross}(x, z)$	`CrossProduct`. Inputs shaped D1 C3 H1 W1; the MIL frontend rejects the op	All families from the M1 onward
Furthest-point sampling	Greedy L2 furthest-point sampling of up to 1024 centroids from up to 8192 points, seeded at the first point, centroids returned channel-major	`FurthestPointSampling`. Keys: `CentroidCount`, `DistanceMetric` (L2 only on this architecture)	All families from the M1 onward
Radius neighborhood search	An L2 ball query returning a points-by-centroids membership matrix, one membership flag per pair	`RadiusSearch`. Two inputs (centroids, points), both D1 C3 H1; key `Radius`	All families from the M1 onward
Stereo cost volume	The L1 matching cost per disparity, $\text{cost}[d,x] = \lvert\text{aux}[x] - \text{ref}[x+d]\rvert$, over $R+1$ disparity planes	`CostVolume`. Keys: `DisparityDirection`, `DisparityRange`; requires reference width $W_r \geq W_a + R$	All families from the M1 onward
Re-strided input view	A contiguous offset window $x[\text{Offset} : \text{Offset} + \text{Size}]$ along one named axis, no data movement	`InputView`. Keys: `Dimension`, `Offset`, `Size`, `Step`; gates `InvalidInputView{Dimension,Offset,Size,Step}`	All families from the M1 onward
Runtime-offset dynamic slice	A window $x[\text{start} : \text{start} + \text{SliceSize}]$ whose start is bound as a constant or runtime index	`DynamicSlice`; reached by the MIL `slice_by_index` path. Keys: `DynamicSliceAxisOrder`, `DynamicSliceInfo`, `CoordinateInfo`, `PaddingInfo`, `BackgroundValue`	Validator callable on the M1, but the code generator rejects `DynamicSlice` there; runs on later families

Layer kind	What it computes	Netplist Type and compiler symbol	Family gate
Tile, concatenate, and reshape utilities	Flatten as an NCHW identity reshape, inference-time dropout as identity, and broadcast of a length-1 axis	`Flatten`, `Dropout` (rate 0), `Broadcast` (keys `Dimension`, `Size`)	All families from the M1 onward

B.2 Arch-gated negatives

Three rows above name layers a later chip accepts and the M1 rejects, by the family gates of chapter 12. `GlobalArgMinMax` is gated to the A15 generation and rejected on the M1. `MinMaxNormalization` with a Channel reduction is arch-gated and rejected on the M1, while its Width and Height reductions run. The texture-engine samplers (resize, crop-and-resize, grid resample, and the affine spatial transform) are accepted from the A14 generation and rejected on the M1, where the compiler reports that the affine transform is not supported on this architecture. They are part of the same gated family but are not authored as netplist Units here.

A second class of rejection is not a family gate but the attested-is-not-reachable rule of chapter 4: the `Sort` and `DynamicSlice` validators are callable on the M1, yet the code generator rejects both, and `TopK` is accepted only outside the $\{3, 4\}$ band. An authored layer is confirmed by a compile-and-run on the target, not by the presence of its descriptor.

B.3 Validator gate set

Every authored layer passes through one per-layer validator, the `_ANECValidate<Op>Layer` family, of which 55 symbols are exported and 50 are per-layer. The compiler runs the same validators in two roles: the segmenter dry-runs them through `_ANECValidateNetworkCreate` to decide engine eligibility, and the backend legalizer re-runs them during a real compile, so the dry-run prediction never drifts from the compile result. The five non-layer exports are `_ANECValidate`, `_ANECValidateNetworkCreate`, `_ANECValidateMPSModule`, `_ANECValidateMPSModuleCreate`, and `_ANECValidateMutableProcedureInfo`.

Each validator reads a fixed bottom (input-tensor) count and a per-chip feature byte from the hardware-abstraction layer, and rejects with a measured literal string. Table B.2 reproduces those gates for the validators that guard the authored and bridge-reachable layers, with the bottom count, constraint, and reject string for each.

Table B.2. The per-layer validator gates.

Validator	Bottoms	Gate and constraint	Reject string
`SDPA`	4 or 5	key and value same shape; mask broadcast-compatible; scale constant	`SDPA layer must have only 4 or 5(optional mask) inputs`
`Conv`	1 plus weight	kernel within the per-chip range; large kernel W and H multiple of 8; channels divisible by groups	`Invalid conv kernel %s = %zd, It should be in [%zd , %zd]`
`MatrixMult`	2	depth 1 on both operands; out-C equals A-C; fits the kernel-memory budget	`depth > 1 is not supported for MatMult`
`Linear`	1 plus weight	input rank below 5	`Linear layer must have only one single input.`
`Pool`	1	window below input; pad below kernel; mode gated per chip	`Pooling mode "%s" is not available on this ANE architecture.`

Validator	Bottoms	Gate and constraint	Reject string
`Neuron`	1	non-linear mode 1 to 46; type in the per-chip list; ReLU-N positive parameters when the gate byte is 0	`This platform doesn't support Neuron %s`
`Reduction`	1	reduce-then-square needs feature byte `0x494` (0 through the M1); each axis at most 4	`square operation after reduction is not supported`
`Softmax`	1	feature byte `0x815` (0 on older); output Float	`Softmax is not supported by this ANE architecture`
`LayerNorm`	1	channels divisible by num-groups; grouped form requires depth 1	`... does not yet support depth > 1`
`InstanceNorm`	1	feature byte `0x816`; spatial axes only	`InstanceNorm layer not supported for this ANE architecture`
`MinMaxNorm`	1	feature byte `0x818`; spatial only, the Channel axis arch-gated	(encoded assert, byte `0x818`)
`LRN`	1	feature byte `0x81a`; channel count of 16 or above fails code generation on the M1	`LRN is not supported on this architecture.`
`ArgMinMax`	1	channel-reduce C at most 2048 fp16; pad below kernel; equal left and right, zero front and back	`ArgMinMax layer must have one input`
`GlobalArgMinMax`	1	feature byte `0x4f2` (1 from the M1 on the bridge route); mode 1 or 2; reduce dimension not 5	(encoded assert, byte `0x4f2`)
`Transpose`	1	permutation valid; extent capped 16384 through the M3, 65536 on the M5; last four dimensions only	`NE Input Transpose is not supported for this arch`
`Concat`	variadic	match input zero on every non-concat axis; constant positive axis; same layout	`Concat layer must have at least 2 inputs`
`Pad`	1	H or W axes only; symmetric and reflect modes need texture byte `0x81d` (0 on the M1)	`Channel padding is not supported on ANE`
`Broadcast`	1	broadcast only from a length-1 axis; depth-axis broadcast needs byte `0x812`	`Broadcast along depth axis is not supported on this architecture`
`Gather`	2	M1 software envelope: data batch 1, depth 1, index channel 3; texture path from the A14	`Cannot decompose layer on this architecture`
`PixelShuffle` / `PixelUnshuffle`	1	depth factor 1; W and H factors in 1, 2, 3, 4, 8; channel divisible by the factor product	`returned invalid:`
`SpaceToBatch` / `BatchToSpace`	1	factors fully factor into 2, 3, 4, 8; batch divisible by the factor product	`Input batch n = %zd is not divisible by factor x = %d * factor y = %d`
`ChannelToSpace`	1	the z dimension is not reorganizable	`ChannelToSpace in z dimension is not supported, current factor.z = %d.`
`Resize`	1	dimension or ratio, not both; sampling axes H and W; texture byte `0x81d` (0 on the M1 takes a software route)	`failed to map resize layer on this arch`

Validator	Bottoms	Gate and constraint	Reject string
`CropResize`	2	index format fp16; texture engine from the A14; same coordinate, method, and padding across axes	`Codegen Error: Invalid Texture CropCfg`
`AffineTransform`	2	matrix fp16; texture byte `0x81d` (0 on the M1)	`affine transform is not supported on this architecture`
`Resample`	2	warp depth 1; warp channel 1 or 2; texture engine from the A14	`Channel size in coordinates should be 1 or 2`
`Sort`	1	direction valid; output fp16 or uint16; validator passes, code generation rejects on the M1	(passes validation, code-generation reject)
`TopK`	1	k in the sort-dimension range; the M1 rejects at code generation, and k in 3 or 4 is forbidden on the M5	(passes validation, code-generation reject)
`Dropout`	1	feature byte `0x4a9` (1 only from the A15); rate in the half-open unit interval	`Dropout layer is not supported on this architecture.`
`Random`	none	feature byte `0x4a9` (1 only from the A15); low below high; output Int8, UInt8, or Float16	`Random layer is not supported on this architecture.`
`RingBufferWriter`	2	the writer must connect to a live-state buffer; circular mode arch-gated	`Circular buffer is not supported on this architecture`
`NMS`	2 or more	boxes channel 4; runs only on a CPU or GPU backend, never engine-native	(passes validation, not engine-native)

The validators are public exported symbols, so they are callable from user space, and this is the basis of the precompile predictor: a callable validator marks a schema-gated layer reachable by direct authoring. A callable validator that accepts the schema does not guarantee the layer compiles, which is why the `Sort`, `TopK`, `DynamicSlice`, and the M1 `CropResize` rows pass the validator and fail at hardware-executable lowering. One opcode-surface gap holds the other way: the RCAS and Reverse operations have an internal semantics validator but no exported per-layer symbol, so they are not reachable by direct authoring through this route.

Appendix C Decoded reference tables

SUMMARY

This appendix collects the decoded values cited throughout the guide into one reference: enum tokens, struct layouts, status codes, the register-init table, command table, register map, and program-container schema. Each section begins with its table and names the research-corpus file that contains the full set when only a representative excerpt is reproduced here.

Each section reproduces the representative and structurally important rows of a larger table. Where a table runs to hundreds or thousands of rows, the full set is in the research corpus and the section names the file that contains it. Every value here is read out of an M1/H13 binary by static analysis.

C.1 Operation-attribute enum tokens to integer values

These are the integer codes the compiler resolves attribute string tokens to. Table C.1 is the front-end activation token map (`MILOpConverter::NeuronTypeFromString`, 33 entries, miss resolves to 0).

Table C.1. The front-end activation token names and the integer neuron codes they resolve to.

token	int	token	int
`relu`	1	`gelu`	23
`leaky_relu`	2	`degamma`	25
`clamped_relu`	3	`trunc`	26
`relu_n` (relu6)	4	`round_nearest`	27
`sigmoid`	5	`floor`	28
`sigmoid_high_precision`	6	`ceil`	29
`tanh`	7	`erf`	31
`silu` (alias `swish`)	9	`threshold_relu`	32
`swish_hard`	10	`gamma`	33
`sqr`	11	`inv`	14
`sqrt`	12	`log2`	15
`rsqrt`	13	`exp2`	16
`elu`	18	`exp`	17
`sin`	20	`sign`	19
`cos`	21	`sigmoid_hard`	22

The serialization dtype enum (`ANECIRDataType`, the `dtype` / `storage_type` attribute) is an 11-code space, given as table C.2.

Table C.2. The serialization data-type codes and the element types they name.

int	type	int	type
0	int4	6	uint16
1	uint8	7	int32
2	int8	8	uint32
3	fp16	9	int64
4	fp32	10	uint64
5	int16		

The MLIR `symbolize*` enums are a length-dispatched chain of packed integer string compares, fully static, each miss resolving to 0, collected in table C.3.

Table C.3. The symbolized attribute tokens and the integer values each one maps to.

attribute	token	int
`padding_style`	`EXPLICIT` / `TF_VALID` / `TF_SAME` / `EXPLICIT_OFFSET` / `ONNX_SAME_LOWER`	0 / 1 / 2 / 3 / 4
`nearest_rounding_mode`	`round_prefer_ceil` / `round_prefer_floor` / `ceil` / `floor` / `round_to_even` / `round_to_odd`	0 / 1 / 2 / 3 / 4 / 5
`reduce_op`	`min` / `max` / `sum` / `prod` / `argMin` / `argMax`	0 / 1 / 2 / 3 / 4 / 5
`data_layout`	`NCHW` / `NHWC` / `OIHW` / `HWIO` / `CHW` / `HWC` / `HW`	0 / 1 / 2 / 3 / 4 / 5 / 6
`data_layout`	`NCDHW` / `NDHWC` / `OIDHW` / `DHWIO`	7 / 8 / 9 / 10
`scatter` mode	`add` / `subtract` / `multiply` / `divide` / `min` / `max` / `set`	0 / 1 / 2 / 3 / 4 / 5 / 6
pool `indices_mode`	`GlobalFlatten1D..4D` / `LocalFlatten1D..4D`	0..3 / 4..7
RNN gate activation	`none` / `relu` / `tanh` / `sigmoid` / `hard_sigmoid` / `scaled_tanh`	0 / 1 / 2 / 3 / 4 / 5
stencil `padding_mode`	`constant` / `mirror` / `mirrorWithEdge` / `clampToEdge` / `zero` / `periodic` / `antiPeriodic`	0 / 1 / 2 / 3 / 4 / 5 / 6
`pixel_format`	`R8Unorm` / `RG8Unorm` / `RGBA8Unorm` / `BGRA8Unorm` / `R16Float`	0 / 1 / 2 / 3 / 4
`pixel_format`	`RG16Float` / `RGBA16Float` / `R32Float` / `RG32Float` / `RGBA32Float`	5 / 6 / 7 / 8 / 9
`arith::RoundingMode`	`to_nearest_even` / `downward` / `upward` / `toward_zero` / `to_nearest_away`	0 / 1 / 2 / 3 / 4
collective reduction	`sum` / `max` / `min` / `prod` / `mean`	1 / 2 / 3 / 4 / 5

A second class of enum is in a packed pointer table in the data segment, the internal `Zin` enums the lowering layer dispatches on, listed in table C.4.

Table C.4. The internal `Zin` enum value-to-token maps the lowering layer dispatches on.

enum	value to token
`ZinIrPoolingType`	1 Avg, 2 Max, 3 ChannelMax, 4 Min, 5 ChannelMin, 6 L1, 7 L2, 8 SpatialAndChannelAvg, 9 SpatialAndChannelMax, 10 SpatialAndChannelMin, 11 SpatialArgMax, 12 ChannelArgMax, 13 SpatialArgMin, 14 ChannelArgMin

enum	value to token
`ZinIrReductionType`	0 Sum, 1 Min, 2 Max, 3 Avg, 4 SatSum, 5 SatSub, 6 ArgMin, 7 ArgMax, 8 BitwiseAnd, 9 BitwiseOr, 10 BitwiseXor
`ZinIrEWType`	1 Add, 2 Mult, 3 Square, 4 Sub, 5 Power, 6 Div, 7 Max, 8 Min, 9 Abs, 10 EqualZero, 11 NotEqualZero, 12 LessThanZero, 13 LessThanEqualZero, 14 GreaterThanEqualZero, 15 GreaterThanZero, 16 Equal, 17 NotEqual, 18 LessThan, 19 LessThanEqual, 20 GreaterThanEqual, 21 GreaterThan
`ZinIrScaledEWType`	1 Add, 2 Mult, 3 SumSquare, 4 Max, 5 Min
`ZinIrPaddingMode`	1 Zero, 2 Negative, 3 Replication, 5 Symmetric, 6 Reflective, 7 Background, 8 DontCare
`ZinIrSamplingMethod`	0 Linear, 1 NearestNeighbor
`ZinIrSamplingGridMode`	0 AlignedCorners, 1 UnalignedCorners, 2 OffsetCorners, 3 Default, 4 OffsetDefault, 5 OffsetDefaultWithNominalScale, 6 StrictAlignedCorners
`ZinIrCoordinateMode`	0 NonNormalized, 1 NormalizedSymmetric, 2 NormalizedReflect
`ZinArgMode`	1 SpatialArgMin, 2 ChannelArgMin, 3 SpatialArgMax, 4 ChannelArgMax
`ZinIrSortDirection`	0 Invalid, 1 Ascending, 2 Descending
`ZinIrTopKType`	0 Invalid, 1 Min, 2 Max
`ZinIrFlattenType`	1 NCHW, 2 NHWC
`ZinIrDimension`	0 N, 1 D, 2 C, 3 H, 4 W

The op-class selector is the `ZinUnitType` table, 79 entries, the engine-unit each layer routes to, given in full as table C.5.

Table C.5. The complete 79-entry `ZinUnitType` op-class table and the engine unit each value selects.

int	unit	int	unit	int	unit
1	Conv	28	LayerNormalization	55	RandomGenerator
2	Pooling	29	LocalResponseNormalization	56	Alias
3	Concat	30	CostVolume	57	CrossProduct
4	ElementWise	31	PixelShuffle	58	Quant
5	ScaledElementWise	32	PixelUnshuffle	59	DeQuant
6	Neuron	33	FurthestPointSampling	60	Linear
7	NeuronCustom	34	SpaceToBatch	61	RingBufferWriter
8	GOC	35	BatchToSpace	62	RingBufferReader
9	DynamicGOC	36	SpaceToChannel	63	BatchNorm
10	ConstMatrixMatrixMult	37	ChannelToSpace	64	Phi
11	Flatten	38	RadiusSearch	65	Condition
12	Unflatten	39	Gather	66	WaitForEvent
13	CrossCorrelation	40	AffineTransform	67	SignalEvent
14	KernelRasterizer	41	Resize	68	NEConv
15	ArgMinMax	42	ResizeAs	69	NEMatMul
16	GlobalArgMinMax	43	Resample	70	NEPool
17	InputView	44	Padding	71	NEBypass
18	MatrixMultiplication	45	Tile	72	PEPool
19	Broadcast	46	CropResize	73	PEElementWise
20	Reduction	47	DynamicSlice	74	PEGOC
21	Transpose	48	PlaneReader	75	AllSlice
22	Reshape	49	PlaneWriter	76	AllGather
23	Shape	50	Sort	77	SDPA
24	Softmax	51	TopK	78	AllReduce

int	unit	int	unit	int	unit
25	InstanceNormalization	52	NMS	79	FunctionCall
26	L2Normalization	53	MatrixDecomposition		
27	MinMaxNormalization	54	Dropout		

The activation non-linear-mode space is a parallel table, `NonLinearModeToString`, 48 slots indexed directly by the lower hardware mode value, reproduced as table C.6.

Table C.6. The complete 48-slot `NonLinearModeToString` table indexed by hardware non-linear-mode value.

idx	mode	idx	mode	idx	mode
0	none	16	rsqrt	32	sin
1	relu	17	clamped_relu_rsqrt	33	cos
2	sigmoid	18	inv	34	gelu
3	sigmoid_high_precision	19	sqr	35	gelu_sigmoid_approximation
4	relu_sigmoid	20	log2	36	degamma
5	sigmoid_hard	21	exp2	37	round_nearest
6	tanh	22	exp	38	trunc
7	clamped_relu	23	elu	39	floor
8	prelu	24	sign	40	ceil
9	relun	25	equal_zero	41	atan
10	swish	26	not_equal_zero	42	atan_part1
11	swish_hard	27	less_than_zero	43	atan_part2
12	dirac	28	less_than_equal_zero	44	erf
13	int	29	greater_than_equal_zero	45	thresholded_relu
14	frac	30	greater_than_zero	46	gamma
15	sqrt	31	custom_lut	47	abs

The micro-op opcode space (`ZinIrOpLayerOpCodeType`, 126 codes, `0x00..0x7d`) has the codes the task-descriptor builder dispatches on, given in full as table C.7.

Table C.7. The complete 126-entry micro-operation opcode table and the layer-kind name each value dispatches on.

op	dec	string	op	dec	string
0x00	0	CONV	0x3f	63	AFFINE_TRANFORM
0x01	1	POOL	0x40	64	PLANE_READER
0x02	2	SCALE_BIAS	0x41	65	PLANE_WRITER
0x03	3	TERNARY_DYNAMIC_GOC	0x42	66	SORT
0x04	4	ACTIVATION	0x43	67	TOP_K
0x05	5	EW	0x44	68	RCAS
0x06	6	SCALED_EW	0x45	69	INDEX
0x07	7	CONCAT	0x46	70	NMS
0x08	8	SPLIT	0x47	71	DROPOUT
0x09	9	COPY	0x48	72	TYPE_CAST
0x0a	10	FLATTEN	0x49	73	STOCHASTIC_ROUND
0x0b	11	UNFLATTEN	0x4a	74	RANDOM_GENERATOR

op	dec	string	op	dec	string
0x0c	12	CROSS_CORRELATIO N	0x4b	75	LINEAR
0x0d	13	CROSS_PRODUCT	0x4c	76	RINGBUFFER_WRITE R
0x0e	14	KERNEL_RASTERIZE R	0x4d	77	RINGBUFFER_READE R
0x0f	15	ARG_MIN_MAX	0x4e	78	CONDITION
0x10	16	GLOBAL_ARG_MIN_M AX	0x4f	79	PHI
0x11	17	MATRIX_MULT	0x50	80	BASICBLOCK_IN
0x12	18	BROADCAST	0x51	81	BASICBLOCK_OUT
0x13	19	FLATTEN_COMPOSIT E	0x52	82	BATCHNORM
0x14	20	UNFLATTEN_COMPOS ITE	0x53	83	WAIT_FOR_EVENT
0x15	21	FPS_WITH_RADIUS_ COMPOSITE	0x54	84	SIGNAL_EVENT
0x16	22	PIXEL_SHUFFLE_CO MPOSITE	0x55	85	ALL_SLICE
0x17	23	PIXEL_UNSHUFFLE_ COMPOSITE	0x56	86	ALL_GATHER
0x18	24	CONV_COMPOSITE	0x57	87	SCALED_DOT_PRODU CT_ATTENTION
0x19	25	MATDECOMP_MATMUL T_COMPOSITE	0x58	88	ALL_REDUCE
0x1a	26	CHANNEL_TO_SPACE _LARGE_FACTOR_CO MPOSITE	0x59	89	PEFUSED_ELEMENTW ISE
0x1b	27	LIVE_IN	0x5a	90	PEFUSED_SECUREFL USH
0x1c	28	LIVEIN_PARAM	0x5b	91	PEFUSED_POOL
0x1d	29	CONST_IN	0x5c	92	PEFUSED_GOC
0x1e	30	LIVE_STATE	0x5d	93	NEFUSED_CONV
0x1f	31	LIVE_OUT	0x5e	94	NEFUSED_KERNEL_R ASTERIZER
0x20	32	REDUCTION	0x5f	95	NEFUSED_CROSS_CO RRELATION
0x21	33	ALIAS	0x60	96	NEFUSED_MATMUL
0x22	34	REINTERPRET_INNE RMOST_DIMENSION	0x61	97	NEFUSED_POOL
0x23	35	REINTERPRET_CAST	0x62	98	NEFUSED_EW
0x24	36	RESHAPE	0x63	99	NEFUSED_DUAL_SOU RCE_EW
0x25	37	VIEW	0x64	100	NEFUSED_BYPASS
0x26	38	TRANSPOSE	0x65	101	NEFUSED_RCAS
0x27	39	SPACE_TO_BATCH	0x66	102	TRANSPOSE_ENGINE _OP
0x28	40	BATCH_TO_SPACE	0x67	103	TE_RESAMPLE
0x29	41	SPACE_TO_CHANNEL	0x68	104	TE_AFFINE_TRANSF ORM
0x2a	42	CHANNEL_TO_SPACE	0x69	105	TE_PAD
0x2b	43	SOFTMAX	0x6a	106	TE_CROP_RESIZE
0x2c	44	INSTANCE_NORM	0x6b	107	TE_SLICE
0x2d	45	L2_NORM	0x6c	108	TE_GATHER
0x2e	46	MINMAX_NORM	0x6d	109	TE_RESIZE
0x2f	47	LAYER_NORM	0x6e	110	TM_WAIT_FOR_EVEN T

op	dec	string	op	dec	string
0x30	48	LRN	0x6f	111	TM_SIGNAL_EVENT
0x31	49	COST_VOLUME	0x70	112	TM_BRANCH
0x32	50	PIXEL_SHUFFLE	0x71	113	TM_FETCH
0x33	51	PIXEL_UNSHUFFLE	0x72	114	TM_STORE
0x34	52	MATRIX_DECOMPOSITION	0x73	115	TM_OPERATE
0x35	53	FPS	0x74	116	TM_USER_SLOT_LOAD
0x36	54	RS	0x75	117	DMA_CONVERT
0x37	55	RESAMPLE	0x76	118	QUANT
0x38	56	GATHER	0x77	119	DEQUANT
0x39	57	TILE	0x78	120	SNE_COND
0x3a	58	SLICE	0x79	121	SNE_GOC
0x3b	59	PAD	0x7a	122	CCDMA_CONST
0x3c	60	RESIZE	0x7b	123	CCDMA_MEMORY
0x3d	61	RESIZEAS	0x7c	124	SPILL_FILL_DUMMY
0x3e	62	CROP_RESIZE	0x7d	125	INVALID

The opcode 0x3f is spelled AFFINE_TRANFORM in the binary, a vendor source typo preserved on the wire.

C.2 Operation-attribute schema and IOKit external-method struct layouts

The attribute schema is string-keyed: the token is the wire encoding the compiler matches on, and most integer constants are not recoverable statically. The converter recognizes 171 literal attribute keys, of which roughly 140 are op-facing. Table C.8 gives representative keys with their value types, meanings, and wire encodings.

Table C.8. Representative operation-attribute keys with their value types, meanings, and wire encodings.

key	type	meaning	value encoding
activation	enum	neuron mode	token into the 22-field PWL descriptor
strides	int[]	per-axis stride	NDCHW int array; deconv restricted to {1,2}
groups	int	group count	channel-wise requires groups == out.C
padding_mode	enum	fill rule	constant / reflect / replicate / symmetric
weights_layout	enum	weight axis order	NCHW / NHWC / OIHW / HWIO; weight buffer is MACI
compressed	bool/enum	weight compression	format set by the MIL-op-count contract
interleave	int	channel-tiling quantum	one of {1,2,3,4,8}
epsilon	float	norm stability	scalar

The host-to-kernel IOKit dispatch key is the (selector, struct-size) tuple, not the selector alone. Table C.9 gives the control-client selectors with their method names and decoded input and output struct sizes.

Table C.9. The IOKit control-client selectors with their method names and decoded input and output struct sizes.

sel	method	in-struct	out/scalar
0	`ANE_DeviceOpen`	104	104
2	`ANE_ProgramSendRequest`	2376 + 1 scalar	40 async
3	`ANE_ProgramCreate`	32	0
4	`ANE_ProgramPrepare`	56	56
6	`ANE_ProgramDestroy`	16	0
7	`ANE_GetStatus`	0	32
8	`ANE_ProgramCreateInstance`	32	0
10	`ANE_GetVersion`	0	1 scalar
21	`ANE_ProgramInputsReady`	3104	0
22	`ANE_MemoryMapRequest`	2080 + 1 scalar	1 scalar

The `ANEDeviceOpen` shared in/out buffer (104 bytes, selector 0) decodes by byte offset as table C.10.

Table C.10. The field layout of the ANEDeviceOpen shared input and output buffer by byte offset.

offset	field
`+0x00`	usage type (1 standard, 2 unsupported) plus session token
`+0x08`	callback function pointer
`+0x10`	receiver context pointer
`+0x18`	timeout `0x2710` (10000)
`+0x48`	version pair `32, 256`
`+0x50`	NumANEs `0, 1`

The full 171-key attribute corpus, the decoded enum-value tables, and the per-selector field layouts for the HW direct-path client are in the research corpus.

C.3 Numeric error, status, and return codes

The ANE stack has no flat numeric status enum. The fixed numeric values that exist are the IOKit return constants and the firmware magic and sentinel words, the first of which table C.11 gives with their meanings on the dispatch path.

Table C.11. The IOKit return constants and their meanings on the engine dispatch path.

macro	hex	ANE path meaning
`kIOReturnSuccess`	`0x00000000`	success
`kIOReturnError`	`0xe0000001`	general failure
`kIOReturnBusy`	`0xe0000007`	gate or command busy
`kIOReturnNoMemory`	`0xe00002bd`	allocation failure
`kIOReturnNoResources`	`0xe00002be`	out of resources, queue or slot exhaustion
`kIOReturnNotPrivileged`	`0xe00002c1`	privilege check failed
`kIOReturnBadArgument`	`0xe00002c2`	typed-args validation failure
`kIOReturnUnsupported`	`0xe00002c7`	disabled or stub path; also what a gated feature returns when its entitlement is absent
`kIOReturnNotReady`	`0xe00002d0`	device or channel not ready
`kIOReturnAborted`	`0xe00002eb`	request aborted

macro	hex	ANE path meaning
kIOReturnNotFound	0xe00002f0	program or process handle not found
kIOReturnTimeout	0xe0000404	firmware op timed out

At every layer the error surface is name-based or message-based. The client-visible surface above IOKit is an error factory that wraps the lower-layer code into a structured error across four domains, named errorDomainCompiler, errorDomainEspresso, errorDomainGeneric, and errorDomainVirtIO. Its factory methods are the taxonomy: a generic wrapper, a missing-code-signing form, program-load and new-instance-load forms that hold the lower-layer code, surface map and unmap forms, and a virtualization-kernel form. The single most look-up-worthy client-visible value is 0xe00002c7 (kIOReturnUnsupported), returned on a disabled or unsupported path and when a gated feature's entitlement is absent.

Table C.12 gives the fixed firmware magic words and sentinel constants with their meanings.

Table C.12. The fixed firmware magic words and sentinel constants with their meanings.

constant	hex	meaning
package magic	0x414E4548 (ANEH)	loader package header
program magic	0x414E4550 (ANEP)	loader program header
section magic	0x414E4553 (ANES)	loader section header
AFPP control magic	0x55AA55AA	AFPP control struct
checksum-valid sentinel	0xFFFFFFFF	command checksum initialized and valid
invalid id	0xFFFFFFFF	ECSneCmdId_Invalid, unbound program or process
padding	0x00000000	command padding must be zero
power-status byte	0xFF / 0x00	fully on / fully off

This section shows the three loader magic words as 32-bit integers; on disk the bytes are little-endian, so a raw byte scan finds HENA, PENA, and SENA (the characters of ANEH, ANEP, ANES reversed). The firmware-to-host notification names, inline status=0x%x print sites, AArch64 and L2C fault-register dump fields, and compiler diagnostic categories are in the research corpus.

C.4 The tunable register-init table

The per-chip register init is a sequence of 12-byte (offset, mask, value) records, each applied as a masked read-modify-write, reg = (reg & ~mask) | value, where reg is the block MMIO base plus the offset. The M1 (ASC AscChinook) firmware has 1994 records across 10 named MMIO blocks, each block reached through a 32-byte descriptor of name, MMIO base, record pointer, and count, the blocks and their counts given in table C.13.

Table C.13. The named MMIO register-init blocks with their bases and record counts.

block name	MMIO base	records
ASC_CHINOOK	0x2_6b00_0000	24
ASCWRAP	0x2_6b40_0000	2
sneCtrl	0x2_6b84_0000	15
ANE	0x2_6bc0_0000	47
aneDpePpt	0x2_6b8e_c000	304
aneDpePptAccp0	0x2_6b8e_d000	528
aneDpePptAccp1	0x2_6b8e_e000	528

block name	MMIO base	records
aneDpePptAccp2	0x2_6b8e_f000	528
aneDpeSys	0x2_6b8f_0000	9
aneDpePpt_soc_dpe_lee	0x2_6b8f_4000	9

Table C.14 gives representative records, one or more per block, with their address, mask, value, and meaning.

Table C.14. Representative register-init records with their address, mask, value, and meaning.

regAddr	mask	value	meaning
0x2_6b14_0020	0xf80000	0x780000	ASC clock or PLL divider field set to 15
0x2_6b40_080c	0x6000_0001	0x6000_0001	fabric clock and QoS enable
0x2_6b84_0028 .. 0044	0x8fff_c000	0x8fff_c000	8 identical SNE QoS and credit words, one per set
0x2_6bc0_d014 .. fec4	0x1	0x1	32 per-tile MAC clock and power enables
0x2_6bc1_400c	0xffff_ff00	0x4010_1000	DMA descriptor base and config word
0x2_6b8e_c000	0xffff	0x267e	peak-power-tracking base budget word
0x2_6b8e_c42c	0x3fff	0x0	DPE trailing-control reg, armed live to 0x3fff
0x2_6b8e_d000	0xffff_ffff	0x0077_3594	per-counter energy scale coefficient (7811476)
0x2_6b8f_0014	0xffff_ffff	0x0000_23e1	DPE config and period word
0x2_6b8f_0038	0xffff_ffff	0x0003_2dcc	DPE accumulation window and divisor (207820)
0x2_6b8f_4000	0x1e	0xc	SoC-level LEE control field

The 32 `mask=1 value=1` records at a regular stride are direct evidence of the 32-tile MAC array geometry, each tile individually clock and power gateable. The DPE system block has seven ascending sampling thresholds (25, 50, 70, 85, 95, 105, 115), and the SoC leakage-estimation block has eight (10, 22, 39, 64, 89, 121, 164, 189): the firmware-side breakpoints of the power model. All 1994 decoded records are in the research corpus.

C.5 The CSNE_CMD_* numeric command table

The host-to-firmware command set is 93 entries, numbered 0x00 through 0x5c, indexed by eCSneCmdId into the firmware command-name string table, whose index is the numeric command identifier. 0xFFFFFFFF is the no-command sentinel. Table C.15 gives the full set; its dir column is H->FW for a host request and FW->H for a firmware notification, and the subsystem codes are lifecycle, power, secure, program, execution, cache, ipc, buffer, property, and stats.

Table C.15. The complete 93-entry host-to-firmware command table with name, direction, subsystem, and purpose.

id	name	dir	subsystem	purpose
0x00	STOP	H->FW	lifecycle	stop the controller
0x01	RESET	H->FW	lifecycle	reset controller state
0x02	CONFIG_GET	H->FW	property	read config blob
0x03	PRINT_ENABLE	H->FW	stats	enable firmware print

id	name	dir	subsystem	purpose
0x04	REG_FILE_LOAD	H->FW	lifecycle	load a register file
0x05	BUILDINFO	H->FW	lifecycle	return firmware build string
0x06	TIMEPROFILE_START	H->FW	stats	begin time-profiling
0x07	TIMEPROFILE_STOP	H->FW	stats	stop time-profiling
0x08	TIMEPROFILE_SHOW	H->FW	stats	dump profile
0x09	FW_RUN_MODE	H->FW	lifecycle	select firmware run mode
0x0a	POWER_DOWN	H->FW	power	full power-down
0x0b	SET_SNE_PMU_BASE	H->FW	power	set PMU MMIO base
0x0c	SET_SNE_RPC_CHECK_CMD	H->FW	property	RPC sanity-check command
0x0d	RPC_ENABLE	H->FW	property	enable the back-channel RPC channel
0x0e	PLATFORM_INFO	H->FW	lifecycle	platform descriptor
0x0f	BOOT	H->FW	lifecycle	bring firmware to booted state
0x10	PING	H->FW	lifecycle	liveness probe
0x11	CONFIG_GET_EXT	H->FW	property	extended config read
0x12	POWER_DEVICE_ON	H->FW	power	power the device on
0x13	POWER_DEVICE_OFF	H->FW	power	power device off
0x14	IPC_ENDPOINT_SET	H->FW	ipc	bind an IPC endpoint
0x15	IPC_ENDPOINT_UNSET	H->FW	ipc	unbind endpoint
0x16	CH_INFO_GET	H->FW	buffer	channel info query
0x17	CH_BUFFER_RECYCLE_MODE_SET	H->FW	buffer	set buffer-recycle mode
0x18	CH_BUFFER_RECYCLE_START	H->FW	buffer	start recycling
0x19	CH_BUFFER_RECYCLE_STOP	H->FW	buffer	stop recycling
0x1a	CH_BUFFER_RETURN	H->FW	buffer	return one pooled buffer
0x1b	CH_BUFFER_POOL_CONFIG_GET	H->FW	buffer	read buffer-pool config
0x1c	CH_BUFFER_POOL_CONFIG_SET	H->FW	buffer	configure buffer-pool
0x1d	CH_DATA_FILE_LOAD	H->FW	buffer	stream a data file over channel
0x1e	CH_PROPERTY_WRITE	H->FW	property	write a register or property
0x1f	CH_PROPERTY_READ	H->FW	property	read a register or property
0x20	TRACE_ENABLE	H->FW	stats	enable tracing
0x21	RESOURCE_INFO_GET	H->FW	lifecycle	query engine resources
0x22	STATS_BUFFER_SIZE_GET	H->FW	stats	compute required stats-buffer size
0x23	SUSPEND	H->FW	lifecycle	suspend engine
0x24	DSID_SET	H->FW	cache	set data-set identifiers for prefetch
0x25	MCACHE_SIZE_GET	H->FW	cache	query memory-cache size
0x26	SECURE_MODE_START	H->FW	secure	enter secure mode
0x27	SECURE_MODE_STOP	H->FW	secure	leave secure mode
0x28	SET_SNE_PMU_BASE2	H->FW	power	version 2 PMU base set

id	name	dir	subsystem	purpose
0x29	IPC_ENDPOINT_SET2	H->FW	ipc	version 2 endpoint bind
0x2a	IPC_ENDPOINT_UNSET2	H->FW	ipc	version 2 unbind
0x2b	CH_DATA_FILE_LOAD2	H->FW	buffer	version 2 data-file load
0x2c	SET_DYNAMIC_POWERGATE	H->FW	power	configure dynamic clock and power gating
0x2d	ANE_DEFAULT_SETTING_SET	H->FW	lifecycle	bulk default-settings push
0x2e	INIT_SHARED_EVENT_INFO	H->FW	ipc	initialize shared-event table
0x2f	EXCLAVE_MODE_START	H->FW	secure	enter exclave mode (stubbed on H13)
0x30	EXCLAVE_MODE_STOP	H->FW	secure	leave exclave mode
0x31	QUIESCE_STATE	H->FW	lifecycle	drain in-flight work
0x32	CPU_LOAD_GET	H->FW	stats	sample CPU load
0x33	SECURE_MODE_RESUME_TRANSITION	H->FW	secure	resume a paused secure transition
0x34	CH_ERROR_NOTIFICATION	FW->H	stats	error notification
0x35	CH_POWER_CONTROL	H->FW	power	channel-level power control
0x36	CH_SIGNPOST_NOTIFICATION	FW->H	stats	32-bit signpost notification
0x37	CH_SIGNPOST_NOTIFICATION_GROUP	FW->H	stats	grouped 32-bit signpost
0x38	CH_RESET_NOTIFICATION	FW->H	stats	reset notification
0x39	CH_SIGNPOST64_NOTIFICATION	FW->H	stats	64-bit signpost
0x3a	CH_SIGNPOST64_NOTIFICATION_GROUP	FW->H	stats	grouped 64-bit signpost
0x3b	CPU_LOAD_NOTIFICATION	FW->H	stats	CPU-load notification
0x3c	TM_SYNC_ERR_NOTIFICATION	FW->H	stats	tile-manager sync error
0x3d	LOAD_PROGRAM	H->FW	program	load a compiled program into a slot
0x3e	UNLOAD_PROGRAM	H->FW	program	unload a program
0x3f	CREATE_PROCESS	H->FW	program	instantiate a process for a program
0x40	TERMINATE_PROCESS	H->FW	program	tear down a process
0x41	PROCEDURE_CALL	H->FW	execution	baseline network invocation
0x42	LOAD_AFPP	H->FW	program	load AFPP prefetch program
0x43	UNLOAD_AFPP	H->FW	program	unload AFPP
0x44	PROGRAM_INTERFACE_VERSION_CHECK	H->FW	program	negotiate program-interface version
0x45	PROCEDURE_CALL_CACHE_REQUEST	H->FW	cache	install a resident cache request
0x46	PROCEDURE_CALL_TRIGGER_CACHE_REQUEST	H->FW	cache	fire an installed cache request
0x47	PROCEDURE_CALL_RECYCLE_OUTPUT_BUFFER	H->FW	cache	return a consumed output buffer
0x48	PROCEDURE_CALL_INVALIDATE_CACHE_REQUEST	H->FW	cache	destroy a cache request

id	name	dir	subsystem	purpose
0x49	PROCEDURE_CALL_WITH_CUSTOM_BARS	H->FW	execution	proc-call with custom barrier array
0x4a	PREMAP_BUFFER	H->FW	cache	pre-map an inference-property buffer
0x4b	PROCEDURE_CALL_CACHE_REQUEST_WITH_CUSTOM_BARS	H->FW	cache	cache request with custom bars
0x4c	PROCEDURE_CALL_CACHE_REQUEST_WITH_SHARED_EVENTS	H->FW	cache	cache request with shared events
0x4d	FORCE_DISABLE_CACHE_REQUESTS	H->FW	cache	global cache-request disable
0x4e	PROCEDURE_CALL_WITH_SIGNAL_EVENTS	H->FW	execution	proc-call with wait and signal events
0x4f	SET_ACTIVE_CACHE_REQUEST_IN_GROUP	H->FW	cache	select active member of a cache-request group
0x50	PROGRAM_EVENT	FW->H	program	per-program event notification
0x51	USER_EVENT	FW->H	stats	user event marker
0x52	DBG_EVENT	FW->H	stats	debug event
0x53	DATA_CHAINING_EVENT	FW->H	cache	data-chaining stage completion
0x54	PREFETCH_DSID_EVENT	FW->H	cache	prefetch completion
0x55	SECURE_MODE_EVENT	FW->H	secure	secure-mode state-change event
0x56	REQUEST_PROGRAM_ID	H->FW	program	allocate a program-id slot
0x57	RETURN_PROGRAM_ID	H->FW	program	free a program-id slot
0x58	REQUEST_PROCESS_ID	H->FW	program	allocate a process-id
0x59	RETURN_PROCESS_ID	H->FW	program	free a process-id
0x5a	INFERENCE_CALL	H->FW	execution	high-level inference submission
0x5b	BACK_CHANNEL_RPC	FW<->H	property	firmware-initiated back-channel RPC
0x5c	DEBUG_COMMAND_DATA_CHECK	H->FW	stats	validate command-data integrity

Two strings the prior corpus counted have no numeric identifier: `CSNE_CMD_START` is a standalone lifecycle log alias, and `CSNE_CMD_IPC_ENDPOINT_TYPE_DATA_CHAINING` is an endpoint-type enum value rather than a command. The per-call numeric limits the command bodies enforce on the M1 are fixed. The dispatch caps are at most 16 signal events per call, at most 32 custom barriers on the wire (128 in the program container), at most 128 custom execute-order entries, at most 16 trigger input buffers, and fewer than 2 active shared events. Priority levels run 0 through 7, split into a privileged band of 0 and 1 and a normal band of 2 through 7. The single dispatch path takes exactly one output buffer set, one task-descriptor partition, and one engine request per list.

Table C.16 gives the fixed-header layout (`sCSneControllerCmdHdr`) that prefixes every ring message, with each field's offset, width, and meaning.

Table C.16. The fixed command-header fields with their offsets, widths, and meanings.

field	offset	width	meaning
`id`	`0x00`	`u32`	the `eCSneCmdId` selector
`size`	`0x04`	`u32`	byte length of the command body
`priority`	`0x08`	`u32`	scheduling band, 0..7 (0..1 realtime, 2..7 normal)
`programId`	`0x0c`	`i32`	loaded-program slot, `-1` invalid
`processId`	`0x10`	`i32`	per-program process instance, `-1` none
`procedureId`	`0x14`	`u32`	index into the program's procedure table

The full 93-entry table with file offsets, the decoded request structs, and the per-call numeric limits are in the research corpus.

C.6 The task-descriptor hardware register map

A captured `ane_reg` record is a `(regAddr, regValue)` pair whose low address selects one of 7 aperture groups. Table C.17 is the aperture map that converts a raw address to a group, with the image base and window of each.

Table C.17. The register-address ranges and the aperture group, image base, and window each maps to.

regAddr range	group	image base	window
`< 0x4c`	G1 dimensions	`0xf4`	19 words
`0x4100 .. 0x4177`	G3 elementwise / planar / pad	`0x264`	30 words
`0x4500 .. 0x4537`	G4 L2 / texture	`0x2e4`	14 words
`0x4900 .. 0x492b`	G5 kernel-fmt / op-mode	`0x324`	11 words
`0x4d00 .. 0x4e13`	G2 tile DMA	`0x148`	69 words
`0x5100 .. 0x5153`	G6 L2-result	`0x358`	21 words
`0x5500 .. 0x5587`	G0 kernel / common	`0x24`	34 words

Table C.18 gives representative G1 dimension fields, which also pack the format and control bits, with the bit range and width of each.

Table C.18. Representative G1 dimension register fields with their bit ranges, widths, and meanings.

regAddr	field	bits	width	meaning
`0x00`	Win	[14:0]	15	input tile width
`0x02`	Hin	[14:0]	15	input tile height
`0x0c`	Cin	[16:0]	17	input channels, max 131071
`0x10`	Cout	[16:0]	17	output channels
`0x10`	CommonInFmt	[1:0]	2	source-1 element format
`0x10`	CommonOutFmt	[5:4]	2	output element format
`0x28`	numGroups	[12:0]	13	convolution groups
`0x38`	CommonTaskType	[7:4]	4	hardware task class (9 valid)

To invert a raw value: DMA strides are 26-bit signed at bits [31:6]; L2-result base and strides are 17-bit at bits [20:4]; a full device address is `(hi << 32) | lo` with `lo` 64-byte aligned and `hi` 10 bits, capped at 42 bits. The complete inventory of roughly 190 register fields across the 7 groups, 11 reloc slots, and on-M1 stubbed engines (CCDMA, atomic scatter, LDTID) is in the research corpus.

Each operation descriptor in the task-descriptor stream has a 32-bit opcode word. Table C.19 gives the words decoded from a live M1 program for three operations.

Table C.19. The version-7 / H13 codegen opcode words for three operations, with the high half-word shared and the low 16 bits selecting the operation.

operation	opcode word
convolution	`0x5042a063`
reduce-mean	`0x5000a021`
matrix multiply	`0x5000b021`

C.7 The `.e5` FlatBuffer schema

The program container is a FlatBuffer whose root table holds four fields, the schema given as listing C.1. The schema reconstructs from the serializer method set and the wire bytes, since the binary strips the reflection schema, and round-trips cleanly through the FlatBuffers tool to 23 tables, 4 enums, and 1 union. The data-type and op-type enums are recovered by name; the numeric ordinals are inferred.

Listing C.1. The reconstructed program-container FlatBuffer schema: root table, type enums, tensor descriptor, section tables, and operation structure.

```
namespace E5RT.fb;

enum TensorDataType : int {
  Invalid = 0, Float16 = 1, Float32 = 2, Int8 = 3, UInt8 = 4,
  Int16 = 5, Int32 = 6, Int4 = 7, Bool = 8, E4M3 = 9, E5M2 = 10
}

enum OpType : int {
  Cast = 0, AneInference = 1, EirInference = 2, CpuInference = 3,
  BnnsCpuInference = 4, MlcCpuInference = 5, MpsGraphInference = 6,
  E5MinimalCpu = 7, Quant = 8, Dequant = 9, Barrier = 10, JitCall = 11
}

table TensorDescriptor {
  dim:[ulong];
  stride:[ulong];
  width:ulong;
  height:ulong;
  channels:ulong;
  batch_number:ulong;
  sequence_length:ulong;
  stride_width:ulong;
  stride_height:ulong;
  stride_channels:ulong;
  stride_batch_number:ulong;
  stride_sequence_length:ulong;
  storage_type:TensorDataType;
  component_pack:int;
}

table BuildInfoEntry { key:string; value:string; }
table BuildInfo { entries:[BuildInfoEntry]; }   // 7 key-value pairs in the sample
```

```
table AliasSymbol { name:string; symbol_index:uint; addr_offset:uint; }

table IOPort { name:string; byte_size:ulong; aperture_va:ulong; }

table Operand { descriptor:TensorDescriptor; }

table CastAttrs { src_dtype:TensorDataType; dst_dtype:TensorDataType; component_pack:int; }

table AneInferenceAttrs {
  procedure_name:string;
  anehash:string;
  program_symbol:string;
  intermediate_buffer_handle:uint;
  compiler_options:string;
}

table Operation {
  name:string;
  op_type:OpType;
  inputs:[uint];
  outputs:[uint];
  arg_frame:string;        // __arg_frame section reference
  attrs_section:string;    // __op_attrs section reference
}

table Block { name:string; operations:[Operation]; }

table Function { name:string; anehash_path:string; blocks:[Block]; }

table Section { name:string; kind:int; }

table E5Program {
  symbol_names:[string];   // field[0] name vector
  build_info:BuildInfo;    // field[1] 7-entry sub-table
  sections:[Section];      // field[2] 6-entry section vector
  format_version:int;      // field[3] inline scalar == 4
}

root_type E5Program;
```

The whole fused graph collapses to a single `AneInference` operation, with the surrounding `Cast` operations holding the input and output dtype conversion. The validation against the round-9 `H13C.e5` sample reads the four root fields, the seven build-info pairs (`built-for-profiling`, `input-file-path`, the component versions, and `on-device-compilation`), and the operation chain `Cast`, `AneInference`, `Cast` straight out of the bytes with no contradiction. The enum ordinals, the `e4m3` and `e5m2` dtypes, segment-chaining fields, and field sets of the ten op-attribute tables other than `CastAttrs` and `AneInferenceAttrs` are inferred rather than byte-confirmed in this single-segment sample.

Appendix D Glossary

SUMMARY

This appendix defines the acronyms, proper nouns, and symbols the guide uses across all nine parts. Read the four core facts first, then the term table, then the family and silicon map.

The term table D.1 is the reference; the notes after it record the core facts that the rest of the guide depends on.

D.1 Core facts

Read these four corrections before the table; several entries below depend on them. The M5 base part is H17, specifically the `h17s` compiler target, not H16s; H16s and H17s are separate targets distinguished only by the generation-tag byte. The multiply-accumulate datapath uses a single wide accumulator of the fp32 class, supplied by radix-4 fp16-rounded input tiles; the accumulator width is fixed hardware on every device and is never a per-chip parameter, so it is not a recursive fp16 reduction tree. On M1/H13 two weight-compression forms stream natively, the int4 palette (int4-LUT) and the sparse form whose mask and values have at least 50 percent zeros; int8 and blockwise-affine weights fold into the descriptor on that generation and only stream natively from later families. The firmware task-queue notification identifier (NID) is 8-bit, taking values 1 through 255.

D.2 Terms

Table D.1. The acronyms, proper nouns, and symbols used across the guide with their definitions.

Term	Definition
A11 through A18, M1 through M5	Apple system-on-chip marketing names, each with an ANE of a specific H-generation, related by $M(n) = H(n + 12)$.
AFPP	The on-device firmware program container: a three-level big-endian FourCC package, `ANEH` then `ANEP` then sections.
`aned`	The system ANE broker daemon at `/usr/libexec/aned` that holds the IOKit access gate, so every unentitled client reaches the engine by sending it a request.
`aneuserd`	The per-user sibling of `aned`, the other holder of the IOKit access gate.
ANEC, `anec.*`	The compiler backend intermediate-language dialect of 97 operations, the target of front-end lowering and the input to task-descriptor codegen.
ANECCompile	The backend compile entry point that direct netplist authoring supplies rather than bypasses.
ANECompiler	The single compiler binary that lowers the front IR to the backend IR and then to task descriptors for every target, so one host can construct any of the 28 targets' hardware-abstraction blobs.
ANECompilerService	The out-of-process compile service; repeated failed compiles in quick succession can stall it, so pace compiles after a failure by about 15 seconds, covered in Part V.

Term	Definition
ANEServices	The user-space framework layer beneath the runtime that marshals requests into IOKit calls.
AppleH11ANEInterface	The kernel driver for the engine, decompiled at version 9.511.3, that holds the IOKit class hierarchy and the user client.
ASC_CHINOOK	The firmware's internal chip codename string for the H13 ANE coprocessor.
bridge ops	Hidden backend layer kinds reached by direct netplist authoring, such as fused attention, fused rank, and fused rearrange, each paired with one frontend bridge module, described in Part II and cataloged in Appendix B.
CCDMA	The cross-chip and cross-engine DMA and event-sync engine; on M1/H13 it is folded, and it is present natively on A15 and later and on M5, where it enables resident state.
CSNE, `CSNE_CMD_*`	The host-to-firmware command protocol, where `CSNE_CMD_*` are the numeric command opcodes the host mailbox issues to the firmware.
DART	The ANE's IOMMU, a 16 KB-page, 3.5 GiB-window unit that maps host physical RAM into the engine's device address space.
DPE	Dynamic Power Estimation: a firmware activity-counter power estimate calibrated by 10 device-tree coefficients and bounded by the peak-power ceiling.
DVA	Device Virtual Address: the address the engine issues, resolved by DART to physical RAM, used interchangeably with IOVA.
dispatch floor	The fixed per-dispatch latency, about 0.23 ms on the M1 anchor and a fitted 0.11 ms on the M5, below which a kernel cannot run regardless of its size, covered in Part III.
`.e5`	The compiled-program dispatch-layer container whose size tracks the segment and dispatch count rather than the operation count.
e5rt	The E5 runtime, the C API that the frontend uses to load and stream programs and the unentitled reachable surface, which still relays to `aned` underneath.
EIR, NitroIR	The runtime's lowered IR, a Lisp-style S-expression node tree serialized on disk, whose pivot type is the fp16 `ndarray<half>`.
Espresso	Apple's cross-backend neural-network runtime and scheduler that hosts the E5 execution engine and the cost-model placement segmenter.
ExeLoop	The firmware's main control execution loop and finite-state machine that fetches and dispatches task descriptors through the RUN, IDLE, and EXEC states.
fp16	Half-precision IEEE float, the engine's native compute and storage type, with a maximum finite magnitude of 65504.
fp16 accumulator	A shorthand tag for the MAC numeric behavior: a single wide accumulator of the fp32 class supplied by radix-4 fp16-rounded input tiles, where the only quantization is fp16 input and partial rounding and the fp16 output grid.
fp8 (E4M3, E5M2)	The 8-bit float weight and activation datapath, present only on H18, which decodes to fp16 before the MAC.
generation-tag	The hardware-abstraction blob's generation byte at offset `0x0`, holding the hex H-number, the decisive discriminator between near-identical targets such as H16s and H17s.
GOC, DynamicGOC	Generate-Output-Channels, the dynamic unit that generates the output-channel-group kernel tiles from a runtime weight, present from M1 onward.
HAL	The Hardware Abstraction Layer: the per-target scalar and byte blob that data-drives nearly all per-family behavior and is the source of truth for capability, limit, and cost, detailed in Part IX.
`.hwx`	The fully lowered hardware-executable container, the counterpart to the `.e5`.

Term	Definition
int4-LUT, palette weights	Palettized 4-bit weight compression, one of the two formats that stream natively on M1/H13 (alongside the sparse form) at about 2.37 times, where int8 and blockwise-affine instead fold into the descriptor.
IOVA	IO Virtual Address: the device virtual address DART produces from host physical RAM, on a 16 KB page over a 3.5 GiB window.
KMEM	The on-chip working and scratch buffer the task descriptor sizes for weights and tiles, gated at 64 KB at legalization and the basis of the working-set threshold.
KV-cache (resident)	An on-device key and value cache that stays resident across dispatches for decode, built on M1 with `share_buffer` rather than native state, covered in Part VIII.
LUT	Lookup table, used both for piecewise-linear activation approximation and for the palette of int4 weight compression.
MAC	Multiply-accumulate, the compute primitive whose datapath is an fp16 multiply, radix-4 fp16-rounded input tiles, and one wide accumulator of the fp32 class.
MIL	The Model Intermediate Language, the front IR in single-assignment form that the compiler segments and lowers to the backend IR.
MLComputePlan	The model-framework introspection surface that reports per-operation device assignment and a cost weight, the readable view of the placement segmenter.
`.mlmodelc`	The compiled model bundle that pairs the runtime net, shapes, weights, and the `.hwx`.
multi-die, AllReduce, AllGather	The multi-die collective-communication layer present on the multi-die H14 through H18 Max and Ultra-class dies, not a base-class feature.
NE core	A neural-engine compute core; the per-family count decodes from the HAL as base 4, then 8, 16, 32, or 64 by suffix, described in Part IX.
NID	The firmware task-queue notification identifier owned by the state machine, 8-bit and taking values 1 through 255.
OCG	The Output-Channel Group, the compiler's output-channel tiling unit sized to the accumulator file, where a larger group means fewer DMA re-bases.
OC/cycle	The per-cycle output-channel throughput of the MAC array, the roofline unit in the cost model.
Path-A	Direct netplist authoring, hand-writing the backend netplist to reach hidden layer kinds, which supplies `ANECCompile` and so cannot reach a lowering the backend rejects, covered in Part II.
palettization	Weight compression that maps each weight to a lookup-table index, the int4 form of which streams natively on M1/H13.
power domains	The five independently gated power domains of the H13 ANE, by which the engine modulates power through the number it energizes.
PPL	Page Protection Layer: the kernel page-table protection layer through which the ANE's DART leaf writes go.
PPT	The peak-power ceiling and throttle under which the Dynamic Power Estimation values are bounded.
`pushTDList`	The firmware function that hands a task descriptor to the hardware and re-enters with an already-built descriptor for resident chains.
roofline	The performance bound that takes the smaller of compute-limited and bandwidth-limited rates as a function of arithmetic intensity, the basis of the cost model in Part III.
RTBuddy	Apple's coprocessor real-time-OS runtime framework, the substrate the ANE firmware app runs on.

Term	Definition
RTKit	The real-time-OS substrate beneath the ANE firmware, providing the task and thread model and the synchronization primitives.
`share_buffer`	The runtime primitive that aliases an output buffer to an input buffer after compile, giving a zero-copy resident cache without native state.
slice ×16 saturation	An H13 codegen defect in which a slice with a nonzero last-axis begin lowers to a scaled kernel that silently sends values above 4094, which is 65504/16, to infinity; H13-only and clean on H17.
SoC T-number	The SoC part number, such as T8103 for the M1 base and T8142 for the M5 base, which maps to an ANE H-generation through the board-type sequence.
sparse-binary, SparseFmt	The sparse-weight compute format, where the weight has a binary sparsity mask; the mask-and-values form streams natively on M1/H13, while the packed sparse-binary palette-index form is absent from the M1 version-5 descriptor and present from A15 and M5.
SPTM	Secure Page Table Monitor: the kernel monitor that, with the Page Protection Layer, governs page-table edits and physical-frame ownership.
stack layers	The top-to-bottom software path from the frontend through the runtime, the framework, `aned`, ANEServices, IOKit, and firmware to silicon, in Part VIII.
styx	The firmware and chip codename for the M1/H13 ANE.
TD, task descriptor	The hardware work unit the firmware loads and the engine executes: a register-image descriptor of DMA sub-blocks and framing, emitted per generation from the compiler's descriptor struct, detailed in Part VII.
TileDMA, KernelDMA	The conv datapath DMA engines: the kernel source streams weight coefficients, the tile source streams input activation tiles, and the tile destination writes outputs.
TM, Tensor-Mover	The firmware tile-manager driver that moves tiles and drives the texture layers.
TQ, task queue	The firmware queue the state machine enqueues a program's task-descriptor partitions into.
wide accumulator	The fp32-class running sum of the MAC, which holds small addends rather than dropping them, so a sum of representable terms stays near-exact, covered in Part III.
Winograd	The Winograd fast-convolution transform the compiler can emit for small kernels, trading multiplies for transforms.
Zin, ZinIr, ZinMir	The compiler's internal class namespaces: the IR-object layer, the mid-IR build layer, and the task-descriptor codegen layer.

D.3 Family and silicon map

The relation $M(n) = H(n + 12)$ is anchored at both ends, with the live M1 reporting `h13g` and the M5 cost-model trees decompiled on the M5 host reporting `H17C` and `H17S`. The compiler-family index drives operation legality, and the per-target HAL drives codegen, limits, and cost. Table D.2 maps each marketing name to its ANE generation, architecture string, compiler family, generation-tag, and core counts; Part IX gives the full table.

Table D.2. The Apple chip marketing names mapped to ANE generation, architecture string, compiler family, generation-tag, and core counts.

Marketing name	ANE H-gen	OS arch string	Compiler family	generation-tag	NE cores by suffix
A13, M1	H13	`h13`, `h13g`	A13	`0x0d`	4, 8 (g)
A14, M2	H14	`h14`, `h14g`, `h14c`	A14	`0x0e`	4, 8, 32
A15, M3	H15	`h15`, `h15g`, `h15c`	A15	`0x0f`	4, 8, 32
A16, M4	H16	`h16`, `h16s`	A16	`0x10`	4, 8, 16, 32
A17, M5	H17	`h17`, `h17s`	A17	`0x11`	4, 8, 16 (M5), 32, 64
A18	H18	`h18`	A18	`0x12`	4

The M5 base is the `h17` runtime arch and the `H17s` compiler target, the 16-core variant, not H16s. The suffixes `g`, `s`, `c`, and `d` decode to NE-core counts of 8, 16, 32, and 64 from a single HAL field. A17 and A18 add no new operation capabilities over A16 and scale only the core count; the A13 to A16 jump was the last capability expansion. The fp8 datapath is H18 only.

Appendix E Provenance

This appendix records the evidentiary basis of every substantive claim in the guide, Part by Part and chapter by chapter. The method is the one given in the Methodology chapter: the engine was reached directly below Core ML, the stack was read by static decompilation of the runtime, compiler, kernel driver, and firmware, and values were taken by live read-only instrumentation and by compile-and-run probing. Two silicon points were measured directly, M1/H13 (Apple M1) as the primary host and M5/H17s (Apple M5) as the second, with an A14-class part (Apple M2, H14) as a middle point where a claim required one. Each claim below is marked one of three ways: measured on a named generation, decompile-derived from a named binary, or predicted from the per-chip tables and not yet confirmed on silicon.

Part I. The Machine

Part I was measured on M1/H13 (Apple M1) unless a chapter notes otherwise.

Chapter 1. What the ANE is

Apple documents the engine only as the `MLComputeUnits` compute-unit selector at developer.apple.com/documentation/coreml/mlcomputeunits, a placement hint with no direct device API and no way to confirm which unit ran a segment; this chapter extends that account by reaching the engine directly below the selector. The roofline figures and the convolution advantage over the GPU (3.8x faster, 9x more energy efficient) are M1/H13 measured. The fp16-product and wide-accumulator result is decompile-derived from the firmware and the compiler and confirmed by the M1/H13 cancellation probe. The Core ML placement planner and the direct Espresso route are decompile-derived.

Chapter 2. Execution model

Apple documents only the load-and-predict surface (`MLModel`, `prediction(from:)`); the autonomous-coprocessor model in this chapter, with its command mailbox, walked segment graph, and resident state across dispatches, has no public counterpart and is reported as reverse-engineered. The compile-once, dispatch-many split and the disk-cached program are decompile-derived and confirmed by M1/H13 dispatch tracing. The mailbox-and-doorbell command channel, the autonomous controller, operand mapping through the address-translation unit, and the walked segment graph with its static-control-flow consequence are decompile-derived from the firmware, kernel-driver, and program-format work. The resident-state mechanism through output-to-input buffer aliasing is M1/H13 measured, observed to persist and update an accumulator and a key-value cache across successive dispatches with no host resubmission.

Chapter 3. Numerics

The fp16 datapath and the type limits are decompile-derived from the firmware and the compiler, with the activation-table coefficients read out of the compiler binary constant section. The wide accumulator and its radix-4 first stage, activation-table behavior and the twenty-three-op accuracy sweep, NaN coercion and the gelu and swish origin biases, round-half-to-even output grid, MAC saturation at exactly 2^{15}, denormal handling, softmax max-subtraction, and bit-deterministic output for a fixed graph and input are all M1/H13 measured. The slice saturation threshold at 4094 is M1/H13 measured and reproduced on A14/H14 (Apple M2), where 4094 stays finite and 4096 overflows; the clean route arrives at A15 and later. The cross-generation accumulator behavior holds family-wide, the M5/H17s difference being a one-unit-in-the-last-place scheduling

and tiling reorder rather than an accumulator-width change. Denormal preservation inside the M5/H17s accumulator, where the M1 flushes to zero inside the multiply-accumulate, is M5/H17s measured and generational. Apple's conversion tooling documents low-precision compression at apple.github.io/coremltools, and the int8 relation here agrees with its affine form $w = s\,(q - z)$; the wide-accumulator and per-product cancellation results have no public counterpart and are reported as reverse-engineered.

Chapter 4. Capability surface

The native classes, type limits, compile-legal envelope, and M1 limits are M1/H13 measured by operation-conformance runs, each native operation compiled and run and each unsupported one rejected on device. The attested-is-not-reachable rule is M1/H13 measured: three-dimensional convolution fails backend lowering on every device mask despite its capability byte, and the top-k, sort, and dynamic-slice validators are callable but code-generation-rejected on the M1. The cumsum result is M1/H13 measured through the curated runtime path, correcting the earlier no-path status. The unsupported-on-every-family set and the per-family unlock points for the texture-engine operations, sin and cos, and the rank and sort bridge are decompile-derived from the operation floors and validators and confirmed on the M1 only at its boundary; the chip that first runs each is predicted from the floor table. Apple documents the convertible operation set at apple.github.io/coremltools and developer.apple.com; this chapter extends and partly corrects it, reporting what compiles and runs on the direct path rather than what the public converter accepts, since some accepted operations, such as three-dimensional convolution, do not lower to the engine.

Part II. Reaching the ANE

Part II was measured on M1/H13 (Apple M1) unless a chapter notes otherwise, with chapter 7 also measured on M2/H14 (Apple M2 Pro) and M5/H17s (Apple M5).

Chapter 5. Software stack

Apple documents the compute-unit selector, model loading, and the placement read-out (`MLComputePlan.load`, `deviceUsage(for:)`, `estimatedCost(of:)`) at developer.apple.com/documentation/coreml, and that read-out agrees with the segmenter decision reported here; the layered runtime beneath it and its internal cost graph are not documented and are reported as reverse-engineered. The layering and the runtime's ownership of compile, library, stream, and descriptors; the shortest-path placement segmenter with its per-operation-by-backend cost graph, learned decision trees, and launch and transfer penalties; the broker model with its single privileged device gate, content-hashed program cache, and time-shared request queue; the 292-export runtime surface and its options dictionary; and the per-inference submit through `IOConnectCallAsyncMethod` selector 2 with the daemon's lifecycle selectors 3 through 6 are decompile-derived from the framework, runtime, and daemon binaries. The reachability of the runtime below the framework, with no placement planner and no entitlement for accepted operations, is decompile-derived and confirmed by M1/H13 dispatch tracing.

Chapter 6. Dispatching without Core ML

Apple documents only the indirect `MLModel` and `prediction(from:)` route; the direct compile, load, bind, and dispatch route here is reported as reverse-engineered and extends that account with a planner-free path that targets the engine on purpose. The five-step workflow and the absence of a placement planner are decompile-derived from the runtime and the format pipeline. The absence of an entitlement requirement for accepted operations is decompile-derived and confirmed M1/H13 measured from ordinary user space, and the single-submission multi-step drive with its performance-neutral result and the resident-state buffer aliasing it rests on are M1/H13 measured.

Chapter 7. Weights and compression

This chapter measures across three endpoints: M1/H13 (Apple M1), A14 on M2/H14 (Apple M2 Pro), and M5/H17s (Apple M5). The int4 and sparse streaming speedups and byte ratios, the int8 matmul latency and

weight-byte halving, and the M1 folding of the int8 and blockwise forms are M1/H13 measured. The A14 int8 and sparse stream and the A14 blockwise fold are M2/H14 measured, and the streaming of all four forms is M5/H17s measured. The weight-reconstruction codecs and the per-format hardware-abstraction-layer streaming gates are decompile-derived from the ANE compiler; the A15 floor at which the blockwise form first streams is predicted from the gate pattern and not yet silicon-confirmed. Apple's conversion tools document palettization, quantization, and pruning at apple.github.io/coremltools as a model-size feature; this finding extends that account, showing the same forms stream on the unentitled engine for a bandwidth gain.

Chapter 8. Entitlement boundary

The four gated features and the layer each gate is at are M1/H13 measured: three-dimensional convolution fails backend lowering on every device mask, a native-state program fails the compile with the counter-and-event engine stubbed in the M1 descriptor, a bf16 input or output fails with an unsupported-dtype rejection and is absent from the eleven-code program-I/O enumeration, and a symbolic dimension parses but fails to lower. The framework cast of bf16 to fp16 and the bucketed fixed-shape handling of flexible shapes are decompile-derived from the framework and system bundles, and the load-time signature boundary is decompile-derived from the kernel driver's corecrypto signature and trustcache vnode-trust checks, with the load rejection code `0xe00002e2` observed on the M1. The arrival of native resident state on a later generation is predicted from the per-family hardware descriptor and not measured on that silicon. Apple documents these capabilities (flexible shapes, the `MLState` type) at the conversion and runtime layer; the finding shows they are not reachable on the unentitled direct path, correcting the impression that a documented capability runs on the engine directly.

Part III. Performance and Fit

Part III was measured on M1/H13 (Apple M1; M1 Max), with a second pass on M5/H17s (Apple M5) and an A14-class middle point on M2/H14 where a cross-device or cross-generation claim required it. Apple publishes no roofline, no per-workload power or efficiency figures, and no cross-processor comparison for the engine, so the figures across this Part are reported as measured rather than against a documented account.

Chapter 9. Roofline

The M1 compute and bandwidth ceilings, the 141 FLOP-per-byte ridge point, 2 MB working-set threshold, 0.23 ms dispatch floor, and conv-throughput scaling are M1/H13 measured from the memory-controller and energy counters and end-to-end timing. The saturating-method figures (the large-matmul compute ceiling, effective peak, wall-clock weight-stream bandwidth matching the compiler's internal 50 GB/s constant, int8 rate, and full-call floor) are M1/H13 measured live in a read-only dtrace session, and are presented alongside the counter-based and slope-based figures with the methodology difference noted. The cross-device ridge points, the standalone and weight-stream bandwidths, and fused-block rate are M5/H17s measured. The fusion-and-floor model is decompile-derived from the analytic cost model and fit to the measured M1 convs within plus or minus 17 percent; cross-chip scaling of the ceilings is predicted from that model, not asserted as a measured M1 fact.

Chapter 10. Power and efficiency

The convolution-stack efficiency figures, the absolute-power comparison on the 4096 matrix multiply, the 2-to-14-times efficiency range, and sustained-load behavior are M1/H13 measured, with package power from hardware instrumentation. The power-utilization model (the zero idle rail, dispatch floor, fp16 and int8 compute-bound draw, regime-dependent points, and operations-per-watt optimum near 0.37 pJ per FLOP) is M1 Max measured with the root power sampler, which exposes only a power reading and a binary on-or-off state, with no frequency or voltage telemetry. The A14-class middle generation is measured on that silicon, and the M5 efficiency figures are M5/H17s measured.

Chapter 11. ANE, GPU, and CPU

The per-class speed and energy verdicts are M1/H13 and M5/H17s measured by a single sixteen-class harness recording minimum latency, idle-subtracted total-package power, and fp16 relative error per class. The saturation peaks, the large-N matrix-multiply falloff, and serving crossovers are M5/H17s (Apple M5 Pro) measured; the per-eval overhead floor and the M1 power gap are M1/H13 measured.

Chapter 12. Across the chip family

The naming rule $M(n) \to H(n+12)$ is decompile-derived from the per-family device tables and confirmed on the measured parts: M1/H13 by the live h13g architecture string, M2/H14g on the live A14 host, and M5/H17s by the resolved target. The family-wide operation limits and the core-and-clock scaling are decompile-derived from the operation floors and the per-target scalar tables. The M5 throughput and working-set threshold, the ten-of-ten cross-silicon prediction pass, the M1-versus-M5 training and inference parity (0.9080 against 0.9070, deterministic across runs), and the one-unit-in-the-last-place fp16 divergence bound are M5/H17s and M1/H13 measured; the M2 campaign measured training accuracy, the four fp16 axes, peak throughput, compression, and the per-op max-dim caps on A14 silicon. The A15 and A16 generations and their M3 and M4 counterparts are decompile-derived from the device tables and not individually measured, the A15/M3 rail being the one generation that remains unmeasured. Apple's product specifications publish a marketing core count per chip, for example a 16-core engine, a different quantity from the four physical compute sets measured on the M1; the H-architecture naming, the mapping, and per-family gates are reported as reverse-engineered. The M5 single-program matmul peak of about 9.5 fp16 TFLOP/s and the weight-stream bandwidth of about 145 GB/s over two DRAM read channels are M5/H17s measured.

Part IV. Workloads

Part IV was measured on M1/H13 (Apple M1 Max) and M5/H17s (Apple M5 Pro), with an A14-class point on M2/H14 where noted, each figure marked for the generation it was taken on.

Chapter 13. Vision, convolution, and encoders

The convolution speed and efficiency, the convolution-stack and roofline figures, the power gap, and per-eval floor are M1/H13 measured; the M5 convolution-stack, ResNet-18, twelve-layer-encoder, and single-sentence-encoder ratios and the serving crossovers are M5/H17s measured. The convolution lowering, Winograd gate, 2 MB working-set constant, per-output-channel fold, and texture-engine operation set with its single A14 family gate are decompile-derived from the ANE compiler and the per-chip parameter table. The Q.4 crop-scale saturation threshold, where $4094 \times 16 = 65504$ reaches the fp16 ceiling, is decompile-derived and confirmed by the fp16 range probe on M1/H13 and on A14 (M2); the clean route arrives on A15 and later. Apple documents vision and image models on the engine only through the compute-unit selector (developer.apple.com/documentation/coreml, developer.apple.com/documentation/vision), with no direct datapath API or cross-processor figures; this chapter extends that account with the measured economics against the GPU and names the texture-engine preprocessing path the public surface does not expose.

Chapter 14. LLM case study

The per-eval dispatch floor, int8-hybrid result, and resident-cache step are M1/H13 measured, the last by a two-proof resident-buffer probe; the batched-decode comparison, the per-projection placement table and position rule, and the speculative-decoding and batched-prefill serving controls are M5/H17s measured. The dispatch and resident-state machinery of the direct path is decompile-derived from the runtime, including the output-to-input buffer aliasing that holds the cache resident, the in-flight cap of 127, and per-process loaded-program cap near 128 (the next load returns `GetANEFModel: must re-compile`). Apple does not publish engine-versus-GPU decode measurements, so the per-batch decode verdict is reported as measured.

Chapter 15. Training on the engine

The gradient audit, differentiable-vocabulary correctness to a cosine of 1.0000, conv weight-gradient saturation threshold, M1 training accuracy and loss-scale curves, and two-generation parity (0.9080 on M1 against 0.9070 on M5 for the identical seeded network) are M1/H13 and M5/H17s measured. The width-axis slice saturation, exact at loss scale 384 and first overflowing at 512 above $65504/16 \approx 4094$, was reproduced on A14 (M2), locating the clean route on A15 and later. The absence of an engine-native backward operation is decompile-derived from the shared compiler, which has gradient operations only in the graphics-processor dialect. Apple documents on-device model update at developer.apple.com/documentation/coreml, a limited fine-tuning surface whose backward pass runs off the engine; this chapter extends that account with a full forward, backward, and optimizer loop running as engine graph operations, optimizer state resident across steps.

Chapter 16. Numerical and scientific computing

The iterative-solver envelope, the size bounds on the unrolled factorizations, the full-spectral-decomposition relative errors, and wide-accumulator behavior are M1/H13 measured, with the fused five-point stencil margin measured on M5/H17s and the DFT-as-matmul throughput on the M2 generation at $N = 1024$. The static-dataflow constraint and the absence of data-dependent control flow are decompile-derived from the operation set and the compiler. The fp16-clean DFT bound near $N = 2048$ is predicted from the wide-accumulator reduction and the fp16 rounding of the matrix entries, an edge of the representable range consistent with the accumulator measurements rather than a single measured cutoff. Apple documents dense linear algebra and signal processing through the accelerate framework at developer.apple.com/documentation/accelerate, all targeting the CPU rather than the engine; this chapter maps which of those kernels fit the engine and which are architecture-limited, a mapping the public documentation does not provide.

Part V. Practice

Part V was measured on M1/H13 (Apple M1), with M5/H17s (Apple M5) as the cross-chip reference where a second generation is needed.

Chapter 17. Model-design rules

The tensor-dimension boundaries, the convolution kernel and stride limits, the arg-min and arg-max 2048 cap, and pooling-window behavior are M1/H13 measured, swept until compile flipped from accept to reject. The per-operation validators, the kernel-format and maximum-dimension fields, the divisibility checks, and working-set and kernel-memory budgets are decompile-derived and joined to those sweeps; the per-family unlock points for the texture-engine padding modes, the square-after-reduction mode, and sin and cos are decompile-derived from the per-chip support flags, confirmed on the M1 only at its boundary and predicted for the chip that first enables each. On the public side, Apple documents the conversion-time constraints and supported converter configurations at apple.github.io/coremltools; this chapter reports the argument, shape, and mode limits the on-device validators enforce, which the public converter does not enumerate.

Chapter 18. Optimization and the cost model

The convolution latency fit and the dispatch floor are M1/H13 measured (Apple M1; M1 Max), and the cross-chip bandwidth, floor, and peak re-fit are M5/H17s measured, with the core-scaled bandwidth confirmed by direct streaming at 57.5 GB/s against the M1's 10.4 GB/s. The compiler's analytic cost functions (the cycles, roofline, and wall-time chain) are decompile-derived with the per-chip parameters walked live from the hardware-abstraction table; the cross-chip scaling of unmeasured targets is predicted from core-count and clock ratios. The cost-model fidelity is M1/H13 measured: a median error near 31 percent with 11 of 68 shapes within plus or minus 17 percent, sound as an ordinal placement tool rather than an absolute-latency oracle, with attention unmodeled and the 9.0 GB/s bandwidth anchor held below the roughly 40 GB/s effective rate because it is jointly calibrated for the convolution fit. There is no public counterpart: Apple

does not publish the engine compiler's cost model or its per-chip parameters, so the model is reported here as decompile-derived and validated against M1/H13 and M5/H17s measured latency.

Chapter 19. Pitfalls and limits

The slice-saturation threshold, dynamic-weight convolution batch boundary, compile-failure back-off, and four-character-code lowering limit are M1/H13 measured, and the A14 generation (M2) was measured to saturate on the same slice, locating the clean route on A15 and above and confirmed clean on the M5. The per-target slice-lowering template and its DMA source-path and patch-width routines are decompile-derived, giving the times-16 fixed-point DMA format and the target-keyed slice route; the clean A15 route is decompile-derived from the per-family lowering and confirmed clean on the M5, not measured on A15 silicon directly. These are failure modes of the private compiler and have no public counterpart: they are reported as measured on the M1 and reverse-engineered from the compiler binary.

Part VI. The Silicon

Part VI was measured on M1/H13 (Apple M1; M1 Max, live architecture string `h13g`); the datapath geometry and memory hierarchy are decompile-derived from the per-chip hardware-abstraction table and the engine compiler, calibrated against the M1 anchors.

Chapter 20. Datapath and MAC geometry

The core count of four, per-core throughput scaling of 1 to 4, power-rail step per core, radix-4 fan-in, wide accumulator, and output-channel-group pass-doubling threshold at 192 to 256 are M1/H13 measured. The accumulator budget of eight and the lane widths of four and eight are decompile-derived and uniform across chips per the hardware-abstraction table, which was carved from the H13 firmware blob and calibrated at the core-count, cycle-divisor, and working-set offsets; the convolution-lowering and performance-model functions are decompile-derived from the ANE compiler (`GetNumOutputChannelsPerCycle`, `GetNumOutputChannelsPerAccumulator`, `ComputeMaxOcgSize`, `ZinMirNECoreAssignment`, `GetNumNeededNEsNextPow2`). Apple publishes a marketing core count per chip, for example a 16-core M1, a different quantity from the decoded `num_nes` of four; the multiply-accumulate geometry has no public counterpart and is reported as reverse-engineered and measured. The int8 compile flag is weight-only quantization that leaves the multiply-accumulate in fp16, neutral on compute-bound work and about 1.5 times faster only on weight-bandwidth-bound matmuls near a 4096-by-4096 weight, M5/H17s measured.

Chapter 21. Memory hierarchy

The 2.28 to 2.34 MB threshold and the 64-byte throughput period are M1/H13 measured on the dispatch path, and the absence of a runtime replacement policy is an M1/H13 measured negative, with sequential and random re-reference order identical at every footprint. The field values are decompile-derived: `0x1b8` (2 MB operand working set), `0x1c8` (64 banks), `0x1c0` (16-byte granule), `0x1f8` (2 MB stride ceiling), `0x1f0` (residency threshold, 0 on M1), and `0x288` (64 KB kernel store); the operand-size comparator, the bank function and conflict model, and inverted residency-buffer gate are decompile-derived from the named compiler routines, with the 2 MB boundary itself the operand-size comparator. The A15-class, A16-class, and M5 values of field `0x1f0` are read from the same table by offset but predicted for those parts, the gate behavior confirmed on the M1 only. The memory hierarchy, the bank function, and residency threshold have no public counterpart and are reported as reverse-engineered and measured. The compiler name for field `0x1b8`, `MemCacheSize` (also `L2Size`) with its `fl2-size` override, is decompile-derived; on the M5 the working-set crossing is smooth in throughput and shows instead in DRAM energy per operation, bottoming near a 2 MB operand, M5/H17s measured.

Part VII. The Toolchain and Encoding

Part VII was measured on M1/H13 (Apple M1; M1 Max, live string `h13g`), with chapter 25 extending to M2/H14 (Apple M2 Pro) and M5/H17s (Apple M5). This Part is mostly decompile-derived from the engine compiler decompile and its constraint-string corpus, the per-chip hardware-abstraction table read by byte offset across 28 target entries, and the runtime serializers and task-descriptor setters; unless a chapter says otherwise, scalar offsets, capability-byte offsets, struct fields, and symbol names are decompile-derived.

Chapter 22. Compiler

The four-phase pipeline, task-descriptor partition budget, allocation-type set, `anec.matmul` and `anec.convolution` lowerings, and fusion rules (the GOC fused unit, seven-slot epilogue, fusable epilogues and the two-live-input, concat, and attention-cut barriers) are decompile-derived from the engine compiler framework (the 9.509 build, a 4.1-million-line decompile) and the constraint-string corpus. The validator export set, the per-layer reject strings, and the code-generation rejections for top-k, sort, dynamic-slice, and three-dimensional convolution are M1/H13 measured. The compiler internals, the backend `anec.*` dialect, and the `_ANECValidate*` surface have no public counterpart and are reported as reverse-engineered; the public tools document only the frontend operation set and conversion passes.

Chapter 23. Program and container format

The two-layer split, four-field root table, cast-inference-cast operation shape, seven register groups, 15-bit and 17-bit dimension widths, relocation-slot model, 44-byte sparse record, and the HWX on-disk layout (the `0xbeefface` Mach-O variant, the segment set, the `ZinAneTd` linked list, the per-lane weight tiles, and shape descriptors) are decompile-derived from the serializer and task-descriptor symbols and cross-confirmed against the vendor's task-descriptor symbol table. The dispatch-descriptor schema was validated by a round-trip through the schema compiler (version 25.12.19), which regenerated an object-API header and a binary reflection schema without error. The decoded identity-linear program with its segment map, port descriptors, register records, and weight bank are M1/H13 measured, parsed byte for byte from real on-disk files in the runtime caches, with the resolved tensor frame read from the post-compile status sidecar. The format has no public counterpart and is reported as reverse-engineered; the full FlatBuffer schema is Appendix C. The custom-bar bit-field relocation, which patches a resolved value into a named descriptor field by bit offset and width, and the range-checked live-in shape and stride parameters that let one program serve a range of input shapes, are decompile-derived from the program loader.

Chapter 24. HAL and capability gates

The scalar offsets, the capability-byte offsets, and family floors are decompile-derived, with every per-target constructor invoked on this host (live string `h13g`) to read byte-exact values for all 28 targets, joined to the minimum-family operation trait and tier assignment read from the same binary. The per-family unlock generations for the texture engine, sin and cos, and the dimension and format-count steps are decompile-derived from the per-target tables and confirmed on the M1 only at its boundary; the generation that first enables each is predicted from the floor table. The attested-is-not-reachable rule is M1/H13 measured: three-dimensional convolution has its HAL kernel-depth attestation at offset `0x70` and fails backend lowering on every device mask, and the top-k, sort, and dynamic-slice validators are callable but code-generation-rejected. The packed-bitfield struct measures `0x938` bytes; its non-flag residual is the cost-model coefficient block at `0x580` through `0x7f0` plus about two soft fp64 coefficients, and the offsets past `0x938`, read in an earlier round as an A12 operation-emulation catalog at `0xa30` through `0xe84`, are a read into zeroed memory beyond the struct and have no table; the capability-flag offsets are decompile-derived from the `ZinIrHalParameters` reader symbols. The HAL table, the capability-byte region, and minimum-family trait have no public counterpart; Apple documents only the model framework and the convertible operation set, not the per-chip capability table or the per-operation family floor.

Chapter 25. Compression internals

The bit-layout and address detail are M1/H13 (table-descriptor codegen version five, family two, A13) unless another version or family is named. The sparse and int4 speedups and byte ratios, the byte-identical compiled program, and the M1 fold of the int8 and blockwise forms are M1/H13 measured; the A14 int8 stream and blockwise fold are M2/H14 (Apple M2 Pro) measured; the all-forms-stream endpoint is M5/H17s (Apple M5) measured. The kernel-format helper tables, affine and palette dequantization codecs, dequantize-to-dense fold path, streaming and palette gates, on-chip-memory budget caps, and Winograd eligibility gate are decompile-derived, with the on-device sparse-format field read from the register map; the A15 floor at which the blockwise form first streams is predicted from the gate pattern, and the resident Winograd transform matrices are predicted, their textbook forms matching the engine's behavior but not byte-confirmable from the binary. Apple's conversion tools document palettization, quantization, and pruning at apple.github.io/coremltools as a model-size feature; this chapter extends that account with the on-device codec arithmetic and the per-family streaming datapath.

Chapter 26. Hidden layers and direct netplist authoring

The fused-attention, ranking, and spatial-rearrange layers were authored, compiled, and dispatched on the M5/H17s byte-exact against a host reference, and the M1 gates and the top-k forbidden band were confirmed on the M1 (measured). The native layer-descriptor catalog, its per-layer `ZinParse<Name>Unit` parsers and `_ANECValidate<Name>Layer` checkers, and the constant-string constraint corpus are decompile-derived, joined to the netplist schema read out of the runtime framework. On the public side, Apple documents the model converter and its intermediate-language operation set at apple.github.io/coremltools, which does not emit these native layer kinds; this chapter authors them directly, the attention, ranking, spatial-rearrange, geometry, and normalization descriptors being present in the compiler and reachable through the network description even though the converter never produces them. The 33-knot activation-LUT format and the gated `NeuronCustom` netplist path (a parser that requires and then rejects the same field sets) are decompile-derived; the rectifier-basis reproduction of an arbitrary pointwise function is M5/H17s measured.

Part VIII. System Internals

Part VIII rests on M1/H13 (Apple M1; M1 Max, and the T6000-generation engine where a multi-die part is needed), with an M2-class kernel cache as the cross-generation reference where cited. Much of it is decompile-derived static analysis with no firmware executed: the unencrypted real-time-kernel preload executable is carved from the on-package firmware image and read for its strings, asserts, and disassembled handlers, and the kernel cache is read for its symbols, dispatch arrays, and call sites.

Chapter 27. Kernel driver and IOKit ABI

The two `IOExternalMethodDispatch2022` arrays are M1/H13 measured, read byte for byte from the kernel cache's read-only data section and corroborated on an M2-class cache where all 26 size tuples are byte-identical, and the control-client open and submit struct sizes are cross-validated against captured user-space call blobs. The selector handlers, four-layer call path, doorbell register write, entitlement-check call sites for `com.apple.ane.iokit-user-access` and `com.apple.ane.allow-dataChaining-access`, driver class hierarchy and device properties, and broker model are decompile-derived from the unstripped kernel-cache symbols, kext property lists, live device registry, and a system-wide entitlement sweep. The driver's user-client ABI has no public counterpart and is reported as reverse-engineered; the IOKit user-client framework and the `IOExternalMethodDispatch2022` structure are public, but this driver's selector numbers, struct sizes, and handler set are not. On the M5 the client-creation gate `ANEClientInfo::create`, its `copyClientEntitlement` stamp of `isPrivileged` and `allowDataChaining`, and six further driver-enforced `com.apple.ane` and `com.apple.private.ane` entitlements are decompile-derived from the M5 kernel driver.

Chapter 28. Address translation and the DART

The leaf word `phys | 0x8000000000000000`, 16 KB granule, active stream set `{0, 1, 2}`, translation-table base `0x90022320`, and host-to-firmware rebase to the `0x1bc4` aperture are M1/H13 measured read-only on the live dispatch path (Apple M1 Pro, T6000-generation DART): the granule and stream set from the live device tree, the base register from function-boundary probes, and the leaf-word template, segment structure, and protection classes from probes across 26178 leaf-map events. The leaf-word bit layout, the fault-register offset map, the panic-terminated fault path (panic confirmed M1/H13 from the disassembly of every fault-path function), the `IODARTErrorInfo` descriptor layout, the `[engine+0xe028]` status predicate, and firmware rebase arithmetic with its three aperture-config offsets are decompile-derived, the rebase arithmetic unicorn-verified. The fault-capture register decode is predicted from the published controller field layout and not measured, because a fault panics the machine. The address-translation unit, its leaf entry format, the rebase boundary, and the fault-capture block have no public counterpart and are reported as reverse-engineered. The per-client isolation contexts, the eight `mapper-ane0` translation mappers in the live IORegistry and the `ANEIsoID1` through `ID7` exclave capabilities that bind them, are M5/H17s measured with System Integrity Protection enabled.

Chapter 29. Firmware

The preload executable is the M1-generation real-time-kernel image, build identity `RTKit-3255.120.11 .release`, chip tag `ASC_CHINOOK`. The task roster, priority bands, heap and pool model, execution-loop command set, scheduler deadline, and fault post-mortem layout; the command-record classes and their `sCSneCmdProcedureCall*` invariants; the doorbell-emit sequence around bit 39 of `S3_3_C15_C8_0`, host-notify site `@0x4c890`, and engine-to-graphics-processor doorbell at `0x2_0646_8000`; the bring-up order, seven MMIO banks, RTBuddy endpoint, and three scratch handshakes; and the per-run statistics buffer with its header, per-engine descriptors, and host-side null gate are decompile-derived from the embedded strings, assertion expressions, and disassembled handlers. The firmware has no public counterpart and is reported as reverse-engineered from the unencrypted image; Apple documents only the model framework and conversion tools above this layer, not the on-engine operating system. The CHINOOK control-CPU register map, its eleven thread contexts, level-two cache, and pipeline error-capture and power-down-save registers, is decompile-derived from the kernel driver.

Chapter 30. Host-to-firmware command protocol

This chapter is static analysis of the unencrypted M1 firmware image, an ARM64e real-time-kernel Mach-O, with no firmware executed. The command vocabulary, numeric identifiers, header layout, and body bounds are read from the ordered command-name string table, struct-size asserts, and log format strings; header byte offsets are inferred from field order and alignment, while field presence, widths, and bounds are read directly from in-binary asserts. The 94-entry `CSNE_CMD` enumeration (93 dispatched command identifiers plus the invalid sentinel), the roughly ten fast-path ids, and the seventy-six-slot dispatch vtable (arm64e auth-rebase chained pointers in `__DATA.__const`, low thirty-two bits giving the target, the procedure-call slot at `+0x200` reaching `0x7374c` and the inference slot at `+0x190` reaching `0x74510`) are decompile-derived. The protocol, the command header, and `CSNE_CMD_*` vocabulary have no public counterpart and are reported as reverse-engineered; the public model framework describes application-level model loading, not the controller command channel. The full numeric command table and the decoded request structs are Appendix C.

Chapter 31. Power and thermal

The clean 0 mW idle, the 176-second saturating loop holding flat power and throughput under nominal thermal pressure, the first-op power-up tax near 0.5 ms past a 100 ms idle gap, and the single held power state across 56,527 dispatches are M1/H13 measured by read-only tracing. The power-block base `0x2_6b8f_0000`, the opaque voltage base `0x2_3b70_c008`, the five-store power-block arm, the `0x11` peak-power control word, the engine and power-manager device-tree nodes (`ane0@84000000`, `compatible "ane,t8020"`, the `0x2_8400_0000` aperture, the `0x2_8E08_0000` power-manager slice), the absence of local DVFS, and firmware seven-step credit sequence are decompile-derived from the H13 firmware Mach-O and live device-tree

enumeration. There is no public counterpart: Apple documents neither the power model, the fixed operating point, nor the thermal behavior, so this account is reported as reverse-engineered and measured.

Chapter 32. Security and isolation

The cross-process timing side-channel (a 2.3 times latency jump under contention and a 20 to 50 bit/s occupancy channel) and the intact data isolation across 9000 concurrent results are M1/H13 measured. The kernel-side trust boundary and its three program checks (code signature, vnode trustcache, client code-signing identity), the firmware's structural-only check, and the secure and exclave method bodies (the secure-mode transition state machine, the `SwitchExclaveMode not supported` stub, the inert `mov w0, #0; ret` exclave selectors) are decompile-derived from the loaded kernel driver and the firmware image. There is no public counterpart: Apple documents neither the secure and exclave transition internals nor the cross-process isolation behavior, so this account is reported as reverse-engineered and measured. On the M5 the exclave is live rather than stubbed: the secure component `com.apple.aneexclave`, its capability-scoped segment access, and the Tightbeam submit path are decompile-derived from the M5 exclave bundle and boot kernelcache, with System Integrity Protection enabled.

Chapter 33. Telemetry and hardware counters

On the Apple M1 and M1 Max, the readable whole-engine channels (DRAM bytes, energy, clock residency), the signpost lifecycle, the all-zero per-run output buffer, and forced-mask load rejection are M1/H13 measured. The `ANEProgramCreateArgs` layout, the `+0x6c` stats-mask offset, the twenty-four per-descriptor counter namespace, the stats-buffer ABI `0x0201`, the `initStatsBufferSection` bail branch, and the free-running engine timebase counter at MMIO `0x2_6b17_8000` (read by the firmware helper `@0x30988`) are decompile-derived from the runtime dylibs and the kernel driver. There is no public counterpart: Apple documents none of the hardware performance-counter block, per-task-descriptor namespace, stats-mask enable, or firmware timestamp, so the block geometry, master enable, readable-versus-walled split, and kernel gate are reported as reverse-engineered and measured. On the M5 the gate actor (the `aned` daemon forcing `statsMask=0` for a `ThirdPartyAppUsingANE` client), the per-channel `DCS BW / ANE L0` and `L1` State-residency bandwidth histograms readable on the unentitled path, and the fuller `ANE_THROTTLE_*` and `VDD_DRAM_VOLTAGE_CHANGE` trigger family are M5/H17s measured with System Integrity Protection enabled.

Part IX. Cross-Silicon Reference

Part IX was measured on M1/H13 (Apple M1; M1 Max) with M5/H17s (Apple M5) as the cross-chip reference; the work is decompilation and static analysis of the one engine compiler binary across its full target set, with boundaries reproduced on silicon where a part was in hand.

Chapter 34. Cross-silicon targets

The 28-target set is measured, extracted by invoking each per-architecture builder on the M1 (host chip irrelevant) and resolving the M5 to `H17s` and the M1 Max to `H13G` by fixed-build-directory compile. The target names, the suffix-to-core decode at HAL offset `0x238`, the interchange-format tables at HAL offset `0x658`, and four-byte format decode are decompile-derived; the A14, A15, A16, and A18 targets and their M-series counterparts are decompile-derived from the per-target tables and not individually measured, and the $M(n) \to H(n+12)$ mapping is confirmed only at the M1 and M5 ends with the middle generations predicted. On the public side, Apple's product specifications publish a marketing core count per chip, for example a 16-core engine, a different quantity from the decoded `num_nes`, which counts per-die compute sets: four on the base M1 against the published sixteen. The 28-target compiler set, the H-architecture naming, the suffix-to-core decode, and interchange tables are not publicly documented and are reported as reverse-engineered.

Chapter 35. Per-family code generation

The slice-saturation threshold, the top-k, sort, and dynamic-slice code-generation rejections, and the operation decompositions are M1/H13 measured; the clean slice route and the native crop-resize, resample, and trig operations are M5/H17s measured. The family enum, `MinimumFamily<N>` trait and its four operation tiers, per-chip hardware-abstraction offsets, task-descriptor patch-width path, and `ConvertSlice<Family>` lowering are decompile-derived from the engine compiler framework (the 9.509 build, 87,874 functions); the A14 and A15 unlock points, M-series families above the M1, and dedicated A15 code-generation branch with its cost table, 45-operation floor, full H15 targets, and YUV420 input are decompile-derived and not measured on A15 silicon. The family enum, minimum-family trait, per-chip parameters, and per-family route selection have no public counterpart; the public conversion tools document the frontend operation set, optimization passes, and compute-unit selector, not the backend per-family lowering reported here.

Chapter 36. Predicted upper tier

The fp8 format converters, E4M3 overflow enumeration, format-register encoders, double-multiply gate, collective dialect operation set, device-mesh and sharding lowering, reduction-to-atomic map, and collective direct-memory-access emitter are decompile-derived, with the family gates read from the 28-target capability bytes; the 64-core ceiling, fp8 `e4m3` and `e5m2` datapath, and Ultra device-mesh collective are decompile-derived and not measured on the upper-tier parts. The E4M3 native multiply, accumulation, and saturation (inferred from the encoders and the H18-only capability byte at offset `0x52d`), the fp16 accumulation of an fp8 multiply (from the absence of any fp8 accumulator field), and the running collective (the enable byte at offset `0x48b` zero on all 28 targets and the register encoding stubbed on every family) are predicted, with no current family materializing the collective. That the load-balancer supports up to four engine dies while the M1 and M1 Max each register a single engine, so cross-die steering engages only on a multi-die part such as the Ultra, is M1 Max measured by the device registry. The fp8 datapath and the multi-die collective layer have no public counterpart in Apple's documentation and are reported as reverse-engineered and explicitly unmeasured.

Back matter

The back-matter chapters rest on M1/H13 as the primary host and M5/H17s for cross-generation scaling.

Methodology

Apple documents the engine only as a compute-unit selector at developer.apple.com/documentation/coreml/mlcomputeunits, with no direct device API; this chapter extends that account by reaching the engine directly below the selector and characterizing it by static analysis and live instrumentation. The direct dispatch route, the four static-analysis artifacts, and the program-binary capture are decompile-derived and confirmed by the kernel trace; the roofline figures and the compile-service rate condition are M1/H13 measured, and the cross-generation scaling and bounded numeric drift are M5/H17s measured.

Open questions

The decoded baseline and the boundary limits are M1/H13, decompile-derived from the compiler, hardware-abstraction tables, kernel driver, and firmware and joined to live instrumentation; M5/H17s confirmed the cross-family predictions. The M3/H15 and upper-tier runtime behavior is predicted, decompile-derived from the gates and not confirmed on silicon.

Appendices

The reference tables are decompile-derived from static, read-only analysis of the M1/H13 binaries: the ANE compiler (`ANECompiler 9.509`), its per-family operation-floor tables, its per-layer validators and parsers, the host runtime, and the unencrypted firmware image (`h13_ane_fw_styx_j5x.im4p`), with no firmware

executed and no engine jobs run. Where a value or status is measured rather than decompile-derived, it was confirmed on physical silicon, primarily M5/H17s and M1/H13, by compiling and dispatching against a host reference.

Appendix A. The operation-by-device matrix

The M1, M2, and M5 columns are measured by operation-conformance runs, each native operation compiled and run and each no-path one rejected on device; the M3 column and the M4 part of the merged M4-and-M5 column are decompile-derived predictions from the per-chip tables. The per-family unlock points for the texture-engine operations, sin and cos, rank and sort bridge, argument reductions, and weight-stream gates are decompile-derived from the operation floors, validators, and symbol-resolution map and confirmed on the M1 only at its boundary; the family that first runs each is predicted from the floor table. Apple documents the convertible operation set at apple.github.io/coremltools and developer.apple.com; this table extends and partly corrects that account, reporting what compiles and runs on the direct engine path, since some accepted operations, such as three-dimensional convolution, do not lower to the engine.

Appendix B. The hidden-layer catalog

Each layer's `Type` tag, descriptor symbol, and `Params` key set are decompile-derived from the compiler export table, parser disassembly, constant-string key atlas, per-layer `ZinParse<Name>Unit` parsers and `_ANECValidate<Name>Layer` checkers, and descriptor-initializer routines `_ANEC<Name>LayerDescInitialize`, joined to the netplist schema read out of the runtime framework. The fused-attention operand contract, the spatial-rearrange channel ordering, the float16-bit-pattern convention for `Alpha`, `Epsilon`, and `Scale`, and point-cloud output contracts are M5/H17s measured by authoring and dispatching the layers, and the M1 arch gates, the `Sort` and `DynamicSlice` rejections, and the top-k $\{3, 4\}$ forbidden band are M1/H13 measured. Apple documents the model converter and its intermediate-language operation set at apple.github.io/coremltools, which does not emit these native layer kinds; this catalog authors the native descriptors directly through the network description.

Appendix C. Decoded reference tables

Every value is read out of an M1/H13 binary by static analysis, with no firmware executed: the attribute and opcode integers and IOKit struct layouts from the compiler decompile (`ANECompiler 9.509`) and the host runtime; the error constants, command table, and tunable table from the unencrypted firmware image (`h13_ane_fw_styx_j5x.im4p`) and the standard IOKit return macros; the register map from the compiler's task-descriptor setters and getters; and the `.e5` schema from the runtime serializer symbols, validated byte-for-byte against a captured sample. The runtime, firmware, compiler, and ABI surface consolidated here is private and undocumented and is reported as reverse-engineered; the public model framework, conversion tools, and intermediate-language reference describe none of these numeric tables.

References

1. AmiraniLabs. “libane: a native Apple Neural Engine runtime.” Repository, https://github.com/AmiraniLabs/libane.
2. Apple. Accelerate and BNNS documentation. https://developer.apple.com/documentation/accelerate.
3. Apple. Active installed base of 2.5 billion devices, reported by T. Cook on the first-quarter fiscal 2026 earnings call, January 29, 2026. apple.com.
4. Apple. Apple silicon technical specifications. https://www.apple.com/mac/compare/.
5. Apple. Core ML framework documentation. https://developer.apple.com/documentation/coreml.
6. Apple. Core ML Tools (coremltools) documentation. https://apple.github.io/coremltools.
7. Apple. Vision framework documentation. https://developer.apple.com/documentation/vision.
8. Apple Machine Learning Research. “Deploying Transformers on the Apple Neural Engine.” Apple Machine Learning Research article, 2022.
9. Benazir, A., and Lin, F. X. “Efficient Mixture-of-Experts LLM Inference with Apple Silicon NPUs.” Preprint, arXiv:2604.18788, 2026.
10. Bi, Z., Chen, X., Sun, L., Yao, Y., Shen, Q., Lou, J., and Deng, C. “RooflineBench: A Benchmarking Framework for On-Device LLMs via Roofline Analysis.” Preprint, arXiv:2602.11506, 2026.
11. Bryngelson, S. H. “ANEForge: Python for direct computation on the Apple Neural Engine.” Preprint, arXiv:2606.17090, 2026.
12. Chen, L., Feng, D., Feng, E., Wang, Y., Zhao, R., Xia, Y., Xu, P., and Chen, H. “Characterizing Mobile SoC for Accelerating Heterogeneous LLM Inference.” ACM SIGOPS Symposium on Operating Systems Principles (SOSP), 2025. arXiv:2501.14794, DOI 10.1145/3731569.3764808.
13. Choi, J. W., Bedard, D., Fowler, R., and Vuduc, R. “A Roofline Model of Energy.” IEEE International Symposium on Parallel and Distributed Processing (IPDPS), 661-672, 2013. DOI 10.1109/IPDPS.2013.77.
14. Community Apple Neural Engine reverse-engineering repositories. johnmai-dev/ANE-LM, mechramc/Orion, skyfallsin/apple-neural-engine-field-guide, and dmaynor/apple-vuln-research. Repositories.
15. Ding, N., and Williams, S. “An Instruction Roofline Model for GPUs.” IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), 7-18, 2019. DOI 10.1109/PMBS49563.2019.00007.
16. Fanariotis, A., Orphanoudakis, T., and Fotopoulos, V. “Evaluating the Energy Efficiency of NPU-Accelerated Machine Learning Inference on Embedded Microcontrollers.” Preprint, arXiv:2509.17533, 2025.
17. Gerganov, G. “whisper.cpp: Whisper inference in C/C++ with Core ML Neural Engine support.” Repository, https://github.com/ggml-org/whisper.cpp.
18. Hollemans, M. “The Neural Engine: What Do We Know About It?” Community-maintained repository, https://github.com/hollance/neural-engine.
19. Hotz, G., and the tinygrad authors. “tinygrad.” Repository, https://github.com/tinygrad/tinygrad.
20. Hübner, P., Hu, A., Peng, I., and Markidis, S. “Apple vs. Oranges: Evaluating the Apple Silicon M-Series SoCs for HPC Performance and Efficiency.” Preprint, arXiv:2502.05317, 2025.
21. Ignatov, A., Timofte, R., Kulik, A., Yang, S., Wang, K., Baum, F., Wu, M., Xu, L., and Van Gool, L. “AI Benchmark: All About Deep Learning on Smartphones in 2019.” Preprint, arXiv:1910.06663, 2019.
22. Ilic, A., Pratas, F., and Sousa, L. “Cache-Aware Roofline Model: Upgrading the Loft.” IEEE Computer Architecture Letters, 13(1), 21-24, 2014. DOI 10.1109/L-CA.2013.6.
23. Jayanth, R., Gupta, N., and Prasanna, V. “Benchmarking Edge AI Platforms for High-Performance ML Inference.” Preprint, arXiv:2409.14803, 2024.

24. Jouppi, N. P., Young, C., Patil, N., Patterson, D. A., et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit." International Symposium on Computer Architecture (ISCA), 1-12, 2017. Also arXiv:1704.04760.
25. Kumaresan, R. "Orion: Characterizing and Programming Apple's Neural Engine for LLM Training and Inference." Preprint, arXiv:2603.06728, 2026.
26. ML.ENERGY / Zeus. "Programmatic Energy Consumption Measurement on Apple Silicon (macOS)." Project issue report (#159), 2025.
27. Moon, S., Cha, J., Park, H., and Kim, J. "Hybe: GPU-NPU Hybrid System for Efficient LLM Inference with Million-Token Context Window." International Symposium on Computer Architecture (ISCA), 808-820, 2025. DOI 10.1145/3695053.3731051.
28. Plyenkov, B. "Decoupling Machine Intelligence from Application in IoT Devices." Master's thesis, Aalto University, 2019.
29. Prashanthi, S. K., Sahoo, K. K., Saikia, A. R., Gupta, P., Joshi, A. V., Pansari, P., and Simmhan, Y. "Pagoda: An Energy and Time Roofline Study for DNN Workloads on Edge Accelerators." Preprint, arXiv:2509.20189, 2025.
30. Singh, M. "Inside the M4 Apple Neural Engine, Part 1: Reverse Engineering." Blog post and repository, 2026, https://github.com/maderix/ANE.
31. Tummalapalli, P., Arayakandy, S., Pal, R., and Kundan, K. "LLM Inference at the Edge: Mobile, NPU, and GPU Performance Efficiency Trade-offs Under Sustained Load." Preprint, arXiv:2603.23640, 2026.
32. Verhelst, M., Benini, L., and Verma, N. "How to Keep Pushing ML Accelerator Performance? Know Your Rooflines!" IEEE Journal of Solid-State Circuits, 2025. DOI 10.1109/JSSC.2025.3553765.
33. Williams, S., Waterman, A., and Patterson, D. A. "Roofline: An Insightful Visual Performance Model for Multicore Architectures." Communications of the ACM, 52(4), 65-76, 2009. DOI 10.1145/1498765.1498785.
34. Xu, D., Zhang, H., Yang, L., Liu, R., Huang, G., Xu, M., and Liu, X. "Fast On-device LLM Inference with NPUs." ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2025. arXiv:2407.05858, DOI 10.1145/3669940.3707239.
35. Yang, C., Kurth, T., and Williams, S. "Hierarchical Roofline Analysis for GPUs: Accelerating Performance Optimization for the NERSC-9 Perlmutter System." Concurrency and Computation: Practice and Experience, 32(20), e5547, 2020. DOI 10.1002/cpe.5547.
36. Yoon, E. "ane: a reverse-engineered Linux driver for the Apple Neural Engine, with anecc." Repository, 2022, https://github.com/eiln/ane.