TECHNICAL WHITE PAPER · TRITON-METAL

triton-metal: A Triton Compiler Backend for Apple Metal

A Swift core behind a C ABI that lowers Triton IR to readable Metal Shading Language: unmodified `@triton.jit` kernels running on a Mac GPU through the pinned Triton v3.7.1, a GEMM at 75–76% of `MPSMatrixMultiplication`, FlashAttention-2 in both directions with verified gradients — and a measured account of which CUDA optimisation techniques transfer to Apple silicon and which invert.

TRITON v3.7.1 | SWIFT / C ABI | METAL SHADING LANGUAGE | FLASHATTENTION-2 FWD + BWD | f16 / bf16 / f32 | MLX ZERO-COPY

ABSTRACT

Triton is how the machine-learning community writes custom fused GPU kernels — FlashAttention variants, MoE routing, quantization kernels — without hand-writing CUDA, and it emits code for CUDA and ROCm and nothing else. That gap is one of the more durable pieces of CUDA's portable-code moat: a large body of research code does not run on Apple silicon at all. This paper describes triton-metal, an out-of-tree compiler backend that lowers Triton IR to Metal Shading Language. All compiler logic is Swift behind a C ABI; the Python package is a ctypes shim that computes nothing. The backend is pinned to Triton v3.7.1 and driven by it — real `@triton.jit` source, Triton's own frontend and MLIR passes, a Mac GPU, no patch to Triton — and ships as one self-contained wheel verified into a fresh virtualenv on a second machine. The lowered matmul tutorial reaches 76% of `MPSMatrixMultiplication` at 1024, 2048 and 4096 square on an M1 Max and 75% on an M1 Pro, up from ~33% at the first working version; FlashAttention-2 forward *and* backward run, and an attention layer written as `@triton.jit` source trains on a Mac GPU with gradients checked twice over. The evaluation is deliberately unflattering where the results are: it reports the negative optimisation results beside the positive ones, names the register file rather than memory traffic as what stands between this kernel and Apple's, and scopes every measurement to M1-generation silicon.

01 Introduction

The interesting property of Triton [1] is not that it generates fast kernels. It is that a Triton kernel is *portable source*: `tl.load`, `tl.dot`, `tl.reduce` and a block-programming model, with no mention of warps, shared-memory banks, or tensor-core fragment layouts. That abstraction is what lets an attention variant be published as forty lines anyone can read, modify, and run — where "anyone" means anyone with an NVIDIA GPU, or lately an AMD one. A researcher on an Apple laptop cannot run the kernel at all, and the workaround is to rewrite it, forking a kernel that was supposed to be portable.

Apple silicon is not a marginal target for this. Unified memory removes the staging copy that dominates small-kernel latency on discrete GPUs; simdgroups are 32 lanes wide, exactly matching a CUDA warp, so `num_warps` maps across without reinterpretation; and threadgroup memory plays the role of CUDA shared memory closely enough that the block model survives intact. The mismatches are bounded: no cross-threadgroup barrier, different atomic dtype coverage, and 8×8 `simdgroup_matrix` operations in place of CUDA's mma fragment shapes.

The goal is narrow, and stated as a negative: make *existing* Triton kernels run unmodified on Metal — not a Metal DSL that resembles Triton, but the same IR, so the upstream file stays the only copy. Three claims organise what follows. The integration is real rather than approximate: the backend attaches through Triton's documented plugin interface, is pinned to v3.7.1 — the release `pytorch/pytorch` release/2.12 pins

— and needs no patch to Triton. The performance is credible on arrival, because a portability story running at a fraction of vendor speed does not get used. And the port produced a result the CUDA literature does not contain: a measured account of which optimisation techniques transfer to M1-generation Apple silicon and which invert, with the mechanism behind every inversion.

02 Scientific Background

2.1 The Block Model, and Why CUDA's Layout System Does Not Transfer

Triton's frontend compiles a decorated function to `ttir`, an MLIR dialect [2] in which tensors are typed values, loads and stores are masked, and the parallel decomposition is a grid of programs rather than of threads. Everything below that — which lane holds which element, where an operand is staged, how a reduction folds — is the backend's problem, which is exactly why the same source can target different hardware. On CUDA, Triton lowers `ttir` → `ttgir` → LLVM IR → PTX, and the `ttgir` layer carries TritonGPU layout encodings that pin down which lane holds which tensor element; on those targets the mapping from tensor elements to registers *is* the optimization.

A Metal backend cannot reuse those encodings, and the reason is what the hardware exposes. CUDA's mma layouts describe a distribution of a matrix fragment over named lanes; Metal's `simdgroup_matrix` deliberately hides it. A `simdgroup_float8x8` is a per-simdgroup object whose 64 elements sit in an opaque arrangement across 32 lanes, populated by `simdgroup_load` from memory rather than assembled from registers the programmer named. Two consequences follow, running in opposite directions. The backend cannot shuffle fragments between lanes, because it does not know where a fragment's elements live — which is why `tt.dot` stages operands through threadgroup memory. But *which* fragment a simdgroup works on becomes an emitter decision rather than a property of a layout encoding, and §4.6 reports that this freedom was worth more than any technique ported from CUDA. What does transfer is everything above `ttgir`: Triton's frontend, inliner, canonicalizer, CSE and loop unrolling.

Two neighbouring systems solve adjacent problems rather than this one. **MLX** [3] is Apple's array framework — a NumPy-shaped API with unified-memory arrays and hand-written Metal kernels underneath — and it is the pragmatic answer today for training something on a Mac, but an MLX version of an attention variant is a *rewrite* of the Triton kernel rather than an execution of it. **MPS** is Apple's vendor library, hand-tuned for the operations Apple chose to tune; `MPSMatrixMultiplication` is the right yardstick for a GEMM and is used as such in §4.5, but a vendor library is the opposite of a custom-kernel story — it is fast at exactly the operations someone already implemented, which is why fused kernels exist at all.

2.2 Numerics: The Default That Silently Changes Answers

The single most consequential fact found in building this backend is that **Metal defaults fast math ON**, and it is easy to miss because everything still runs. With it enabled, `exp`, division and denormal handling drift from an IEEE reference — enough to fail a tolerance-checked comparison against a CPU implementation, and enough that a numerically careful kernel such as an online softmax stops being one. The backend sets `MTLCompileOptions.mathMode = .safe`, lowers `math.*` to Metal's `precise::` namespace wherever one exists, and never emits `fast::`. Metal has no `erf` at all, so it lowers to a generated helper using the Abramowitz & Stegun 7.1.26 approximation [4], accurate to ~1.5e-7 absolute. The payoff is measurable: the fused softmax's worst absolute error against a CPU reference over twelve rows of one hundred columns is **7.45e-08**, about one float ULP.

2.3 Online Softmax and FlashAttention-2

The kernels that decide whether a Triton backend is useful beyond elementwise work are the fused-attention family, and their structure dictated most of what the compiler had to grow. A naive implementation materialises the $S \times S$ score matrix, costing memory quadratic in sequence length and, more importantly, the bandwidth of writing it out and reading it back. FlashAttention [5] tiles over key blocks and carries a running maximum and sum so the softmax normalisation applies incrementally — the online-softmax formulation [6] — and FlashAttention-2 [7] restructures the loop so the output accumulator is rescaled rather than the probabilities. Two properties bear on a compiler: the running statistics cross a loop containing cross-lane reductions, so tensor state must survive a loop whose body includes a barrier; and the two `tt.dot`s share their axes crosswise, so no fixed (M, N, fresh-K) axis assignment describes the kernel.

The backward pass is where a Triton backend stops being an inference target. It is three recompute-based kernels: `Delta = dO·O`, then `dQ` over key blocks, then `dK`/`dV` over query blocks. The forward stores one `BLOCK_M`-wide per-row logsumexp vector per query block and the backward rebuilds P from it with the same `exp2` expression the forward evaluated, so the score matrix never exists on either side. Because `dQ` sums over key blocks while `dK`/`dV` sum over query blocks, no single program owns both ends; the directions run as separate programs rather than accumulating `dQ` with `tt.atomic_rmw fadd`. Determinism decides that, not speed — and §6 states the cost, since the split formulation recomputes $Q\,K^{T}$ in each direction, seven GEMMs' worth of arithmetic where the mathematics needs five.

03 System Architecture

3.1 The Lowering Pipeline, and Why It Emits Text

The obvious "serious compiler" choice is to emit AIR, Metal's LLVM bitcode dialect, and skip a parse. triton-metal emits Metal Shading Language *source text* instead and compiles it with `MTLDevice.makeLibrary(source:)` at runtime. The reason is debuggability: every SSA value becomes a named variable, so `%13 = arith.addf %9, %12` becomes `float v13 = v9 + v12;` and a diff between the input IR and the generated kernel is a line-for-line diff, which makes a lowering bug something one can *read* rather than bisect. The costs are accepted rather than denied — a text round-trip through Metal's front end on every compile, and no control below the MSL level. A second path, `xcrun metal` producing `.metallib` bytes offline, exists for callers wanting a cacheable artifact; `xcodebuild` is never used.

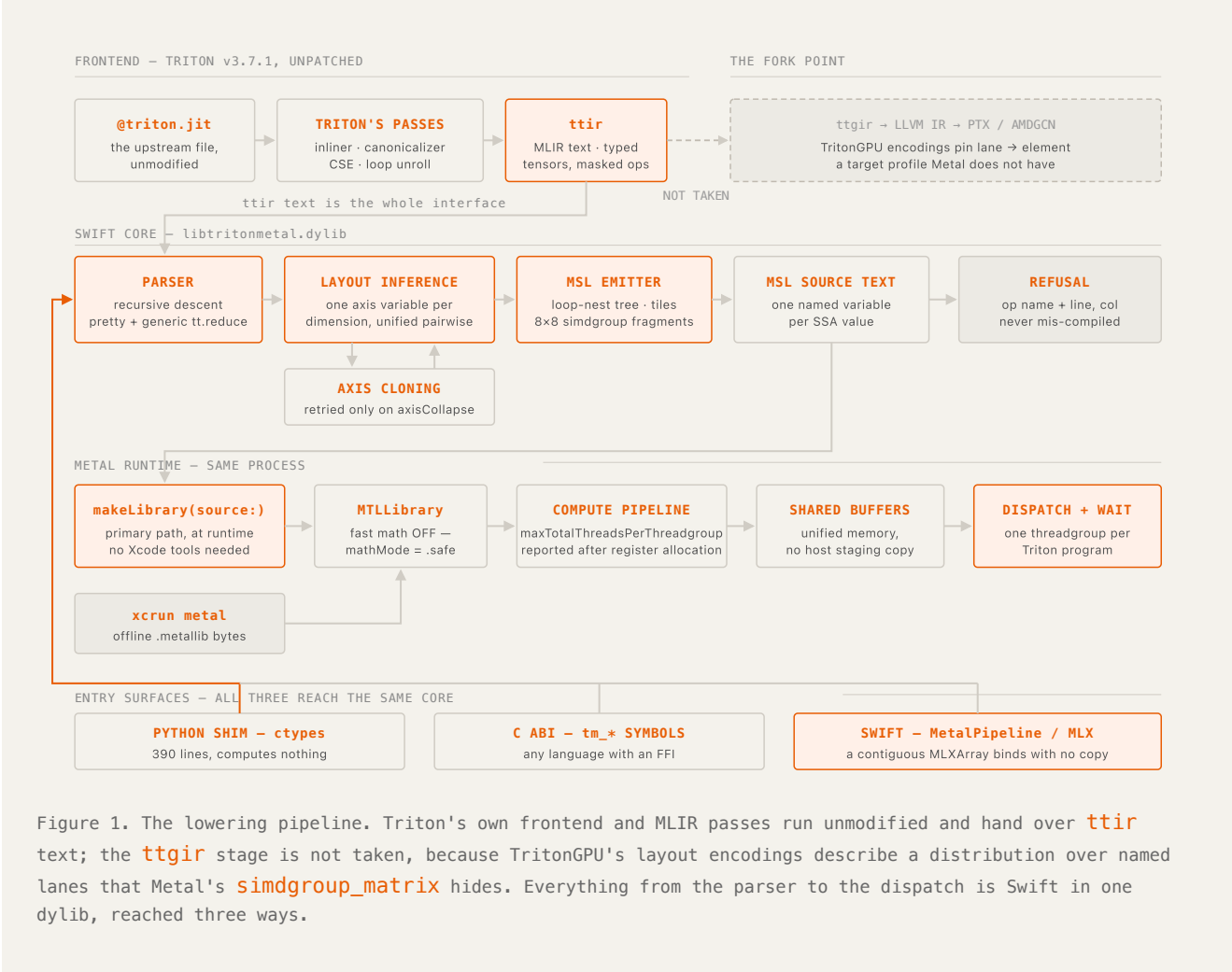

Figure 1. The lowering pipeline. Triton's own frontend and MLIR passes run unmodified and hand over `ttir` text; the `ttgir` stage is not taken, because TritonGPU's layout encodings describe a distribution over named lanes that Metal's `simdgroup_matrix` hides. Everything from the parser to the dispatch is Swift in one dylib, reached three ways.

3.2 Language Policy: Swift Core, C ABI, No-Logic Shim

Everything that does work is Swift, exported as `tm_*` symbols in `libtritonmetal.dylib`. The Python package is a 390-line ctypes binding that marshals arguments, translates a documented failure sentinel into a `RuntimeError` carrying `tm_last_error`, and computes nothing. This is a policy, not an accident: a

backend written half in Python and half in a compiled core drifts, launch geometry gets computed on the Python side "just for now", and eventually the core cannot be used from anything but Python. The load-bearing example is `tm_kernel_info`, which returns launch metadata as JSON. Threadgroup size in particular is *not* `num_warps * 32`: it is that product clamped to the innermost block dimension, rounded **up** to a whole simdgroup, and capped at Metal's 1024 — and a `tt.dot` kernel skips the clamp, which is an occupancy heuristic for elementwise kernels. The rounding is not cosmetic: `simd_shuffle_down` reads lanes that must be live, so a threadgroup is never a partial simdgroup.

3.3 Execution Model and Layout Inference

The execution model is one Metal threadgroup per Triton program. Every tensor is a slice of one **block** — an index space whose dimensions are called axes — and a value is emitted inside a nest of loops over exactly the axes it varies along. The last axis of a nest is distributed over the threadgroup's threads; the axes outside it are walked by every thread. A value is therefore emitted at the shallowest depth its dimensions allow, so row-constant work is computed once per row and only column-varying work reaches the innermost loop; a store at a uniform depth is guarded with `tm_thread_id.x == 0u`.

The subtle part is *which* block dimension a value spans. `tl.arange(0, M)` is rank 1 even inside a rank-2 kernel, and nothing about it says whether it indexes rows or columns — only the `tt.expand_dims` consuming it does. So the mapping is inferred: every dimension of every tensor gets an axis *variable*, and the operations relating two tensors **unify** them — elementwise dimension-wise, `tt.expand_dims` and `tt.reduce` around the inserted or dropped dimension, `tt.trans` through its permutation, and `tt.dot` by pinning its operands to (M, K), (K, N) and (M, N). A rank-deficient value that never reaches a full-rank one is an error, not a guess. Two rules there were found by running real Triton IR rather than by reasoning: a size-1 dimension must carry no axis identity, and two varying dimensions of one value must not collapse onto one axis — the value would then be a diagonal of itself, and is refused by name.

FlashAttention-2 forced both structural changes. Its `p` spans (`BLOCK_M`, `BLOCK_N`) and its `acc` spans (`BLOCK_M`, `HEAD_DIM`); neither contains the other, and one three-deep nest would recompute each of them `HEAD_DIM` or `BLOCK_N` times over, so nests had to become a *tree* whose siblings share their common prefix — a change to how every statement in the emitter is placed. The second value class the dot needed is a *tile*: a tensor living in a threadgroup array instead of per-lane registers, padded to whole 8×8 fragments with the padding zero-filled during staging, which is what makes block dimensions that are not multiples of 8 work without a separate edge-tile path.

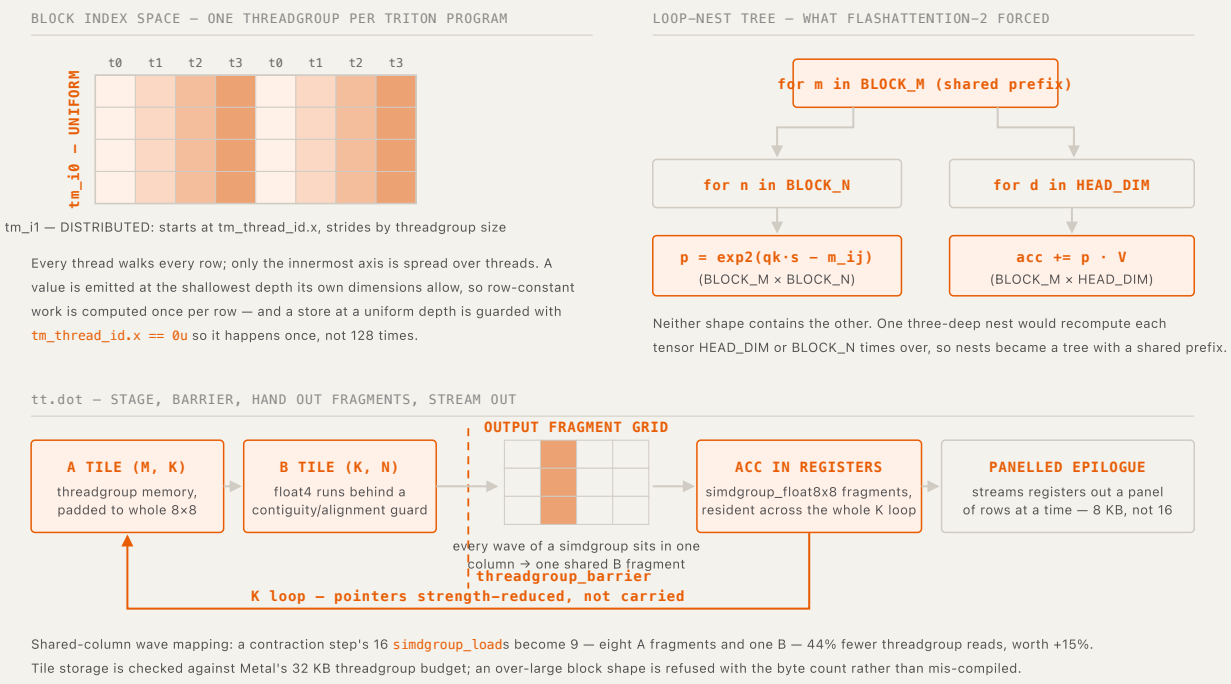

Figure 2. The execution model, the loop-nest tree, and the `tt.dot` lowering. Because Metal moves whole fragments through threadgroup memory rather than distributing them over named lanes, *which* fragment a simdgroup takes is an emitter decision — the freedom the shared-column mapping exploits, and one with no CUDA analogue.

04 Verification Strategy

4.1 What the Suite Asserts, and at What Sizes

The suite is 228 Swift cases across seventeen suites plus 23 Python ones, sixteen of which exercise the shim over C-ABI round trips while seven drive real `@triton.jit` kernels through the pinned Triton, including an attention layer's forward *and* backward. Every GPU result is checked against an independently computed CPU reference.

Test sizes divide *nothing*: the matmul tutorial runs at `129x257x65` and `37x41x43`, single-tile products go down to `5x3x7` — a dot smaller than one 8×8 fragment in every dimension — and softmax reduces rows 100 columns wide inside a 128-wide block. A backend whose masks or edge handling are wrong passes every power-of-two test and fails these. Refusal is tested as first-class behaviour too: about a quarter of the suite asserts that something is rejected with a message naming the offender and its line — a rank-deficient value, two varying dimensions collapsing onto one axis, a tile over Metal's 32 KB threadgroup budget reported with its byte count.

4.2 Gradients Checked Twice, Against Tolerances That Were Derived

Every gradient is checked twice independently: against an analytic CPU reference in `Double`, and against central **finite differences** of the forward, which know nothing of the backward formulas — the second catches a reference wrong in the same way the kernel is. Eight shapes, including `s127`, `s33`, `d80` and `d20` plus two f16-in/f32-accumulate ones, across `num_warps` 1..8, with the real forward's own logsumexp feeding the backward.

The tolerances are **derived, not tuned**. For a gradient element that is a sum of n terms the recursive-summation bound is $(n-1)\cdot u\cdot\Sigma|term|$, so the per-tensor bound is $n\cdot u\cdot A$, with A the cancellation amplification computed from the `Double` reference itself and u either 2^{-24} in f32 or 2^{-11} for the f16 dot operands. Nothing in the suite is a tolerance chosen because a smaller one failed.

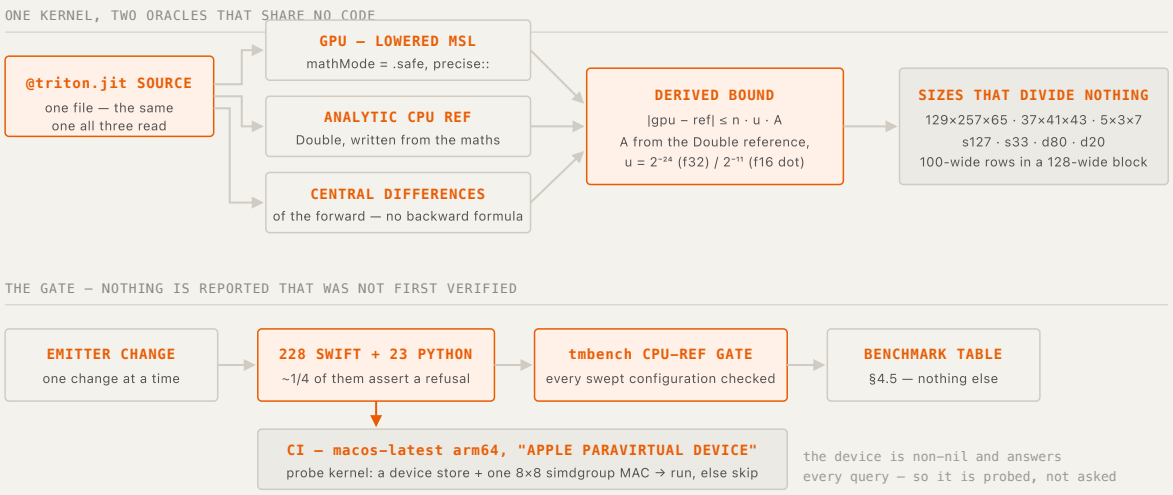

Figure 3. The verification path. A result is admitted only when two references sharing no code agree with it inside a bound computed from the reference rather than chosen, at sizes dividing neither the block shape nor the 8×8 fragment. The CI gate is a probe rather than a device query, because a virtualised GPU answers every question a query could ask.

4.3 Coverage, and What the Uncovered Fraction Is Made Of

Scope	Region	Function	Line
`Compiler.swift`	96.80%	97.22%	98.72%
`Layout.swift`	89.49%	77.00%	94.40%
`MSLEmitter.swift`	84.53%	86.33%	91.97%
`AxisCloning.swift`	56.57%	66.67%	62.50%
Swift core, total	**83.43%**	**85.23%**	**89.50%**

Over roughly 6,900 lines of Swift core against 3,900 of tests; `TritonMetalMLX` is excluded, since its suite skips without MLX's shader library. What the uncovered fraction is *made of* matters more than the number.

`Runtime.swift`'s gap is entirely environment-failure branches unreachable on a machine where the suite passes. `AxisCloning` is lowest because most of what is uncovered there is two mechanical operand-remapping switches with one arm per operation in the language, of which only the handful an index-arithmetic cone contains is exercised: the behaviour is covered, the switch arms are not.

4.4 Continuous Integration on a Virtualised GPU

CI runs `swift build`, `swift test` and the Python suite on `macos-latest`, an arm64 runner, on every push. That runner's GPU is virtualised: it reports an *Apple Paravirtual device* whose `MTLCreateSystemDefaultDevice()` is non-nil and which answers every query, so gating on "is there a device" would give a suite that fails rather than skips. The gate is a **probe** instead — one kernel using both halves of what every emitted kernel needs, an ordinary device store and an 8×8 `simdgroup` multiply-accumulate through threadgroup memory, cached once per process. That the paravirtual device passes it is a measured fact about the current runner image rather than an assumption.

4.5 Throughput, Measured Against the Vendor Library

Benchmarks run through `tmbench`, which verifies every configuration against a CPU reference at non-multiple-of-tile sizes before reporting it. Dispatches are packed into one command buffer until each timed sample is about 25 ms of GPU work; median of three, each side timed twice per size with the faster taken, since MPS is the denominator of every ratio and its first run at a size is cold. Machine peaks are measured rather than read off a spec sheet: 6.2 TFLOP/s f32 and 371.5 GB/s on the M1 Max, 3.45 TFLOP/s and 166.4 GB/s on the M1 Pro.

Machine	Size	Baseline	Round 2	Round 3	of MPS	of measured peak
M1 Max	1024	1.85 TF	4.33 TF	**4.30 TF**	**76%**	69%
M1 Max	2048	1.99 TF	4.66 TF	**4.64 TF**	**76%**	75%
M1 Max	4096	1.97 TF	4.64 TF	**4.65 TF**	**76%**	75%
M1 Pro	1024	1.02 TF	2.27 TF	**2.33 TF**	**75%**	67%
M1 Pro	2048	1.11 TF	2.33 TF	**2.43 TF**	**75%**	71%

f32 square GEMM against `MPSMatrixMultiplication` [8]. Published Triton reaches 62–82% of a CUDA-kernel stack end-to-end on its native target [9] and 80–100% of cuBLAS on GEMM specifically [10], [11], after years of NVIDIA-specific pipeline work: three optimisation rounds put a new backend inside the lower band and not the upper one. MPS reaches 92–99% of the same machine's measured f32 peak from the same API surface, this kernel 75%.

Attention is the other kind of comparison, a *fused* kernel against the *composite* it replaces: for the forward, `Q` K^T + `MPSMatrixSoftMax` + `P V`, three dispatches per head through a real $S{\times}S$ f32 score matrix; for the backward, five `MPSMatrixMultiplication`s with a softmax and a softmax gradient between them, through *two* live $S{\times}S$ matrices. Both sides are credited the same $5{\cdot}2{\cdot}S^2{\cdot}D$ per head, though the fused backward performs seven GEMMs.

Shape	Forward			Backward		
	fused	MPS	ratio	fused	MPS	ratio
`b1 h8 s512 d64` f32	1358 GF	381 GF	**357%**	977 GF	515 GF	**190%**
`b1 h8 s1024 d64` f32	1678 GF	1575 GF	**107%**	1054 GF	1519 GF	69%
`b1 h16 s2048 d64` f32	1761 GF	2586 GF	68%	1172 GF	2494 GF	47%
`b1 h8 s512 d64` f16	—	—	—	1293 GF	588 GF	**220%**
`b1 h8 s1024 d64` f16	2089 GF	1582 GF	**132%**	1419 GF	1445 GF	98%

The crossover sits near `s1024` in the forward and earlier in the backward — 69% where the forward is at 107% — with the 40% arithmetic premium of recomputing $Q\ K^T$ in both directions most of the reason; f16 moves it outward. The composite's own readings move, the same f32 composite measuring 1519 and 1445 GF at `s1024` minutes apart, so those ratios are worth about ±5 points. Underneath both is the end-to-end result: an attention layer written as `@triton.jit` source, run on a Mac GPU in both directions, its gradients agreeing with a hand-written numpy autograd to 6.2e-07 and with finite differences to 3.7e-08, and eight steps of gradient descent on $0.5 \cdot ||O - T||^2$ taking the loss from **1541.73 to 1482.86**.

4.6 Which CUDA Techniques Transfer, and Which Invert

Measured one change at a time on an M1 Max at 2048, winners and losers alike. **Transferred:** register-resident accumulators (+13%); staging locality with literal trip counts (+16%); the accumulator tile doubling as the operand staging arena (+6%, and the enabler for 64×128 blocks inside the 32 KB budget); vectorized `float4` global loads (+20%, the largest single change, though most of it is the *address arithmetic* the vector path skips rather than the loads).

Inverted: double buffering is a wash — 3633 against 3627 GFLOP/s — and a 26% *loss* once vector staging lands, because ping-ponging makes each tile's base a variable pointer where the vector path wants a constant, and Metal has no `cp.async` copy engine, so the prefetch competes for the same issue slots. Register blocking past 1×1 along **N** inverts hard: at 64×128, 1×1 beats 2×2 by 53% at identical fragment counts, 4327 against 2836 — the opposite of CUTLASS guidance [12]. Blocking along **M** is the exception that proves the rule, since 2×1 keeps the shared column and is worth +8%.

Bigger tiles do not pay, which is the informative negative result. The panelled epilogue removed threadgroup memory as a constraint on block shape — 64×64 costs 8 KB instead of 16, and a 128×128 f32 accumulator, twice Metal's entire budget, becomes emittable — and it is worth 0% at the winning shape, +15% at 128×64, while the 128×128 tile it unlocks runs **21% slower** than 64×64. A tile that halves operand traffic per output element and loses is the kernel saying operand traffic is not what it waits on. What it waits on is legible in `maxTotalThreadsPerThreadgroup`, which Metal fixes after register allocation: 1024 for the winner, 832 at 128×64, 768 at 128×128, 448 for 128×128 with 2×2 blocking, which therefore cannot launch its own 512 threads. Occupancy is how this GPU hides latency, registers are what occupancy costs, and that one mechanism is behind every inversion above.

No CUDA analogue at all: the shared-column wave mapping, +15%. Because Metal moves whole 8×8 fragments through threadgroup memory instead of distributing them over named lanes, which fragment each simdgroup wants is an emitter decision — and choosing it so every wave of a simdgroup shares one B fragment cuts a contraction step from sixteen threadgroup reads to nine. There is nothing to port, because the CUDA equivalent of that decision is made by the mma layout rather than the kernel.

76%

OF MPSMatrixMultiplication — 1024, 2048 AND 4096 SQUARE, M1 MAX

75% at 1024 and 2048 on an M1 Pro, up from ~33% at the first working version. Every figure here is an M1-generation GPU-ALU number against an M1-generation MPS; `tmbench --sweep full` re-measures all of it on other silicon in one command.

05 Application Domains and Economic Analysis

5.1 The Kernel Development Loop on Hardware Already Owned

The economic argument is about where a kernel is *edited*, not where it finally runs. A Triton kernel is iterated far more often than it is run at scale, and on CUDA every one of those iterations requires an NVIDIA GPU — rented, queued for, or bought — because the kernel does not execute at all without one. With this backend installed the loop runs on a Mac the team already owns, and rented NVIDIA time is spent on the runs that need the hardware rather than on the ability to run anything. Nothing about the file changes when it moves between the two, because it is the same file.

The second domain is inference and small-scale training on owned hardware: FlashAttention-2 in both directions, GEMM, softmax, elementwise work and atomics on `f32` and `i32` — the kernels a training loop is made of — over unified memory with no host staging copy. Adoption costs one artifact: `Tools/bundle-wheel.sh` stages `libtritonmetal.dylib` into `triton/backends/metal/` with the ctypes shim alongside it, so the recipient installs one file and nothing else, verified into a fresh virtualenv on a second machine with no repository checkout on the path. The dollar side is left qualitative on purpose: this paper measures kernels rather than invoices.

5.2 The Swift-Native Path, Sized Honestly

Because the compiler is a Swift library rather than a Python extension, the Python frontend is optional in a way it is not on CUDA, where `@triton.jit` is a decorator and the launcher a Python object. `TritonMetalMLX` — a separate target and product, so the core acquires no dependency — runs an emitted kernel directly on MLX [3] tensors, checking each argument's dtype against the kernel's own metadata. A contiguous `MLXArray` reaches a kernel with **no copy at all**: its storage came from MLX's Metal allocator, so

`makeBuffer(bytesNoCopy:)` returns the array's own address and the kernel's stores land in the array — verified by writing through the bound buffer and reading back through MLX at every size down to 12 bytes, rather than inferred from pointer equality. A strided array is the one case that copies, and the binding reports which it did.

The latency consequence is smaller than the architectural one and is reported as measured: with arguments bound, the same kernel from byte-identical IR launches in 231–233 µs from Swift against 282–286 µs through the ctypes shim on an M1 Max, 360–367 against 371–378 on an M1 Pro — **3–18%**, because both paths are dominated by a synchronous command-buffer round trip they share. What the Swift path removes unconditionally is not the interpreter but the *copy*: a numpy array must be staged into a Metal buffer and read back, 141–216 µs at `[64, 512] @ [512, 2048]`. The demonstration is an interpretability op rather than a toy — a sparse-autoencoder encoder fused into one kernel, agreeing with MLX's own ops to 8.3e-07 worst relative error over shapes ragged in all three dimensions.

5.3 Positioning After WWDC 2026

Apple's **TensorOps** is a Metal Shading Language API that accelerates tensor operations including matrix multiplication and convolution, taking full advantage of the neural accelerator in the M5 chip family — a hardware block sited directly in each shader core [13]. What it asks of a developer is an MSL kernel. Apple shipped no Triton frontend, so this backend's bet — run the ecosystem's *existing* Triton source unmodified — is untouched in kind.

What TensorOps does supersede is one thing precisely: the *destination* that `tt.dot` lowers to. On Metal-4-capable silicon the `simdgroup_matrix` path of §3.3 stops being the best lowering of the same `ttir`, because TensorOps reaches a compute unit `simdgroup_matrix` cannot. Emitting TensorOps there is the named next-generation lever — a change of destination inside an unchanged pipeline rather than a change of thesis, since the frontend, the layout inference and the loop-nest tree are indifferent to which matrix instruction the dot ends at. It is spec-level and nothing more: no M4 or M5 part was available here, and no number in this paper is measured on such hardware. It also bounds §4, every ratio of which is an M1-generation number against an M1-generation MPS — a second arithmetic path moves both sides of that fraction at once.

06 Limitations and Roadmap

Not a framework. "Training-capable" here means the attention forward and backward, the GEMM, softmax and elementwise kernels, and atomics on `f32` and `i32` — no more. There is no autograd: a caller writes or generates the backward kernel, as a Triton user does. There are no optimizer-state kernels; Adam's moment updates are ordinary elementwise work that nothing in the design objects to, but none is written or measured here. There is no dropout, because there is no `tl.rand` and no RNG lowering at all. Each is implementable and unimplemented, which is a different statement from unsupported.

Type and atomic boundaries. `bf16` lowers to Metal's native `bfloat`, including as a `simdgroup_matrix` operand type; there is no `f64`, because Metal has no `double`, and no block pointers. Atomics are 32-bit only — Metal has no 16- or 64-bit atomics and no floating-point `fetch_max`, which goes through a

compare-exchange loop — and expose one memory order, so a kernel relying on an atomic's release edge to publish ordinary stores is not correctly lowered.

The determinism trade is open. Accumulating `dQ` with `tt.atomic_rmw fadd` would remove one of the two QK^T recomputations, and with it most of the fused backward's 40% arithmetic premium, at the cost of a gradient that is no longer bit-reproducible. The atomics exist for it; the trade is unmeasured, and the crossover in §4.5 is the number that would decide it.

Installation is not `pip install triton`. Triton publishes no macOS wheel, so someone with a Mac and a Swift toolchain builds it once — about nine minutes, no CUDA toolchain, no Xcode, no patch to Triton. That yields the single wheel everyone downstream installs, which moves the cost from every user to one without removing it; an upstream macOS wheel would.

Every performance finding is M1-generation-scoped. All of §4.5 and §4.6 was measured on an M1 Max and an M1 Pro. The reasons given for the inversions — no `cp.async` copy engine, occupancy as the latency-hiding mechanism, fragments moving through threadgroup memory rather than named lanes — are architectural properties later parts may have changed. Nothing here should be assumed to hold on later silicon in *either* direction until re-measured, and `tmbench --sweep full` is that re-measurement.

The named next performance lever. The remaining four points of hardware utilisation are an issue-slot problem at *fixed occupancy*: less non-arithmetic work per unit of matrix arithmetic, bought without spending registers. The largest candidate is loading the B operand's 8×8 fragments straight from device memory — MSL's `simdgroup_load` takes a `device` pointer as happily as a `threadgroup` one [14], and under the shared-column mapping each simdgroup reads exactly one B fragment per contraction step with no two reading the same one, so B crosses the memory system once either way and staging it buys only masking, at the cost of an entire staging pass. A must stay staged, because every simdgroup reads all of it. The price is emitting the contraction loop twice around a hoisted guard: not a layout redesign, and local to the dot emitter.

Pin maintenance. The backend is pinned to Triton v3.7.1 and vendors that tag's adapter signatures, because the out-of-tree plugin interface churns between releases; tracking a newer Triton is deliberate work rather than a version bump. A subset of Triton's conformance suite also remains unported.

07 Conclusion

Triton IR text goes in; readable Metal Shading Language, a Metal library, a compute pipeline, unified-memory buffers, a dispatch and correct results come out — driven from Python, from Swift, or from any language that can call C. The tutorial kernels match CPU references at sizes dividing neither the block shape nor the simdgroup fragment, and FlashAttention-2 runs in both directions: an attention layer written as `@triton.jit` source trains on a Mac GPU, its gradients checked against a numpy autograd to 6.2e-07 and finite differences to 3.7e-08.

Two results deserve restating plainly. The numerics are good: fast math off, `precise::` where Metal has it, a softmax agreeing with a CPU reference to about one ULP, and gradient tolerances derived from a summation bound rather than chosen to pass. The performance is good rather than adequate: 75–76% of

MPS on a GEMM across two machines, inside the band published Triton reaches end-to-end on its own target; 357% of the MPS composite on attention forward at `s512` and 190% on the backward, each with a measured crossover rather than a headline ratio. The GEMM number did not move in the third optimisation round, and what that round produced instead is the mechanism behind it — the register file, not memory traffic, is what stands between this kernel and Apple's.

The boundary is as plain as the results. This compiles and runs the kernels a training loop is made of; it is not itself a training framework, and every performance figure is single-generation. The claim is not that the portable-kernel moat has been drained, but that it is crossable: the ecosystem's existing Triton source, unmodified, now compiles and trains on a Mac.

08 References

[1] Tillet, P., Kung, H.T., and Cox, D., "Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations," Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL), pp. 10–19, 2019.

[2] Lattner, C., et al., "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation," IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 2–14, 2021.

[3] Hannun, A., Digani, J., Katharopoulos, A., and Collobert, R., "MLX: Efficient and Flexible Machine Learning on Apple Silicon," Apple Machine Learning Research, 2023. https://github.com/ml-explore/mlx

[4] Abramowitz, M., and Stegun, I.A., *Handbook of Mathematical Functions*, National Bureau of Standards Applied Mathematics Series 55, 1964, §7.1.26.

[5] Dao, T., Fu, D.Y., Ermon, S., Rudra, A., and Ré, C., "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness," Advances in Neural Information Processing Systems 35 (NeurIPS), 2022.

[6] Milakov, M., and Gimelshein, N., "Online normalizer calculation for softmax," arXiv:1805.02867, 2018.

[7] Dao, T., "FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning," International Conference on Learning Representations (ICLR), 2024.

[8] Apple Inc., "MPSMatrixMultiplication — Metal Performance Shaders," Apple Developer Documentation. https://developer.apple.com/documentation/metalperformanceshaders

[9] PyTorch, "CUDA-Free Inference for LLMs," PyTorch Blog, 2024. https://pytorch.org/blog/cuda-free-inference-for-llms/

[10] Zhang, X., et al., "Hexcute: A Tile-Based Programming Language with Automatic Layout and Task-Mapping Synthesis," arXiv:2504.16214, 2025.

[11] Triton project, "int8 GEMM performance versus cuBLAS," triton-lang/triton issue #4876, 2024. https://github.com/triton-lang/triton/issues/4876

[12] NVIDIA, "CUTLASS: CUDA Templates for Linear Algebra Subroutines — Efficient GEMM," NVIDIA Corporation. https://github.com/NVIDIA/cutlass

[13] Apple Inc., "Explore Metal 4 machine learning APIs," WWDC 2026 session 330, and "Machine learning passes," Apple Developer Documentation. https://developer.apple.com/videos/play/wwdc2026/330/

[14] Apple Inc., *Metal Shading Language Specification*, version 3.2, Apple Inc., 2024, §6.16 (SIMD-group matrix functions).