Discovering Faster Linear Solvers for FLASH Magnetic-Diffusion Systems

HRISHIKESH GARUD, Autodidakt AI

We built a system to discover faster linear solvers for magnetic-diffusion systems arising in *FLASH*. To ground the discovery process, we assembled an initial dataset of linear systems from parameter variations of *AlWire* problems 1 and 2 and the *ZPinch* example, together with two *MagDiff* unit tests: the implicit-AMR test and the edge test. **We achieved a 1.546× speedup (54.6%) over baseline**.

1 Introduction

Large systems of linear equations underpin some of the most important scientific and engineering problems across multiple domains. They often form the most expensive part of production simulation stacks and any algorithmic progress in linear solvers will directly translate into advances in respective fields and will be a net positive for society. Therefore, we are building autonomous systems that enable us to optimize current algorithms and discover novel ones to solve them.

This report provides details on the work towards discovering linear solvers for *magnetic diffusion* systems arising in *Magnetic / Magneto-Inertial Confinement Fusion* power devices.

2 Dataset

We prepared a dataset of 386 magnetic-diffusion linear systems from 14 *FLASH* simulation configurations. These configurations are variations of *FLASH*'s supplied *AlWire*, *ZPinch*, and *MagDiff* test problems (table 1). These configurations do not represent production parameters, however the structure and types of resultant matrices are a good proxy for real production systems. Table 2 shows the *AlWire* and *ZPinch* configurations used.

For each *AlWire* and *ZPinch* run, we collected every magnetic-diffusion linear system in timesteps 1 - 32, including systems before and after mesh refinement. For each *MagDiff* unit test, we collected the first linear system to check numerical correctness.

The dataset contains matrices with $16,420 - 2,153,480$ nonzeros. We determined all matrices to be numerically nonsymmetric by measuring the relative difference between each matrix and its transpose using the Frobenius norm:

$$\delta(A) = \frac{\|A - A^{\mathsf{T}}\|_F}{\|A\|_F}$$

We computed a high-accuracy reference solution, x_{ref}, for each system using *SciPy*'s 'spsolve' for sparse LU factorization and triangular solves. We cross-checked each solution with the independent iterative solver *GCROT(m,k)*.

3 Results

We measure speedup against *HYPRE*'s *GMRES* solver with restart length 50 and *BoomerAMG* preconditioning.

3.1 Harness 1

The core of this harness is an RL optimization loop over open-source LLMs. We use the *GPT-OSS 120B* model for experiments with this harness.

This harness explored *GMRES* and *BiCGSTAB* implementations, varying restart length, orthogonalization, and *AMG* settings. Its best solver achieved 1.051× aggregate speedup (5.1%) using *GMRES(30)*, one-pass classical *Gram–Schmidt* with batched inner products, and *BoomerAMG*'s default forward/backward ℓ_1-*Gauss–Seidel* smoothing.

3.2 Native Harnesses

GPT-5.6-Sol (High) with *Codex* tested proposals covering *GMRES* restart lengths 20-50, one-pass versus two-pass orthogonalization, batched vector operations, and *AMG* coarsening, smoothing, and sparsity settings. Its best solver achieved 1.153× aggregate speedup (15.3%) using *GMRES(30)*, one-pass modified *Gram–Schmidt*, and an *AMG* strength threshold of 0.5 instead of 0.25, while retaining symmetric *Gauss–Seidel* smoothing.

Claude Fable 5.1 (High) with *Claude Code* tested proposals covering *Jacobi-* and *ILU*-preconditioned *BiCGSTAB* stages followed by *GMRES*, *AMG* coarsening, interpolation, and smoothing settings, and *Krylov* basis allocation and vector operations.

Its best solver achieved 1.106× aggregate speedup (10.6%) using *GMRES(40)* with one *BoomerAMG* V-cycle per iteration. It allocated the basis as needed in contiguous batches of eight vectors and used classical *Gram–Schmidt* with fused inner products and vector updates, selective reorthogonalization, and a fused solution update. Its *AMG* configuration uses *HMIS* (Hybrid Modified Independent Set) coarsening to select coarse-grid unknowns and extended+i interpolation to transfer corrections between levels, with at most four interpolation weights per row. It uses a strength threshold of 0.25, forward/backward *Gauss–Seidel* smoothing with coarse/fine ordering, and a direct solve on the coarsest grid of at most 64 unknowns. It explicitly computes the true residual before accepting convergence.

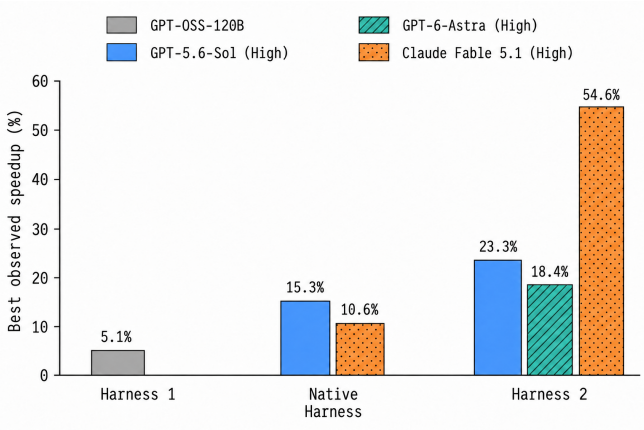

Fig. 1. Speedup over baseline.

FLASH test problem	Collected systems	Unknowns per system	Physics and simulation interval
AlWire problem 1	4 runs × 32 = 128	12,288–184,512	Cylindrical wire; constant conductor resistivity with a conductor/vacuum jump. Fixed 10 ns steps through 320 ns.
AlWire problem 2	4 runs × 32 = 128	12,288–196,608	Cylindrical wire; spatially varying conductor resistivity. Fixed 1 ns steps through 32 ns.
ZPinch	4 runs × 32 = 128	9,984–44,544	Cylindrical MHD with a prescribed current and Davies–Wen state-dependent resistivity. Initial timesteps grow from 1 to 19.19 ps, reaching 0.201 ns.
MagDiff: `implicit_amr.par`	1	7,680	Cartesian implicit magnetic diffusion with adaptive mesh refinement; first magnetic solve at 1 ns.
MagDiff: `edgeTest/flash.par`	1	4,224	Cartesian magnetic-diffusion test with edge-centered cross derivatives; first magnetic solve at 1 ns.

Table 1. FLASH test problems used to collect magnetic-diffusion linear systems.

FLASH parameter	AlWire	ZPinch
Time-integration parameter θ (`diff_magThetaImplct`)	0.5 or 1	0.5 or 1
Blocks at refinement level 1 (`nblockx` × `nblocky`)	8×8, 16×16, or 32×32	2×2 or 4×4
Maximum refinement level (`lrefine_max`)	1–3	3–4
Current input	Constant or smoothly varying	Supplied `current.dat` waveform
Vacuum magnetic diffusivity (`res_vacRes`)	10^5, 10^8, or 10^{12} cm^2/s	10^{11} or 10^{12} cm^2/s

Table 2. FLASH parameter values used in the AlWire and ZPinch runs.

3.3 Harness 2

This harness doesn't use an RL optimization loop, which enables us to use frontier models that pair well with aspects of this harness that are ideal for scientific research and discovery tasks.

We evaluated this harness with *GPT-5.6-Sol (High)*, *GPT-6-Astra (High)*, and *Claude Fable 5.1 (High)*. Their best solvers achieved speedups of 1.233× (23.3%), 1.184× (18.4%), and 1.546× (54.6%), respectively.

With *GPT-5.6-Sol (High)*, this harness explored *GMRES* and *BiCGSTAB*, *AMG* and *incomplete-LU* preconditioning, restart size, and vector-operation costs. Its best solver achieved 1.233× aggregate speedup (23.3%) using *GMRES(24)*, batched classical *Gram–Schmidt* with selective reorthogonalization, and guarded norm estimates. Its *AMG* configuration combines a strength threshold of 0.5, interpolation truncation of 0.1, forward/backward *Gauss–Seidel* smoothing, and symmetric *Gauss–Seidel* on the coarsest grid. The solver also checks the initial residual before allocating the full *Krylov* workspace or building *AMG*, avoiding that work when the supplied initial guess already meets tolerance.

With *GPT-6-Astra (High)*, this harness explored *BiCGSTAB* and *GMRES*, switching between *ILU* and *AMG* preconditioning, coarsening and smoothing settings, and batched orthogonalization and basis allocation. Its best solver achieved 1.184× aggregate speedup (18.4%) using a guarded *BiCGSTAB* stage with *BoomerAMG* preconditioning, followed by *GMRES(32)* when progress stalls or the recurrence breaks down.

The *GMRES* correction uses batched classical *Gram–Schmidt* with selective reorthogonalization and allocates its basis only when needed. Both stages share an *AMG* hierarchy with *HMIS* coarsening, extended+i interpolation, a strength threshold of 0.25, forward/backward *Gauss–Seidel* smoothing, and a direct coarse-grid solve. For systems with at least 32,768 unknowns, it adds one aggressive coarsening level.

With *Claude Fable 5.1 (High)*, this harness explored adaptive switching between inexpensive iterations and *AMG*, warm-start rescaling, convergence-rate estimates, and multigrid hierarchy settings. Its best solver achieved 1.546× aggregate speedup (54.6%) using a cost-adaptive solver. It attempts a least-squares rescaling of the supplied initial guess and uses a cost estimate to decide whether to try inexpensive conjugate-gradient iterations with optional *Jacobi* scaling.

The solver monitors residual progress and estimates the remaining work to decide when to switch to multigrid. It builds the *BoomerAMG* hierarchy only when needed, then uses guarded preconditioned conjugate-gradient iterations with a *GMRES(30)* fallback on loss of definiteness or stagnation. Its *AMG* configuration uses *HMIS* coarsening, two aggressive coarsening levels, a strength threshold of 0.5, forward/backward *Gauss–Seidel* smoothing, and a direct coarse-grid solve.

These results identify useful combinations of established solver techniques and show promising signs of improved performance by further scaling up compute.