Do You Need a GPU to Speed Up a Backtest Parameter Sweep? A Controlled Speed Ladder for a Path-Dependent Kernel

Eugen Soloviov*

Abstract

The compute bill of systematic-strategy research is dominated by mass parameter search: one backtest kernel re-run across many configurations, multiplied again by walk-forward loops. The practitioner reflex when a sweep takes an hour is to ask for a GPU. We measure a different claim: for a path-dependent kernel of this shape, the bottleneck is the *engine* and the *orchestration*, not the absence of accelerator hardware. We take one fixed strategy kernel—an HMA/HMA3 moving-average-cross sweep with a stateful, bar-by-bar event loop (open on a cross, close on the opposite cross, 0.09% round-trip fee)—and implement it five ways, from the naive profile up: `pandas rolling().apply()` with a Python bar loop; vectorized `numpy` (BLAS pinned to one thread, so the rung is verifiably single-core); single-threaded `numba`; `numba` parallelized across combos with `prange`; and a process pool of `numba` workers. All five rungs are cross-checked to produce identical per-combo (PnL, trade-count) outputs—trade counts exactly equal, PnL to a true absolute tolerance of 10^{-6}—so the ladder varies speed and nothing else. On 150,000 synthetic bars × 80 parameter combinations on a 12-core Apple M2 Max laptop, the naive rung takes 69.92 s (1.1 combos/s; measured in full, not extrapolated) and the top rung 0.23 s (340.9 combos/s): 297.9× end to end with zero hardware change—and still 13.1× measured from the already-vectorized `numpy` rung, so the headline does not stand on the naive baseline alone. The decomposition is instructive: vectorization alone is worth 22.7×, single-threaded `numba` 35.3×, and orchestration across 12 cores supplies the rest (217.6× with threads, 297.9× with processes). The GPU question we answer from a measured split and a modeled bound: the weighted-moving-average feature stage accounts for 99.3% of the compiled kernel's time and, at a computed arithmetic intensity of 10.78 FLOP/byte (a deliberately GPU-favorable idealization), it is exactly the kind of work a GPU *could* batch. What a GPU lacks here is not suitability but leverage: the exploitable parallelism is only 80 combos wide, and the whole optimized job is 0.23 s, so kernel-launch and synchronization overhead has nothing to amortize against. Wall-clock speedups are hardware-specific and we say so; what reproduces deterministically is the equivalence gate and the ordering of the ladder. The practical conclusion for a practitioner on a laptop: before renting an accelerator, replace `rolling().apply()` and the Python loop with a JIT-compiled kernel and parallelize across the sweep—two orders of magnitude are already on the table.

1 Introduction

A parameter sweep is the workhorse of quantitative strategy research: the same backtest kernel evaluated across a grid of configurations, often inside a walk-forward or cross-validation loop that multiplies the evaluation count by another order of magnitude [6]. Sweep throughput therefore sets

*Independent Researcher. ORCID: 0009-0006-3148-111X. Correspondence: suenot@gmail.com. Code to reproduce every number and figure: `https://github.com/suenot/backtest-speed-ladder`.

the pace of the research loop itself, and when a sweep takes an hour on a laptop the conversation reliably turns to hardware—specifically, to GPUs.

This paper measures a prior question: what does the hour consist of? The typical first implementation of a technical-analysis backtest is not a tuned engine. It is `pandas` with `rolling().apply()` callbacks for the indicators [7] and an interpreted Python `for` loop over bars for the trade logic, run serially on one core. That profile is not a strawman; it is the natural expression of the computation in the standard data stack, and—for weighted moving averages, which have no fused `pandas` primitive—`rolling().apply()` with a Python-level callback is the idiomatic spelling. Between that starting point and any accelerator conversation stands a factor we measure directly: 297.9× on the same machine, with the same results, checked to 10^{-6}. And because a naive baseline can flatter any speedup, we co-headline the stricter number: from an already-vectorized, single-core `numpy` implementation to the top rung is still 13.1×.

The design is deliberately controlled. We fix *one* strategy kernel—a Hull-moving-average cross with a stateful, bar-by-bar event loop in which the open position carries across bars—and implement it five ways: naive `pandas` (M0), vectorized `numpy` (M1) [4], single-threaded JIT-compiled `numba` (M2) [5], the same kernel parallelized across parameter combos with `numba prange` threads (M3), and the same kernel parallelized across combos with a process pool (M4). Every rung is cross-checked to produce identical per-combo (PnL, trade-count) outputs on all 80 combos before its timing counts. The gate matters because speed comparisons in this space routinely compare engines that quietly compute different things—different fill conventions, fee timing, or warm-up handling; here, only speed differs.

The GPU discussion is then grounded in measurement rather than a slogan. Two facts frame it. First, the feature stage—the weighted-moving-average arithmetic—has a computed intensity of 10.78 FLOP/byte under a deliberately GPU-favorable model, so this is *not* a memory-bound workload that a GPU could not help in principle. Second, a direct time split of the compiled kernel shows that this batchable feature stage is where the CPU actually spends its time: 99.3%, against 0.7% in the stateful event loop. A GPU could therefore batch essentially all of the compute. The case that it is still not the missing rung rests on leverage, not suitability (Section 5): the exploitable parallelism is only as wide as the 80-combo grid, which the 12-core CPU pool already covers with lanes to spare, and the whole optimized job is 0.23 s, so launch and synchronization overheads have nothing to amortize against [3]. The branchy, path-dependent event loop remains a poor SIMT fit [2], but at 0.7% of the kernel it is a porting nuisance, not the argument.

Throughout, we are explicit about what is and is not claimed. Wall-clock timings are machine-specific; we commit one representative best-of-3 run on disclosed hardware and treat the *ordering* of the ladder and the *equivalence* of its rungs—not the exact seconds—as the reproducible findings. The study is one kernel family on one host with synthetic data; the limitations are collected in Section 6.

Contributions.

1. A controlled five-rung speed ladder for a fixed path-dependent backtest kernel, with an equivalence gate that makes all rungs output-identical on every combo (trade counts exact, PnL to a true absolute 10^{-6}), so the comparison isolates implementation speed from everything else (Section 3).
2. The measured ladder on disclosed laptop hardware: 22.7× from `numpy` vectorization alone, 35.3× from single-threaded `numba`, 217.6× from `prange` threads, and 297.9× from a process pool—69.92 s down to 0.23 s with zero hardware change, and 13.1× even from the vectorized `numpy` rung (Section 4).

3. An honest decomposition of where the orders of magnitude come from, with the single-core rungs verifiably single-core (BLAS pinned to one thread): vectorization is most of the "free" win, compilation adds a further 1.55× over `numpy`, and orchestration across cores contributes the final 6.2×–8.4×; on this host the process pool edges out `prange` threads (297.9× vs. 217.6×), the one ordering we flag as least portable (Section 4.3).
4. A reframing of the GPU question from measurement: a measured feature/event-loop time split (99.3%/0.7% of the compiled kernel) plus a computed arithmetic-intensity bound (10.78 FLOP/byte idealized, 11.07 as executed) showing that a GPU *could* batch the dominant stage—and that what kills the payoff here is the 80-wide parallelism and the 0.23 s budget, with the conditions under which a GPU *would* pay stated explicitly (Section 5).

2 Related work

The scientific-Python performance stack. The rungs of our ladder are the standard tools. `pandas` supplies the DataFrame and rolling-window machinery that most practitioners reach for first [7]; its `rolling().apply()` executes a Python-level callback per window, which is the naive profile we time as M0. `numpy` provides array programming whose operations execute in compiled loops [4]; our M1 expresses the weighted moving average as a sliding-window matrix–vector product and the trade logic as a vectorized reducer. `numba` JIT-compiles numerical Python to LLVM machine code [5], including explicitly parallel loops via `prange`; it is our M2/M3. Columnar engines such as Polars [10] attack the same interpreter overhead from the DataFrame side; we do not benchmark them here because our kernel's bottleneck after M1 is the bespoke compiled kernel, not DataFrame operations.

Vectorized backtesting. The observation that backtests can be dramatically accelerated by array programming is embodied in vectorbt [9], which combines `numpy` layouts with `numba`-compiled kernels and reports orders-of-magnitude gains over event-driven frameworks. Our study is complementary rather than competing: we could not benchmark vectorbt in this environment (a Python-version constraint), and more importantly our aim is not to rank frameworks but to decompose—on one fixed kernel, with an equivalence gate—*where* such gains come from: vectorization, compilation, or orchestration. The ladder shows each layer's contribution separately.

Performance models. The roofline model [11] relates achievable throughput to arithmetic intensity (FLOPs per byte of memory traffic) and is the standard lens for asking whether a workload is compute- or memory-bound; we compute the intensity of our feature stage explicitly and disclose the model behind it. Amdahl's law [1] bounds the return on parallel resources by the *serial fraction* of the work, and we invoke it only in that sense; the binding constraint on our sweep is simpler and cruder—finite width: 80 independent combos cannot occupy more than 80 coarse-grained lanes, however wide the device.

GPU computing. CUDA established the GPU as a general throughput processor whose model wants tens of thousands of fine-grained threads [8]. Two well-documented failure modes matter for our kernel: control-flow divergence, where threads of a warp serialize on branchy code [2], and host–device data movement, which Gregg and Hazelwood [3] showed can dominate reported GPU speedups when left unaccounted. Our GPU discussion (Section 5) is an application of these papers to a measured workload profile, not a new measurement of GPU hardware.

Why sweeps are large at all. The demand for throughput comes from validation practice: walk-forward and cross-validation protocols re-run the sweep per fold [6]. The same literature warns that search breadth inflates selection bias; we return to this in Section 6, because a faster engine is an amplifier for multiple testing, not a substitute for validation discipline.

3 One kernel, five implementations

3.1 The strategy kernel

Data. Performance of this kernel is size-bound, not value-bound—the work per bar is fixed by the window lengths, not by the prices—so we use a synthetic close series: geometric Brownian motion with zero log-drift, per-bar log-return volatility 0.0008, initial price 30,000, length 150,000 bars, seed 42. Using synthetic data makes the benchmark self-contained and the equivalence fingerprint deterministic.

Features. All indicators are built from the linearly weighted moving average

$$\mathrm{WMA}_p(x)_t \;=\; \frac{\sum_{j=1}^{p} j\, x_{t-p+j}}{\sum_{j=1}^{p} j}, \tag{1}$$

undefined (NaN) for $t < p$. With $r(\cdot)$ denoting rounding to the nearest integer, the Hull moving average of length L and its HMA3 variant (windows floored at 1; $q = L/2$) are

$$\mathrm{HMA}_L(x) \;=\; \mathrm{WMA}_{r(\sqrt{L})}\big(2\,\mathrm{WMA}_{r(L/2)}(x) - \mathrm{WMA}_L(x)\big), \tag{2}$$

$$\mathrm{HMA}^3_L(x) \;=\; \mathrm{WMA}_{r(q)}\big(3\,\mathrm{WMA}_{r(q/3)}(x) - \mathrm{WMA}_{r(q/2)}(x) - \mathrm{WMA}_{r(q)}(x)\big), \tag{3}$$

which issue 7 WMA passes per parameter combo, 6 of them distinct: since $q = L/2$, the inner HMA3 window $r(q)$ coincides with the HMA half-window $r(L/2)$, so the WMA of the raw series at that window is computed twice, identically in every rung (the duplication is part of the fixed kernel and leaves the equivalence untouched). Section 3.5 counts the arithmetic both ways.

Event loop. The trading rule is a stateful cross. On each bar t where both features are defined, the direction is $d_t = +1$ if $\mathrm{HMA}_L(x)_t < \mathrm{HMA}^3_L(x)_t$ and -1 otherwise. The first valid bar opens a position in direction d_t at the close; whenever d_t flips, the open position is closed at the current close and a new one opened in the new direction. Each round trip books

$$\mathrm{pnl} \;=\; \pm\,\frac{c_{\mathrm{exit}} - c_{\mathrm{entry}}}{c_{\mathrm{entry}}} \times 100 \;-\; \phi, \qquad \phi = 0.09, \tag{4}$$

the signed return in percentage points minus a 0.09% round-trip fee (matching a production fee model). The per-combo output is the pair (total PnL, trade count). The loop is genuinely path-dependent: the position state carries from bar to bar, so bars cannot be processed independently *within* a combo. This is the property that blocks naive vectorization of the trade logic—though, as Section 3.5 measures, it is *not* where the compiled kernel spends its time.

Parameter grid. The sweep runs 80 combos: integer HMA lengths spread uniformly over $[6, 200]$, in ascending order (cheap windows first, expensive last—a detail that will matter for load balancing in Section 4.3).

3.2 The five rungs

Each rung computes exactly the sweep above.

- **M0 pandas** (the naive profile). Each WMA is `rolling(p).apply()` with a dot-product callback executed per window in Python; the event loop is an interpreted Python `for` over all bars. One core.
- **M1 numpy** (vectorized). Each WMA is a `sliding_window_view` matrix–vector product. The event loop is replaced by an *equivalent* vectorized reducer: direction flips are located with `diff`/`flatnonzero`, and all trades are booked in one closed-form array expression. One core, verified: BLAS is pinned to a single thread (Section 3.4).
- **M2 numba** (compiled). Both the WMA and the bar-by-bar event loop are `@njit`-compiled; the loop retains its stateful form—no vectorization trick is needed once the loop is machine code. One thread.
- **M3 numba-prange** (threads). M2's kernel unchanged, with the *combo* loop parallelized by `numba prange` across 12 threads. Parallelism is across combos only; each combo's event loop stays serial.
- **M4 mp+numba** (processes). M2's kernel unchanged, with combos distributed over a `ProcessPoolExecutor` of 12 workers, handed out one combo at a time (chunk size 1 at this grid size), i.e. dynamically load-balanced.

M1 is the one rung whose *code shape* differs (a reducer instead of a loop); the equivalence gate below is what licenses calling it the same computation.

3.3 The equivalence gate

Before any timing is reported, every rung's per-combo (PnL, trade-count) pairs are compared against the M1 reference on all 80 combos—the pandas baseline included, since it too is measured in full. Trade counts must match exactly; PnL must agree within an absolute tolerance of 10^{-6} percentage points, with the comparison's relative-tolerance term set to zero, so the bound is genuinely absolute (the rungs in fact agree far more tightly). In the committed run all checks pass. The result is a ladder in which the only thing that varies is speed: any speedup that survives the gate cannot come from computing something cheaper.

The fingerprint of the first grid point ($L = 6$) is PnL -5165.58 percentage points over 57,029 trades. We record it as a *correctness* fingerprint, not tradable alpha—see Section 4.4.

3.4 Timing methodology and hardware

All timings are wall-clock (`perf_counter`), best of 3 repeats. The minimum is a deliberate choice of estimator: it is the noise-minimized estimate of the achievable steady-state time, and it discards the cold first pass (page cache, JIT load) by construction. JIT compilation is excluded throughout (each `numba` kernel is warmed on a short prefix before its timer starts, with on-disk caching enabled; pool workers are warmed with dummy tasks that trigger compilation before the timed map). The M0 baseline is measured *in full*—all 80 combos, best-of-3, like every other rung—and its outputs enter the equivalence gate on all 80 combos. (A development version of this benchmark extrapolated M0 from 5 uniformly sampled combos; that estimate, 70.20 s, is within half a percent of the full measurement, 69.92 s, but the committed number is measured, not estimated.)

The host is an Apple M2 Max (12 CPU cores, a mix of performance and efficiency cores), macOS 26.3 (arm64), Python 3.14.6, `numpy` 2.4.3 built against Apple's Accelerate BLAS, `numba` 0.64.0 with 12 threads, and 12 pool workers. BLAS is pinned to a single thread at import (`OMP_NUM_THREADS` and its siblings set to 1): M1's sliding-window product is a GEMV that Accelerate would otherwise multithread, silently smuggling parallelism into the "vectorization alone" rung. With the pin, M0–M2 are verifiably single-core and the vectorization/compilation/orchestration decomposition of Section 4.2 is honest. This machine has no NVIDIA GPU; no CUDA is used or required anywhere in the study, and the GPU section is analytical. We commit one representative run of this configuration as `results.json`; every number quoted in this paper is mechanically checked against that file.

3.5 Where the kernel spends its time, and its arithmetic intensity

The measured feature/loop split. The compiled kernel factors exactly into the WMA feature stage and the event loop, so we time the feature stage alone over the same grid and take the ratio: the features account for 99.3% of the single-thread `numba` kernel's wall time, the stateful event loop for 0.7%. This measurement matters twice. It locates the engine's work in the batchable arithmetic, not in the serial loop—a fact the GPU discussion must respect (Section 5)—and it explains why compiling the loop pays modestly while parallelizing across combos pays largely: the loop was never the bulk of the compiled kernel's time.

A computed, GPU-favorable intensity bound. For the roofline framing we model the feature stage's arithmetic intensity: each WMA_p over n bars costs $2p$ FLOPs per bar (p multiplies, p adds) and, with perfect sliding reuse, streams its length-n float64 input once (8 bytes per bar). Counting the 6 distinct WMAs per combo over 80 combos at $n = 150{,}000$ (480 distinct passes), the sweep's feature stage totals 6.21 GFLOP against 576 MB of traffic:

$$\mathrm{AI} \;=\; \frac{\sum_{\text{combos}} \sum_{\text{windows } p} 2\,p\,n}{\sum_{\text{combos}} 6 \cdot 8n} \;=\; 10.78 \text{ FLOP/byte}; \tag{5}$$

counting the 7 passes the kernel actually issues (560 in the sweep, the duplicated window included) gives 11.07. We stress that the model is a deliberately GPU-favorable *hybrid*: it pairs naive-recompute FLOPs ($2p$ per bar, no sliding-sum or prefix tricks) with best-case streamed traffic (8 bytes per bar, each input read once), while the as-executed inner loop re-reads p values per bar (mostly from cache), so real memory traffic is higher and the true intensity lower. 10.78 is a computed bound, not a measurement. The point of reporting it is the *opposite* of a memory-bound story: even generously idealized, the ratio is far above one, so the feature math is exactly the kind of work a GPU could batch. The honest GPU argument must therefore rest elsewhere (Section 5).

4 Results

4.1 The ladder

Table 1 and Figure 1 are the study. The naive profile needs 69.92 s for the 80-combo sweep—0.87 s per combo—and the top rung needs 0.23 s, or 2.9 ms per combo: a 297.9× end-to-end speedup on identical hardware with output-identical results. And because that headline stands on a deliberately naive baseline, we state the stricter one alongside it: from the competently vectorized M1—a single-core `numpy` implementation many would already call fast—to the top rung is 13.1×. For scale: a 10,000-combo overnight-sized search at the M0 rate is a 2.4-hour job; at the M4 rate it is a coffee break. Nothing about the strategy, the data, or the results changed between the two rows.

Table 1: The speed ladder: one kernel, five implementations, identical per-combo outputs (150,000 bars, 80 combos, best-of-3 wall clock, Apple M2 Max). Every rung, the pandas baseline included, is measured in full and passes the equivalence gate of Section 3.3 on all 80 combos; the single-core rungs run with BLAS pinned to one thread.

Rung	Implementation	Wall (s)	Speedup	Combos/s	Parallelism
M0	pandas `rolling().apply()` + Python loop	69.92	1.0×	1.1	1 core
M1	numpy sliding-window + vectorized reducer	3.07	22.7×	26.0	1 core
M2	numba `njit` WMA + `njit` event loop	1.98	35.3×	40.4	1 thread
M3	M2 kernel, `prange` across combos	0.32	217.6×	248.9	12 threads
M4	M2 kernel, process pool across combos	0.23	297.9×	340.9	12 processes

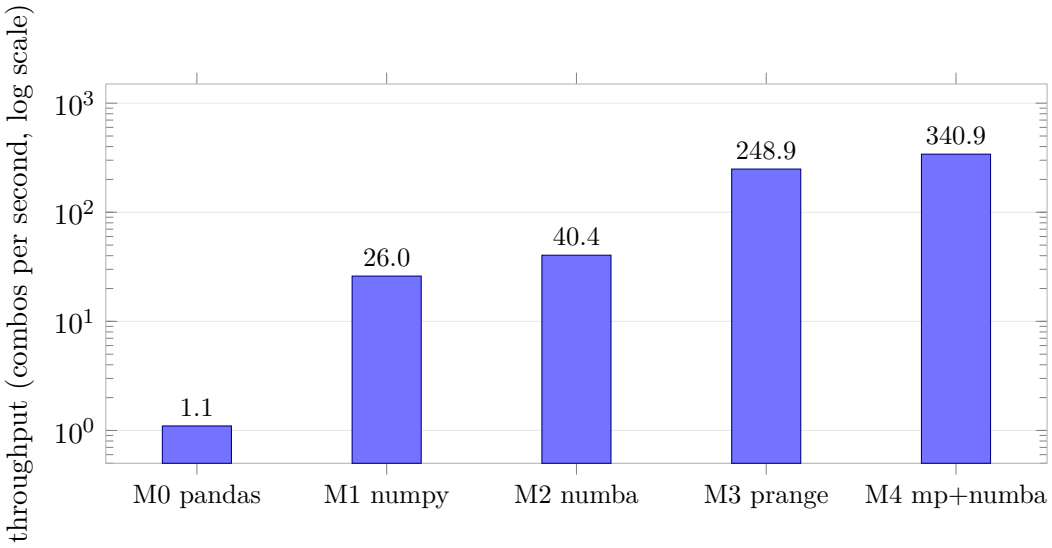

Figure 1: Sweep throughput by rung, log scale (the range spans 1.1 to 340.9 combos/s). Same kernel, same host, identical per-combo outputs at every rung; best-of-3 wall clock; every rung measured in full.

4.2 Where the orders of magnitude come from

The ladder decomposes the gain into three distinct mechanisms. The decomposition is honest by construction: BLAS is pinned to one thread (Section 3.4), so no hidden parallelism leaks into the single-core rungs.

Vectorization is most of the "free" win (22.7×). Moving from `rolling().apply()` and an interpreted bar loop to `numpy`—no compiler, no parallelism, one pinned core—already buys 22.7×. The mechanism is the removal of per-window and per-bar Python interpreter dispatch: M0 crosses the C/Python boundary once per window per WMA (plus once per bar in the trade loop), while M1 crosses it once per array operation. This rung requires re-expressing the event loop as a vectorized reducer, which is possible for this rule (flips can be located, then trades booked in closed form) but is exactly the step that does not generalize to richer path dependence—which is why M2 matters.

Compilation adds 1.55× over numpy (35.3× total) and removes the shape constraint. `numba` compiles the *loop form* of the kernel: the WMA becomes a fused multiply–add loop with

no temporaries, and the event loop runs as machine code in its natural stateful shape. The speed gain over M1 is a modest 1.55×—M1's inner product already runs in compiled code, and M1 pays mainly for materialized temporaries and strided window reads—but the structural gain is larger than the factor suggests: M2 no longer requires the trade logic to be expressible as array algebra. Any path-dependent rule that fits in a loop now runs at compiled speed.

Orchestration supplies the final 6.2×–8.4×. The remaining rungs change no kernel code at all; they only schedule the 80 independent combos across the 12 cores: 217.6× with `prange` threads, 297.9× with a process pool. This is the cheapest order of magnitude in the ladder—a dozen lines of orchestration around an unchanged kernel.

4.3 Threads versus processes

Against the single-thread M2, the thread rung scales 6.2× on 12 threads (51% parallel efficiency) and the process rung 8.4× (70%); M4 beats M3 by 1.37×. Sub-linear scaling is expected—the combo costs are skewed (window length, and hence cost, grows along the grid), the host mixes performance and efficiency cores, and the sweep is short enough that end-of-sweep straggling is visible. The direction of the M3/M4 gap is consistent with scheduling: the process pool hands out combos one at a time (chunk size 1), dynamically balancing the skewed costs, while `prange` partitions the combo range statically, so a thread that draws the expensive tail windows finishes last. We flag this ordering as the least portable claim in the paper: it is one run on one heterogeneous-core host, we did not instrument the schedules, and on other hosts (or other grid orderings) threads may well win—threads share the feature arrays, while processes pay serialization and per-worker warm-up that our timing deliberately excludes. Our tests assert only that both parallel rungs beat the single thread, not their mutual order.

4.4 What the equivalence gate certifies—and what it does not

All equivalence checks pass in the committed run: every rung, the pandas baseline included, matches M1 on all 80 combos (trade counts exactly; PnL to an absolute 10^{-6} with zero relative tolerance). The committed fingerprint of the first grid point ($L = 6$) is a PnL of -5165.58 percentage points over 57,029 trades.

We stress what this number is: a *correctness* fingerprint, not tradable alpha. On driftless GBM noise the shortest-window cross flips roughly every 2.6 bars, and 57,029 round trips at a 0.09-point fee cost 5,132.61 points; the gross PnL before fees is -32.97 points, i.e. near zero, exactly as a moving-average cross on a martingale should gross. The strategy loses its fee bill, as it must; the fingerprint's value is that five independent implementations agree on it to 10^{-6}. No performance claim about HMA crosses is made or implied anywhere in this paper.

5 Why a GPU is not the missing rung

The ladder ends at the CPU, and the reader may ask where the GPU rung is. The honest answer must begin with two concessions, one modeled and one measured. At a computed 10.78 FLOP/byte (11.07 as executed) the feature stage is not memory-bound, so the lazy dismissal—"backtests just stream data, GPUs can't help"—is wrong for this kernel [11]. More pointedly, the measured time split of Section 3.5 shows the compiled kernel spends 99.3% of its time in exactly this batchable WMA arithmetic: all 480 distinct window convolutions of the sweep could in principle be evaluated in a few large batched launches [8]. A familiar version of the no-GPU argument claims that the

serial event loop is where an optimized backtest spends its remaining time; our measurement refutes that—the loop is 0.7% of the kernel. A GPU could batch essentially all of the compute. The case that it is still not the missing rung is about leverage, not suitability, and it rests on two structural facts.

(a) The exploitable parallelism is 80 combos wide. The sweep exposes one unit of coarse-grained parallelism per combo; within a combo the feature passes are few and the event loop is a serial dependency chain. A GPU is a throughput machine that wants tens of thousands of resident threads to hide latency [8]; mapped one combo per thread the device would idle, and even the batched-GEMV formulation of the feature stage offers only 80 columns of independent work at a time. This is not an Amdahl serial-fraction effect [1]—almost everything here parallelizes—it is finite width: 80 tasks cannot occupy more than 80 coarse lanes, however wide the device. Nor is the CPU pool the binding constraint: at 51–70% parallel efficiency the work is finite and skewed, and more lanes—CPU or GPU alike—mostly buy idle lanes.

(b) The budget left to beat is 0.23 s. The input is small—150,000 float64 bars is 1.2 MB—so bulk bandwidth is not the issue; the issue is that the end-to-end time a GPU implementation must beat is a quarter of a second. On this host's unified memory the transfer term is minor, which moves kernel-launch and synchronization overhead to the foreground; on discrete-GPU CUDA systems host–device transfer joins it, the accounting Gregg and Hazelwood [3] showed routinely erases claimed GPU advantages on small kernels. Fixed overheads that are noise against the naive rung's 69.92 s are structural against 0.23 s: there is almost nothing left to amortize them over.

The event-loop residue. The stateful loop remains a poor SIMT fit—branchy, divergent, position carried bar to bar [2]—so a full-device port must either leave it on the CPU (adding a device–host synchronization to every sweep) or accept divergent scalar code on the device. But at 0.7% of the kernel this is a porting cost, not a performance argument, and we deliberately do not lean on it.

Platform note and the honest carve-out. On this host (Apple silicon) the would-be GPU path is MLX or PyTorch-MPS, not CUDA, and either requires rewriting the hot path for the device. We state the boundary of the claim plainly: for *this* workload shape—a modest combo grid, one series, path-dependent per-combo state, a quarter-second optimized budget—the missing rungs between 69.92 s and 0.23 s were engine and orchestration, and hardware was constant. Workloads shaped differently can absolutely justify a GPU: sweeps of 10^5–10^6 configurations, cross-sectional feature pipelines over many assets, stateless or batchable PnL rules, or ML training inside the loop are width- or compute-rich in exactly the way this kernel is not [8]. And we do not close the door here either: recasting this sweep as one large batched tensor program—every combo's features as columns of a single device-resident matrix computation, with the thin loop stage reformulated or left on the host—is a genuinely promising direction that deserves its own controlled measurement, not a dismissal from a laptop (Section 7).

6 Limitations

One host, wall-clock numbers. Every timing in this paper is a wall-clock measurement on one Apple M2 Max laptop; none of the constants (69.92 s, 297.9×, 51–70% efficiency) should be expected to transfer to other machines, and the M3-vs-M4 ordering in particular is host-specific

(Section 4.3). We commit one representative best-of-3 run and disclose the host, the BLAS backend and its thread pinning, and the library versions. What is deterministic and checked by the released tests is the equivalence of the rungs and the coarse ladder ordering (M0 < M1 < M2 < parallel rungs in combos/s), not the seconds.

One kernel family, synthetic data. The kernel is a single-asset HMA/HMA3 cross with seven WMA passes per combo and a two-state event loop, on GBM noise. Kernels with heavier per-bar state (order books, multi-asset portfolios, intrabar fill models) have different profiles—in particular the M1 rung's closed-form reducer does not exist for most of them, making the M0→M2 compiled-loop path the relevant one, and the 99.3%/0.7% feature/loop split of Section 3.5 would shift toward the loop. The data being synthetic is by design (perf is size-bound and the fingerprint deterministic), but it also means no claim here touches real-market behavior.

The baseline is deliberately naive. M0 is the profile of a first implementation, not of an expert `pandas` user; anyone who replaces `rolling().apply()` with array expressions has effectively moved to M1. The baseline is nonetheless measured in full and equivalence-checked on every combo—its naivety is in its code shape, not in any estimation—and the 13.1× M1→M4 figure is the reader's guard against strawman amplification.

The GPU analysis is analytical, not measured. We did not implement the batched MLX/MPS reformulation; Section 5 combines a measured CPU time split with a modeled, deliberately GPU-favorable intensity bound, and its conclusion is scoped to this workload shape on this class of budget. The batched "big-matrix" GPU direction remains untested here and is marked as future work, not refuted.

vectorbt and other engines are not measured. Our environment's Python version predates available vectorbt builds, so its published gains [9] are discussed as prior art only; the ladder does not rank frameworks.

Speed amplifies multiple testing. A 297.9× faster sweep evaluates 297.9× more configurations per unit of researcher patience, and the expected maximum of a search grows with its breadth [6]. Throughput is an instrument; without deflation and out-of-sample discipline it mostly manufactures overfitting faster. This paper prices the instrument, not the research program.

7 Conclusion

We built a five-rung speed ladder for one fixed, path-dependent backtest kernel and gated every rung on output equivalence over every combo, so that exactly one thing varies: implementation. On a 12-core laptop the ladder spans 69.92 s to 0.23 s—297.9× from the naive profile, and still 13.1× from a competently vectorized single-core baseline—with 22.7× from vectorization alone, 35.3× from a single compiled thread, and the rest from scheduling 80 independent combos across cores (217.6× threads, 297.9× processes; the process pool's dynamic balancing wins on this host, an ordering we do not claim travels). The GPU question, reframed by measurement: 99.3% of the compiled kernel is batchable feature arithmetic a GPU could genuinely run—the payoff dies on the 80-wide parallelism and the quarter-second budget, not on any unsuitability of the math. We therefore do not write the GPU off: recasting the sweep as one large batched tensor program over a much wider combo grid, or for a feedback-free kernel family, is a separate and promising direction

that deserves a dedicated measurement. The bottleneck in mass parameter search of this shape is the engine and the orchestration; the hardware was never the constraint. For the practitioner the recipe is three steps long: vectorize what vectorizes, JIT-compile the loop that does not, and parallelize across the sweep—and only then, if the workload's width and shape have genuinely outgrown the CPU, price an accelerator against the profile you measured.

Reproducibility. A single command regenerates `results.json`: `python scripts/run_all.py`, whose default `--repeats 3` matches the committed run. It measures all five rungs in full and re-runs the equivalence gate on every combo, while `--quick` runs a small smoke version and writes `results_quick.json`, never clobbering the canonical file. `scripts/check_paper_numbers.py` verifies that every number quoted in this paper matches the *committed* `results.json` at its printed precision (and that each quoted constant literally appears, with word-boundary matching, in the LaTeX source); `tests/` re-runs a reduced ladder and asserts the deterministic invariants—equivalence, ladder ordering, and the arithmetic-intensity bound—without asserting machine-specific wall times. The committed `results.json` is one representative best-of-3 run on the disclosed host (Apple M2 Max, 12 cores, `numba` 0.64.0, `numpy` 2.4.3 on Accelerate pinned to one BLAS thread, Python 3.14.6, seed 42).

References

[1] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18–20, 1967, Spring Joint Computer Conference (AFIPS '67 (Spring))*, pages 483–485, New York, NY, USA, 1967. ACM. doi: 10.1145/1465482.1465560.

[2] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 40)*, pages 407–420. IEEE Computer Society, 2007. doi: 10.1109/MICRO.2007.30.

[3] Chris Gregg and Kim Hazelwood. Where is the data? why you cannot debate CPU vs. GPU performance without the answer. In *2011 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 134–144. IEEE, 2011. doi: 10.1109/ISPASS.2011.5762730.

[4] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020. doi: 10.1038/s41586-020-2649-2.

[5] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: a LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC (LLVM '15)*, pages 1–6, New York, NY, USA, 2015. ACM. doi: 10.1145/2833157.2833162.

[6] Marcos López de Prado. *Advances in Financial Machine Learning*. John Wiley & Sons, Hoboken, NJ, 2018. ISBN 978-1-119-48208-6.

[7] Wes McKinney. Data structures for statistical computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference (SciPy 2010)*, pages 56–61, 2010.

[8] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, 2008. doi: 10.1145/1365490.1365500.

[9] Oleg Polakow. vectorbt: Vectorized backtesting and technical analysis for Python. Software, https://github.com/polakowo/vectorbt, 2021.

[10] Ritchie Vink and Polars contributors. Polars: DataFrames for the new era. Software, https://github.com/pola-rs/polars, 2025. Zenodo, https://doi.org/10.5281/zenodo.15313193.

[11] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009. doi: 10.1145/1498765.1498785.