Check for updates

Software: Practice and Experience

WILEY

RESEARCH ARTICLE OPEN ACCESS

ANEForge: Python for Direct Computation on the Apple Neural Engine

Spencer H. Bryngelson[1,2]

[1]Walker Department of Mechanical Engineering, The University of Texas at Austin, Austin, Texas, USA | [2]Oden Institute for Computational Engineering and Sciences, The University of Texas at Austin, Austin, Texas, USA

Correspondence: Spencer H. Bryngelson (shb@utexas.edu)

Received: 16 June 2026 | **Revised:** 14 October 2026 | **Accepted:** 18 October 2026

Keywords: Apple Neural Engine | hardware accelerators | machine-learning compilers | on-device training | Python

ABSTRACT

Background: The Apple Neural Engine (ANE) is the fixed-function neural accelerator on every recent Apple device, and the only sanctioned route to it is CoreML. CoreML treats the engine as a scheduling option: no configuration requires it, the CPU remains a permitted fallback, and no runtime interface reports which unit ran a call.

Objective: We set out to program the ANE directly from Python, with the compute unit fixed so that each measurement belongs to the engine, and to report what the engine computes and at what cost.

Methods: ANEForge compiles a lazy tensor graph into one ANE program and dispatches it through Espresso, the private runtime that CoreML is built on. It needs no entitlement and leaves system-integrity protection on. The package reaches the engine's native fused-attention layer, streams int8, int4, and sparse weights, keeps decoder and optimizer state on the engine across calls, runs the forward pass, backward pass, and optimizer update of training on the engine, and compiles fixed-iteration Krylov solvers and a factored Fourier transform as static-dataflow programs. We measured latency, energy per inference, and agreement with framework references on an M5 Pro, against the GPU and CPU at matched precision.

Results: A pretrained ResNet-18 forward runs in 0.33 ms and uses 2.3 mJ per inference, about a factor of 9 less than the GPU at matched fp16 precision (paired runs give 5.5 to 12.7). A call costs about 74 µs whether the program fuses one operator or 128. ResNet-18, a Vision Transformer, and a sentence encoder each compile to one program and match their framework references at cosine similarity 1.0000. One forward pass of the Stable Diffusion U-Net and VAE decoder agrees with the reference to a relative error of 0.041 and 0.021.

Conclusions: The engine's advantage comes from fusion. A single memory-bound operator finishes sooner on the CPU or the GPU, while a fused program pays the per-call floor once. Fixed-iteration methods whose program size does not depend on the problem dimension compile to the engine, and half precision limits how ill-conditioned a problem they can solve.

1 | Introduction

The Apple Neural Engine (ANE) is a fixed-function neural accelerator that ships on every recent iPhone, iPad, and Apple Silicon Mac [1, 2]. On the M-series Mac it draws the least power of the system-on-chip's three programmable compute blocks for the workloads it accepts. The only sanctioned route to the ANE is CoreML, which compiles a model and selects a compute unit at runtime [3].

CoreML treats the engine as a scheduling option. A caller can exclude the GPU and can inspect placement offline, per layer

This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

© 2026 The Author(s). *Software: Practice and Experience* published by John Wiley & Sons Ltd.

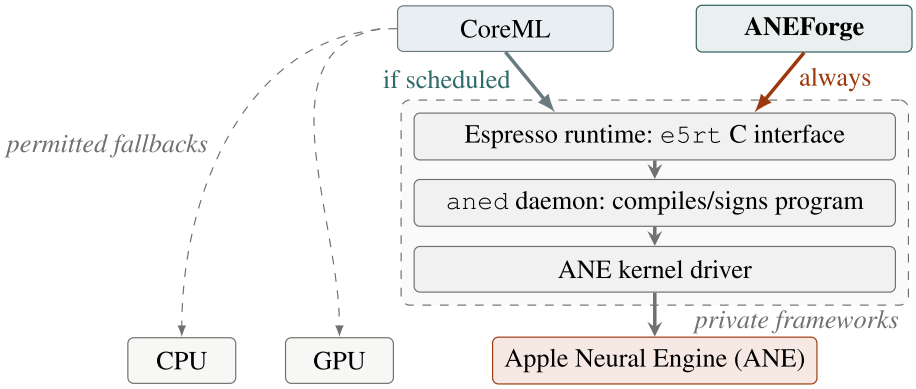


FIGURE 1 | Where ANEForge enters the system. CoreML schedules a compiled model across the CPU, the GPU, and the engine, and may serve any call on a fallback unit (dashed); ANEForge enters the private stack at the `e5rt` interface and dispatches to the engine only (solid). Either way, every program is compiled and signed by Apple's `aned`.

in Xcode's performance reports or as the estimate `MLComputePlan` returns. However, no configuration requires the ANE, the CPU is a permitted fallback, and no runtime reports which unit executed a call [1, 3]. A workload accepted by the engine often runs efficiently can therefore be served on a slower, less efficient unit with no diagnostic at the call site.

The hardware is reached through a stack of private, undocumented Apple frameworks that CoreML, MPSGraph, and Espresso call internally (Figure 1). That stack is the programming surface of the ANE, and it is not exposed to application code. Determining what the ANE can compute and creating a frontend that targets it directly, requires working at this level.

Here, we present ANEForge, a Python package that compiles an operator graph and dispatches it to the ANE through private symbols, without loading CoreML. The caller builds a graph, compiles it to an ANE program, and runs it. The package exposes its dispatch paths and a machine-readable record of its operators with evidence that each runs on the device.

Section 2 places the package among the projects that reach the engine below CoreML and states what distinguishes its entry point. Section 3 describes the access path, the operator surface, and the compile-and-dispatch pipeline, and Section 4 lists what the package does. Section 5 reports latency, energy, and agreement against reference frameworks on an M5 Pro, with the dispatch floor and the attention rewrite treated in detail, and Section 6 reports how the private surface has behaved across operating-system releases. Section 7 states the limitations and Section 8 concludes.

2 | Prior Direct Access to the Engine

Table 1 compares earlier projects with ANEForge. `tinygrad` dispatched single operators to the engine from macOS in 2020–2021 by extracting compiled `hwx` programs and submitting them through the ANE services framework and its IOKit interface, but only by signing a restricted Apple entitlement into the Python interpreter after patching the entitlement check [4].

The Asahi Linux effort documents the hardware and drives it on Linux through an open kernel driver and a user-space library; its compiler converts programs that macOS's own ANE compiler produced [5]. A community knowledge base documents the engine as observed through CoreML [1]. Two 2026 projects program the M4 engine through the private `_ANEClient` and `_ANECompiler` classes of the AppleNeuralEngine framework: A reverse-engineering series with a training repository in Objective-C [6], and Orion, an Objective-C runtime with a 27-operation graph intermediate representation that lowers through five passes to MIL and trains a 110M-parameter transformer with the forward and backward passes on the engine and the loss, weight gradients, and Adam update on the CPU [7].

ANEForge enters the private stack one layer above these projects. CoreML sits on the Espresso runtime, whose execution runtime `e5rt` owns compilation, the program library, the execution stream, and the operand descriptors, and `e5rt` in turn is a client of the AppleNeuralEngine framework's `_ANEClient` classes, which hand programs to the `aned` daemon and the kernel driver [8]. CoreML is one client of that runtime; an application that names the engine as its target is another. Both entries converge on `aned`, which compiles, signs, and caches every program, and reach one engine. The `e5rt` compile call accepts MIL with the options Espresso passes and returns a program that `aned` has lowered and signed, so the fused route gets Apple's operator fusion and layout. Its ports bind to buffer objects, and binding an output port to an input port's buffer keeps a decoder's key-value cache or a trainer's optimizer state on the engine between calls. The in-memory models of `_ANEClient` are compiled by the same `aned`, and its requests take IOSurface buffers. Whether a caller can reuse those buffers across calls to keep state on the engine has not been tested [8], so Orion's host-side key-value cache (Table 1) may be a design choice of that project. At `e5rt` the package reaches compilation and buffer binding through the C interface CoreML uses. The `_ANEClient` classes are entered only for the native bridge layers of Section 3.2, whose network description the public compiler never emits, and there ANEForge uses the in-memory-model surface that Orion and Singh use. Unlike the other projects in Table 1, ANEForge keeps a capability registry that pairs its operators with evidence from the hardware (Section 3.2).

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.70124, Wiley Online Library on [29/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 1 | Projects that reach the ANE below CoreML, by what they enter through and what they run.

	`tinygrad` [4]	Asahi `ane`, `anecc` [5]	Singh [6]	Orion [7]	ANEForge
Entry surface	`hwx` via ANE services and IOKit	Linux driver; `hwx` compiled on macOS	`_ANEClient`, `_ANECompiler`	`_ANEClient`, `_ANECompiler`	`e5rt` (Espresso); in-memory model for bridge sub-programs
Unentitled	– (Entitlement, patched `amfid`)	Linux	✓[a]	✓[a]	✓
Language	Python	C, Python	Objective-C	Objective-C	Python; Objective-C++ shim
Graph compiler	– (Single ops)	– (CoreML compiles)	Hand-written MIL kernels	27-op IR, five passes	lazy graph; 100 fused ops, 19 native layers
Training on engine	—	—	F, B	F, B (O on CPU)	F, B, O
Native fused attention	—	—	– (Decomposed)	– (Decomposed)	✓
Compressed weights	—	—	int8 (W8A8)	Not stated	int8, int4, sparse
Resident state	—	—	Not stated	KV cache on host	KV cache, optimizer state
Numerical kernels	—	—	—	—	Krylov, FFT, Cholesky, eigenvalue
Models validated	None end to end	CoreML models	Stories110M, Qwen3-0.6B	GPT-2 124M, Stories110M	ResNet-18, ViT-B/16, MiniLM, SD U-Net and VAE
License	MIT	MIT	MIT	MIT	MIT

Note: "Entry" is the private surface the project calls; "Unentitled" means no Apple entitlement and no change to code signing or system-integrity protection. "Training" lists which of the forward pass (F), backward pass (B), and optimizer update (O) run on the engine. "Not stated" marks a feature the project's documentation does not claim; a dash marks a feature it does not have.

[a]Inferred from the classes each project calls. Singh and Orion use the in-memory-model classes of the AppleNeuralEngine framework, which the companion guide finds callable without an entitlement [8], but neither project states its entitlement status.

3 | Design

The package is organized around one path: A lazy operator graph is lowered to a single ANE program and dispatched to the hardware. This section describes the access path that carries the dispatch (Section 3.1), the operator surface the caller builds graphs from (Section 3.2), the compile-and-dispatch pipeline (Section 3.3), and the cost model, optimizer, and autograd modules that operate over it (Section 3.4).

3.1 | The Access Path

The caller constructs a graph of `Tensor` nodes from the operator surface. The compiler lowers the whole graph to the model intermediate language (MIL) that the ANE compiler consumes, packs the weights into one binary blob, and produces a callable program handle. Dispatch goes through `e5rt`, a C interface exported by Apple's private Espresso framework, into the `aned` system daemon and the ANE kernel driver that Apple's own frameworks use. The program handle is reused across calls, so a model compiles once and runs many times.

A small Objective-C++ shim carries the dispatch; it is compiled on the user's machine because it links private Apple frameworks, and the Python package loads it lazily on the first call. Access is unentitled. The shim resolves the symbols of the private frameworks at run time, carries no Apple entitlement, and leaves code signing and system-integrity protection in place. Every program binary is produced and signed by Apple's own `aned`, and the shim loads nothing that `aned` did not sign. The shim uses about 30 of the 235 `e5rt` symbols Espresso exports, and the system log shows each dispatch reaching the driver with the code-signing identity of a third-party application.

3.2 | The Operator Surface

We call the *operator surface* the set of operators a caller can build a graph from. It is a subset of what the hardware accepts, and the capability registry records both sets. The evidence for the MIL operators is a sweep of all 166 operators through this path, stored in the repository, and the bridge operators have recorded runs of their authored layers [8]. The registry is generated from the frontend's operator tables, and on hardware a release re-runs only the operator corpus of Section 6.

Two routes feed the dispatch path (Figure 2). The fused route lowers 100 MIL operators, listed by category in Table 2, and four attention composites assembled from them (`sdpa`, `mha`, `cross_attention`, and `geglu`) into one program. Those operators are the subset of the 166-operator MIL vocabulary that a sweep of every operator found to compile and run on the engine through this path (116 of 166) and that the frontend exposes. The bridge route adds 19 native operators that the public toolchain does not emit, including the fused-attention layer, `argmax`, `topk`, `sort`, and several geometry and point-cloud layers, by authoring the compiler's internal network description directly and running the result as a sub-program through a graph cut. The 19 are promoted from 26 native hardware layer types authored and run this way; those 26 sit among the 45 layer descriptors the ANE compiler carries, each paired with

10970024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.70124, Wiley Online Library on [29/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

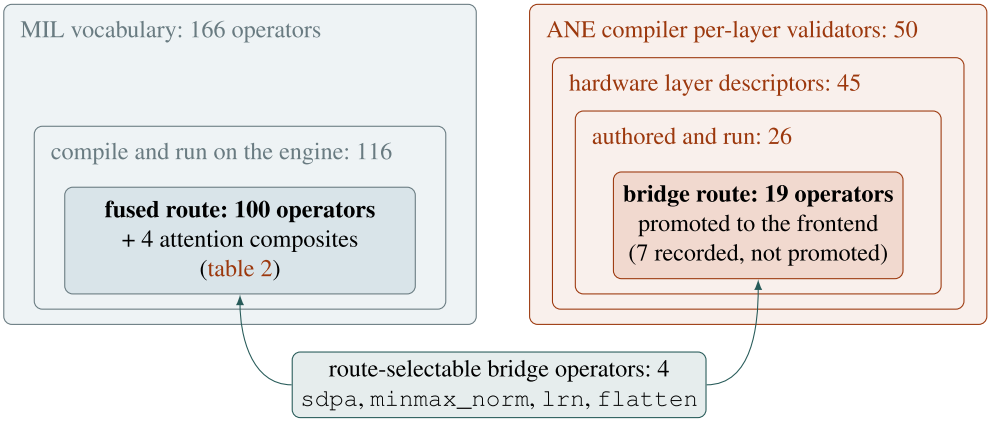


FIGURE 2 | How the operator counts relate. The fused route (left) exposes 100 of the 116 MIL operators that compile and run on the engine, out of a 166-operator vocabulary; the bridge route (right) exposes 19 of the 26 native hardware layers authored below the public compiler, among the 45 layer descriptors and 50 per-layer validators the ANE compiler carries [8]. The four operators at the bottom have a lowering on each route.

TABLE 2 | The fused operator surface, by category, and the native bridge operators.

Category	Operators
Convolution, pooling	`conv`, `conv_transpose`, `dynamic_conv`, `linear_activation`, `max_pool`, `avg_pool`, `upsample`
Matrix multiply	`matmul`, `bmm`, `bmm_w`, `einsum`
Activations, rounding	`relu`, `relu6`, `clamped_relu`, `leaky_relu`, `thresholded_relu`, `prelu`, `elu`, `gelu`, `silu`, `sigmoid`, `sigmoid_hard`, `tanh`, `scaled_tanh`, `softplus`, `softsign`, `softmax`, `clip`, `threshold`, `log`, `exp`, `exp2`, `erf`, `ceil`, `floor`, `round`, `sign`
Elementwise arithmetic	`add`, `sub`, `mul`, `real_div`, `adds`, `muls`, `abs`, `inverse`, `maximum`, `minimum`, `pow`, `rsqrt`, `sqrt`, `square`
Comparison, select	`equal`, `not_equal`, `greater`, `greater_equal`, `less`, `less_equal`, `logical_not`, `select`
Normalization	`batch_norm`, `instance_norm`, `layer_norm`, `channel_layer_norm`, `rms_norm`, `group_norm`, `l2_norm`, `local_response_norm`
Reductions	`reduce_sum`, `reduce_mean`, `reduce_max`, `reduce_min`, `reduce_l1_norm`, `reduce_log_sum`, `reduce_log_sum_exp`, `reduce_sum_square`
Shape, structural	`reshape`, `transpose`, `squeeze`, `expand_dims`, `flatten2d`, `slice_by_size`, `concat`, `stack`, `split`, `tile`, `reverse`, `crop`, `pixel_shuffle`, `pixel_unshuffle`, `space_to_depth`, `depth_to_space`
Resize, geometry	`affine`, `resize_nearest_neighbor`, `resize_bilinear`, `upsample_bilinear`
Trigonometric, dtype	`sin`, `cos`, `atan`, `cast`, `const_array`
Attention (composite)	`sdpa`, `mha`, `cross_attention`, `geglu`
Native bridge (graph cut)	`sdpa`, `argmax`, `topk`, `sort`, `fps`, `radius_search`, `cross_product`, `cross_correlation`, `cost_volume`, `minmax_norm`, `lrn`, `space_to_channel`, `channel_to_space`, `space_to_batch`, `batch_to_space`, `dynamic_slice`, `input_view`, `flatten`, `scaled_elementwise`

Note: Arithmetic, comparison, matrix-multiply, and reduction operators are also reached through the usual Python operators and methods on tensors (`+`, `@`, `.mean`, and so on). `af.tune` can move `sdpa`, `minmax_norm`, `lrn`, and `flatten` from the bridge into the fused program (Section 3.2).

one of its 50 per-layer validator classes [8]. The remaining seven authored layers are recorded in the registry with their evidence but not promoted, because the fused route already covers them or they have no inference-time use. Four bridge operators have an equivalent fused lowering, and the optimizer of Section 3.4 picks one of the two per node. On the fused route, `sdpa` lowers to the `sdpa` composite, `minmax_norm` to a composite of `reduce_min`, `reduce_max`, `sub`, `adds`, and `real_div`, `lrn` to `local_response_norm`, and `flatten` to `reshape`.

3.3 | Compilation and Dispatch

Figure 3 shows the path from a graph to the hardware, and Listing 1 writes it as code, one call per stage, with the implementing module named in each comment. Building a graph runs nothing; `af.compile` lowers it and packs the weights into one program, and calling that program dispatches it to the engine. The optimizer and the trainer are entered through one call each, `af.tune` and `af.Trainer`, and use this compile-and-dispatch

10970024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.70124, Wiley Online Library on [29/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

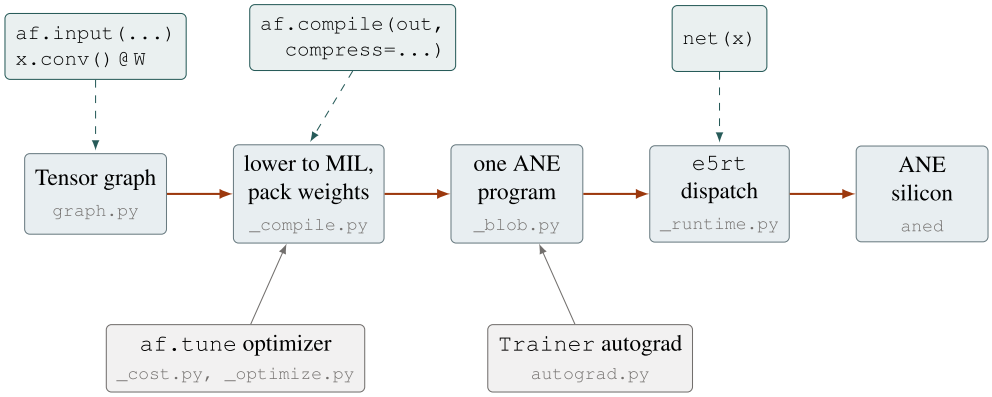


FIGURE 3 | The compilation and dispatch path: The Python the caller writes (top, green) drives each pipeline stage (middle, with the package module that implements it), and the supporting modules (bottom) feed compilation. A small Objective-C++ shim (`_lib/*.mm`) carries the `e5rt` dispatch into the `aned` daemon.

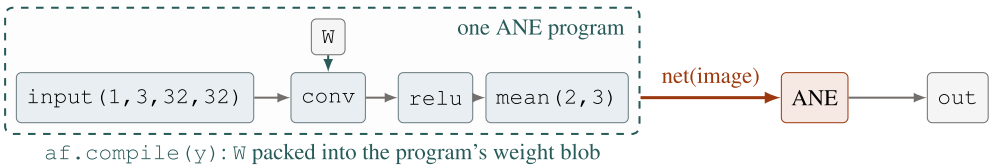


FIGURE 4 | How Listing 1 evaluates. `input`, `conv`, `relu`, and `mean` are the operator nodes of the lazy graph that lines 1–2 build; building runs nothing. `W` is the convolution's weight tensor, packed into the program's weight blob. Each program call `net(image)` is one dispatch to the engine, which returns the output array `out`.

path. Figure 4 traces the listing's first four lines: The lazy graph they build, the single fused program `af.compile` produces from it, and the one dispatch each call makes.

LISTING 1 | The pipeline of Figure 3 in code, one stage per line, with the module that implements each.

```
import aneforge as af

x   = af.input((1, 3, 32, 32))
      # graph.py    : a lazy Tensor graph node
y   = af.conv(x, W).relu().mean((2, 3))
      # graph.py    : ops extend the graph
net = af.compile(y, compress="int8")
      # _compile.py : lower to MIL  (+ _blob.py:
      #               pack weights -> one blob)
out = net(image)
      # _runtime.py : dispatch to aned / ANE
      #              (handle reused across calls)

fast = af.tune(y)
      # _cost.py + _optimize.py : af.tune
       optimizer
trainer = af.Trainer(loss, params, lr=1e-3,
    device_optimizer=True)  # autograd.py :
    Trainer
```

3.4 | Cost Model, Optimizer, and Autograd

Three modules operate over the two routes. A structural cost model estimates the time of each node from its shape, dtype, and a calibrated per-architecture table. Section 5.5 measures how well the ranking agrees with the hardware. The optimizer `af.tune` uses that estimate to rank the equivalent lowerings of a graph (route flips between the fused and bridge routes, global or per-weight int8, and constant folding), prunes a lossy candidate whose estimate exceeds 1.5 times the best estimate, measures the survivors on the engine within a budget of eight, validates each against the unoptimized baseline, and caches the result keyed by structure and shape. A reverse-mode autograd module emits ordinary tensor operators for each gradient, so the backward pass and the optimizer update compile to ANE programs and run on the engine alongside the forward pass.

4 | Functionality

The package provides the following capabilities, each tested on Apple Silicon against a NumPy or framework reference.

Graph compilation and dispatch: `af.compile` lowers a graph to one ANE program and returns a callable; `af.input` and `af.image_input` declare float and uint8 input ports, the latter emitting an on-engine dequantization so raw camera or decoded-video bytes feed the model directly.

Weight compression: `af.compile(…, compress=)` streams int8, int4 lookup-table, or unstructured-sparse weights from the engine's own dequantization path rather than folding them to half precision at compile time; the weight footprint drops by about 4 times for int4 and 2 for int8 (a 4096-square matmul's weight blob shrinks from 33.6 to 8.4 MB). Compression is off by default, and the default path's program is byte-identical to plain half precision.

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.70124, Wiley Online Library on [29/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Native attention: `af.sdpa` targets the engine's dedicated fused-attention layer, for causal attention, the single-query decode shape over a cached key and value, and a runtime additive mask. The public toolchain has carried a scaled-dot-product-attention operator since coremltools 8 [9], but Apple's MIL compiler decomposes it into matmul, softmax, matmul and never emits the native fused layer, which `af.sdpa` reaches directly.

Resident on-engine state: Two kinds of state stay resident on the engine across steps through output-to-input buffer aliasing: A zero-copy key-value cache for autoregressive decoding, and optimizer state for training. The host feeds only the new input each step and reads results back at checkpoints. CoreML reaches key-value-cache residency through its sanctioned stateful-model interface (`MLState`, macOS 15 and later) [10]; the buffer aliasing here is lower-level and carries optimizer state over that mechanism.

On-engine training: The autograd module runs the forward pass, the backward pass, and the optimizer update on the engine; a fused softmax-cross-entropy uses the analytic, half-precision-stable gradient, and the differentiable operator set spans the structural and linear-algebra operators, the common activations, the unary math operators, and the layer, RMS, and group normalizations.

Pretrained model loaders: `af.load` imports a sentence encoder, `af.load_resnet18` a torchvision classifier with batch normalization folded at load, and helpers cover a Vision Transformer and a Stable Diffusion U-Net and VAE decoder. Further loaders cover Llama- and Qwen-family decoders with resident key-value caches (since v0.2.0) and the Whisper encoder and decoder (since v0.4.0). We do not evaluate these loaders here.

Numerical and spectral methods: `aneforge.linalg` and `aneforge.fft` run fixed-iteration Krylov solvers, Cholesky and eigenvalue routines, and a factored fast Fourier transform as static-dataflow graphs, so dense linear algebra and spectral methods run on the engine at a program size independent of the problem dimension. Unit tests bound the relative error of the Cholesky reconstruction by 5×10^{-3} and of the symmetric eigenvalues by 1.5×10^{-2} at $n = 8$. We report no other measurement of these two routines.

Precompile validation: `af.compile(…, validate=True)` runs the ANE compiler's own per-layer validators (the 50 of Figure 2) on a graph before compiling it, rejecting an invalid shape or dtype much faster than a round-trip compile would.

5 | Evaluation

This section reports what the package measures on one host. Section 5.1 states the configuration and the energy accounting, Section 5.2 the pretrained models, Section 5.3 ResNet-18 on all three compute units at matched precision, Section 5.4 the dispatch floor and what it bounds, Section 5.5 the attention rewrite and the cost model behind it, Section 5.6 training, decode, and numerical kernels, and Section 5.7 the rows that were repeated on two older engine generations.

5.1 | Measurement Configuration

All timings and energies in this section were taken on an Apple M5 Pro under macOS 26.6.2, with ANE compiler 9.509.0 and Espresso 3525.1.1, running ANEForge v0.4.0; each regenerates from a script in the paper's experiments/ directory or from the example that reports it. Latency is the minimum end-to-end call time over 30 repetitions after warmup, or over 50 for the dispatch-floor probe of Section 5.4. Energy per inference is idle-subtracted active package power, the CPU, GPU, and ANE rails as `powermetrics` reports them at 500 ms intervals over a sustained 6 s window, multiplied by the mean per-call time inside that window; the two quantities come from one measurement, and each arm is repeated eight times, interleaved. GPU baselines are torchvision on Metal Performance Shaders (torch-MPS) in fp32 and fp16 and a graph-compiled MLX port in fp16; CPU baselines are torchvision in fp32, fp16, and bf16.

5.2 | Pretrained Models

The build-then-compile-then-call pattern of Listing 1 runs unchanged on the pretrained networks of Table 3. A pretrained vision model runs in two lines, and one keyword switches its weights to a streamed, on-engine-dequantized encoding (Listing 2). The `compress` argument selects int8, int4 lookup-table, or sparse weights, accuracy-gated against a tolerance and falling back to a coarser encoding or to half precision when a layer does not meet it (Section 4).

TABLE 3 | Pretrained models run end-to-end on the engine, each fused into one program and validated against its reference framework.

Model	Ops → program	Agreement vs. reference	Reference
ResNet-18	49	cosine 1.0000, top-1 match	torchvision
ViT-B/16	288	cosine 1.0000, top-5 5/5	torchvision
MiniLM (L6) encoder	144	cosine 1.0000	`transformers`
Stable Diffusion U-Net (one forward)	308	relerr 0.041	`diffusers`
Stable Diffusion VAE decoder (one forward)	72	relerr 0.021	`diffusers`

Note: The classifiers and the encoder report cosine similarity of the output against the framework's float32 output and, for the classifiers, label agreement; the two diffusion components validate one forward pass and report relative error, because a full classifier-free-guidance sampling loop does not close in half precision (Section 7).

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.70124, Wiley Online Library on [29/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 4 | ResNet-18 ($1 \times 3 \times 224^2$) on the three compute units of an M5 Pro. Latency is the minimum over 30 calls.

Unit	Precision	Latency (ms)	Sustained (ms)	Power (W)	Energy (mJ/inf)	Energy ratio	Relative error
ANE (ANEForge)	fp16	0.334	0.341	6.89	2.34 ± 0.16	1	2.3×10^{-3}
GPU (torch-MPS)	fp16	1.922	2.467	9.13	21.8 ± 6.1	9.3	1.7×10^{-3}
GPU (MLX, compiled)	fp16	1.832	1.875	18.40	34.5 ± 1.4	14.7	—
GPU (torch-MPS)	fp32	1.658	2.295	11.65	25.6 ± 7.5	10.9	7.0×10^{-7}
CPU (torch)	fp32	5.934	6.205	17.12	106.3 ± 1.6	45.3	0 (reference)
CPU (torch)	fp16	511.7	—	—	—	—	1.8×10^{-3}
CPU (torch)	bf16	505.1	—	—	—	—	1.5×10^{-2}

Note: The sustained column gives the mean call time inside a 6 s power window, over which idle-subtracted active package power (0.88 W idle) is sampled. Energy is their product, given as mean and standard deviation over eight interleaved repetitions. Relative error is against the torchvision fp32 CPU output, and every arm returns the same top-1 label; the MLX port carries timm weights, so its output is not compared. The two slow CPU arms were not sampled for energy.

LISTING 2 | Pretrained inference and on-engine weight compression.

```
import aneforge as af

clf = af.load_resnet18()
      # torchvision weights, BN folded
label = clf(image)[0].argmax()
        # one fused ANE program

# 4x smaller weights, streamed and dequantized on
     the engine, accuracy-gated:
clf4 = af.load_resnet18(compress="int4",
    compress_atol=0.05)
```

5.3 | ResNet-18 on Three Units at Matched Precision

Table 4 reports the pretrained ResNet-18 forward of Table 3 on the engine, the GPU, and the CPU at fp32 and fp16, with energy per inference for each case. At fp16 the GPU runs the forward in 1.92 ms under torch-MPS and 1.83 ms under graph-compiled MLX, against 0.334 ms on the engine, a latency ratio of about 5.5–5.8; at fp32 torch-MPS is faster than its own fp16 path, 1.66 ms, and the ratio is 5.0. The precision match therefore changes the GPU comparison by less than a factor of 1.2, since, ResNet-18 at batch one is latency-bound on the GPU. On the CPU, PyTorch's fp16 and bf16 convolution paths run the forward in 512 and 505 ms, about 80 times slower than its fp32 path at 5.9 ms, so the fp32 row is the fairer CPU baseline.

The engine completes a forward in 2.34 mJ, drawing 4.43 W on the ANE rail and 2.43 W of host CPU during the dispatch loop, 6.9 W of package in all; torch-MPS fp16 draws 21.8 mJ, torch-MPS fp32 25.6 mJ, compiled MLX 34.5 mJ (faster, but at 15 W on the GPU rail), and the fp32 CPU 106 mJ. The engine thus uses about a factor of 9 less energy than torch-MPS at matched fp16 precision, 15 less than compiled MLX, and 45 less than the CPU at fp32. These factors divide mean energies, and the torch-MPS fp16 energy, 21.8 ± 6.1 mJ, varies most. Pairing each engine repetition with the torch-MPS fp16 repetition run next to it gives ratios from 5.5 to 12.7, median 8.5, against 13.4–16.0 for compiled MLX and 41.5–48.5 for the fp32 CPU. The sustained time that energy uses is within 5% of the minimum latency for every arm except torch-MPS, where it is longer by a factor of 1.28 at fp16 and 1.38 at fp32.

5.4 | The Dispatch Floor

Every call to the engine pays a fixed cost for the host-to-daemon submission and the firmware round trip, whatever the size of the work, and dispatch is single-in-flight, so concurrency does not amortize it. The size of that floor sets the smallest call for which the engine pays, and it is measured three ways in Figure 5.

A chain of n small 1×1 convolutions fused into one program runs in about 74 µs whether n is 1 or 128; the linear fit has an intercept of 74.0 µs and a slope indistinguishable from zero. The floor is paid once per program, and the 128 operators inside it cost nothing measurable at this size. A single `relu` over 2^k fp16 elements runs in about 77 µs (median) from 32 B to 128 KiB, after which the host transfer term appears and the latency reaches 1.75 ms at 16 MiB, a rate of about 9.6 GB/s for the copy into and out of the bound buffers. In a sustained loop the per-call time is about 2.6 times the minimum-latency probe (195 µs against 74 µs); the latencies of Table 4 are probe minima. The companion guide reports an M1 floor of 0.19–0.23 ms per call [8]. The lower figure is the mean of a hot loop over a small convolution graph with host copies, and the upper comes from a slope fit. About 98% of the M1 floor is dispatch overhead, 130 µs of it the firmware round trip, and a fused stack there stays flat from one layer to thirty-two.

That `relu` on the other two units sets the crossover. On the CPU (NumPy) it completes in under 1 $mathrm{\mu s}$ at the smallest sizes and crosses the engine's floor at 64 KiB; on the GPU (MLX) it pays a floor of its own, $130 - -150$ µs, which the engine beats below 512 KiB and loses to above it, because the GPU's transfer term is smaller. For a single memory-bound operator the engine is thus not the fastest unit at any size tested; the CPU finishes first below 64 KiB and the GPU above 512 KiB, so the engine's advantage comes from fusion. A program the cost model predicts to be floor-bound should do more work per call, by increasing the batch or by chaining more operators into one program, and the package warns about this at compile time.

5.5 | The Attention Rewrite and the Cost Model

The optimizer rewrites an attention block by folding its query, key, value, attention, and output projections into one fused

10970024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.70124, Wiley Online Library on [29/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

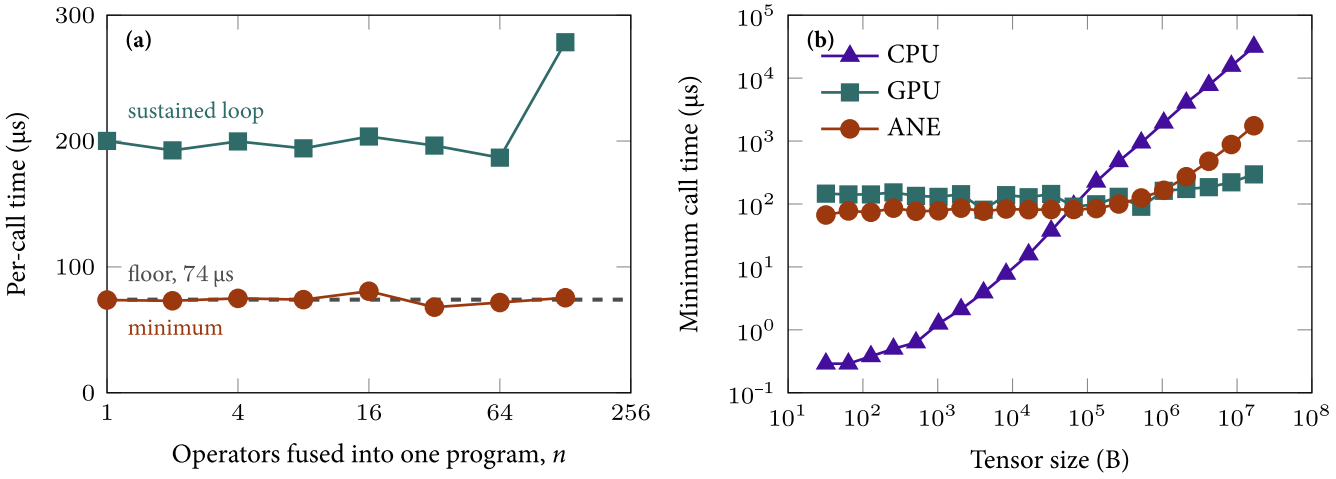


FIGURE 5 | The per-program dispatch floor on the M5 Pro. (a) Minimum (over 50 calls) and sustained per-call time of a chain of n small 1×1 convolutions fused into one program, with the intercept of the linear fit, 74.0 µs, dashed. (b) Minimum time of one `relu` over 2^k fp16 elements on the three units, as labeled.

TABLE 5 | The attention rewrite across sequence lengths S on the M5 Pro.

Heads	S	Head dim	Unoptimized (µs)	Tuned (µs)	Speedup
8	16	64	498	104	4.78×
8	32	64	492	92	5.34×
8	64	64	503	104	4.82×
8	128	64	615	125	4.93×
8	256	64	844	220	3.83×
8	512	64	605	607	1.00× (native kept)
8	1024	64	1746	1747	1.00× (native kept)
12	197	64	917	226	4.05×

Note: Latency is the minimum over 30 calls; the tuned column is the lowering `af.tune` selected, and "native" means it kept the unrewritten graph. The tuned output matched the unoptimized output to a relative max-abs difference of 0.0000 in every row.

TABLE 6 | The cost model as a pruning heuristic on the M5 Pro.

Graph	Variants	Pruned	Optimum pruned	Tuner's choice
ResNet-18	2	0	No	fp16 (measured optimum)
MiniLM encoder ($S = 32$)	2	0	No	fp16 (int8 variant rejected on accuracy)
Attention, eight heads, $S = 32$	3	1	No	fused rewrite (measured optimum)
Attention, eight heads, $S = 256$	3	1	No	fused rewrite (measured optimum)
Attention, 12 heads, $S = 197$	3	1	No	fused rewrite (measured optimum)
CIFAR-10 CNN ($B = 128$)	1	0	No	fp16 (only variant)

Note: Every legal lowering of each graph was measured; the tuner's rule (prune a lossy candidate estimated at more than 1.5 times the best estimate, measure at most eight) was then replayed against the measurements.

program, a transformation verified bit-identical to the unoptimized graph. Table 5 reports the rewrite across sequence lengths for an eight-head block with 64-dimensional heads, and for the attention shape of ViT-B/16. For sequence lengths up to 256 the rewrite runs about 3.8–5.3 times faster, because the unrewritten block is five dispatches deep and each pays the floor of Section 5.4. At 512 and 1024 the tuner keeps the native route, since the attention itself then accounts for most of the block's cost and the graph cut no longer costs a measurable fraction; the measured speedup of the rewrite is 1.00. On the ViT-B/16 shape the rewrite runs 4.05 times faster. The whole pass is one call: `af.tune(out)` prunes the equivalent lowerings of a graph with the cost model, measures the survivors on the engine, caches the result keyed by structure and shape, and validates each against the unoptimized baseline, returning the lossless selection by default. An explicit tolerance, `af.tune(out, atol=0.1)`, admits an int8 rewrite only when it also clears a speedup margin.

Table 6 tests the cost model as a pruning heuristic. For each benchmark graph, every legal lowering was compiled and

10970724x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.70124, Wiley Online Library on [29/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

measured with no pruning and no budget, and the tuner's rule was then replayed against the full table. The variant space is small, one to three lowerings per graph, because the legal variants are the route flips and the global int8 rewrite. In the three attention graphs the rule pruned the global-int8 variant, and the measurement confirms it was slower than the route flip the model ranked first; in no graph was the measured optimum pruned, and the tuner's selection was the measured optimum in every case. The estimate is coarse in absolute terms since it carries no dispatch term; the rewritten attention block, for example, is estimated at 63 µs and measures 106 – −227 µs. The model's ranking agreed with measurement in every graph with more than one variant. The calibrated table is one curve per ANE architecture, fourteen at this release (H11 through H17s), interpolated from recorded sweeps, so on another chip the model uses that chip's curve and the M5 Pro table is not applied elsewhere. Whether that curve ranks as well on another chip was not measured.

5.6 | Training, Decode, and Numerical Kernels

Listing 3 trains a convolutional network on the engine. The convolution is assembled from primitives so its weight is a graph parameter, and the group normalization is built from primitives so its affine trains at any batch size; the forward pass, the backward pass, and the Adam update all run on the engine. The network reaches 71.2% test accuracy on CIFAR-10, within 1.7 points of a PyTorch model of the same layer topology at 72.9%; the convolution and group-normalization gradients match PyTorch to 1.0 cosine similarity.

LISTING 3 | Train a convolutional network forward, backward, and optimizer step on the engine.

```
import aneforge as af

x, logits, params = af.cifar_cnn(batch=128)
      # conv -$>$ GroupNorm -$>$ relu -$>$ pool
target = af.input((128, 10))
obj = af.softmax_cross_entropy(logits, target)
      # fp16-stable analytic gradient
tr = af.Trainer(obj, params, lr=3e-3, optimizer="
    adam",   # fwd + bwd + Adam on ANE
                device_optimizer=True,
    data_inputs=first_batch)
for images, labels in batches:
      # host feeds only data + lr
    tr.data[x], tr.data[target] = images, onehot(
    labels)
    tr.step()
      # one on-engine training step
acc = tr.accuracy(test_images, test_labels)
      # 71.2% (PyTorch peer: 72.9%)
```

The single-query decode shape lets a decoder run autoregressively on the engine (Listing 4): One new query attends over a cached key and value. The shipped example `examples/gpt_generate_ane.py` wraps this primitive in a full decode loop whose key-value cache stays resident on the engine across steps. The host feeds only the new token and a position mask, the cache never returns to the host, and the output matches a NumPy reference token for token.

LISTING 4 | Single-query decode-shape attention, the public primitive behind on-engine autoregressive decode.

```
import aneforge as af

# one new query (seq_q = 1) attends over the
    cached keys and values of length S
a = af.sdpa(q.reshape(1, H, 1, dh),
    # the new token's query
            Kc.reshape(1, H, S, dh),
            # cached keys
            Vc.reshape(1, H, S, dh))
            # cached values
# examples/gpt_generate_ane.py wraps this in a
    full loop whose KV cache stays
# resident on the ANE across steps (share_buffer)
    ; host feeds only the new
# token + a position one-hot, and the output
    matches numpy token for token.
```

The engine has no data-dependent control flow, but a fixed-iteration numerical method is static dataflow and compiles to one program whose size does not depend on the problem dimension. The package runs a fixed-iteration Krylov solve for symmetric positive-definite and general systems on the engine, and a fast Fourier transform staged as a product of factors and carried as real and imaginary pairs (Listing 5). The test suite checks both against NumPy in double precision. The solve must reach a relative error of 5×10^{-3} at $n = 48$ and condition number 10 (5×10^{-2} at condition number 100), and the two-dimensional transform 2×10^{-2} at sizes up to 64×64.

LISTING 5 | Dense linear algebra and spectral methods on the engine.

```
from aneforge.linalg import conjugate_gradient
from aneforge.fft import fft as ane_fft

x = conjugate_gradient(A, b, iters=40)
    # fixed-iteration CG; same program at any n
Xre, Xim = ane_fft(sig_re, sig_im, N)
    # factored DFT; matches np.fft to fp16
```

The repository's `examples/` directory contains runnable versions of these demonstrations, and the operator corpus is the project's correctness gate. The operator counts and reference agreements of Table 3 are emitted by the corresponding scripts (examples/resnet18.py, vit.py, sentence_embeddings.py, sd_unet.py, sd_vae.py), the attention-rewrite speedups and the pruning audit by experiments/attention_sweep.py and experiments/cost_model_pruning.py, and the three-unit and dispatch-floor measurements by experiments/resnet18_three_units.py and experiments/dispatch_floor.py, so each number regenerates on Apple Silicon.

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.70124, Wiley Online Library on [29/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 7 | Rows of the repository's single-stream device map repeated on three engine generations under one harness (the script and the committed result files are named in Section 5.7).

Workload	M1 Max	M2 Pro	M5 Pro
GEMM floor ($M = 64$, $K = 256$, $N = 256$), ms	0.164	0.110	0.092
Convolution stack ($C = 256$, 32^2, 3×3, depth 16), ms	1.918	1.265	0.757
Attention ($S = 197$, $D = 768$, 12 heads), ms	0.613	0.589	0.230
Layer norm (197×768), ms	0.321	0.283	0.195
ResNet-18 forward, ms	—	1.064	0.330
ResNet-18 forward, mJ/inf	—	2.64	2.20

Note: Latency is the minimum over 30 calls on the engine (15 for ResNet-18), and energy per inference follows Section 5.1 over a single 6 s window. The rows were recorded in June 2026 with a pre-release version of the package. Table 4 reports 2.34 mJ for ResNet-18, an average of eight windows under v0.4.0.

5.7 | Across Engine Generations

The timings above come from one M5 Pro, and the package is verified on an M1 Max only by the operator corpus. The repository's single-stream device map, a fixed set of synthetic workloads the bench tooling runs on each host (bench/device_compare_wattcomplete.py), was however run on an M1 Max, an M2 Pro, and the M5 Pro under one harness, with the results committed as bench/results/device_compare_wattcomplete_results_M{1,2,5}.json, and Table 7 reproduces the rows that bear on this paper. On the M1 Max, M2 Pro, and M5 Pro, the floor row takes 164, 110, 92 µs, the ResNet-like convolution stack 1.92, 1.27, 0.76 ms and the ViT attention shape 0.61, 0.59, 0.23 ms. In the same harness the M2 Pro runs the pretrained ResNet-18 forward in 1.06 ms at 2.6 mJ per inference, and the M5 Pro in 0.33 ms at 2.2 mJ. The map ranks the units the same way for each workload on all three hosts, with multipliers that change by generation. From the M1 Max to the M5 Pro the floor row falls by a factor of 1.8 and the convolution stack by a factor of 2.5. Apple lists a 16-core Neural Engine for both chips. The ANE compiler, which assigns output channels across engine units, plans for eight units on the M1 Max target (H13G) and 16 on the M5 Pro target (H17s), with cost-model clocks of 1.14 and 1.89 GHz [8]. The floor row times one $64 \times 256 \times 256$ matmul that moves 32 KiB each way, against 2 KiB for the convolution chain of Section 5.4, and on the M5 Pro the `relu` probe puts the cost of that larger transfer at about 8 µs. The other 10 µs of the gap between 74 and 92 µs was not isolated. The timings of Sections 5.3–5.5 are thus single-host results, and Table 7 is the extent of what was repeated elsewhere.

6 | Stability of the Private Surface

The package calls private, undocumented Apple symbols that carry no API contract and can change between operating-system releases. This section reports what has been observed across releases.

Each release records the macOS and ANE-compiler versions it was verified against, and the operator corpus is run on the engine before a release ships. The corpus has 80 small graphs of one to ten operators, and ten of them are also compiled with int8 weights, for 90 variants in all. A variant passes when it compiles, runs on the engine, and matches a NumPy reference. The graphs use 67 of the 100 fused operators and the bridge operators `sdpa` and `argmax`. Separate on-engine tests compare the two lowerings of `sdpa`, `minmax_norm`, `lrn`, and `flatten`. Dispatch is verified on the M5 Pro and the M1 Max under macOS 26.5 (ANE compiler 9.509.0, Espresso 3520.4.1) and on the M5 Pro under macOS 26.6.2, where the Espresso component had moved to 3525.1.1 and the compiler had not, with the operator corpus (90 of 90 variants) and every measurement of Section 5 passing on the newer release. The package requires macOS 14 or later, and on macOS 14 and 15 the verification is narrower. Hosted continuous-integration runners virtualize macOS without exposing the engine, so since July 2026 a weekly scheduled job builds the dispatch shim, imports the package, and runs every hardware-free test on macOS 14, macOS 15, and the newest hosted image, and every one of its six runs to date has passed on all three. That job shows the shim still links and the package still imports on those releases; it does not dispatch, and the corpus gate on macOS 14 or 15 hardware has not been run.

Across the releases the package has run on, only one dispatch symbol changed. macOS 26 introduced e5rt_execution_stream_submit_async and deprecated the earlier asynchronous submit, and the shim resolves whichever is present. The compiler and the silicon have broken more often than the symbols. A differential fuzzer that compiles random graphs on the engine and compares them against NumPy found six cases in which Espresso accepts a program and miscompiles it, each now carried as a workaround in the lowering with a regression test. Five further behaviors differ silently between hosts or releases and are recorded as pitfalls with their probes: A slice kernel keyed to the compiler target that differs between M1 and M5, saturation of slices above 4094 elements on M1, dynamic-weight convolution at batch two or more, a compile-service restart of about 15 s, and four-character-code image input [8]. The package is for research and local development, since, software linking private frameworks cannot ship through the App Store.

7 | Limitations

The package presented here has limitations. The route is interoperability research on hardware the user owns, not a circumvention. The package carries no Apple entitlement, leaves code signing and system-integrity protection in place, and redistributes no Apple material, with the dispatch shim compiled on the user's machine. An archived source capsule accordingly preserves the source, the corpus, and its expected

10970024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.70124, Wiley Online Library on [29/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

outputs. Executability requires an Apple Silicon machine within the verified range, because the shim links the operating system's resident private frameworks and the program binary comes from its ANE compiler.

The compute path is half precision; accumulation error stays acceptable for the validated models, but half precision sets the dataplane's numerical reach. A full classifier-free-guidance diffusion sampling loop, for example, does not close in half precision: The guided difference it amplifies is a fraction of a percent of the signal, below the half-precision step error.

On the machines tested the engine exposes a single hardware lane, so a host dispatch drives one program at a time and the throughput gains come from fusing work into one program, not from concurrent dispatch. ANEForge is not a general tensor library. A graph that uses an operator outside the operator surface fails at compile time. The timings of Section 5 are single-host results, and Section 5.7 bounds what was repeated on other generations.

8 | Conclusions

ANEForge gives Python direct access to the Apple Neural Engine through Espresso, with the compute unit fixed and no entitlement. The engine's operator census is narrower than its compiler's vocabulary, 116 of 166 MIL operators and 26 native layers below the public compiler, and the per-program floor of about 74 µs sets the smallest unit of work for which a dispatch pays. A single memory-bound operator finishes sooner on the CPU below 64 KiB and on the GPU above 512 KiB, so the engine's advantage comes from fusion; a program of 128 operators pays the floor once. On ResNet-18 the engine uses about a factor of 9 less energy per inference than the GPU at matched precision, with paired runs from 5.5 to 12.7. A fixed-iteration method compiles to the engine if its program size is independent of the problem dimension and it uses only operators from the operator surface. Dense Krylov solvers and a factored Fourier transform qualify, and half precision limits how ill-conditioned a problem they can solve.

Author Contributions

Spencer H. Bryngelson: conceptualization, methodology, software, validation, formal analysis, investigation, data curation, visualization, writing – original draft, writing – review and editing.

Funding

The author has nothing to report.

Conflicts of Interest

The author declares no conflicts of interest.

Data Availability Statement

ANEForge v0.4.0 is available under the MIT license at https://github.com/sbryngelson/ANEForge, with documentation at https://aneforge.readthedocs.io. The version this paper describes is archived at https://doi.org/10.5281/zenodo.20672609. The package is written in Python and Objective-C++ and requires an Apple Silicon Mac running macOS 14 or later, Python 3.10 or later, and NumPy; ANE dispatch is verified on macOS 26 (Section 6).

References

1. M. Hollemans, "The Neural Engine — What Do We Know About It?," (2020), https://github.com/hollance/neural-engine.

2. Apple Machine Learning Research, "Deploying Transformers on the Apple Neural Engine," (2022), https://machinelearning.apple.com/research/neural-engine-transformers.

3. Apple Inc., "Core ML," (2024), https://developer.apple.com/documentation/coreml.

4. G. Hotz and Tinygrad Contributors, "Tinygrad Apple Neural Engine Backend (ACCEL/ANE)," (2021), https://github.com/tinygrad/tinygrad.

5. E. Yoon, "ANE: Reverse-Engineered Linux Driver for the Apple Neural Engine (+ANECC)," (2022), https://github.com/eiln/ane, https://github.com/eiln/anecc.

6. M. Singh, "Inside the M4 Apple Neural Engine, Part 1: Reverse Engineering," (2026), https://maderix.substack.com/p/inside-the-m4-apple-neural-engine, repo, https://github.com/maderix/ANE.

7. R. Kumaresan, "Orion: Characterizing and Programming Apple's Neural Engine for LLM Training and Inference" 2026.

8. S. H. Bryngelson, "The Apple Neural Engine: A Complete Guide," 2026 arXiv preprint arXiv:2606.22283.

9. Apple Inc., "Core ML Tools (coremltools)," (2024), https://github.com/apple/coremltools.

10. Apple Machine Learning Research, "On Device Llama 3.1 With Core ML," (2024), https://machinelearning.apple.com/research/core-ml-on-device-llama.

10970024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.70124, Wiley Online Library on [29/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License