ISSN (Print) 1226-1750
ISSN (Online) 2233-6656
Journal of Magnetics 31(2), 204-213 (2026) https://doi.org/10.4283/JMAG.2026.31.2.204

Optimization of MuMax3 by Using Claude Code: A CUDA-Graph-Based Case Study in AI-Assisted Performance Engineering

Chun-Yeol You*

Department of Physics and Chemistry, Daegu Gyeongbuk Institute of Science and Technology, Daegu 42988, Republic of Korea

(Received 15 June 2026, Received in final form 26 June 2026, Accepted 26 June 2026)

MuMax3 is a widely used open-source GPU-accelerated micromagnetic simulator whose computational core — CUDA kernels and cuFFT-based demagnetization convolutions — has changed little since its original release. We report a case study in which an agentic large-language-model coding assistant (Claude Code, Anthropic) was used, under continuous human supervision, to profile and optimize this mature CUDA/Go codebase. Profiling with NVIDIA Nsight Systems revealed that for small and medium grids (64×64×1 to 256×256×1 cells), 75–79 % of step time is spent in CPU-side cuLaunchKernel **driver calls rather than in GPU computation, because every solver step launches roughly 27 kernels sequentially on a single CUDA stream. Building on this finding, we implemented a "split-graph" execution strategy: the time-step-independent torque-evaluation kernel sequences of each solver stage are captured once as CUDA Graphs and replayed via** cudaGraphLaunch**, while the time-step-dependent update, error estimate, and adaptive-step-size logic remain ordinary stream-ordered calls. The optimization is exposed transparently through** Run()/Steps()**, guarded by a compatibility check (constant excitation, zero thermal field, no custom field terms, time-independent material parameters, mesh size below a tunable threshold) that falls back silently to the original code path when violated. Across five solvers (Heun, RK23, RK45DP, RK56, Backward Euler), the optimization yields up to 5.4× throughput for a 64×64×1 grid with a fixed time step, 2.4–3.6× for adaptive time-stepping, decreasing smoothly to ≈1.0× near 10^6 cells, essentially independent of which physical field terms (exchange, anisotropy, DMI) are active. All results were verified bit-for-bit identical (ndiff=0) against the unmodified code, and the full 176-script mumax3 regression suite passes with zero failures. We discuss the workflow itself — including three episodes in which the assistant autonomously diagnosed and corrected its own defects — as a template for AI-assisted optimization of legacy scientific HPC codes.**

Keywords : micromagnetics, spintronics, mumax3, AI-assisted

1. Introduction

Among the available micromagnetic simulation packages, MuMax3 [1] — an open-source, GPU-accelerated finite-difference solver written in Go with CUDA kernels — has become a *de facto* standard, owing to its combination of high performance on consumer GPUs, an accessible scripting interface, and extensive verification against the micromagnetic standard problems and the earlier OOMMF package [2, 3]. Since its description in 2014 [1, 4], MuMax3's computational core has remained largely stable even as GPU hardware has advanced through several generations (Pascal, Turing, Ampere, Ada, and now Blackwell). Such micromagnetic simulation is a central tool in the study of nanoscale magnetism, from domain-wall dynamics and spin-torque oscillators to skyrmion-based racetrack memories [5-10].

Re-optimizing a mature, 16,000-line mixed Go/CUDA codebase for new hardware is, in practice, a significant undertaking: it requires GPU profiling expertise, familiarity with the CUDA driver API, and — critically — the ability to verify that any change preserves numerical results to within floating-point precision across a large regression suite. This combination of requirements is precisely the kind of work that recent agentic large-language-model (LLM) coding assistants are increasingly used for, with mixed but improving success on benchmarks of real-world software-engineering tasks [14-22].

In this paper we report a case study in which one such assistant, Claude Code (Anthropic), was used under continuous human supervision to analyze and optimize a fork

*Corresponding author: Tel: +82-53-785-6522
e-mail: cyyou@dgist.ac.kr

of MuMax3 (based on commit 3fe3d41) running on an NVIDIA RTX 5060 Ti GPU. Rather than a narrow micro-benchmark, the assistant was given an open-ended brief — "find and implement performance improvements, with full correctness verification" — and allowed to drive its own profiling-implement-verify cycle over an extended, multi-session project. The work proceeded in three stages of increasing impact:

1. Two small, low-risk fixes identified directly from source-code inspection: halving the launch range of the demagnetization kernel-multiplication step (exploiting a symmetry the kernel already used internally), and removing a per-step host–device synchronization that computed a diagnostic quantity (MaxTorque) not used by any solver's step-size control.
2. A much larger optimization, discovered only through GPU profiling rather than code inspection: for small and medium grids, 75–79 % of step time is consumed by CPU-side CUDA driver call overhead (cuLaunchKernel), because MuMax3 issues 27 kernel launches per torque evaluation on a single CUDA stream. This motivated a CUDA-Graph-based "split-graph" capture-and-replay strategy (Secs. IV and VII).
3. A safety-and-generalization phase in which the assistant extended the optimization from a single proof-of-concept solver/configuration to all five common solvers, integrated it transparently into the normal simulation-run API, and — critically — built an explicit compatibility-guard system so that configurations outside the verified envelope fall back automatically to the original, unmodified code path (Sec. VI).

The remaining sections provide: a brief background and glossary for readers unfamiliar with either micromagnetic simulation internals or CUDA execution models (Sec. II); a description of the AI-assisted workflow itself, including the assistant's persistent project memory and three illustrative autonomous-debugging episodes (Sec. III); a summary of the implemented optimizations (Sec. IV); benchmark results (Sec. V); the correctness-verification methodology and safety guards (Sec. VI); the size- and configuration-dependence of the speedup (Sec. VII); a discussion of limitations and directions not pursued (Sec. VIII); and code/build availability (Sec. IX).

2. Background and Glossary

This section briefly summarizes the concepts needed to follow Secs. III–VIII, for readers whose primary expertise is in magnetism rather than GPU computing, or vice versa.

2.1. Micromagnetic time integration

MuMax3 evolves the magnetization field $\mathbf{m}(\mathbf{r}, t)$ according to the Landau–Lifshitz–Gilbert (LLG) equation [12], $\partial \mathbf{m}/\partial t = -\gamma \mathbf{m} \times \mathbf{B}_{\text{eff}} + \alpha \mathbf{m} \times \partial \mathbf{m}/\partial t$, using an explicit, adaptive-step Runge–Kutta-family solver. The default solver is the embedded Dormand–Prince RK45 method [13] (RK45DP, "solver 5"); MuMax3 also offers Heun ("solver 2"), Bogacki–Shampine RK23 ("solver 3"), a 6th-order RK pair ("solver 6", RK56), and an implicit Backward Euler method ("solver −1") for strongly damped problems. Each solver evaluates the right-hand side of the LLG equation — the **effective field** $\mathbf{B}_{\text{eff}}$ and the resulting **torque** — once per Runge–Kutta *stage*; we refer to this as a *torque evaluation*, implemented in MuMax3 as torqueFn.

2.2. Effective field terms

$\mathbf{B}_{\text{eff}}$ is accumulated from several physical contributions, each computed by one or more dedicated CUDA kernels: the demagnetizing (dipolar) field, computed via an FFT-based convolution with a precomputed demagnetizing kernel derived from the Newell tensor [11]; the exchange field, computed by a finite-difference stencil kernel; magnetocrystalline anisotropy; interfacial or bulk Dzyaloshinskii–Moriya interaction (DMI) terms, relevant to skyrmion physics [8, 9]; the Zeeman field from an external applied field $\mathbf{B}_{\text{ext}}$; and, optionally, thermal and magnetoelastic contributions. Each term is added into the same GPU buffer sequentially.

2.3. Adaptive time-stepping

With FixDt == 0 (the default), the step size Dt_si is adjusted every step from an error estimate, typically err = MaxVecDiff(·,·) × Dt_si, compared against a tolerance MaxErr; if the error is too large the step is rejected and retried with a smaller Dt_si. With FixDt != 0, the step size is constant and this error-estimation logic becomes a no-op (for Heun) or is entirely absent (for the fixed-step variants discussed in Sec. VII).

2.4. CUDA execution model

A CUDA *kernel* is a GPU function; *launching* it (cuLaunchKernel) is a CPU-side driver call that has fixed overhead (typically several microseconds) independent of how little work the kernel does. MuMax3 issues all kernels and cuFFT operations on a single CUDA *stream* — a FIFO queue in which operations execute in issue order, with no explicit cross-operation synchronization needed. A *reduction* (e.g., MaxVecNorm, MaxVecDiff,

used for error estimates and diagnostics) requires copying a scalar result from device to host memory (cuMemcpyDtoH); the non-asynchronous form of this call blocks the CPU until all prior work on the stream completes — a *host–device synchronization*.

2.5. CUDA Graphs

A CUDA Graph is a representation of a sequence of GPU operations (and their dependencies) that can be recorded once, via *stream capture*, and then replayed many times with a single cudaGraphLaunch call. Because the entire sequence is submitted to the driver as one unit, graph replay amortizes the per-kernel launch overhead described above. Two operations are *incompatible with stream capture* and will abort it if encountered: non-asynchronous host–device memory copies (such as the reduction readback above) and certain memory-pool allocations (cuMemAlloc) — both relevant to Secs. III and VI.

2.6. Correctness verification tools used in this work

mumax3 -test runs MuMax3's built-in self-test suite. The test/ directory ships 176 regression scripts (.mx3 files) exercising a broad range of physical configurations. ovfdiff is a small Go utility written for this project that performs an element-by-element comparison of two OVF-format field-data files and reports the number of differing elements (ndiff) and the maximum absolute difference (maxDiff); ndiff=0 indicates the two files are bit-for-bit identical.

3. Methodology: an AI-assisted optimization workflow

3.1. Workflow and persistent project memory

The optimization work was carried out across multiple sessions of an agentic coding assistant (Claude Code) operating directly on the MuMax3 source tree, with build, test, and benchmark commands executed in a local PowerShell environment. Each session followed the same loop:

1. **Profile or hypothesize.** Either re-examine a previously identified bottleneck, or run NVIDIA Nsight Systems (nsys profile --trace=cuda + nsys stats) on representative .mx3 benchmark scripts (64×64×1, 256×256×1, 2048×2048×1, all using the Heun solver with demagnetization, exchange, and a constant Zeeman field) to obtain a quantitative breakdown of step time.
2. **Implement.** Modify Go and/or CUDA source files, regenerate CUDA wrapper code where needed (cuda2go + nvcc), and rebuild (go build -o mumax3.exe ./cmd/mumax3).
3. **Verify correctness.** Run mumax3 -test; for any change touching numerical results, run matched before/after scripts and compare field outputs with ovfdiff, targeting ndiff=0 (bit-identical) or differences attributable to the documented ~10^{-8}–10^{-7} floating-point reduction-order noise floor.
4. **Benchmark.** Measure throughput (S_PER_STEP, seconds per simulation step) over multiple repetitions, reporting both mean and median to reduce sensitivity to scheduling outliers.
5. **Run the regression suite.** Execute all 176 .mx3 scripts in test/ and inspect the per-script logs for expect()/expectv() failures or Go panics.
6. **Document.** Record the result — including negative results and design decisions not taken — in a persistent project log (a Markdown file checked into the repository) that is re-loaded at the start of every subsequent session.

This persistent log proved important for a project of this length: it is what allowed, for example, a numerical tolerance (~10^{-8}, attributable to GPU reduction accumulation order) identified in an early step to be recognized as "the same noise floor" when it reappeared in a later, unrelated change, rather than being mistaken for a new regression.

3.2. Three autonomous-debugging episodes

A recurring pattern in this project was that the assistant introduced a defect during an implementation step, *detected it itself* via the verification procedure above (Sec. III A, steps 3 and 5), traced it to a root cause in the existing codebase, and corrected it — without the root cause having been anticipated in advance. Box 1 summarizes three representative episodes, condensed from the project log.

Box 1. Three autonomous-debugging episodes

1. **FSAL state leakage in the RK45DP graph runner.** After implementing a CUDA-Graph runner for the default RK45DP solver, a 64×64×1 comparison of steps(500) against the new RunGraph(500) showed matching step counts and final simulation times, but the y-component of the averaged magnetization differed by 3.7 % — an order of magnitude above the ~10^{-8} noise floor established for the previously verified Heun runner. Tracing the discrepancy, the assistant found that MuMax3's standard run loop (RunWhile) calls stepper.Free() before iterating, which for *RK45DP sets the cached first-stage slope

k1 to nil, forcing it to be recomputed from the *current* magnetization on the next call. The new graph runner never called Free(), so it silently reused a stale k1 computed at the end of a 5-step warm-up — *before* the benchmark script's m = uniform(1,0,0) reset. The fix was a single added call, rk.Free(), mirroring the existing convention; re-verification with ovfdiff then returned ndiff=0.

2. **Buffer-pool ordering crash on first full-suite run (20/176 failures).** Immediately after integrating the graph path transparently into Run()/Steps(), a full 176-script regression run produced 20 access-violation crashes, all in scripts that took 20 steps (the new path's activation threshold). The assistant traced the crashes to cuMemAlloc being invoked from inside an active CUDA stream capture — a hard error in the CUDA driver. The root cause lay in MuMax3's existing size-keyed GPU buffer pool, which grows (via cuMemAlloc) only the first time a given allocation size is requested, based on the high-water mark of concurrently checked-out buffers. Each new graph runner performed a "warm-up" torque evaluation *before* checking out its own permanent buffers (the Runge–Kutta stage slopes, etc.), so the pool's high-water mark was set too low; the first permanent-buffer request during the subsequent capture then triggered a fresh, capture-illegal cuMemAlloc. The fix reordered all five graph runners so that the warm-up evaluation runs only *after* all permanent buffers are checked out — resolving 19 of the 20 failures.
3. **The 20th failure:** NoDemagSpins **and** CUDA_ERROR_STREAM_CAPTURE_UNSUPPORTED**.** The one remaining failure, test/nodemagspins.mx3, raised panic: Unknown CUresult: 900 (cudaErrorStreamCaptureUnsupported). The assistant traced the panic through the call stack SetDemagField → setMaskedDemagField → data.Copy → cuda.MemCpy → cuda.Sync, identifying that MuMax3's masked-demagnetization code path — used whenever any region has NoDemagSpins != 0 — issues a blocking host–device synchronization, which (like the reduction readback identified earlier in the project) cannot execute during graph capture. Rather than special-casing this rare configuration inside the capture itself, the assistant added one clause to the existing compatibility-guard function (Sec. VI): if NoDemagSpins is nonzero anywhere, the transparent graph path is skipped and the script silently falls back to the original execution path. With this change, all 176/176 regression scripts passed.

In all three cases, the assistant's own change had exposed a pre-existing assumption in the original MuMax3 codebase (about FSAL caching, buffer-pool growth, and synchronous masked-demag updates, respectively) that had simply never been stressed in that way before; none required guidance beyond the standard "verify, and if verification fails, diagnose and fix" instruction.

3.3. AI contribution disclosure

All source-code modifications, profiling analyses, benchmark scripts, and figure-generation scripts described in this paper were produced through the iterative, human-supervised process described above, using Claude Code (Anthropic). The author directed each session's objectives, reviewed every code change, and independently ran the verification procedure of Sec. III A before accepting any change. A complete, dated log of work items — one row

Table 1. Summary of implemented optimizations.

#	Target	Mechanism	Measured effect
1	cuda/conv_kernmul.go + kernmulrsymm2d{xy,z}.cu (demag kernel multiplication)	Halve the launch grid (N_y/2+1 rows) and have each thread compute both a row and its mirror row, exploiting the Hermitian symmetry the kernel already relied on	0.75 % throughput gain on a 2048×2048×1 grid; negligible at small/medium sizes
2	engine/{heun,rk4,rk23,rk45dp,rk56,backwardeuler,minimizer}.go, cuda/slice.go	Remove a per-step setMaxTorque() call (a MaxVecNorm reduction + host sync) that is not used by any step-size-control logic, replacing the GUI's maxtorque readout with the existing independent lazy getter; remove a redundant duplicate Sync() around MemCpyDtoH	5–8 % (small grid, median), 11–19 % (medium grid, median); <1 % (large grid)
3	cuda/cu/graph.go (new), cuda/init.go, engine/graphrun.go (new), per-solver Step() mirrors	"Split-graph" CUDA Graph capture/replay of the time-step-independent torque-evaluation kernels in each solver stage, transparently dispatched from Run()/Steps()	Up to 5.4× (small, FixDt!=0); 2.4–3.6× (adaptive); see Sec. V/VII

per project-log section referenced in Secs. IV–VII — is provided as Supplementary Material.

4. Summary of Implemented Optimizations

Three optimizations were implemented and verified, summarized in Table 1 and described below.

Optimizations 1 and 2 were identified directly from source-code inspection (an explicit // TODO-style comment in the case of #1, and a developer comment flagging an unused reduction in the case of #2) and required only localized, low-risk changes. Their effect, while real, is modest and concentrated in different size regimes: #1 matters only for the compute-bound large-grid case (where the kernel-multiplication step is a non-trivial fraction of GPU time), while #2 matters only for small/medium grids (where host synchronization overhead is a larger fraction of an otherwise short step).

Optimization 3 — the focus of the remainder of this paper — was motivated by a profiling observation rather than code inspection. For the small (64×64×1) and medium (256×256×1) benchmark grids, a cuLaunchKernel-by-cuLaunchKernel breakdown of one torque evaluation showed that **CPU-side driver call overhead, not GPU computation, dominates step time** (Fig. 1). Each torque evaluation issues 27 kernel launches sequentially on MuMax3's single CUDA stream; at 6.8 µs/launch, this driver overhead alone accounts for 75.5 % of the 250 µs/eval small-grid step time and 78.7 % of the 284 µs/eval medium-grid step time. (The bars in Fig. 1(b) do not sum to 100 % because CPU-side driver-call durations and GPU-side kernel execution overlap under MuMax3's asynchronous dispatch; each bar is an independent fraction of total wall-clock step time, not a partition of it.) By contrast, for the large (2048×2048×1) grid, GPU kernel execution accounts for 96 % of step time, and is itself dominated by the demagnetization FFT (68.9 %) and kernel-multiplication (12.8 %) steps (Fig. 1(a)).

This observation reframes the optimization target for small/medium simulations from "make the kernels faster" to "issue fewer driver calls per step" — which is exactly what CUDA Graph capture/replay provides, and is the subject of Secs. V–VII.

4.1. The split-graph design

A naive application of CUDA Graphs — capturing an entire adaptive solver step, including its error estimate and step-size control — fails outright, because the error-estimate reduction (MaxVecDiff/MaxVecNorm) requires a blocking host readback that stream capture forbids (Sec. II). The solution implemented here, illustrated in Fig. 2 for the Heun solver, is a **split graph**: only the *time-step-independent* torque-evaluation kernel sequences — which depend solely on the current magnetization buffer addresses, not on Dt_si — are captured, once, before the step loop begins. The time-step-dependent buffer update

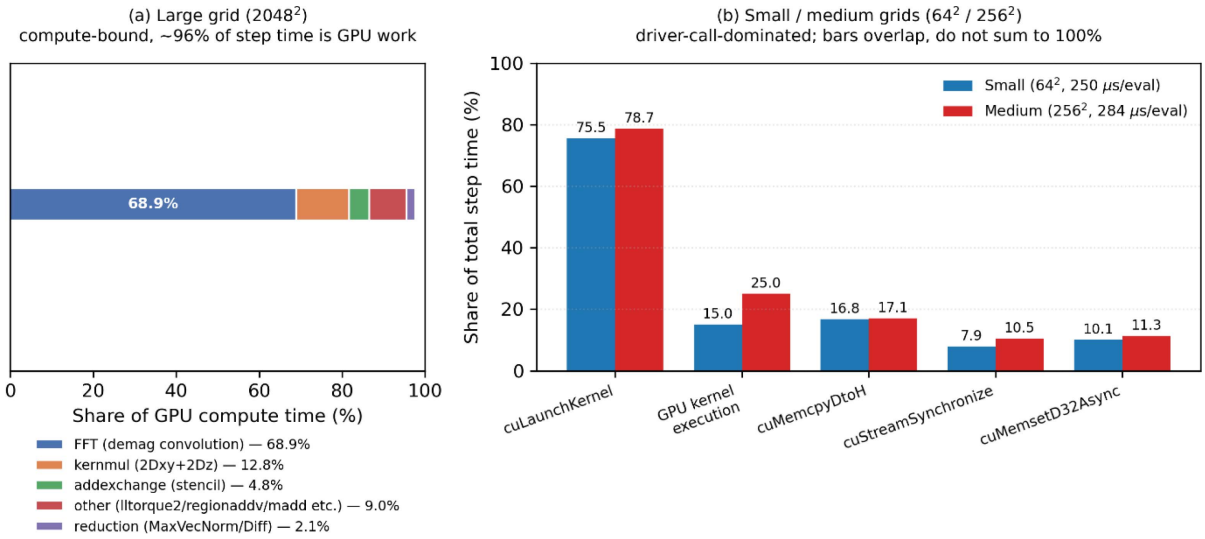

Fig. 1. (Color online) GPU/host time breakdown of one torque evaluation. (a) Large grid (2048×2048×1, compute-bound): a single stacked bar partitions 96 % of step time spent in GPU kernels among the demagnetization FFT, kernel multiplication, exchange stencil, reduction, and remaining torque/buffer-arithmetic kernels. (b) Small/medium grids (64×64×1/256×256×1, driver-call-dominated): grouped bars show each component's share of total step time (250 µs and 284 µs, respectively); cuLaunchKernel alone accounts for 75.5 %/78.7 %. Because CPU and GPU work overlap asynchronously, panel (b)'s bars sum to more than 100 % and should not be read as a partition.

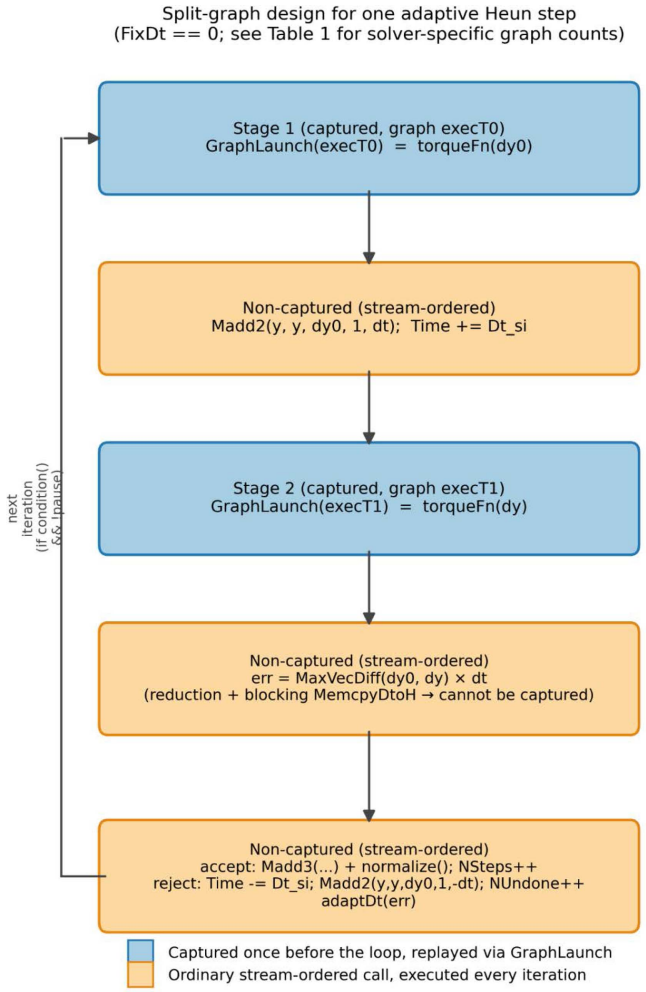

Fig. 2. (Color online) Split-graph design for one adaptive Heun step. The two dt-independent torque evaluations (top and middle, blue) are captured once before the loop and replayed via GraphLaunch; the dt-dependent update, error estimate (which requires a capture-incompatible blocking readback), and accept/reject + step-size adaptation (orange) execute as ordinary stream-ordered calls every iteration.

(Madd2/Madd3), the error estimate and its host readback, the accept/reject decision, and the adaptive step-size update remain ordinary stream-ordered calls on the same capture stream, executed fresh every iteration. Because the magnetization, slope, and error buffers are allocated once (with defer-based recycling at the end of the run) and never reallocated mid-loop, the addresses captured into each graph remain valid for the lifetime of the run.

This design generalizes directly to MuMax3's other explicit solvers by capturing one graph per Runge–Kutta stage's torque evaluation (Table 2); only Backward Euler, which is restricted to FixDt != 0, has just two captured graphs corresponding to its predictor and corrector evaluations.

4.2. Transparent integration and safety guards

The split-graph runners were initially exposed as an explicit, opt-in script function, RunGraph(n), restricted by an AssertMsg to a single verified solver/configuration. In the final implementation, Run()/Steps() dispatch to the graph path automatically via a non-throwing helper, tryRunGraph, subject to:

- a **compatibility check**, graphCompatible() (detailed in Sec. VI), verifying that the current configuration matches the conditions under which bit-identical results were established;
- a **cell-count threshold**, graphMaxCells (Sec. VII), below which the fixed cost of graph capture is reliably amortized; and
- a **minimum step count**, graphMinSteps = 20, so that capture overhead is amortized over enough replays.

If either check fails, Steps(n)/Run(t) fall back, with no observable difference to the user, to MuMax3's original RunWhile() loop.

5. Benchmark Setup and Results

All benchmarks were run on a single workstation: an NVIDIA GeForce RTX 5060 Ti (Blackwell, GB206, 36 SM, 4608 CUDA cores, 448 GB/s GDDR7, compute capability 12.0), driver 596.49 (CUDA 13.2 runtime), Windows 11, with the fork built directly from source via go build (Go 1.26.4, CUDA Toolkit 13.2, MinGW-w64 GCC for cgo). For each configuration, S_PER_STEP

Table 2. Number of captured graphs per solver (split-graph design).

Solver	solver() index	Captured graphs	Notes
Heun	2	2	torqueFn(dy0), torqueFn(dy)
RK23 (Bogacki–Shampine)	3	3	one per intermediate stage
RK45DP (Dormand–Prince, default)	5	5	stage 7 reuses the stage-2 graph (FSAL)
RK56	6	8	stage 9 (the 6th-order solution) needs no torque evaluation
Backward Euler	–1	2	FixDt != 0 only

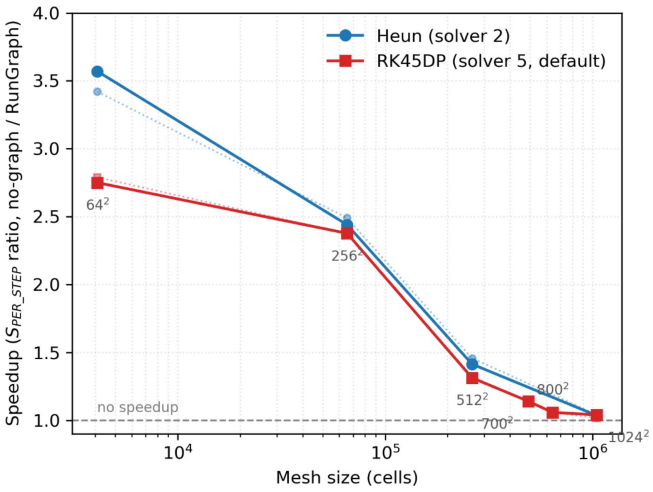

Fig. 3. (Color online) Speedup (ratio of S_{PER_STEP} without/with the optimization; median, with mean shown as a lighter dotted overlay) vs. mesh size (cells, log scale), for Heun and RK45DP with adaptive time-stepping (FixDt == 0). Both series decrease smoothly from 2.4–3.6 at 4096 cells to 1.0 near 10^6 cells.

(wall-clock seconds per simulation step) was measured over 10 repetitions (after a 5-step warm-up) for small/medium/large grids, and both the mean and median are reported; the median is used as the primary figure of merit because run-to-run driver/OS scheduling noise occasionally produces 2× outliers in individual repetitions.

Figure 3 shows the speedup — the ratio of S_PER_STEP without the optimization to S_PER_STEP with it — as a function of mesh size, for adaptive time-stepping (FixDt == 0), for the Heun and RK45DP (default) solvers. Both series decrease smoothly and monotonically with cell count: Heun from 3.6× (64^2) to 2.4× (256^2) to 1.46× (512^2) to 1.04× (1024^2); RK45DP from 2.8× (64^2) to 2.4× (256^2) to 1.32× (512^2) to 1.06× (800^2) to 1.02× (1.024^2). RK45DP's somewhat smaller speedups relative to Heun reflect that its split-graph design (Table 2) leaves a larger fraction of each step (5 stage updates, 5 normalizations, the error estimate, and the FSAL copy) outside the captured graphs.

For a fixed time step (FixDt != 0), where the error-estimate logic is entirely absent from the captured-graph path (Heun only), speedups are substantially larger (Table 3).

6. Correctness Verification and Safety Guards

6.1. Bit-identical verification

For every solver and configuration listed in Table III and Fig. 3, the magnetization and demagnetizing-field outputs (m000000.ovf, B_demag000000.ovf) of a run using the original code path were compared element-by-element, via ovfdiff, against the same script using the optimized path. In every case the result was ndiff=0, maxDiff=0 — bit-for-bit identical. Scalar diagnostics such as m_avg occasionally differ in their last 1–2 significant digits (~10^{-8}–10^{-7} relative); this was traced to the accumulation order of the m.average() GPU reduction and is reproducible even between two runs of the *unmodified* code, i.e., it is a pre-existing run-to-run noise floor unrelated to this work.

6.2. Regression suite

The 176 .mx3 scripts in MuMax3's test/ directory were executed after each code change. Because a pre-existing issue with the PowerShell test harness causes its summary file to misreport exit codes for *every* script (independent of this work), each run's _results/*.log and *.err files

Table 3. Additional configurations.

Configuration	Solver	Cells	Speedup (mean/median)	Notes
Fixed Dt=2×10^{-13} s	Heun	4,096 (64^2)	5.13× / 5.36×	error estimate entirely absent from captured path
Fixed Dt=2×10^{-13} s	Heun	65,536 (256^2)	3.32× / 3.45×	
3D ((64×64×16), demag+exchange+Zeeman	Heun	65,536	2.06× / 2.08×	3D demagnetization FFT
3D (64×64×16), demag+exchange+Zeeman	RK45DP	65,536	1.98× / 1.95×	
2D 256×256×1 (same cell count, for reference)	Heun	65,536	2.49× / 2.44×	thin-film reference
2D 256×256×1 (same cell count, for reference)	RK45DP	65,536	2.37× / 2.38×	
Realistic field set: demag+exchange+Zeeman+K_{u1}/AnisU+DMI	Heun	65,536	2.54× / 2.53×	within noise of the demag+exchange+Zeeman case
Realistic field set: demag+exchange+Zeeman+K_{u1}/AnisU+DMI	RK45DP	65,536	2.35× / 2.31×	

were inspected directly for expect()/expectv() failure messages ("have: ... want: ...") and for Go panics/fatal errors. The final implementation passes 176/176 with zero such messages.

6.3. The compatibility guard, graphCompatible()

Because the bit-identical verification above necessarily covers only a finite set of configurations, the transparent integration (Sec. IV B) is gated by an explicit compatibility check, graphIncompatibilityReason(), that returns a non-empty, human-readable reason whenever the current simulation configuration falls outside the verified envelope — in which case tryRunGraph returns false and execution proceeds via the original, unmodified RunWhile() loop with no change in behavior. The guard rejects:

- a nonzero thermal field (Temp != 0), whose thermal-noise generation (cuRAND) inside the captured graph has unverified replay semantics;
- a time-dependent applied field $\mathbf{B}_{ext}$ (any extraTerms, or a per-region value set via a function of t);
- any custom field term added via AddFieldTerm;
- any of 21 region-wise material or torque parameters (e.g., M_{sat}, A_{ex}, D_{ind}, D_{bulk}, $K_{u1,2}$, $K_{c1,2,3}$, AnisU/AnisC1/AnisC2, $B_{1,2}$, α, and the spin-transfer-torque parameters ξ, P, Λ, ε' [23, 24], FrozenSpins, FreeLayerThickness) set as a function of time via SetRegionFn; and
- a nonzero NoDemagSpins mask (Sec. III B, episode 3), whose masked-demagnetization path issues a capture-incompatible blocking synchronization.

If a script changes any of these settings *during* a run (e.g., via the GUI or an Inject-based script callback) after the graph path has already started, a second mechanism, checkInject, detects the pending callback at the top of each replay iteration; if it executes without merely pausing the run, the remaining steps fall back (fellBack) to RunWhile(), which always re-evaluates the torque under the current (possibly changed) configuration.

Two known gaps in this guard remain (Sec. VIII): the magnetoelastic strain excitations (exx, eyy, ezz, exy, exz, eyz) are not covered by the time-independence check applied to the 21 parameters above, even though they enter the captured torque evaluation when the magnetoelastic coupling constants B_1/B_2 are nonzero.

7. Size Dependence of the Optimization

The smooth decrease in speedup with cell count seen in Fig. 3 reflects a simple competition: the work eliminated by the split-graph optimization — 27 driver calls and one reduction readback per step — is a roughly *fixed* per-step overhead, while the GPU compute time it is compared against grows with the cell count. As the latter grows, the former becomes a smaller fraction of total step time, and the achievable speedup approaches 1×.

This trend is consistent across all axes of generalization examined (Table III): doubling the linear grid size from 64^2 to 1024^2 (a 256× increase in cell count) reduces the Heun speedup from 3.6× to 1.0× and the RK45DP speedup from 2.8× to 1.0×, with intermediate points at 256^2 (2.4), 512^2 (1.3–1.5), and (RK45DP only) 700^2/800^2 (1.14×/1.06×). Switching from a 2D thin-film grid to a 3D grid of the *same total cell count* (64×64×16 vs. 256×256×1, both 65,536 cells) reduces the speedup from to 2.0 for both solvers — consistent with the 2.4–2.5 larger per-cell GPU cost of a 3D demagnetization FFT increasing the denominator of the speedup ratio in the same way that a larger 2D grid does. By contrast, adding physically significant field terms (uniaxial anisotropy and interfacial DMI) at fixed cell count changes the speedup by less than the run-to-run noise (2.54×/2.35× vs. 2.49×/2.37× for the demag+exchange+Zeeman baseline) — the *composition* of the captured torque evaluation has essentially no effect on the achievable speedup, only its *cost relative to cell count* does.

This analysis directly motivated the choice of graph-MaxCells = 800{,}000: at 640,000 cells (800×800×1), the RK45DP speedup is 1.06 (a 6 % gain, confirmed in a same-session before/after comparison at 1.10), while at 1,048,576 cells (1024^2) it is 1.0–1.04 (no longer worthwhile). The threshold of 800,000 cells thus includes configurations with at least a 5 % expected gain while excluding the ~10^6-cell regime where graph capture's one-time cost is no longer reliably amortized.

8. Discussion and Future Directions

8.1. What worked, and why

The largest gain in this project (the CUDA-Graph split-graph design, Sec. IV) came not from the two changes initially suggested by source-code inspection (Table 1, items 1–2, which together yield only 1–10 % depending on grid size), but from a profiling-driven observation — that small/medium-grid step time is dominated by CPU-side driver overhead rather than GPU compute (Fig. 1) — that was not evident from reading the source code alone. This suggests that, for agentic-AI-assisted optimization of mature HPC codebases, *profiling-first* workflows may be substantially more productive than *code-reading-first* workflows, even though the latter are cheaper to set up.

8.2. Limitations and known gaps

One safety-relevant gap remains in the compatibility guard (Sec. VI C): the magnetoelastic strain excitations are not checked for time-independence, even though — like the 21 material parameters that *are* checked — they enter the captured torque evaluation when the corresponding coupling constants are nonzero. No script in the 176-script regression suite combines a time-dependent strain excitation with a nonzero magnetoelastic coupling, so this gap is not currently triggered, but closing it (by extending hasTimeDependentRegion-style checking to the ScalarExcitation parameters exx/eyy/ezz/exy/exz/eyz) is recommended before this fork is used for magnetoelastic simulations.

8.3. Directions considered and not pursued

Two further directions were analyzed but not implemented, for documented reasons. First, a multi-stream refactor (computing independent effective-field contributions — exchange, anisotropy, DMI, Zeeman — concurrently on separate streams before summing) was estimated to address at most 31 % of large-grid GPU time (the complement of the FFT-dominated 68.9 %, Fig. 1(a)), at the cost of a substantial redesign of MuMax3's single-stream buffer-pool ownership model, with additional risk that the RTX 5060 Ti's 36 SMs may already be near saturation for these kernels, limiting realizable concurrency. Second, replacing cuFFT with a third-party FFT library (VkFFT) was considered as the only third-party-library option with a large theoretical upper bound (the 68.9 % FFT share), but was judged too high-risk relative to its uncertain benefit over an NVIDIA-tuned library on current hardware; CPU libraries (MKL, GSL) and CUB-based reductions were found to have no applicable target, since MuMax3's entire numerical core already executes on the GPU and reductions account for only 2 % of large-grid GPU time.

8.4. Broader implications

This case study suggests that agentic coding assistants can meaningfully extend the useful lifetime of mature, hardware-sensitive HPC codebases — not merely by applying textbook optimizations, but by combining profiling-driven discovery with the disciplined, exhaustive regression verification that such changes require, including the autonomous detection and correction of self-introduced defects (Box 1). The explicit compatibility-guard design (Sec. VI C) — verify a finite envelope exhaustively, then gate the optimization on staying within it, falling back silently otherwise — may be a useful general pattern for deploying AI-assisted optimizations of numerical codes where an unverified silent wrong answer would be far worse than no optimization at all.

9. Code Availability and Installation

The modified source tree is a fork of MuMax3 [1] (github.com/mumax/3, commit 3fe3d41fcb6bb6f933a744d3c609b1bcc0368022), available at https://github.com/mirryou-maker/mumax-CO. The fork builds with the standard Go toolchain (Go 1.22) and the CUDA Toolkit (verified with 13.2); on Windows, a single one-line patch to cuda/cu/context.go (cuCtxCreate → cuCtxCreate_v2, required for CUDA 13.0's ABI change) is needed before running:

```
go build -o mumax3.exe ./cmd/mumax3
```

No additional dependencies beyond those required by upstream MuMax3 are introduced. The CUDA-Graph optimization (Sec. IV) is on by default and requires no script changes; it can be disabled for a given run by violating any condition of graphCompatible() (Sec. VI C), e.g., by adding a (zero-valued) custom field term, which is also how the "no-graph" baselines in Sec. V were constructed for cell counts within graphMaxCells.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) (Grant No. RS-2026-25472340). All code modifications, benchmark scripts, and figure-generation scripts described in this work were produced in collaboration with Claude Code (Anthropic), an agentic large-language-model coding assistant, under the author's direction and review; see Sec. III C for the disclosure statement and the Supplementary Material for a complete log of AI-assisted work items.

Supplementary Material

A complete, dated log of AI-assisted work items (one row per project-log section referenced above) is provided as supplementary.csv, listing for each item: the date, the task, the assistant's role, and the human verification step performed before acceptance.

References

[1] A. Vansteenkiste, J. Leliaert, M. Dvornik, M. Helsen, F. Garcia-Sanchez, and B. Van Waeyenberge, AIP Adv. **4**, 107133 (2014).

[2] M. J. Donahue and D. G. Porter, Interagency Report NISTIR 6376, National Institute of Standards and Tech-

nology, Gaithersburg, MD (1999).
[3] C.-Y. You, J. Magn. **17**, 73 (2012).
[4] A. Vansteenkiste and B. Van de Wiele, J. Magn. Magn. Mater. **323**, 2585 (2011).
[5] L. Moreels, I. Lateur, D. De Gusem, *et al.*, npj Comput. Mater. **12**, 71 (2026).
[6] F. Bruckner, S. Koraltan, C. Abert, and D. Suess, Sci. Rep. **13**, 12054 (2023).
[7] J. Leliaert and J. Mulkers, J. Appl. Phys. **125**, 180901 (2019).
[8] A. Fert, V. Cros, and J. Sampaio, Nat. Nanotechnol. **8**, 152 (2013).
[9] J. Sampaio, V. Cros, S. Rohart, A. Thiaville, and A. Fert, Nat. Nanotechnol. **8**, 839 (2013).
[10] C. Back, V. Cros, H. Ebert, *et al.*, J. Phys. D: Appl. Phys. **53**, 363001 (2020).
[11] A. J. Newell, W. Williams, and D. J. Dunlop, J. Geophys. Res. **98**, 9551 (1993).
[12] T. L. Gilbert, IEEE Trans. Magn. **40**, 3443 (2004).
[13] J. R. Dormand and P. J. Prince, J. Comput. Appl. Math. **6**, 19 (1980).
[14] C. J. García-Cervera, Bol. Soc. Esp. Mat. Apl. **39**, 103 (2007).
[15] NVIDIA Corporation, version 12.x, section on CUDA Graphs, https://docs.nvidia.com/cuda/cuda-c-programming-guide/ (2024).
[16] HPC CUDA Graph overhead-reduction study — placeholder
[17] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, in *Proc. ICLR 2024*, arXiv:2310.06770.
[18] Anthropic, "Claude Code" (2025), https://www.anthropic.com/claude-code.
[19] Anthropic, "Claude Opus 4 & Claude Sonnet 4 System Card" (May 2025).
[20] A. Ouyang, S. Guo, S. Arora, A. L. Zhang, W. Hu, C. Ré, and A. Mirhoseini, arXiv:2502.10517 (2025).
[21] Sakana AI, Technical Report (2025), pub.sakana.ai/static/paper.pdf.
[22] "CUDA-LLM: LLMs Can Write Efficient CUDA Kernels," arXiv:2506.09092 (2025).
[23] H. Kim and C.-Y. You, J. Magn. **21**, 491 (2016).
[24] C.-Y. You, Appl. Phys. Lett. **100**, 252402 (2012).