UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS

DÉPARTEMENT DES SCIENCES ADMINISTRATIVES

WORKING PAPER NO. 40

RAID: A Metal-Accelerated C++ Framework for Reproducible High-Frequency Financial Econometrics

Batched, Fused and Streaming Estimation on Apple Silicon without Sacrificing Inference

Simon-Pierre Boucher

Département des sciences administratives
Université du Québec en Outaouais

simon-pierre.boucher@uqo.ca

Gatineau – Pavillon Alexandre-Taché
283, boulevard Alexandre-Taché
Gatineau, Québec, Canada J9A 1L8

November 2026

Version 1.0

Abstract

We ask whether heterogeneous high-frequency econometric workloads—realized measures, rolling and batched regressions, HAR forecasting models, GARCH-family quasi-maximum likelihood, block bootstrap inference—can be reorganised into batched, fused, streaming computation on Apple Silicon GPUs without sacrificing estimator accuracy, statistical inference, or reproducibility. Apple GPUs expose no 64-bit floating point. RAID answers with a C++20 library whose FP64 CPU backend is the reference, whose Metal backend accumulates every statistic in compensated "double-float" arithmetic and combines chunked partial sums in FP64, and whose estimators report backend, precision, convergence and provenance in every result. Tier A methods reproduce `highfrequency`, `rugarch`, `statsmodels` and `arch` to documented tolerances; Monte Carlo experiments confirm bias, RMSE and coverage; GPU and CPU results agree to 10^{-6} relative and are bitwise reproducible across runs and thread counts. Using one year of 1-minute bars for a seeded universe of 1,000 US equities served by a private FirstRate Data lakehouse, we measure break-even points on an M5 Max laptop and across eleven Apple Silicon generations (M2 to M6). The GPU wins on fused realized measures beyond roughly 4×10^4 elements and on batched GARCH likelihood evaluation when many parameter vectors are evaluated per series; the CPU wins on rolling sums, on single-vector GARCH recursions and on small panels. Conversion and buffer traffic, not kernel time, dominate GPU wall-clock: unified memory is not free computation. We publish the negative results, the tolerances, and a reproducible benchmark protocol.

Keywords: high-frequency econometrics, realized volatility, HAR, GARCH, GPU computing, Apple Metal, reproducibility, numerical precision

JEL Classification: C58, C63, C87, G17

1. Introduction

High-frequency financial econometrics is computationally repetitive in a particular way. The same small estimator—a sum of squared returns, a bipower product, a four-regressor regression, a GARCH recursion—is evaluated on thousands of series, thousands of days, and thousands of bootstrap or Monte Carlo replicas. The established toolchains that practitioners trust for correctness (`highfrequency` in R, Boudt et al., 2022; `rugarch`, Ghalanos, 2024; `arch` and `statsmodels` in Python, Sheppard, 2025; Seabold and Perktold, 2010) are organised around one series at a time and double-precision CPU arithmetic. Meanwhile the hardware on most researchers' desks has changed: Apple Silicon machines couple a multicore CPU with a wide GPU in a single unified memory space. The GPU, however, has no 64-bit floating-point unit (Apple Inc., 2026), and econometric estimators are not obviously tolerant of single precision.

This paper studies whether that hardware can be used for high-frequency econometrics *without* giving up what makes the established toolchains trustworthy: estimator accuracy, valid inference and reproducibility. We frame the question as three falsifiable research questions and build the software needed to answer them.

RQ1 (break-even). At what data and batch sizes does Metal beat an optimised multicore CPU once I/O, shader compilation, synchronisation and host–device transfer are counted?

RQ2 (batching and inference). Does batched estimation raise throughput while preserving coverage, convergence and the numerical agreement with FP64 references?

RQ3 (pipeline). Can an Arrow/Parquet-to-estimator pipeline materially reduce copies, memory pressure and end-to-end latency relative to the columnar engines researchers use today?

RAID (Research Architecture for Intraday Data) is the instrument: an open C++20 library with (i) an FP64 CPU backend that serves as the reference implementation of every method, (ii) an optional Metal backend in which every accumulator is carried as a compensated double-float pair (Dekker, 1971; Neumaier, 1974) and chunked partial sums are combined in FP64 on the CPU, (iii) a method registry that pins the exact convention of each estimator and the reference software it is compared with, (iv) nanobind Python bindings and a command-line tool, and (v) a secure client for the FinData OS lakehouse that serves FirstRate Data 1-minute bars with dataset versions, request identifiers and checksums.

Our contributions are deliberately engineering and measurement contributions, not new econometric methodology. First, a precision strategy for FP32-only GPUs that keeps realized measures within 10^{-6} of FP64, regression sufficient statistics within 10^{-6} and GARCH likelihoods within 2×10^{-5}, with bitwise reproducibility. Second, a validation protocol: every Tier A method is compared with at least one established implementation under identical conventions, with tolerances recorded in the registry and reproduced by the test suite. Third, a benchmark protocol with

randomised order, warm-ups, repeated measurements, bootstrap confidence intervals of medians, explicit cold and cached shader compilation, and manifests recording hardware, compiler, load and thermal state. Fourth, measured answers to RQ1–RQ3 on an M5 Max laptop and on eleven Apple Silicon generations, including the cases where the CPU wins.

The remainder of the paper is organised as follows. Section 2 positions the work relative to the econometric and systems literatures. Section 3 describes the architecture and API, Section 4 the estimator definitions and conventions, Section 5 the Metal execution and precision strategy, Section 6 the statistical correctness evidence. Sections 7–10 present the experimental setup, microbenchmarks, end-to-end benchmarks and financial applications. Sections 11–13 discuss threats to validity, reproducibility and licensing, and conclude.

2. Related literature

Realized measures and their software. The realized variance of Andersen et al. (2003) and Barndorff-Nielsen and Shephard (2002), bipower variation (Barndorff-Nielsen, 2004, 2005), realized semivariance (Barndorff-Nielsen et al., 2010), realized quarticity and jump proxies (Andersen et al., 2007; Huang, 2005) are the building blocks of Tier A. Their finite-sample constants and sampling conventions vary across implementations; Liu et al. (2015) document how much the choice of sampling scheme matters. Noise-robust estimators—two-scale (Zhang et al., 2005), realized kernels (Barndorff-Nielsen et al., 2008), pre-averaging (Jacod et al., 2009)—require tick or quote data and are outside the admissible inputs of this paper, which uses 1-minute OHLCV bars only. The reference implementation for our comparisons is the `highfrequency` package (Boudt et al., 2022).

HAR and GARCH. The HAR model of Corsi (2008) and its HARQ extension (Bollerslev et al., 2016), the signed-jump variants of Patton and Sheppard (2015), and the GARCH family (Engle, 1982; Bollerslev, 1986; Nelson, 1991; GLOSTEN et al., 1993) with quasi-maximum likelihood inference (Bollerslev and Wooldridge, 1992; Fiorentini et al., 1996) are the forecasting and conditional-variance workhorses we implement. We compare with `rugarch` (Ghalanos, 2024) and `arch` (Sheppard, 2025). Forecast losses follow Patton (2011); the horse race of Hansen and Lunde (2005) and the comparison framework of Diebold and Mariano (2002); Hansen et al. (2011a) motivate the loss-based evaluation, although model selection is not the object of this paper. Realized GARCH (Hansen et al., 2011b), HEAVY (Noureldin et al., 2011), DCC (Engle, 2002) and GARCH-MIDAS (Engle et al., 2013) are listed in the registry as not implemented.

Inference. Heteroskedasticity-robust (White, 1980), HAC (Newey and West, 1987, 1994) and cluster-robust (Cameron et al., 2011) covariances are implemented with the `statsmodels` con-

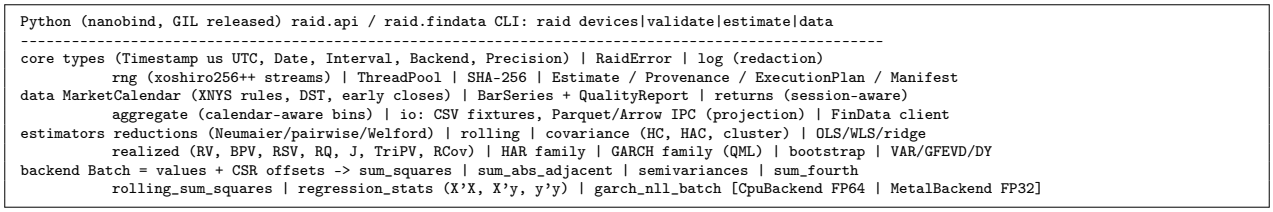


Figure 1. Layers of RAID. Estimators consume the backend interface through an `ExecutionPlan` (`auto|cpu|metal`); every result is an `Estimate` carrying coefficients, loss, inference, convergence, warnings, runtime, backend, precision and provenance.

ventions. Dependent-data resampling uses the stationary bootstrap (Politis and Romano, 1994); the wild bootstrap for realized volatility (Gonçalves and Meddahi, 2009) is a natural extension we do not pursue. Connectedness measures follow Diebold and Yilmaz (2012); Diebold and Yılmaz (2014) with the generalized FEVD of Koop et al. (1996) and Pesaran and Shin (1998); covariance shrinkage follows Ledoit and Wolf (2004).

Numerical computing and GPUs in economics. Compensated summation goes back to Kahan (1965), Dekker (1971) and Neumaier (1974); Higham (1993) gives the error analysis we rely on for the double-float accumulators. Aldrich et al. (2011) introduced GPU computing to dynamic equilibrium models; the econometric literature on GPUs has concentrated on Monte Carlo and Bayesian computation rather than on the estimators studied here. Our systems dependencies are Eigen (Guennebaud et al., 2025) with Apple Accelerate BLAS/LAPACK (Blackford et al., 2002), Apache Arrow and Parquet (Apache Software Foundation, 2025), nanobind (Jakob, 2025), and the Metal Shading Language (Apple Inc., 2026). The comparison engines for the pipeline experiment are pandas, Polars and DuckDB on top of NumPy (Harris et al., 2020) and SciPy (Virtanen et al., 2020).

3. Architecture and API

RAID is a C++20 library (MIT licence) with four layers (Figure 1): a *core* of strong types and infrastructure, a *data* layer that turns provider bars into validated, calendar-aware return panels, a *statistics/estimators/models* layer that implements the registry, and a *backend* layer that exposes a small set of batched numerical primitives with a CPU reference implementation and a Metal implementation. Python bindings (nanobind) and a command-line tool sit on top.

Core types. Timestamps are microseconds since the Unix epoch in UTC and are never mixed with raw integers; dates use a proleptic-Gregorian day count; intervals are an enumeration. Every estimator returns an `Estimate`: named coefficients, a loss (residual sum of squares or negative

log-likelihood, documented per model), an optional `InferenceResult` (standard errors, t-statistics, p-values, covariance, covariance type), a `Convergence` record (status, iterations, function evaluations, gradient norm, message), warnings, runtime, the backend and precision actually used, and a `Provenance` record (dataset identifier and version, request identifiers, input hashes, code version). `Estimate::ok()` requires convergence; a failed optimisation is never reported as a success. An `ExecutionPlan` selects the backend, the precision policy (FP32 chunk length, compensated accumulation, validation tolerance), determinism, threads and fallback behaviour.

Data layer. `BarSeries` stores OHLCV columns, bar-open timestamps and a session index. The `MarketCalendar` implements the NYSE rules (weekday holidays with the Saturday/Sunday observance rules, Good Friday from the Gregorian Easter, Juneteenth since 2022, special closures, early closes at 13:00, US daylight-saving transitions before and after 2007) and is validated set-for-set against `exchange_calendars` over 2000–2026. `validate()` counts OHLC inconsistencies, non-positive prices, grid misalignment, bars outside the session policy, non-trading days, duplicates, price jumps and the number of regular bars missing relative to the calendar (391 on a normal day, 211 on an early close, the 16:00 closing print included); prices are never altered and bars are never manufactured. Returns are computed within sessions only; the overnight return and the daily open-to-close and close-to-close returns are separate outputs. Aggregation to coarser intervals bins bars from the session open and keeps the closing print as its own bin, which reproduces the provider's derived 5-minute series exactly (Section 7).

Panels and batches. Batched kernels take a *CSR batch*: one contiguous array of values and an offsets array delimiting segments (a segment is a session, a series, or a window). Ragged panels—different session lengths, early closes, delisted series—are therefore first-class, and no padding is needed. Rectangular panels get a dense column-major view for linear algebra.

Backend contract. The backend interface has seven primitives (Figure 1). Each has an FP64 CPU implementation (compensated per-segment sums, parallel over segments) and a Metal implementation. Estimators are backend-independent: `realized_measures_batch`, `har_batch` and the GARCH grid screen call the primitives and finish in FP64 on the CPU (finite-sample constants, solves, inference). `resolve(plan)` maps `auto` to Metal when a device is available and falls back to the CPU otherwise, recording the reason.

FinData client. The data provider is FinData OS, a private lakehouse serving FirstRate Data bars through a versioned REST API (Section 7). The client authenticates with a bearer key read from the environment only, redacts secrets from logs, applies a token bucket (5 requests per second against a

600 per minute limit), retries with exponential backoff and `Retry-After`, follows cursor pagination of Arrow IPC pages, chunks requests to the documented caps (31 days of 1-minute bars per call, 12 years per export), polls asynchronous Parquet exports, and keeps a checksum-verified cache keyed by dataset version. Every exchange's request identifier and dataset version are kept for provenance.

Python and CLI. The nanobind module releases the GIL around every call, accepts NumPy arrays as views and returns NumPy arrays and JSON-like dictionaries. `raid.findata` exposes the client; `raid.api` exposes the estimators with a `backend` argument. The CLI offers `raid devices`, `raid validate`, `raid estimate realized|har|garch|ols` and `raid data catalog|bars|export|sessions|c`

4. Estimator definitions and conventions

The method registry (`docs/methods/registry.md` in the repository) records, for each method, the equation, the inputs, the reference implementation and its options, the test tolerance and the validation status. We reproduce the Tier A definitions here because conventions—not formulas—are where implementations diverge.

Returns and sessions. With bar closes $p_{t,0},\dots,p_{t,M_t}$ in session t (regular hours 09:30–16:00 ET inclusive of the 16:00 closing print), intraday log returns are $r_{t,j}=\log(p_{t,j}/p_{t,j-1})$, $j=1,\dots,M_t$; a normal session has $M_t=390$ and an early close $M_t=210$ when no minute is missing. Zero-volume minutes are absent at the source and are left absent.

Realized measures. For one session with M returns,

$$\mathrm{RV}=\sum_{j=1}^{M} r_j^2, \qquad \mathrm{BPV}=\mu_1^{-2}\,\frac{M}{M-1}\sum_{j=2}^{M}|r_j|\,|r_{j-1}|, \quad \mu_1=\sqrt{2/\pi}, \tag{1}$$

$$\mathrm{RSV}^-=\sum_j r_j^2 \mathbb{1}\{r_j<0\}, \qquad \mathrm{RSV}^+=\sum_j r_j^2 \mathbb{1}\{r_j>0\}, \qquad \mathrm{RQ}=\frac{M}{3}\sum_{j=1}^{M} r_j^4, \tag{2}$$

$$J=\max(\mathrm{RV}-\mathrm{BPV},0), \qquad \mathrm{TriPV}=\mu_{4/3}^{-3}\,\frac{M}{M-2}\sum_{j=3}^{M}|r_j r_{j-1} r_{j-2}|^{4/3}, \tag{3}$$

with $\mu_p=\mathbb{E}|Z|^p$ for $Z\sim N(0,1)$. The finite-sample factors are options. Two conventions of `highfrequency` differ: `rBPCov` with `makeReturns = FALSE` applies no $M/(M-1)$ factor, and `rQuar` uses $(M+1)/3$, the number of prices, instead of $M/3$; the reference tests compare after conversion. Realized covariance on synchronised returns is $\sum_j r_j r_j'$ without demeaning.

Regression and inference. OLS, WLS and ridge are solved by column-pivoting Householder QR (SVD when rank-deficient; ridge by augmentation). Covariances follow `statsmodels`: classical $s^2(X'X)^{-1}$; HC0–HC3 with the leverage corrections; HAC with Bartlett weights and L lags (default $\lfloor 4(n/100)^{2/9} \rfloor$, no small-sample correction); one-way cluster with $\frac{G}{G-1}\frac{n-1}{n-k}$. p-values use $t(n-k)$. Rolling OLS refits each window.

HAR family. With daily RV_t, lags $(1,5,22)$ and horizon h,

$$y_{t+h} = \beta_0 + \beta_d \mathrm{RV}_t + \beta_w \overline{\mathrm{RV}}_t^{(5)} + \beta_m \overline{\mathrm{RV}}_t^{(22)} \; [+ \beta_q (\sqrt{\mathrm{RQ}_t} - \sqrt{\overline{\mathrm{RQ}}}) \mathrm{RV}_t] + \varepsilon_{t+h}, \qquad \overline{\mathrm{RV}}_t^{(k)} = \tfrac{1}{k} \sum_{j=0}^{k-1} \mathrm{RV}_{t-j}, \tag{4}$$

where y_{t+h} is the average of $\mathrm{RV}_{t+1}, \ldots, \mathrm{RV}_{t+h}$ (the single-day target is an option) and the bracketed HARQ term follows Bollerslev et al. (2016) in the demeaned form used by `highfrequency`; log-HAR applies the same regression to $\log \mathrm{RV}$ and back-transforms with exp (the smearing factor is reported, not applied). Overlapping targets for $h > 1$ induce $\mathrm{MA}(h-1)$ errors, so the HAC lag is at least $h-1$. Rolling estimation is strictly causal: the model fitted on observations up to t forecasts y_{t+h}. Forecast losses are MSE and $\mathrm{QLIKE} = y/\hat{y} - \log(y/\hat{y}) - 1$ (Patton, 2011).

GARCH family. With $\varepsilon_t = r_t - \mu$,

$$\mathrm{GARCH}(1,1): \sigma_t^2 = \omega + \alpha \varepsilon_{t-1}^2 + \beta \sigma_{t-1}^2, \qquad \mathrm{GJR}: \; \sigma_t^2 = \omega + (\alpha + \gamma \mathbb{1}\{\varepsilon_{t-1} < 0\}) \varepsilon_{t-1}^2 + \beta \sigma_{t-1}^2, \tag{5}$$

$$\mathrm{EGARCH}: \log \sigma_t^2 = \omega + \alpha(|z_{t-1}| - \mathbb{E}|z|) + \gamma z_{t-1} + \beta \log \sigma_{t-1}^2, \quad z = \varepsilon/\sigma. \tag{6}$$

Errors are Gaussian or standardised Student-t with $\nu \in (2.05, 500)$ (the bounds of `arch`); σ_0^2 is the sample variance of ε (as `rugarch`; `arch` uses a backcast, which explains the 5×10^{-4} log-likelihood tolerance against it). Estimation is quasi-maximum likelihood by BFGS on transformed parameters (log for ω, logistic for persistence and the α/β share, tanh for the EGARCH β), with central-difference gradients validated across step sizes; up to two alternative starting points are tried on non-convergence, and the number tried is reported. Standard errors are the Bollerslev–Wooldridge sandwich $H^{-1} \mathrm{OPG} H^{-1}$ by default (Hessian and OPG are options). Forecasts are analytical for GARCH/GJR and by seeded simulation beyond one step for EGARCH. The `rugarch` EGARCH maps its `alpha1` to our γ and its `gamma1` to our α.

Bootstrap. The stationary bootstrap of Politis and Romano (1994) draws blocks of geometric length (mean b, default $n^{1/3}$) starting at uniform indices with circular wrap; the circular block bootstrap uses fixed length; an iid option exists for comparison only. Replica r uses the stream

(seed, r) of a counter-seeded xoshiro256++ generator, so replicas are independent of the thread that computes them.

Tier B. VAR(p) with intercept by multivariate least squares, generalized FEVD (Koop et al., 1996; Pesaran and Shin, 1998) normalised row-wise, Diebold–Yilmaz total, from, to and net connectedness (Diebold and Yilmaz, 2012; Diebold and Yılmaz, 2014) with rolling windows, and Ledoit–Wolf shrinkage toward the scaled identity (Ledoit and Wolf, 2004). Realized GARCH, HEAVY, DCC, GARCH-MIDAS, state-space and change-point methods, pre-averaging, realized kernels and order-flow methods are *not* implemented; the registry says so.

5. Metal execution and precision strategy

The constraint. Apple GPUs and the Metal Shading Language have no 64-bit floating-point type; a kernel declaring a `double` does not compile ("`'double' is not supported in Metal`"). Econometric accumulators—sums of 4×10^5 squared returns of magnitude 10^{-6}, Gram matrices $X'X$ with regressors of very different scales, log-likelihoods summed over thousands of observations—lose several digits in naive FP32. The strategy has four parts.

1. Double-float accumulation. Every accumulator on the GPU is a pair (h, ℓ) of `floats` representing $h + \ell$ with $|\ell| \le \frac{1}{2}\,\mathrm{ulp}(h)$. Additions use the error-free transformation TwoSum (Dekker, 1971; Neumaier, 1974): $s = h + v$, $e = (h - (s - (s - h))) + (v - (s - h))$, followed by a renormalisation. The pair carries about 48 significant bits; the error analysis of Higham (1993) gives a bound of order $n\,u^2$ rather than $n\,u$ for a sum of n terms with unit roundoff u. The Metal compiler must not reassociate: the kernels are compiled with `-fno-fast-math` offline and with `fastMathEnabled = NO` at runtime. Per-element inputs are still rounded to FP32 on the way in; for a sum of squares of M independent returns this input rounding contributes a relative error of order $2u\sqrt{3/M}$, i.e. about 10^{-8} for $M = 390$, which is what we observe (Section 6).

2. Chunked partial sums combined in FP64. A segment longer than the chunk length (4 096 elements by default, a `PrecisionPolicy` parameter) is split into chunks; one threadgroup of 256 threads reduces each chunk with a fixed-order tree reduction of double-float values and writes a `float2`; the host adds the chunks in FP64 in a fixed order and adds the chunk-boundary cross terms ($|r_j||r_{j-1}|$ across a boundary) in FP64. The combination order is deterministic, so results are bitwise reproducible across runs for a given chunk length and threadgroup width (verified in Section 6).

3. Mixed precision by design. Quantities that are cheap and precision-critical stay in FP64 on the CPU: the finite-sample constants, the initial variance of a GARCH recursion, the Student-t

density constants (Metal has no `lgamma`), the regression solve from the GPU-accumulated sufficient statistics, and all inference. The batched GARCH kernel scales each series to unit variance on the host (FP64) and un-scales the likelihood by $n \log s$, which conditions the FP32 recursion for returns of magnitude 10^{-2}.

4. Kernel fusion. The realized kernel computes the five accumulators of (1)–(2) in a single pass over the data; the four-kernel variant exists only for the ablation. Regression statistics accumulate the upper triangle of $X'X$, $X'y$ and $y'y$ ($k \leq 8$) in registers. The GARCH kernel runs one recursion per thread for one (series, parameter vector) pair—the parallelism comes from the number of parameter vectors, not from the recursion.

Host side. The backend owns one device, one command queue, one library and one pipeline cache per process; buffers come from a pool keyed by power-of-two size class in shared storage (unified memory). The shader library is loaded from a prebuilt `.metallib` when present and compiled from the embedded MSL source otherwise; both paths are measured. GPU time is read from the command buffer's `GPUStartTime`/`GPUEndTime`; wall-clock includes FP64→FP32 conversion into the shared buffer, encoding, commit and completion. Metal does not expose the GPU core count; the manifest records it from `system_profiler`.

What the GPU does not do. Optimisation drivers (BFGS line searches) run on the CPU; the GPU evaluates batches of likelihoods. Rolling sums use a direct $O(w)$-per-output kernel, which the experiments show to be slower than the CPU's $O(1)$ running sum—a limitation we report rather than hide. Nothing in the Metal path is a new estimator.

6. Statistical correctness

Every Tier A method satisfies five requirements: an independent FP64 CPU reference, an analytical or simulation truth where one exists, agreement with established software under identical conventions, GPU/CPU agreement within a documented tolerance, and adversarial tests. Table 1 summarises the reference comparisons on committed synthetic fixtures (no vendor data); every row is a test in the repository that runs on every build.

Monte Carlo. Table 2 (experiment E5, $R = 1\,000$ replications) reports bias, RMSE and the empirical coverage of 95 % intervals. For GARCH(1,1) with $(\omega, \alpha, \beta) = (0.02, 0.08, 0.90)$ and $n = 2\,000$, biases are at most 0.005 in absolute value, coverage of the QMLE intervals is 0.93–0.97, and all 1 000 fits converge; at $n = 500$ the familiar finite-sample downward bias of β (-0.026) and

Table 1. Reference validation on synthetic fixtures (all tests pass).

Method	Reference (version)	Quantity compared	Tolerance
XNYS calendar	`exchange_calendars` 4.13.2	sessions, holidays, early closes 2000–2026 (6 790 sessions)	exact sets
Session aggregation 1m→5m	FinData derived 5m (live)	OHLCV of 280 regular 5-minute bars (AAPL, Nov. 2024)	exact
RV, BPV, RSV$^{\pm}$, RQ	`highfrequency` 1.0.3 / NumPy	per-session values (10 sessions)	10^{-10} / 10^{-12} rel.
OLS, HC0–HC3, HAC, cluster	`statsmodels` 0.15.0	coefficients, SE, p-values, R^2, SSR	10^{-10} / 10^{-9}
t p-values	`scipy` 1.18.1	$2t$.sf	10^{-12}
HAR-RV $h = 1, 5, 22$, log-HAR, HARQ (HAC)	`statsmodels` 0.15.0; `highfrequency` `HARmodel`	coefficients, HAC SE	10^{-9}; 10^{-8} ($h = 1$), 10^{-6} (HARQ, $h = 5$)
GARCH, GJR, EGARCH, GARCH-t	`arch` 8.0.0; `rugarch` 1.5.6	log-likelihood; parameters; $h \leq 5$ forecasts	5×10^{-4} / 2×10^{-4}; 5 % + 0.01; 5 %
VAR(2), FEVD, GFEVD, DY, Ledoit–Wolf	`statsmodels` 0.15.0 (GFEVD by hand)	coefficients, Σ_u, llf, FEVD, GFEVD, totals, shrinkage	10^{-9} (10^{-7} totals)
Metal vs CPU FP64	RAID CPU backend	RV, BPV, RSV$^{\pm}$ (n = 390–39 000, chunked)	10^{-6} rel. (Σx^4: 10^{-5})
		rolling sums; $X'X$, $X'y$, $y'y$ and the solve	10^{-5}; 10^{-6} (solve 10^{-5})
		batched GARCH NLL (GJR, Student-t); argmin over a grid	2×10^{-5} (5×10^{-5}); identical

Conventions converted before comparison: `rQuar` uses $(M+1)/3$; `rBPCov(makeReturns=FALSE)` applies no $M/(M-1)$; HARQ uses $(\sqrt{\mathrm{RQ}_t} - \sqrt{\overline{\mathrm{RQ}}})\mathrm{RV}_t$; `rugarch` EGARCH `alpha1`/`gamma1` are swapped relative to (6); `arch` initialises σ_0^2 by backcasting. Student-t ν is compared only when identified ($\nu < 100$).

under-coverage of α (0.90) appear. For HAR with overlapping $h = 5$ targets and HAC inference, coverage of the pseudo-true projection coefficients is 0.90–0.94 ($T = 1\,200$), the usual mild under-coverage of HAC. Realized variance converges at the $\sqrt{2/M}$ rate: the Monte Carlo RMSE is within 10 % of $\sqrt{2/M}$ IV for $M \in \{39, 390, 3\,900\}$. The stationary-bootstrap interval for the mean of an AR(1) covers in 89 % of replications ($R = 100$, $n = 800$, $b = 10$).

Reproducibility. With the same seed, the 2 000 bootstrap statistics are bitwise identical when computed with 1, 4 or 18 threads; the CPU realized measures are bitwise identical with 1 or 18 threads; two Metal runs give bitwise identical RV, BPV and RQ; and the maximum relative difference between Metal and CPU FP64 realized variance on 50 segments of Student-$t(5)$ returns is 3.7×10^{-8}, consistent with the input-rounding bound of Section 5.

Adversarial cases. The test suite covers zero and missing volume, holidays, DST transitions (both rules), the 16:00 print, early closes, duplicated and unsorted bars, rank-deficient and ill-conditioned designs (warned, minimum-norm solution), singular $X'X$ from sufficient statistics (pseudo-inverse, warned), invalid GARCH parameters ($+\infty$ likelihood on both backends), non-convergence (never reported as success), insufficient samples (typed exceptions), nonexistent Metal devices and forced fallback. Numerical gradients of the GARCH likelihood agree across step sizes 10^{-4}, 10^{-5} and 10^{-6} to 10^{-3} relative.

Table 2. E5 — Monte Carlo bias, RMSE and 95 % coverage ($R = 1\,000$; HAR coverage listed for the four coefficients).

DGP	parameter	bias	RMSE	coverage 95%	converged
GARCH(1,1) $n = 500$	μ	-0.0005	0.0383	0.950	1000/1000
GARCH(1,1) $n = 500$	ω	+0.0201	0.0487	0.954	1000/1000
GARCH(1,1) $n = 500$	α	+0.0021	0.0304	0.896	1000/1000
GARCH(1,1) $n = 500$	β	-0.0256	0.0709	0.916	1000/1000
GARCH(1,1) $n = 2000$	μ	+0.0001	0.0190	0.961	1000/1000
GARCH(1,1) $n = 2000$	ω	+0.0034	0.0093	0.972	1000/1000
GARCH(1,1) $n = 2000$	α	+0.0007	0.0137	0.932	1000/1000
GARCH(1,1) $n = 2000$	β	-0.0048	0.0196	0.936	1000/1000
HAR $h = 1$ (HAC)	$\beta_0, \beta_d, \beta_w, \beta_m$	–	–	0.945, 0.939, 0.946, 0.941	1000/1000
HAR $h = 5$ (HAC)	$\beta_0, \beta_d, \beta_w, \beta_m$	–	–	0.909, 0.936, 0.931, 0.900	1000/1000

7. Experimental setup

Data. All market data come from FinData OS, the author's private lakehouse (`https://findata.spboucher.ai/api/v1`) built on FirstRate Data US equity bars under a private-use licence; raw bars are not redistributed and only aggregates appear in this paper. Bars are 1-minute OHLCV, stamped at the bar *open* in UTC, with the extended session 04:00–20:00 ET and the regular session 09:30–16:00 ET (the 16:00 bar is the closing print); zero-volume minutes are absent at the source. The universe (experiment E0) is a seeded random sample of 1 000 symbols among the 6 046 of the `market_bars.stock.1m.raw` dataset that were active through all of 2024 (first session on or before 2024-01-02, last on or after 2024-12-31, delisted suffixes excluded); nested sub-samples of 1, 10 and 100 symbols are its first elements. Twenty asynchronous Parquet exports (50 symbols each, 43.3 million bars, 0.63 GB, dataset version 17 375, 437 s including polling) are cached with their SHA-256 and request identifiers. Of the 1 000 symbols, 888 have at least 200 regular sessions with positive realized variance and 926 at least 200 sessions of returns; the experiments report the counts they use. For E3 we add ten years of daily bars (2015–2024, 1.82 million rows) for the same symbols. All experiment inputs are listed with hashes in `experiments/manifests/`.

The provider's own conventions were verified live before use (opt-in integration tests): the per-session expected bar counts (391; 211 on early closes) equal RAID's calendar for every session queried; page downloads equal exports bar for bar; and RAID's calendar-aware 1-minute to 5-minute aggregation in extended-session mode reproduces the provider's derived 5-minute dataset exactly (280 of 280 regular-window bars, OHLCV), while in regular-session mode the only difference is the closing bin, where the provider's 16:00 bar also contains the 16:01–16:04 post-close minutes.

Hardware and software. The development and benchmark machine is an Apple M5 Max laptop: 18 CPU cores, 40 GPU cores, 48 GB unified memory, macOS 27.0, Xcode 27.2, Apple clang 21, Metal 4 (GPU families Apple1–Apple10). Dependencies: Eigen 5.0.1 with Accelerate BLAS/LAPACK, Apache Arrow/Parquet 26.0.0, Catch2 3.16.1, nanobind 3.1.0, Python 3.12 with NumPy 2.5.4, pandas, Polars 2.1.0, DuckDB 2.0.0, statsmodels 0.15.0, arch 8.0.0, SciPy 1.18.1; R 4.6.1 with `highfrequency` 1.0.3 and `rugarch` 1.5.6. For E7, eleven other Apple Silicon machines of the author's cluster (M2 to M6, 8 to 96 GB, 10 to 60 GPU cores) run a self-contained bundle built with `-mcpu=apple-m1` and a macOS 26 deployment target; nothing is installed on them, and the hosts that serve production services were excluded.

Protocol. Each benchmark case is run twice as warm-up and then R times ($R = 5$ on the laptop, 3 on cluster nodes); case order is shuffled with a fixed seed; the reported statistic is the median of wall-clock seconds with a 2 000-draw bootstrap 95 % confidence interval of the median. A backend "wins" when the intervals do not overlap; otherwise the case is a tie. Wall-clock for Metal includes the FP64→FP32 conversion into shared buffers, encoding, commit and completion; GPU device time is reported separately. Shader compilation is measured apart (cold, cache-busted with a salted source; warm; and prebuilt library). Each run writes JSON records and a manifest with host, OS, compiler, library versions, load average, thermal state, seed and input hashes; peak resident set size is read from `getrusage`. The laptop was shared with other sessions during the runs (load average 2.5–4 on 18 cores), which the manifests record; cluster nodes were idle. Energy was not measured (no instrument), and we make no energy claim.

Experiments. E1: batched realized measures versus panel width and segment length. E2: HAR rolling forecasts and batched estimation. E3: GARCH-family panel estimation and a GPU likelihood grid. E4: realized covariance and rolling connectedness. E5: Monte Carlo inference and reproducibility (Section 6). E6: the Arrow/Parquet-to-estimator pipeline against pandas, Polars and DuckDB. E7: the generation sweep. Micro-benchmarks of the seven primitives precede them.

8. Microbenchmarks

Primitives on the laptop. Table 3 (abridged; the full grid is in the repository) and Figure 2 report the fused realized kernel, regression sufficient statistics ($k = 4$), the rolling sum of squares ($w = 22$) and the batched GARCH negative log-likelihood with one ($P = 1$) and 64 ($P = 64$) parameter vectors per series, for series counts $S \in \{1, 10, 100, 1\,000, 10\,000\}$ and lengths $n \in \{390, 3\,900, 39\,000, 390\,000\}$ of Student-$t(5)$ synthetic returns.

Four findings answer RQ1 at the kernel level. First, the fused realized kernel is faster than

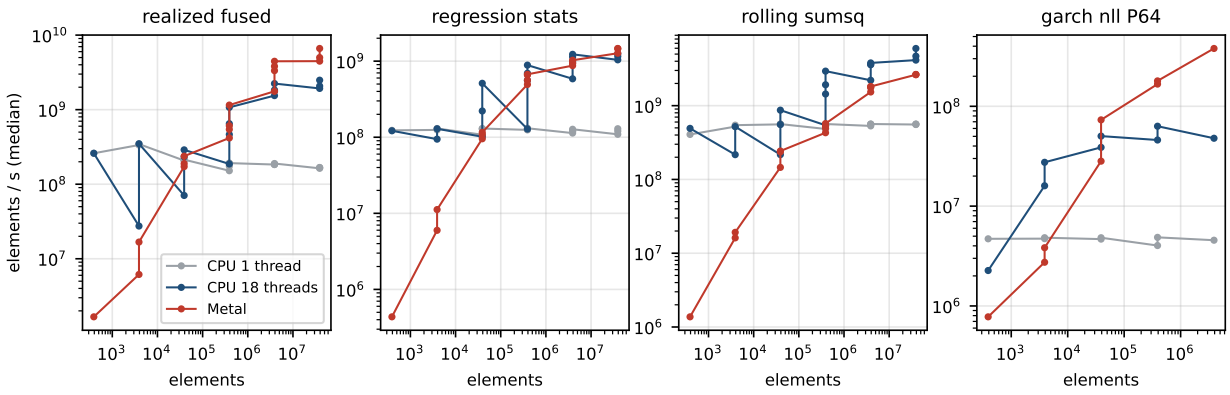


Figure 2. Throughput of the primitives (elements per second, medians) versus problem size on the laptop.

the 18-thread CPU from about 4×10^4 elements upward (100×390: 2.4×) and reaches 2.0–3.4× between 3.9×10^6 and 3.9×10^7 elements; a few shapes at 3.9×10^5 elements are ties. Second, regression statistics break even only around 4×10^5 elements for a single long series (3.9×), are ties for most shapes between 4×10^5 and 4×10^6, and gain 1.1–1.4× at 4×10^7: the per-row cost on the CPU is already small and the GPU pays conversion and launch overhead. Third, the rolling sum of squares is a negative result: the direct $O(w)$ kernel never beats the CPU's $O(1)$ running sum (0.4–0.9× of CPU MT at every size). Fourth, batched GARCH likelihoods are the clearest GPU case, but only when many parameter vectors are evaluated per series: with $P = 1$ the CPU wins everywhere (the GPU is launch-bound and the recursion is sequential), with $P = 64$ Metal wins from 100 series (1.9×) up to 7.9× at $1\,000 \times 3\,900$. For the reductions, device time is 5–20 times smaller than wall-clock: FP64→FP32 conversion and buffer fills, not the kernels, set the wall-clock. Unified memory removes the copy to the device but not the conversion.

Shader compilation and library loading. Loading the prebuilt library takes 0.2 ms and the first pipeline creation 14 ms on the laptop; compiling the embedded MSL source at runtime takes 39 ms when the system shader cache is warm and 123 ms cold (compiler service start included). On cluster nodes the cold compile is 37–49 ms (110 ms on the 8 GB M2 mini) and the warm compile 0.3–0.6 ms (Table 4). These costs are paid once per process and matter only for workloads of a few milliseconds.

Generation sweep (E7). Table 4 and Figure 3 report the same bundle on eleven machines. All Metal tests pass on every machine with both the prebuilt library and runtime compilation, including the 8 GB M2 mini without developer tools. The speedup of the fused realized kernel at $1\,000 \times 3\,900$ ranges from 4.0× (M2, 10 GPU cores, 8 CPU cores) and 3.6× (M2 Max) down to 1.1× on the M5 Max and M5 Pro; GARCH $P = 64$ from 7.8× (M2 Max) to 1.8× (M5 Pro); the rolling kernel

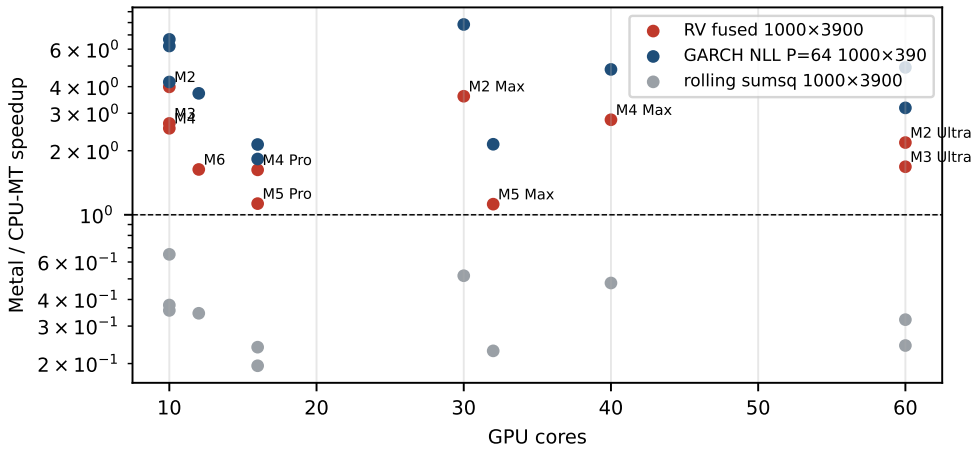


Figure 3. E7 — Metal/CPU-MT speedup versus GPU core count for three kernels on eleven Apple Silicon machines.

loses everywhere (0.2–0.65×). The pattern is not "more GPU cores, more speedup": the ratio is governed by the CPU's multicore throughput, which grew faster across generations than the GPU throughput of these memory-bound kernels, and by the conversion overhead, which is CPU-bound. The GPU helps most on the machines with the weakest CPUs.

9. End-to-end benchmarks

E1 — batched realized measures on the real panel. Table 5 and Figure 4 report the full estimator path (`realized_measures_batch`: the five fused accumulators, finite-sample constants, jump proxy, tripower quarticity and daily returns) on the 2024 panel, with segments of one day, one week, one month or one year of regular-session 1-minute returns, for panel widths of 1, 10, 100 and 926 symbols, and on a synthetic 10 000-series panel. Three facts stand out. First, at the estimator level the GPU advantage is much smaller than at the kernel level: on 40 million real returns the 18-thread CPU needs 29–30 ms and Metal 20–28 ms (1.07–1.41×), and at 4.8×10^6 returns or fewer the two are statistically tied or the CPU wins; the single-thread CPU is 8× slower than either. The CPU post-processing that the GPU does not remove—conversion to FP32, tripower in FP64, the per-segment finishing—bounds the achievable speedup (Amdahl's law), and the fused kernel's 2–3× on raw reductions shrinks accordingly. Second, kernel fusion matters more than the device: the four-kernel Metal variant is 1.4–3.2× slower than the fused one on the same data, because each extra pass re-reads the panel and pays its own launch and synchronisation. Third, accuracy is unaffected: the maximum relative difference between Metal and FP64 realized variance over all 926×252 session segments is 1.6×10^{-7} and decreases with segment length (1.4×10^{-8} for yearly segments), exactly the input-rounding behaviour predicted in Section 5. Longer segments

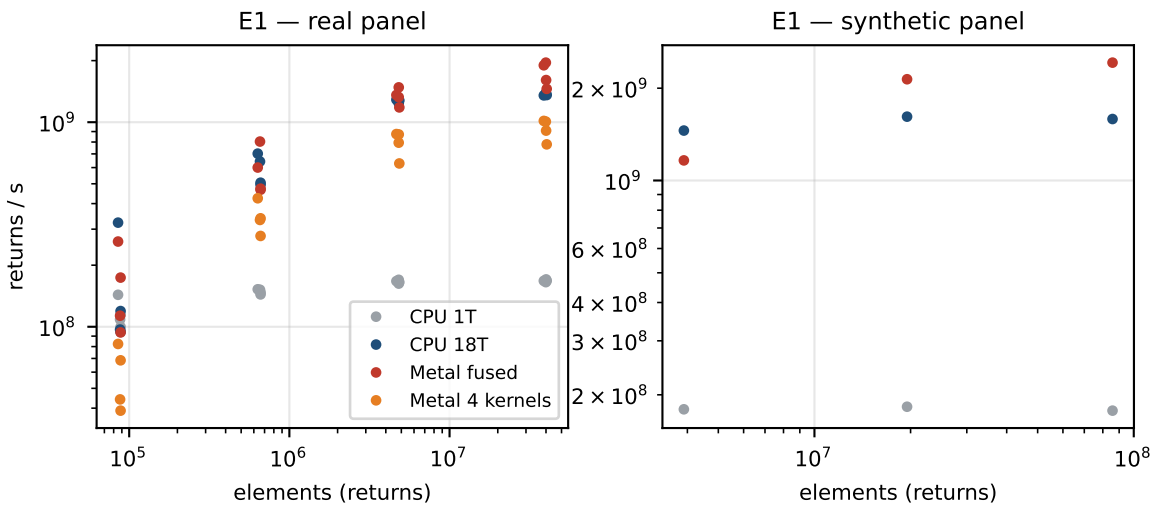


Figure 4. E1 — throughput (returns per second) of the realized-measures estimator by backend on the real and synthetic panels.

favour the GPU (month and year segments are the Metal wins); many tiny segments favour the CPU.

E6 — the Arrow/Parquet-to-estimator pipeline. Table 6 compares, on four of the cached export files (2.2–2.5 million bars, 50 symbols, one year each), the RAID pipeline—Parquet read with column projection and threaded decoding, session filtering with the exact calendar, within-session log returns, and the five realized measures per session—with the same computation written in pandas, Polars (lazy scan) and DuckDB (SQL window functions). The baselines use an approximate early-close rule (fixed dates) because none of them ships an exchange calendar. RAID on the CPU processes a file in 0.118 s (20 million bars per second), Polars in 0.165 s and DuckDB in 0.164 s (both 14 million bars per second), pandas in 2.56 s. Within RAID, decoding the Parquet columns takes 0.065 s, session filtering and returns 0.021 s and the estimators 0.031 s on the CPU; Metal makes the estimator stage slower (0.037 s) because the stage is below the break-even size of E1 and the conversion is paid again. The answer to RQ3 is therefore moderate: a typed, projected, calendar-exact C++ pipeline is about 1.4× faster than the best columnar engines and 22× faster than pandas on this task, and the GPU plays no role in it. Before column projection was added, the same pipeline took 0.35 s—reading eight of twenty columns instead of all of them, and avoiding the per-row string column, was worth more than any kernel. True zero-copy from Arrow to the estimators was possible for every file (single-chunk, null-free numeric columns), but the session filter necessarily materialises a selection; we count four column copies per file.

10. Financial applications

The applications are not forecasting studies; they show that the batched estimators deliver the same statistical content as per-series estimation on real data, and where batching pays.

E2 — HAR forecasting and batched estimation. For the first 100 symbols of the universe, HAR-RV, log-HAR and HARQ are re-estimated every day on a causal 120-session window and forecast the average realized variance over the next $h \in \{1, 5, 22\}$ sessions; the benchmark is the random-walk forecast $\widehat{\mathrm{RV}}_{t+h} = \mathrm{RV}_t$ (Table 7). With only one year of data the windows are short, yet the median QLIKE of HAR-RV is 0.196, 0.146 and 0.134 at $h = 1, 5, 22$ against 0.251, 0.239 and 0.301 for the random walk, and HAR beats the random walk for 84 %, 86 % and 78 % of symbols; log-HAR is best at the longer horizons (0.138 and 0.112), consistent with the literature. Table 8 reports the runtime of estimating HAR-RV ($h = 1$, 252 daily observations per series) on 10, 100 and 781 series: per-series QR fits take 0.08–6.0 ms on one thread and 0.05–0.68 ms on 18 threads; the batched path (sufficient statistics on the backend, FP64 solve) takes 0.10–3.9 ms on the CPU and 0.32–4.7 ms on Metal. With 252 observations per series the designs are tiny (252×4 per series), the GPU is launch-bound, and the per-series multithreaded QR is the fastest option; batching would pay only with long daily histories or many more series. Coefficients from the batched path agree with per-series QR to 10^{-9} (CPU) and 10^{-4} relative (Metal, FP32 sufficient statistics), the documented tolerance.

E3 — GARCH-family panel. GARCH(1,1), GJR, EGARCH and GARCH-t are fitted by QML to the daily close-to-close log returns of 888 symbols (2015–2024 daily bars, median 2 459 training observations; the last 40 sessions of 2024 are held out). Table 9 reports convergence rates, iterations, time per fit, persistence, and the QLIKE of the fixed-parameter one-day-ahead variance forecasts against the next day's realized variance. GARCH and GARCH-t converge for all 888 series, GJR for 98.4 % and EGARCH for 92.8 % (after up to two restarts; the failures are reported as such, not as estimates), in 2.9–18 ms per fit; persistence medians are 0.956–0.973. Daily GARCH forecasts have a median QLIKE of 0.39–0.40 against 0.27–0.28 for the realized-variance random walk: a one-day-ahead forecast built from intraday data beats one built from daily returns, as the literature since Andersen et al. (2003) would predict, and the gap is not a numerical artefact. The GPU's contribution in E3 is the likelihood grid: evaluating 256 (ω, α, β) combinations per series for all 888 series takes 23 ms on Metal against 117 ms on 18 CPU threads (5.1×); the grid argmin is identical on both backends for all 888 series, with a maximum relative difference of 2×10^{-3} in the likelihood at the least-likely grid points (near the non-stationarity boundary).

E4 — realized covariance and connectedness. For $N \in \{10, 30, 60, 100\}$ symbols with at least 95 % regular-bar coverage, daily realized covariance matrices are formed from synchronised 5-minute returns (78 per day) on the days where every symbol has the full grid, and rolling Diebold–Yilmaz total connectedness is computed from a VAR(1) on the daily log-RV panel (120-day windows, $H = 10$). Table 10 shows the two constraints the charter asked for. Memory is not one of them: the rolling connectedness over 133 windows takes 0.5, 2.3, 8.9 and 29 ms for $N = 10, 30, 60, 100$ on the CPU and never fails. Conditioning is: the median condition number of the daily realized covariance grows from 81 ($N = 10$) to 422 ($N = 30$) and 5.9×10^3 ($N = 60$), and the matrix is singular for $N = 100 > 78$ bins; the Ledoit–Wolf shrinkage intensity rises from 0.16 to 0.33 accordingly. Strict synchronisation also discards 70–164 of the 252 days as N grows. Connectedness itself is not GPU work at these sizes; the GPU-relevant part is the realized covariance of large panels, which reduces to the batched sufficient statistics of E1.

11. Threats to validity

Construct. Speedups compare RAID's own CPU backend with its own Metal backend; the CPU backend is a careful but not a vendor-tuned implementation (Accelerate is used for linear algebra, not for the reductions). A hand-vectorised AVX/NEON CPU kernel could narrow the gaps further. Conversely, the GPU kernels are first implementations: the rolling kernel in particular is known to be the wrong algorithm.

Internal. The laptop was shared with other processes during the runs (load average 2.5–4 on 18 cores); medians with bootstrap intervals and shuffled order mitigate but do not remove this, and the cluster nodes, which were idle, show the same qualitative pattern. Thermal state was recorded (no throttling warnings) but power mode was not controlled. The real panel is one year of one market; the synthetic panels have Student-$t(5)$ returns with no intraday seasonality.

Statistical. The forecasting results of E2 and E3 use short windows and a fixed-parameter evaluation and are illustrations of equivalence, not model comparisons; no model confidence set or Diebold–Mariano test is reported, and none should be inferred. The provider's point-in-time guarantees apply to fundamentals, which this paper does not use; corporate actions are irrelevant to within-session returns but the close-to-close daily returns of E3 are from unadjusted bars and contain split days as outliers (the QML fits are robust to a handful of such days, but we did not remove them).

External. All data are US equities from a single vendor; all hardware is Apple Silicon. The conclusions about FP32 accumulation transfer to other FP32-only accelerators in principle, the

timing conclusions do not.

Scope. Nothing here is a microstructure result: 1-minute bars cannot support noise-robust estimators or order-flow measures, and the registry marks them as not implemented. Energy was not measured.

12. Reproducibility and licensing

The repository (`wp40_raid`, git remote on the author's SPB Git server) contains the library, kernels, bindings, CLI, tests, benchmark drivers, every raw benchmark record (`experiments/results/raw/*.jsonl`) with its manifest (host, OS, compiler, library versions, load, thermal state, seeds, input hashes), the scripts that generate every table and figure of this paper from those records (`experiments/scripts/make_paper_`
the SHA-256 of each input and output is written to `experiments/manifests/paper_assets.json`), and `RESEARCH_STATUS.md`, which lists what is implemented, verified, experimental and deliberately not implemented. No number in the text was typed by hand: the prose quotes `experiments/results/key_number`

A clean checkout builds with CMake presets (`macos-release` with Metal and Arrow; `cpu-only` for Linux) and passes 55 test cases; the Python package builds with scikit-build-core. The synthetic fixtures and the reference values produced by `statsmodels`, `arch`, `exchange_calendars`, `highfrequency` and `rugarch` are committed, so the validation suite runs without network access or vendor data. Reproducing E1–E6 requires an API key for the FinData OS lakehouse; the universe selection is seeded and the dataset version is pinned, so the same bars are served to a reader with access. E7 requires only the bundle and SSH access to the machines.

RAID is released under the MIT licence; Eigen (MPL-2.0), Apache Arrow (Apache-2.0), nanobind (BSD-3) and Catch2 (BSL-1.0) are its dependencies. Apple Metal and Accelerate are proprietary system frameworks; the Metal backend is optional and the CPU backend is portable. FirstRate Data bars are used under a private licence and are not redistributed; the paper reports aggregates only, and we thank FirstRate Data for the data.

13. Conclusion

Can high-frequency econometric workloads be moved to an FP32-only GPU without losing accuracy, inference or reproducibility? For the estimators studied here the answer is yes on accuracy and reproducibility and "sometimes" on speed. Compensated double-float accumulation, chunked FP64 combination and mixed-precision design keep Metal results within 10^{-6} of FP64 for realized measures and regression statistics and within 2×10^{-5} for batched GARCH likelihoods, bitwise reproducible across runs and thread counts, with Monte Carlo coverage and reference comparisons

indistinguishable from the CPU path. The speed answer is specific: the fused realized kernel beats an 18-thread CPU beyond about 4×10^4 elements (2–3.4$\times$ on raw reductions, 1.1–1.4$\times$ once the estimator's own FP64 post-processing is counted), batched GARCH likelihood grids gain 5–8$\times$, rolling sums and single-vector recursions lose, small panels tie, and the pipeline from Parquet to estimates is governed by column projection and calendar logic rather than by any kernel. Across eleven Apple Silicon generations the GPU helps most where the CPU is weakest.

The practical lesson for econometric software on unified-memory machines is that the device is cheap and the conversion is not: a library that wants GPU speedups must keep FP32 copies resident across calls, fuse its passes, and reserve the GPU for problems whose parallelism is in the number of series or parameter vectors, not in the length of a recursion. The methodological lesson is that none of this requires new estimators—only explicit conventions, references and tolerances, which is where most implementations differ.

References

Aldrich, Eric M., Jesús Fernández-Villaverde, A. Ronald Gallant, and Juan F. Rubio-Ramírez, "Tapping the supercomputer under your desk: Solving dynamic equilibrium models with graphics processors," *Journal of Economic Dynamics and Control*, 2011, *35* (3), 386–393.

Andersen, Torben G., Tim Bollerslev, and Francis X. Diebold, "Roughing It Up: Including Jump Components in the Measurement, Modeling, and Forecasting of Return Volatility," *Review of Economics and Statistics*, 2007, *89* (4), 701–720.

_ , _ , _ , and Paul Labys, "Modeling and Forecasting Realized Volatility," *Econometrica*, 2003, *71* (2), 579–625.

Apache Software Foundation, "Apache Arrow: a cross-language development platform for in-memory analytics, C++ library version 26.0.0," https://arrow.apache.org/ 2025. Accessed 2026-11-02.

Apple Inc., "Metal Shading Language Specification, version 4," https://developer.apple.com/metal/ 2026. Accessed 2026-11-02; Metal does not define a 64-bit floating-point type.

Barndorff-Nielsen, O. E., "Power and Bipower Variation with Stochastic Volatility and Jumps," *Journal of Financial Econometrics*, 2004, *2* (1), 1–37.

_ , "Econometrics of Testing for Jumps in Financial Economics Using Bipower Variation," *Journal of Financial Econometrics*, 2005, *4* (1), 1–30.

Barndorff-Nielsen, Ole E. and Neil Shephard, "Econometric analysis of realized volatility and its use in estimating stochastic volatility models," *Journal of the Royal Statistical Society: Series B*, 2002, *64* (2), 253–280.

_ , Peter Reinhard Hansen, Asger Lunde, and Neil Shephard, "Designing Realized Kernels to Measure the ex post Variation of Equity Prices in the Presence of Noise," *Econometrica*, 2008, *76* (6), 1481–1536.

_ , Silja Kinnebrock, and Neil Shephard, "Measuring Downside Risk: Realised Semivariance," in Tim Bollerslev, Jeffrey Russell, and Mark Watson, eds., *Volatility and Time Series Econometrics: Essays in Honor of Robert F. Engle*, Oxford University Press, 2010, pp. 117–136.

Blackford, L. Susan, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, Michael Heroux, Linda Kaufman, Andrew Lumsdaine, Antoine Petitet, Roldan Pozo,

Karin Remington, and R. Clint Whaley, "An updated set of basic linear algebra subprograms (BLAS)," *ACM Transactions on Mathematical Software*, 2002, *28* (2), 135–151.

Bollerslev, Tim, "Generalized autoregressive conditional heteroskedasticity," *Journal of Econometrics*, 1986, *31* (3), 307–327.

_ **and Jeffrey M. Wooldridge**, "Quasi-maximum likelihood estimation and inference in dynamic models with time-varying covariances," *Econometric Reviews*, 1992, *11* (2), 143–172.

_ **, Andrew J. Patton, and Rogier Quaedvlieg**, "Exploiting the errors: A simple approach for improved volatility forecasting," *Journal of Econometrics*, 2016, *192* (1), 1–18.

Boudt, Kris, Onno Kleen, and Emil Sjørup, "Analyzing Intraday Financial Data in R: The `highfrequency` Package," *Journal of Statistical Software*, 2022, *104* (8).

Cameron, A. Colin, Jonah B. Gelbach, and Douglas L. Miller, "Robust Inference With Multiway Clustering," *Journal of Business & Economic Statistics*, 2011, *29* (2), 238–249.

Corsi, F., "A Simple Approximate Long-Memory Model of Realized Volatility," *Journal of Financial Econometrics*, 2008, *7* (2), 174–196.

Dekker, T. J., "A floating-point technique for extending the available precision," *Numerische Mathematik*, 1971, *18*, 224–242.

Diebold, Francis X. and Kamil Yilmaz, "Better to give than to receive: Predictive directional measurement of volatility spillovers," *International Journal of Forecasting*, 2012, *28* (1), 57–66.

_ **and Kamil Yılmaz**, "On the network topology of variance decompositions: Measuring the connectedness of financial firms," *Journal of Econometrics*, 2014, *182* (1), 119–134.

Diebold, Francis X and Robert S Mariano, "Comparing Predictive Accuracy," *Journal of Business & Economic Statistics*, 2002, *20* (1), 134–144.

Engle, Robert, "Dynamic Conditional Correlation: A Simple Class of Multivariate Generalized Autoregressive Conditional Heteroskedasticity Models," *Journal of Business & Economic Statistics*, 2002, *20* (3), 339–350.

Engle, Robert F., "Autoregressive Conditional Heteroscedasticity with Estimates of the Variance of United Kingdom Inflation," *Econometrica*, 1982, *50* (4), 987.

_ **, Eric Ghysels, and Bumjean Sohn**, "Stock Market Volatility and Macroeconomic Fundamentals," *Review of Economics and Statistics*, 2013, *95* (3), 776–797.

Fiorentini, Gabriele, Giorgio Calzolari, and Lorenzo Panattoni, "Analytic derivatives and the computation of GARCH estimates," *Journal of Applied Econometrics*, 1996, *11* (4), 399–417.

Ghalanos, Alexios, "rugarch: Univariate GARCH Models, R package version 1.5-6," CRAN 2024.

GLOSTEN, LAWRENCE R., RAVI JAGANNATHAN, and DAVID E. RUNKLE, "On the Relation between the Expected Value and the Volatility of the Nominal Excess Return on Stocks," *The Journal of Finance*, 1993, *48* (5), 1779–1801.

Gonçalves, Sílvia and Nour Meddahi, "Bootstrapping realized volatility," *Econometrica*, 2009, *77* (1), 283–306.

Guennebaud, Gaël, Benoît Jacob et al., "Eigen v5.0.1: a C++ template library for linear algebra," https://eigen.tuxfamily.org 2025. Accessed 2026-11-02.

Hansen, Peter R. and Asger Lunde, "A forecast comparison of volatility models: does anything beat a GARCH(1,1)?," *Journal of Applied Econometrics*, 2005, *20* (7), 873–889.

_ , _ , and James M. Nason, "The model confidence set," *Econometrica*, 2011, *79* (2), 453–497.

Hansen, Peter Reinhard, Zhuo Huang, and Howard Howan Shek, "Realized GARCH: a joint model for returns and realized measures of volatility," *Journal of Applied Econometrics*, 2011, *27* (6), 877–906.

Harris, Charles R., K. Jarrod Millman, Stéfan J. van der Walt et al., "Array programming with NumPy," *Nature*, 2020, *585*, 357–362.

Higham, Nicholas J., "The accuracy of floating point summation," *SIAM Journal on Scientific Computing*, 1993, *14* (4), 783–799.

Huang, X., "The Relative Contribution of Jumps to Total Price Variance," *Journal of Financial Econometrics*, 2005, *3* (4), 456–499.

Jacod, Jean, Yingying Li, Per A. Mykland, Mark Podolskij, and Mathias Vetter, "Microstructure noise in the continuous case: The pre-averaging approach," *Stochastic Processes and their Applications*, 2009, *119* (7), 2249–2276.

Jakob, Wenzel, "nanobind 3.1.0: tiny and efficient C++/Python bindings," https://github.com/wjakob/nanobind 2025. Accessed 2026-11-02.

Kahan, W., "Pracniques: further remarks on reducing truncation errors," *Communications of the ACM*, 1965, *8* (1), 40.

Koop, Gary, M. Hashem Pesaran, and Simon M. Potter, "Impulse response analysis in nonlinear multivariate models," *Journal of Econometrics*, 1996, *74* (1), 119–147.

Ledoit, Olivier and Michael Wolf, "A well-conditioned estimator for large-dimensional covariance matrices," *Journal of Multivariate Analysis*, 2004, *88* (2), 365–411.

Liu, Lily Y., Andrew J. Patton, and Kevin Sheppard, "Does anything beat 5-minute RV? A comparison of realized measures across multiple asset classes," *Journal of Econometrics*, 2015, *187* (1), 293–311.

Nelson, Daniel B., "Conditional Heteroskedasticity in Asset Returns: A New Approach," *Econometrica*, 1991, *59* (2), 347.

Neumaier, Arnold, "Rundungsfehleranalyse einiger Verfahren zur Summation endlicher Summen," *ZAMM – Zeitschrift für Angewandte Mathematik und Mechanik*, 1974, *54* (1), 39–51.

Newey, Whitney K. and Kenneth D. West, "A Simple, Positive Semi-Definite, Heteroskedasticity and Autocorrelation Consistent Covariance Matrix," *Econometrica*, 1987, *55* (3), 703.

_ **and** _ , "Automatic lag selection in covariance matrix estimation," *Review of Economic Studies*, 1994, *61* (4), 631–653.

Noureldin, Diaa, Neil Shephard, and Kevin Sheppard, "Multivariate high-frequency-based volatility (HEAVY) models," *Journal of Applied Econometrics*, 2011, *27* (6), 907–933.

Patton, Andrew J., "Volatility forecast comparison using imperfect volatility proxies," *Journal of Econometrics*, 2011, *160* (1), 246–256.

_ **and Kevin Sheppard**, "Good Volatility, Bad Volatility: Signed Jumps and The Persistence of Volatility," *Review of Economics and Statistics*, 2015, *97* (3), 683–697.

Pesaran, H.Hashem and Yongcheol Shin, "Generalized impulse response analysis in linear multivariate models," *Economics Letters*, 1998, *58* (1), 17–29.

Politis, Dimitris N. and Joseph P. Romano, "The Stationary Bootstrap," *Journal of the American Statistical Association*, 1994, *89* (428), 1303–1313.

Seabold, Skipper and Josef Perktold, "statsmodels: Econometric and statistical modeling with Python," in "Proceedings of the 9th Python in Science Conference" 2010, pp. 92–96.

Sheppard, Kevin, "arch 8.0.0: ARCH models in Python," https://github.com/bashtage/arch 2025. Version 8.0.0, accessed 2026-11-02.

Virtanen, Pauli, Ralf Gommers, Travis E. Oliphant et al., "SciPy 1.0: fundamental algorithms for scientific computing in Python," *Nature Methods*, 2020, *17*, 261–272.

White, Halbert, "A Heteroskedasticity-Consistent Covariance Matrix Estimator and a Direct Test for Heteroskedasticity," *Econometrica*, 1980, *48* (4), 817.

Zhang, Lan, Per A Mykland, and Yacine Aït-Sahalia, "A Tale of Two Time Scales: Determining Integrated Volatility With Noisy High-Frequency Data," *Journal of the American Statistical Association*, 2005, *100* (472), 1394–1411.

Appendices

A. Kernel pseudocode

```
df two_sum(float a, float b): s = a + b; bb = s - a; e = (a - (s - bb)) + (b - bb); return (s, e)
df df_add_f(df a, float v): (h, l) = two_sum(a.hi, v); l += a.lo; h2 = h + l; return (h2, l - (h2 - h))
kernel realized_fused(x, chunks, out): // one threadgroup (256 threads) per (segment, chunk)
  c = chunks[gid]; s2 = adj = neg = pos = s4 = (0, 0)
  for i in c.begin + lid .. c.end step 256:
    v = x[i]; q = v*v; s2 += q; s4 += q*q; if v < 0: neg += q elif v > 0: pos += q
    if i > c.begin: adj += |v| * |x[i-1]|
  tree-reduce each df over the threadgroup in fixed order; thread 0 writes 5 float2
host: for each chunk c of segment s: acc[s] += (double)hi + (double)lo (fixed order);
     adj[s] += |x[c.begin]| * |x[c.begin-1]| in FP64 when c.begin > segment begin
kernel garch_nll(r, seg, params[8], var0, jobs, out, model): // one thread per (segment, parameter vector)
  (mu, omega, alpha, gamma, beta, nu, kconst, Eabs) = params[job]; s2 = var0[seg]
  validity checks -> +inf; nll = (0, 0); e_prev = r[b] - mu
  for i in 0..n-1: if i > 0: s2 = recursion(model); if !(s2 > 0): return +inf
    e = r[b+i] - mu; ll = kconst - 0.5 log s2 - (student ? 0.5 (nu+1) log(1 + e^2/((nu-2) s2)) : 0.5 e^2/s2)
    nll += -ll
  out[job] = nll.hi + nll.lo // host adds n log(sd) for the per-series scaling
```

B. Optimiser details

GARCH-family QML: parameters are transformed to an unconstrained space ($\omega = e^u$; persistence $p = \sigma(u_p)(1-10^{-9})$ and share $s = \sigma(u_s)$ so that $\alpha = sp$, $\beta = (1-s)p$ for GARCH; for GJR $\alpha+\gamma = e^{u_g}$ and $\beta = p - \alpha - \gamma/2 \geq 0$; for EGARCH ω free and $\beta = \tanh(u)$; $\nu = 2.05 + 497.95\,\sigma(u_\nu)$). BFGS with Armijo backtracking (factor 0.5, up to 60 halvings), central-difference gradients with step $10^{-5}\max(1,|u|)$, convergence when $\|g\| < 10^{-5}$, or when the relative change of the negative log-likelihood is below 10^{-8} (or the step below 10^{-9}) with $\|g\| < 10^{-2}$ in the transformed space; at

most 500 iterations; up to three starting variants (persistence 0.95, 0.80, 0.985; opposite EGARCH γ sign). Covariance from the numerical Hessian (central differences, step $10^{-4}\max(1,|\theta|)$) and the OPG of per-observation scores. Returns are scaled to unit variance before optimisation and the estimates un-scaled (EGARCH: $\omega \to \omega + (1-\beta)\log s^{-2}$).

C. Finite-sample conventions

Regular session = 09:30 to 16:00 ET inclusive of the 16:00 closing print (391 one-minute bars; 211 on 13:00 early closes); extended session 04:00–19:59 ET (17:00 on early closes). Intraday returns are close-to-close within the session (M = bars -1). BPV factor $M/(M-1)$ on by default; TriPV factor $M/(M-2)$; RQ $= (M/3)\sum r^4$. HAR windows of 1, 5 and 22 trading days; targets averaged over h sessions; HAC lags $\max(h-1, \lfloor 4(n/100)^{2/9} \rfloor)$. HARQ regressor $(\sqrt{\mathrm{RQ}_t} - \sqrt{\overline{\mathrm{RQ}}})\mathrm{RV}_t$ with $\overline{\mathrm{RQ}}$ over the estimation sample. GARCH σ_0^2 = sample variance of ε. Stationary bootstrap mean block $n^{1/3}$ unless stated.

D. Benchmark configuration

Laptop: Apple M5 Max (6P+12E cores reported by `hw.perflevel`, 18 hardware threads), 40-core GPU, 48 GB, macOS 27.0 (26A5425a), Xcode 27.2 (27B52), Apple clang 21.0.0 (clang-2100.4.1.1), Metal toolchain 32023.922, CMake 4.3.2, `-O3 -mcpu=native -ffp-model=precise`, Eigen 5.0.1 + Accelerate, Arrow 26.0.0. Metal: 256-thread threadgroups for reductions, 4 096-element chunks, 64-thread threadgroups for GARCH jobs, shared storage mode, `fastMathEnabled = NO`. Repetitions: 5 (laptop), 3 (nodes); 2 warm-ups; shuffled order with seed 20261102; medians and 2 000-draw bootstrap 95 % CIs. Cluster bundle: `-mcpu=apple-m1`, deployment target macOS 26.0, Arrow disabled, Metal enabled, prebuilt metallib plus embedded MSL.

E. Hardware inventory (E7)

Node	Chip	GPU cores	CPU threads	RAM (GB)	macOS
m2m8a	Apple M2	10	8	8	26.5.1
M2M32c	Apple M2 Max	30	12	32	26.5.1
M2U64	Apple M2 Ultra	60	24	64	26.5
M3BA24	Apple M3	10	8	24	26.5.1
M3U96b	Apple M3 Ultra	60	28	96	26.4
m4md	Apple M4	10	10	16	26.6.2
m4mm	Apple M4 Pro	16	12	48	26.5.1
M4M64c	Apple M4 Max	40	16	64	26.4
m5ma	Apple M5 Pro	16	15	24	27.0
M5M36	Apple M5 Max	32	18	36	27.0
m6ma	Apple M6	12	12	24	27.0

F. API provenance

FinData OS API `https://findata.spboucher.ai/api/v1` (OpenAPI 3.1 snapshot of 2026-11-02 in `docs/evidence/`); datasets `market_bars.stock.1m.raw` (version 17 375 at export time; 6.70 billion rows, 16 195 symbols, 2000–2026) and `market_bars.stock.1d.raw`; bars schema v1 (`timestamp_utc` = bar open, UTC); exports as Parquet with the dataset version in the job record; request identifiers of every exchange are kept in the cache manifests; API key identified by a SHA-256 fingerprint only. Data © FirstRate Data, private-use licence, no redistribution.

G. Numerical tolerance tables

See Table 1 for the reference tolerances. GPU/CPU tolerances by primitive: realized accumulators 10^{-6} relative (Σx^4: 10^{-5}); rolling sums 10^{-5}; regression statistics 10^{-6} absolute on $\max(1, |\cdot|)$ and 10^{-5} on the solved coefficients; batched GARCH NLL 2×10^{-5} relative (Student-t: 5×10^{-5}) on simulated series with $n = 2\,000$; on the E3 real-data grid the maximum relative difference is 2×10^{-3} at boundary grid points and the argmin agrees for 888/888 series.

Table 3. Primitive kernels, laptop (M5 Max). Medians of wall-clock milliseconds over 5 repetitions; CPU MT uses 18 threads; speedup $= T_{\text{CPU MT}}/T_{\text{Metal}}$; verdict by non-overlapping bootstrap CIs.

Kernel	$S \times n$	elements	CPU 1T (ms)	CPU MT (ms)	Metal (ms)	speedup	verdict
realized_fused	1×390	390	0.001	0.001	0.234	0.01	CPU
realized_fused	1×3900	3,900	0.012	0.011	0.633	0.02	CPU
realized_fused	10×390	3,900	0.011	0.142	0.233	0.61	CPU
realized_fused	1×39000	39,000	0.183	0.167	0.207	0.81	CPU
realized_fused	10×3900	39,000	0.188	0.135	0.164	0.82	CPU
realized_fused	100×390	39,000	0.187	0.552	0.226	2.44	Metal
realized_fused	1×390000	390,000	2.032	2.086	0.935	2.23	Metal
realized_fused	10×39000	390,000	2.573	0.363	0.718	0.51	tie
realized_fused	100×3900	390,000	2.043	0.602	0.641	0.94	tie
realized_fused	1000×390	390,000	2.236	0.840	0.338	2.48	Metal
realized_fused	10×390000	3,900,000	20.892	2.527	1.022	2.47	Metal
realized_fused	100×39000	3,900,000	20.591	2.190	1.169	1.87	Metal
realized_fused	1000×3900	3,900,000	21.245	1.742	0.875	1.99	Metal
realized_fused	10000×390	3,900,000	21.238	2.126	2.213	0.96	tie
realized_fused	100×390000	39,000,000	232.587	18.793	8.720	2.16	Metal
realized_fused	1000×39000	39,000,000	238.772	20.247	5.892	3.44	Metal
realized_fused	10000×3900	39,000,000	233.672	15.694	7.803	2.01	Metal
regression_stats	1×390	390	0.003	0.003	0.894	0.00	CPU
regression_stats	1×3900	3,900	0.030	0.030	0.348	0.09	CPU
regression_stats	10×390	3,900	0.031	0.041	0.652	0.06	CPU
regression_stats	1×39000	39,000	0.302	0.384	0.409	0.94	tie
regression_stats	10×3900	39,000	0.300	0.076	0.362	0.21	CPU
regression_stats	100×390	39,000	0.361	0.176	0.336	0.52	CPU
regression_stats	1×390000	390,000	2.973	3.043	0.790	3.85	Metal
regression_stats	10×39000	390,000	3.084	0.562	0.696	0.81	CPU
regression_stats	100×3900	390,000	2.998	0.440	0.595	0.74	CPU
regression_stats	1000×390	390,000	3.121	0.697	0.581	1.20	tie
regression_stats	10×390000	3,900,000	32.663	6.655	4.117	1.62	Metal
regression_stats	100×39000	3,900,000	32.191	3.174	4.469	0.71	tie
regression_stats	1000×3900	3,900,000	34.396	3.422	3.796	0.90	tie
regression_stats	10000×390	3,900,000	30.644	3.959	4.191	0.94	tie
regression_stats	100×390000	39,000,000	301.369	33.760	30.709	1.10	Metal
regression_stats	1000×39000	39,000,000	324.036	31.241	26.743	1.17	Metal
regression_stats	10000×3900	39,000,000	356.914	37.490	26.574	1.41	Metal
rolling_sumsq	1×390	390	0.001	0.001	0.284	0.00	CPU
rolling_sumsq	1×3900	3,900	0.007	0.007	0.242	0.03	CPU
rolling_sumsq	10×390	3,900	0.007	0.018	0.203	0.09	CPU
rolling_sumsq	1×39000	39,000	0.069	0.070	0.165	0.43	CPU
rolling_sumsq	10×3900	39,000	0.069	0.045	0.161	0.28	CPU
rolling_sumsq	100×390	39,000	0.070	0.180	0.268	0.67	tie
rolling_sumsq	1×390000	390,000	0.691	0.719	0.790	0.91	tie
rolling_sumsq	10×39000	390,000	0.710	0.133	0.736	0.18	CPU
rolling_sumsq	100×3900	390,000	0.807	0.203	0.682	0.30	CPU
rolling_sumsq	1000×390	390,000	0.719	0.271	0.908	0.30	CPU
rolling_sumsq	10×390000	3,900,000	6.914	1.766	2.544	0.69	CPU
rolling_sumsq	100×39000	3,900,000	7.338	1.087	2.227	0.49	CPU
rolling_sumsq	1000×3900	3,900,000	6.954	1.027	2.145	0.48	CPU
rolling_sumsq	10000×390	3,900,000	7.048	1.072	2.163	0.50	CPU
rolling_sumsq	100×390000	39,000,000	69.622	9.413	14.852	0.63	CPU
rolling_sumsq	1000×39000	39,000,000	70.263	8.312	14.729	0.56	CPU
rolling_sumsq	10000×3900	39,000,000	69.226	6.543	14.753	0.44	CPU
garch_nll_P1	1×390	390	0.002	0.002	0.918	0.00	CPU
garch_nll_P1	1×3900	3,900	0.016	0.017	1.349	0.01	CPU
garch_nll_P1	10×390	3,900	0.022	0.029	0.343	0.09	CPU
garch_nll_P1	10×3900	39,000	0.216	0.069	1.430	0.05	CPU
garch_nll_P1	100×390	39,000	0.192	0.161	0.624	0.26	CPU
garch_nll_P1	100×3900	390,000	1.615	0.207	1.649	0.13	CPU
garch_nll_P1	1000×390	390,000	1.639	0.277	1.153	0.24	CPU
garch_nll_P1	1000×3900	3,900,000	18.310	1.498	7.207	0.21	CPU
garch_nll_P1	10000×390	3,900,000	17.741	2.243	6.755	0.33	CPU

Table 4. E7 — generation sweep (3 repetitions, shuffled; speedups are CPU MT / Metal medians at the stated shape; tests = Metal test suite with the prebuilt metallib / with runtime MSL compilation).

Node	chip	GPU cores	CPU thr.	GB	tests (metallib / runtime)	cold / warm compile (ms)	RV fused 1000×3900 speedup	regression speedup	GARCH P=64 speedup	rolling speedup
m2m8a	Apple M2	10	8	8	pass / pass	110 / 1	3.99	1.61	6.67	0.65
M2M32c	Apple M2 Max	30	12	32	pass / pass	48 / 1	3.61	1.35	7.84	0.52
M2U64	Apple M2 Ultra	60	24	64	pass / pass	49 / 1	2.18	1.08	4.92	0.32
M3BA24	Apple M3	10	8	24	pass / pass	44 / 0	2.69	1.68	6.21	0.36
M3U96b	Apple M3 Ultra	60	28	96	pass / pass	47 / 1	1.68	0.78	3.18	0.24
m4md	Apple M4	10	10	16	pass / pass	42 / 0	2.55	1.34	4.21	0.38
m4mm	Apple M4 Pro	16	12	48	pass / pass	39 / 0	1.63	0.91	2.14	0.24
M4M64c	Apple M4 Max	40	16	64	pass / pass	40 / 0	2.80	1.19	4.82	0.48
m5ma	Apple M5 Pro	16	15	24	pass / pass	44 / 0	1.13	0.55	1.83	0.20
M5M36	Apple M5 Max	32	18	36	pass / pass	37 / 0	1.12	0.60	2.14	0.23
m6ma	Apple M6	12	12	24	pass / pass	40 / 0	1.63	0.79	3.72	0.34

Table 5. E1 — batched realized measures (estimator path), laptop. Medians of 5 repetitions; "real" = 2024 FinData panel; "synthetic" = Student-$t(5)$ returns; max rel. err. = largest $|\mathrm{RV}_{\mathrm{Metal}} - \mathrm{RV}_{\mathrm{FP64}}|/\mathrm{RV}_{\mathrm{FP64}}$ over segments.

Panel	S	segment	elements	CPU 1T (ms)	CPU MT (ms)	Metal fused (ms)	Metal 4 kernels (ms)	max rel. err. RV
real	1	day	88,350	0.77	0.94	0.94	2.27	3.2e-08
real	1	week	87,732	0.81	0.91	0.77	1.99	1.5e-08
real	1	month	85,202	0.59	0.26	0.33	1.03	5.5e-09
real	1	year	88,350	0.88	0.74	0.51	1.29	1.2e-09
real	10	day	660,818	4.59	1.33	1.40	1.95	1.2e-07
real	10	week	655,478	4.32	1.02	0.81	1.97	4.3e-08
real	10	month	634,608	4.16	0.90	1.06	1.49	1.5e-08
real	10	year	660,818	4.49	1.31	1.41	2.38	3.2e-09
real	100	day	4,857,139	29.41	3.80	4.11	7.73	1.5e-07
real	100	week	4,814,500	28.38	3.75	3.63	6.06	1.4e-07
real	100	month	4,656,568	27.86	3.61	3.43	5.32	6.5e-08
real	100	year	4,817,403	29.64	3.89	3.25	5.53	1.2e-08
real	926	day	40,408,948	240.89	29.77	27.77	51.85	1.6e-07
real	926	week	40,062,893	235.07	29.19	24.93	44.04	1.4e-07
real	926	month	38,769,255	231.20	28.67	20.43	38.21	6.5e-08
real	926	year	39,917,613	241.63	28.71	20.41	39.65	1.4e-08
synthetic	10000	day	3,900,000	21.74	2.68	3.35	–	4.9e-08
synthetic	10000	week	19,500,000	106.64	12.07	9.12	–	3.4e-08
synthetic	10000	month	85,800,000	483.17	54.08	35.41	–	2.8e-08

Table 6. E6 — end-to-end pipeline per export file (median of 4 files $\times$ 3 repetitions; 2.37 million bars per file).

Engine	median total (s)	95% CI	bars / s	notes
RAID cpu	0.118	[0.097, 0.128]	20.0M	Arrow projection, exact calendar, 5 measures
RAID metal	0.125	[0.101, 0.134]	19.0M	same, fused Metal kernel
polars	0.165	[0.146, 0.172]	14.4M	approximate early-close rule, lazy scan
duckdb	0.164	[0.158, 0.165]	14.5M	approximate early-close rule, SQL window
pandas	2.558	[2.355, 2.634]	0.9M	approximate early-close rule, groupby.apply

Table 7. E2 — rolling causal HAR forecasts, 100 symbols, 2024 (120-session window, median QLIKE across symbols).

h	symbols	HAR QLIKE	log-HAR QLIKE	HARQ QLIKE	RW QLIKE	HAR beats RW (%)
1	100	0.196	0.205	0.203	0.251	84.0
5	100	0.146	0.138	0.150	0.239	86.0
22	100	0.134	0.112	0.132	0.301	78.0

Table 8. E2 — runtime of HAR-RV estimation ($h = 1$), milliseconds (medians of 5 repetitions).

S	T	per-series 1T (ms)	per-series 18T (ms)	batched CPU (ms)	batched Metal (ms)
10	252	0.077	0.053	0.102	0.320
100	252	0.765	0.145	0.651	0.753
781	251	6.034	0.684	3.861	4.703

Table 9. E3 — GARCH-family QML on 888 daily return series (2015–2024); forecasts evaluated on the last 40 sessions of 2024 against realized variance.

Model	fits	converged (%)	median iterations	median time (ms)	median persistence	QLIKE vs RV	QLIKE RW
garch.n	888	100.0	14	2.86	0.962	0.403	0.276
gjr.n	888	98.4	18	4.00	0.971	0.399	0.273
egarch.n	888	92.8	17	18.15	0.956	0.396	0.272
garch.t	888	100.0	18	6.75	0.973	0.391	0.276

Table 10. E4 — daily realized covariance (5-minute, synchronised) and rolling Diebold–Yilmaz connectedness, 2024.

N	days used	cond. median	cond. max	LW shrinkage	DY windows (120 d)	mean total (%)	time (ms)
10	182	81	1114	0.16	133	37.2	0.5
30	129	422	1982	0.26	133	69.7	2.3
60	95	5918	[illegible]	0.33	133	87.5	8.9
100	88	[illegible]	[illegible]	0.33	133	97.2	28.8