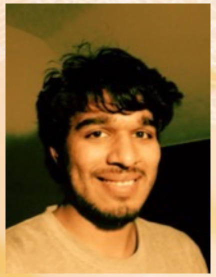
Batched Iterative Solvers in Plasma Fusion Simulations

JLESC 2023
Bordeaux, France

Hartwig Anzt
Innovative Computing Lab, University of Tennessee

Isha Aggarwal

Aditya Kashi


Pratik Nayak

Dhruva Kulkarni

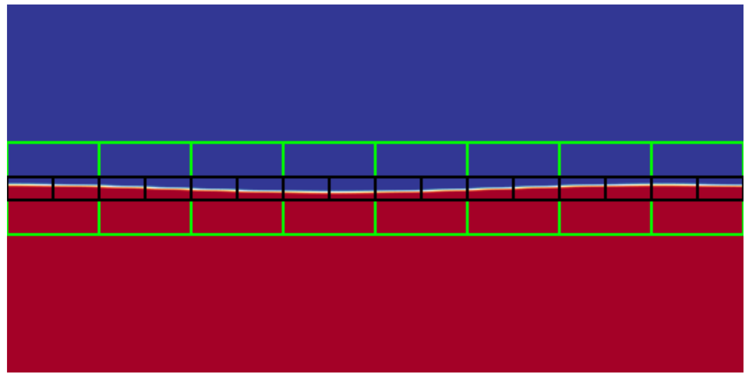
Paul Lin

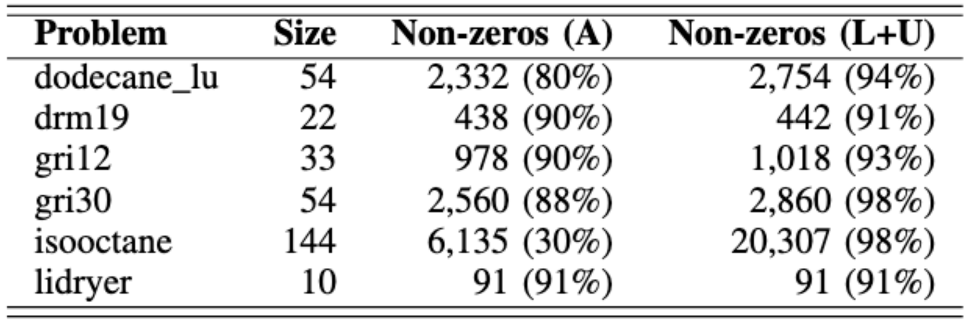
This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, the Horizon2020 Program of the European Commission, and the Helmholtz Impuls und VernetzungsfondVH-NG-1241.

Motivation: Combustion simulations

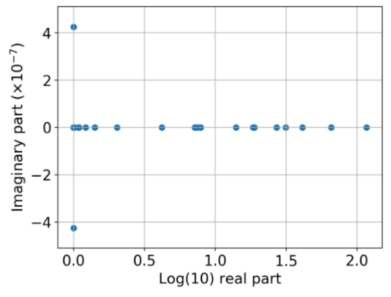
PeleLM is a parallel, adaptive mesh refinement (AMR) code that solves the reacting Navier-Stokes equations in the low Mach number regime. The core libraries for managing the subcycling AMR grids and communication are found in the AMReX source code.

https://amrex-combustion.github.io/PeleLM/overview.html

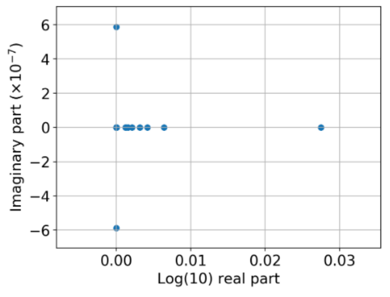




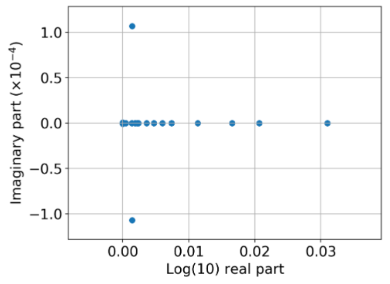
Problem	Size	Non-zeros (A)	Non-zeros (L+U)
dodecane_lu	54	2,332 (80%)	2,754 (94%)
drm19	22	438 (90%)	442 (91%)
gri12	33	978 (90%)	1,018 (93%)
gri30	54	2,560 (88%)	2,860 (98%)
isooctane	144	6,135 (30%)	20,307 (98%)
lidryer	10	91 (91%)	91 (91%)



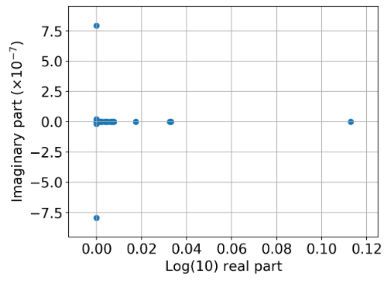
(a) dodecane_lu



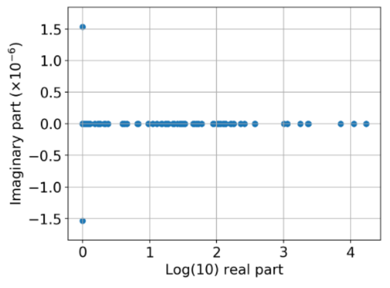
(b) drm19



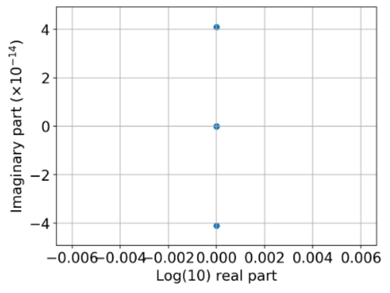
(c) gri12



(d) gri30



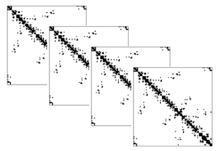
(e) isooctane



(f) lidryer

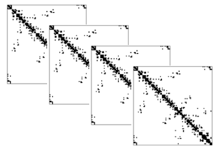
Batched Iterative Solver Setting

- Many sparse problems of medium size have to be solved concurrently.
  - ~ 50 – 2,000 unknowns, < 50% dense;
  - All sparse systems may share the same sparsity pattern;
  - An approximate solution may be acceptable (e.g., inside a non-linear solver);

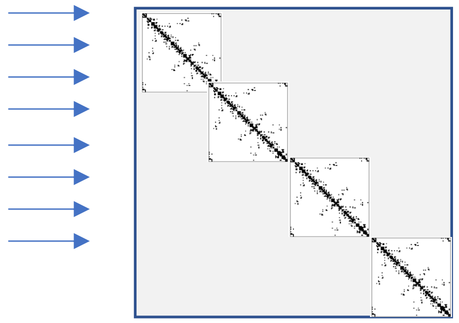


Batched Iterative Solver Setting

- Many sparse problems of medium size have to be solved concurrently.
  - ~ 50 – 2,000 unknowns, < 50% dense;
  - All sparse systems may share the same sparsity pattern;
  - An approximate solution may be acceptable (e.g., inside a non-linear solver);

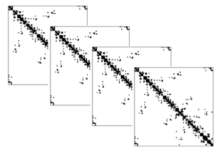


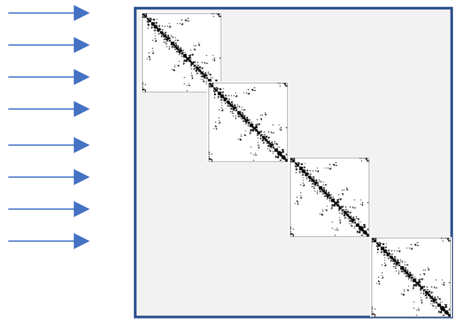
- One solution is to arrange the individual systems on the main diagonal of one large system.
  - Convergence determined by the “hardest” problem;
  - No reuse of sparsity pattern information;
  - Global synchronization points;



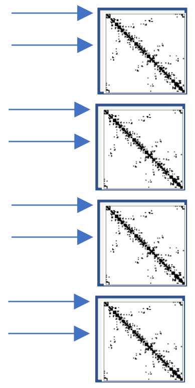
Batched Iterative Solver Setting

- Many sparse problems of medium size have to be solved concurrently.
  - ~ 50 – 2,000 unknowns, < 50% dense;
  - All sparse systems may share the same sparsity pattern;
  - An approximate solution may be acceptable (e.g., inside a non-linear solver);





- One solution is to arrange the individual systems on the main diagonal of one large system.
  - Convergence determined by the “hardest” problem;
  - No reuse of sparsity pattern information;
  - Global synchronization points;
- Better approach: design batched iterative solve functionality that solves all problems concurrently.
  - Problem-dependent convergence accounted for;
  - No global synchronization;
  - Reuse of sparsity pattern information;
  - Parallelize across individual problems;



Performance aspects of batched kernels

1. Batched functionality is generally memory-bound;
   -> Urgent need to minimize main memory access;

Implication for (dense) direct methods

- Algorithm steps need to be merged into one kernel (e.g. Gauss-Jordan-Elimination for inversion);

Implication for (sparse) iterative methods

- Interfacing external components via main memory impacts performance;
- All algorithm components have to be in-lined in the kernel code;

Performance aspects of batched kernels

	Implication for (dense) direct methods	Implication for (sparse) iterative methods
1. Batched functionality is generally memory-bound; *-> Urgent need to minimize main memory access;*	• Algorithm steps need to be merged into one kernel (e.g. Gauss-Jordan-Elimination for inversion);	• Interfacing external components via main memory impacts performance; • All algorithm components have to be in-lined in the kernel code;
2. Different problems have different resource requirements; *-> hard to predict the register/shared memory requirement;*	• Different kernels for different problem sizes;	• Sparse matrix memory needs unknown; • Caching can only be use for const data; • Shared memory space for intermediate vectors unknown;

Performance aspects of batched kernels

	Implication for (dense) direct methods	**Implication for (sparse) iterative methods**
1. Batched functionality is generally memory-bound; *-> Urgent need to minimize main memory access;*	• Algorithm steps need to be merged into one kernel (e.g. Gauss-Jordan-Elimination for inversion);	• Interfacing external components via main memory impacts performance; • All algorithm components have to be in-lined in the kernel code;
2. Different problems have different resource requirements; *-> hard to predict the register/shared memory requirement;*	• Different kernels for different problem sizes;	• Sparse matrix memory needs unknown; • Caching can only be use for const data; • Shared memory space for intermediate vectors unknown;
3. Different problems may result in different algorithm behavior; -> unpredictable algorithm execution;	• Pivoting can result in some branching in the kernel execution;	• Need to monitor iterative solver convergence for each problem individually and complete early; • Need to schedule problems “appropriately”;

Performance aspects of batched kernels

```
$\boldsymbol{r} \leftarrow \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}, \hat{\boldsymbol{r}} \leftarrow \boldsymbol{r}, \boldsymbol{p} \leftarrow \boldsymbol{0}, \boldsymbol{v} \leftarrow \boldsymbol{0}$
$\rho' \leftarrow 1, \omega \leftarrow 1, \alpha \leftarrow 1$
for $i < N_{iter}$ do
    if $\|\boldsymbol{r}\| < \tau$ then
        Break
    end if
    $\rho \leftarrow \boldsymbol{r} \cdot \boldsymbol{r}'$
    $\beta \leftarrow \frac{\rho' \alpha}{\rho \omega}$
    $\boldsymbol{p} \leftarrow \boldsymbol{r} + \beta(\boldsymbol{p} - \omega \boldsymbol{v})$
    $\hat{\boldsymbol{p}} \leftarrow \mathrm{PRECOND}(\boldsymbol{p})$
    $\boldsymbol{v} \leftarrow \boldsymbol{A}\hat{\boldsymbol{p}}$
    $\alpha \leftarrow \frac{\rho}{\hat{\boldsymbol{r}} \cdot \boldsymbol{v}}$
    $\boldsymbol{s} \leftarrow \boldsymbol{r} - \alpha \boldsymbol{v}$
    if $\|\boldsymbol{s}\| < \tau$ then
        $\boldsymbol{x} \leftarrow \boldsymbol{x} + \alpha \hat{\boldsymbol{p}}$
        Break
    end if
    $\hat{\boldsymbol{s}} \leftarrow \mathrm{PRECOND}(\boldsymbol{s})$
    $\boldsymbol{t} \leftarrow \boldsymbol{A}\hat{\boldsymbol{s}}$
    $\omega \leftarrow \frac{\boldsymbol{t} \cdot \boldsymbol{s}}{\boldsymbol{t} \cdot \boldsymbol{t}}$
    $\boldsymbol{x} \leftarrow \boldsymbol{x} + \alpha \hat{\boldsymbol{p}} + \omega \hat{\boldsymbol{s}}$
    $\boldsymbol{r} \leftarrow \boldsymbol{s} - \omega \boldsymbol{t}$
    $\rho' \leftarrow \rho$
end for
```

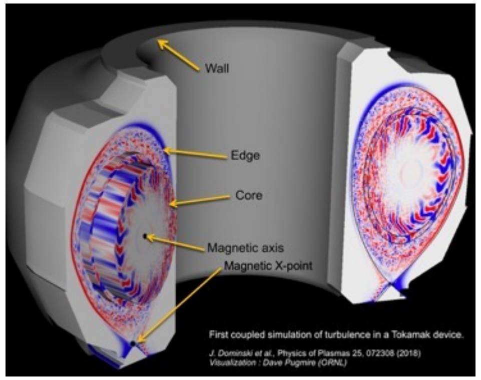
Ginkgo

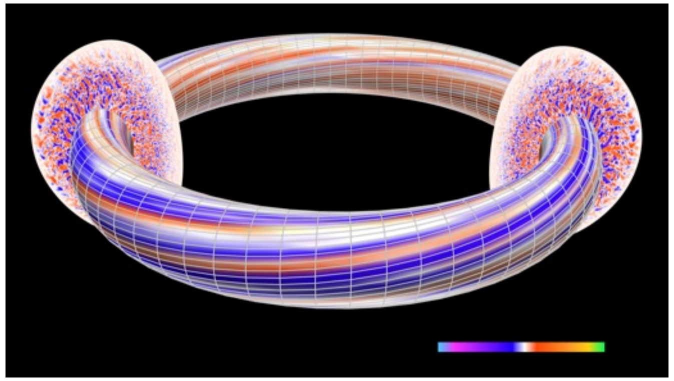
- Red objects: Intermediate vectors in SpMV: High priority
- Blue objects: Other vectors: Low priority
- Green objects: Constant matrices or vectors (cache)

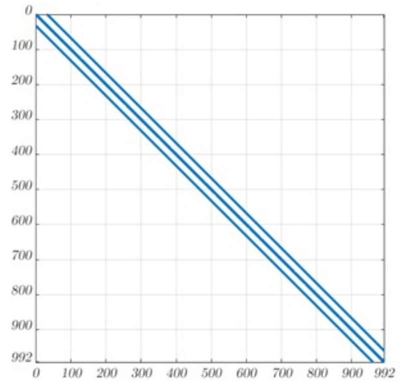
First experiences with Ginkgo’s batched iterative solvers in XGC

XGC is a gyrokinetic particle-in-cell code, which specializes in the simulation of the edge region of magnetically confined thermonuclear fusion plasma. The simulation domain can include the magnetic separatrix, magnetic axis and the biased material wall. XGC can run in total-delta-f, and conventional delta-f mode. The ion species are always gyrokinetic except for ETG simulation. Electrons can be adiabatic, massless fluid, driftkinetic, or gyrokinetic.

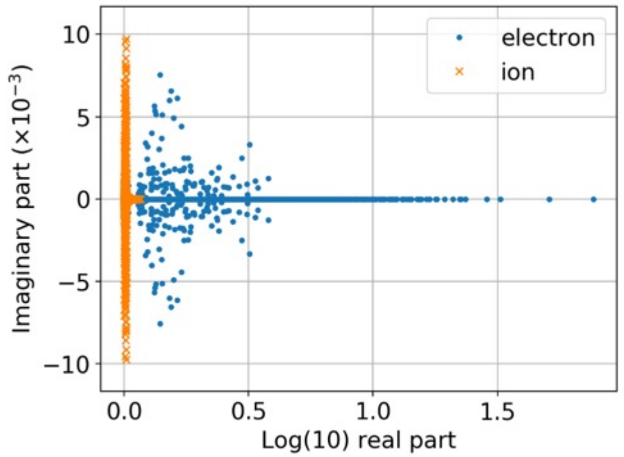
Source: https://xgc.pppl.gov/html/general_info.html


Wall
Edge
Core
Magnetic axis
Magnetic X-point
First coupled simulation of turbulence in a Tokamak device.
J. Dominski et al., Physics of Plasmas 25, 072308 (2018)
Visualization : Dave Pugmire (ORNL)



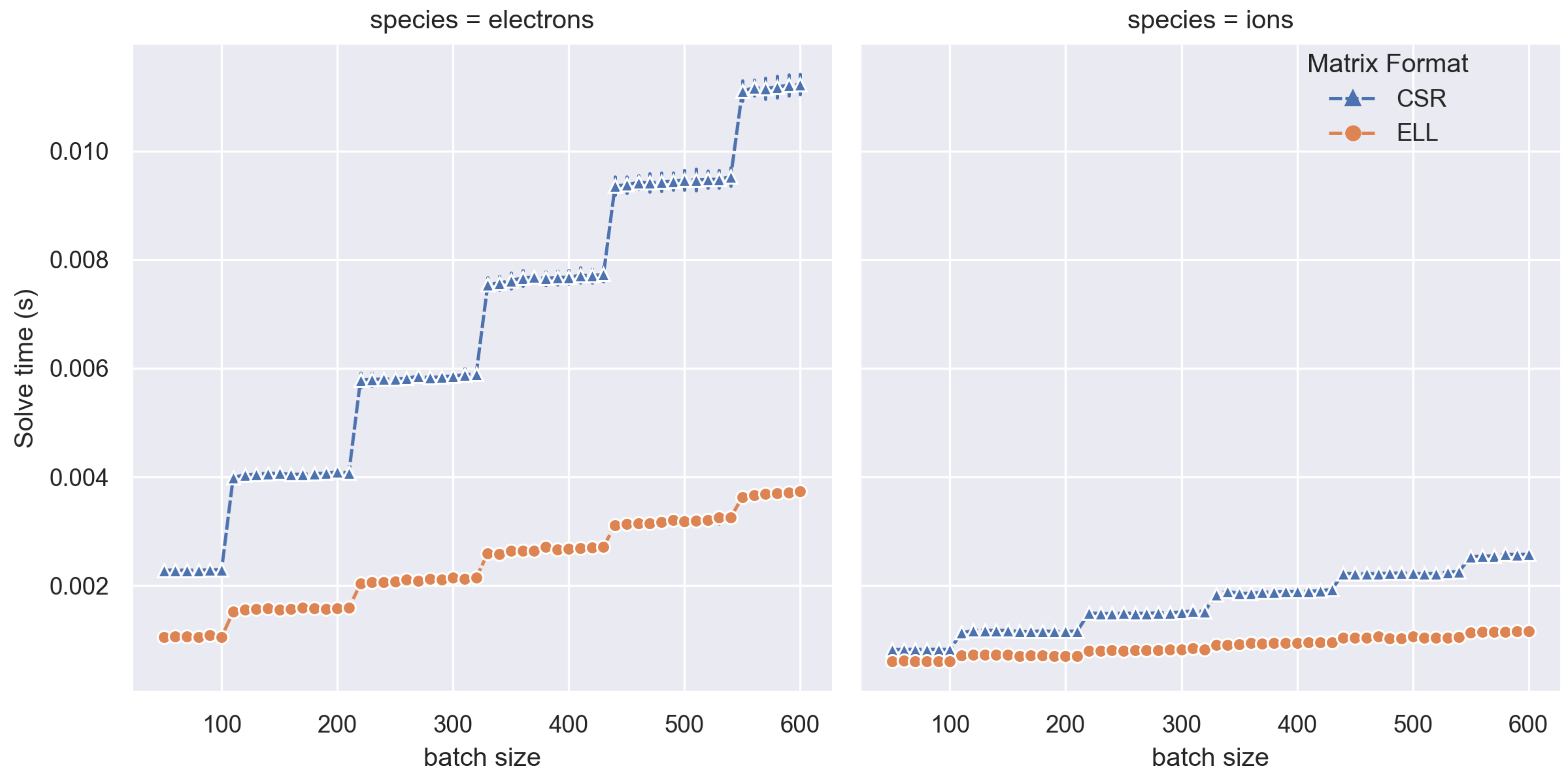


- Two species
- Ions easy to solve
- Electrons hard to solve
- Banded matrix structure
- Non-symmetric, need BiCGSTAB
- n = ~1,000
- nz = ~9,000

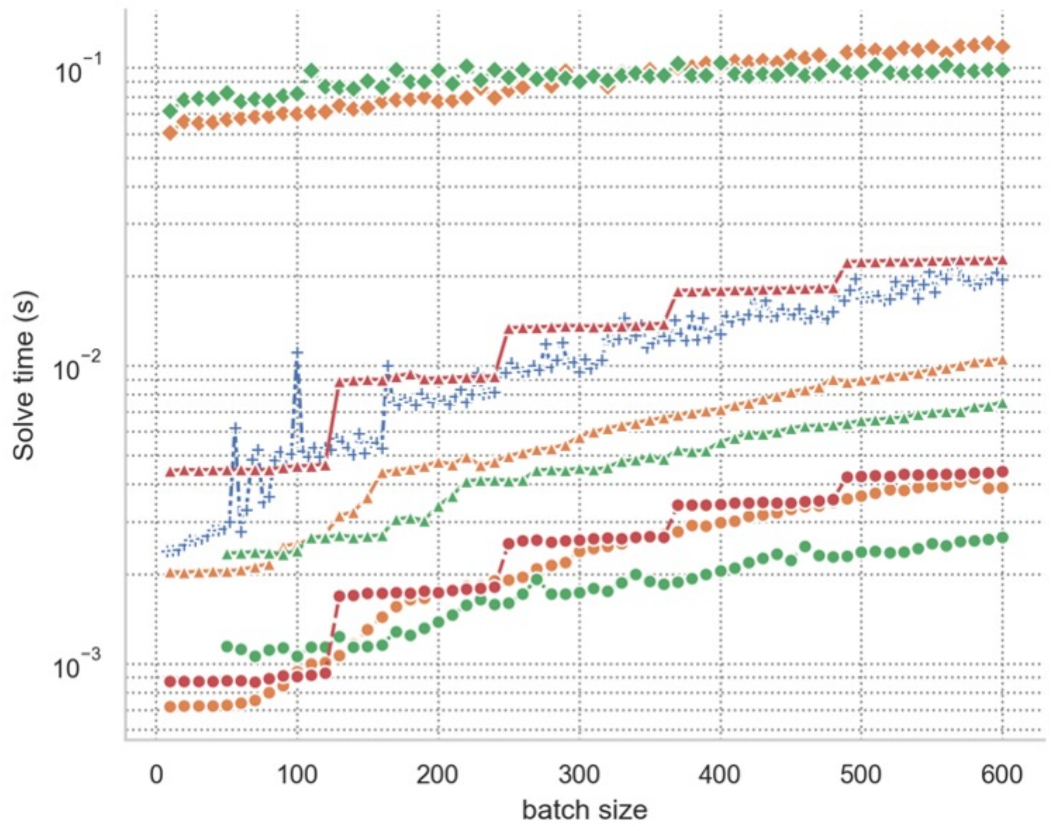


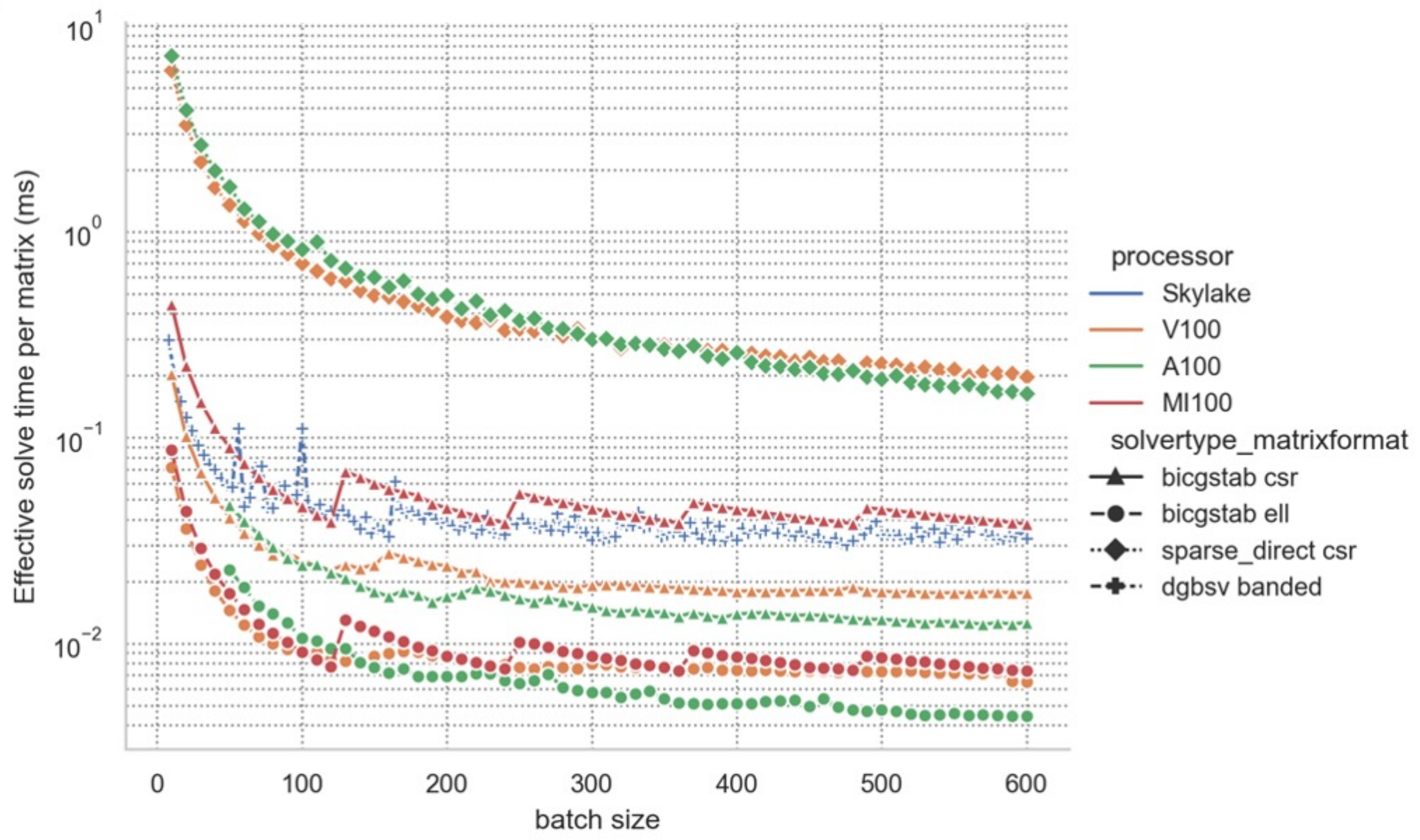
First experiences with Ginkgo's batched iterative solvers in XGC

NVIDIA A100 GPU



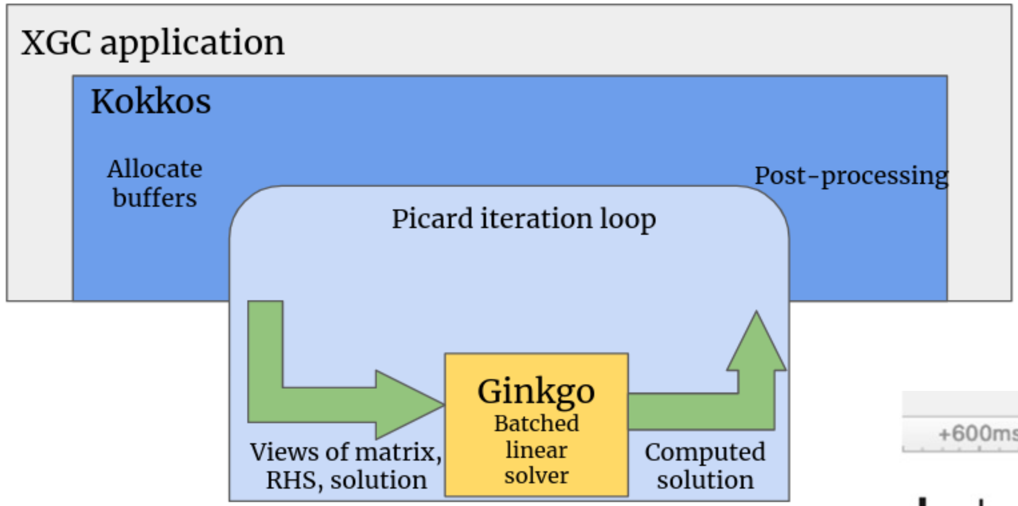
First experiences with Ginkgo’s batched iterative solvers in XGC

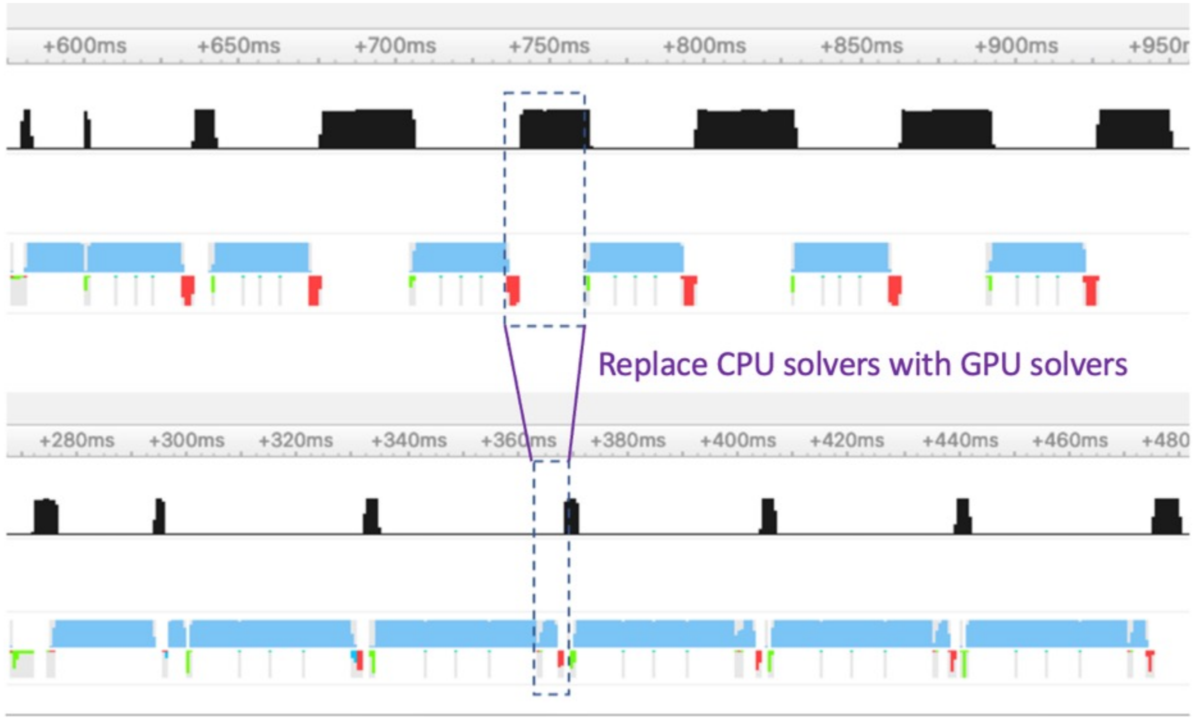




Aditya Kashi, Pratik Nayak, Dhruva Kulkarni, Aaron Scheinberg, Paul Lin, and Hartwig Anzt. **Batched sparse iterative solvers on gpu for the collision operator for fusion plasma simulations**. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 157–167. IEEE, 2022.

First experiences with Ginkgo's batched iterative solvers in XGC

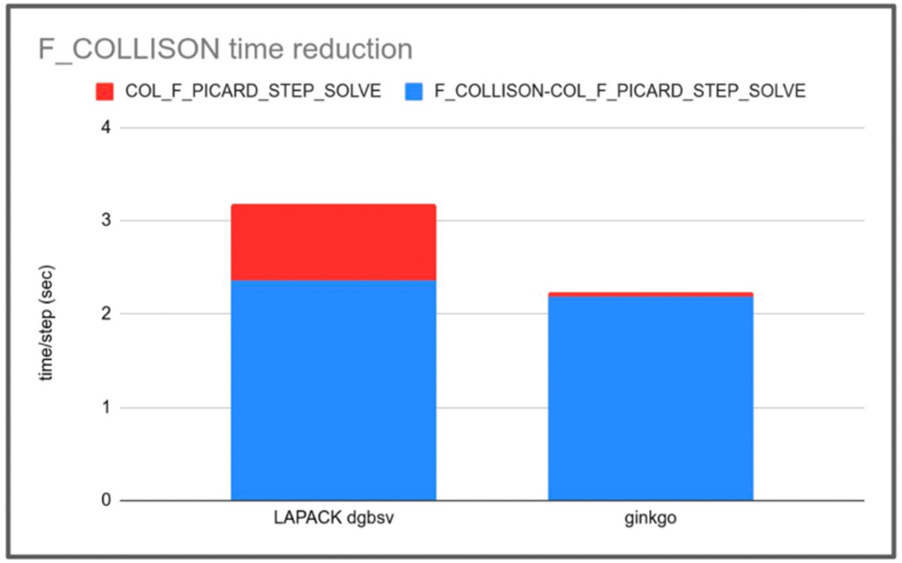




XGC collision operator solve LAPACK vs. Ginkgo: XGC pe459_d3d_EM_heatload test case

- XGC pe459_d3d_EM_heatload (Aaron's test case; used for Summit, Perlmutter and Crusher scaling studies)
- Preliminary study on 32 nodes of Perlmutter (128 A100s)
  - 2 poloidal planes (216k nodes per plane); 22.4M ptl/GPU, 89.6M ptl/node (ptl_num=700k)
  - Ran 20 time steps, collisions calculated every other time step

	per time step (s)	
	dgbsv	ginkgo
MAIN_LOOP	19.05	18.26
MAIN_LOOP-F_COLLISION	15.87	16.03
F_COLLISON	3.18	2.23
F_COLLISON-COL_F_PICARD_STEP_SOLVE	2.37	2.18
COL_F_PICARD_STEP_SOLVE	0.82	0.05
COL_F_SOLVER_CONVERT_BANDED	0.08	
COL_F_SOLVER_DGBSV	0.53	

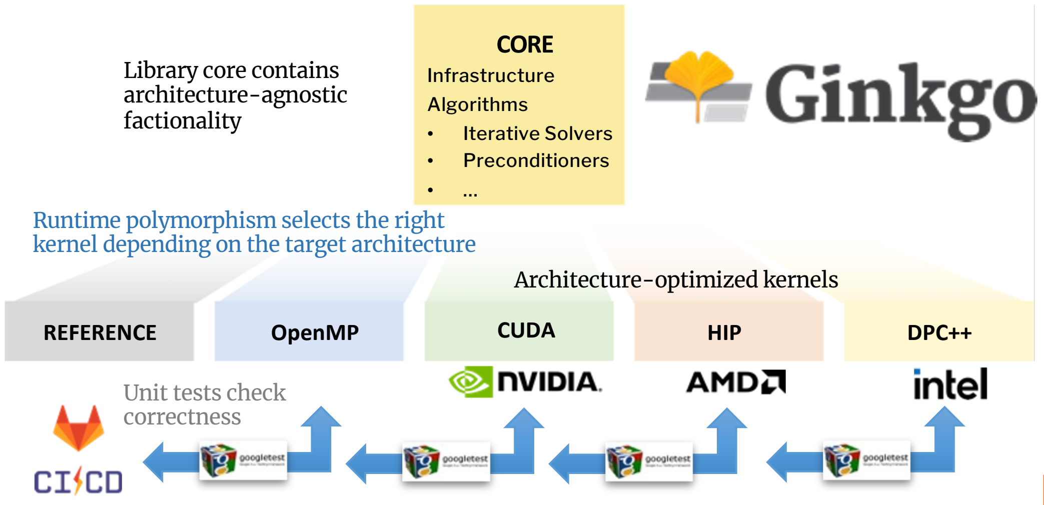


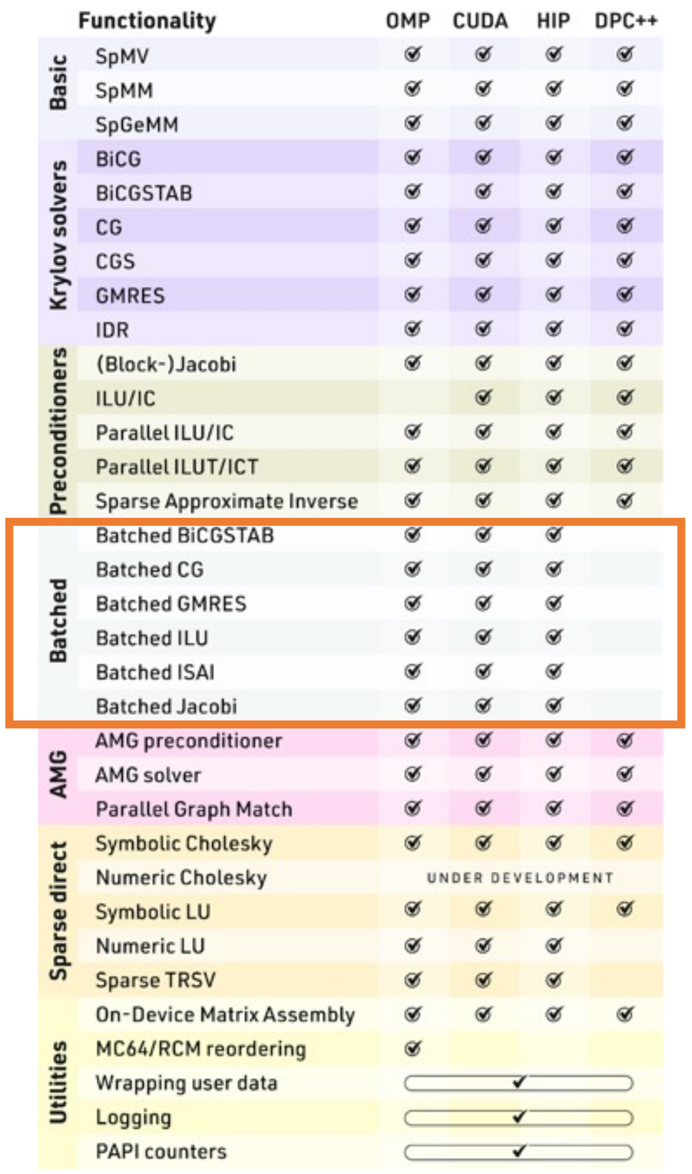
- With CPU LAPACK dgbsv
  - F_COLLISON is 17% of MAIN_LOOP time
  - COL_F_PICARD_STEP_SOLVE is 24% of F_COLLISON time and 4.3% of MAIN_LOOP time
  - COL_F_SOLVER_DGBSV is 66% of COL_F_PICARD_STEP_SOLVE
  - COL_F_SOLVER_CONVERT_BANDED is 10% of COL_F_PICARD_STEP_SOLVE
- Replacing CPU LAPACK dgbsv by GPU Ginkgo
  - COL_F_PICARD_STEP_SOLVE reduced from 0.82s to 0.046s per step; reduction of 94%
  - F_COLLISON reduced from 3.18s to 2.23s per step; reduction of 30%
  - MAIN_LOOP time reduced by 4.1%

Velocity grid: 33x39; matrices: 1287 rows

XGC collision operator solve performed on GPU

Status and open questions


Library core contains architecture-agnostic factionality
CORE
Infrastructure
Algorithms
• Iterative Solvers
• Preconditioners
• …
Ginkgo
Runtime polymorphism selects the right kernel depending on the target architecture
Architecture-optimized kernels
REFERENCE
OpenMP
CUDA
HIP
DPC++
NVIDIA
AMD
intel
Unit tests check correctness
CI/CD
googletest



	Functionality	OMP	CUDA	HIP	DPC++
Basic	SpMV	✓	✓	✓	✓
	SpMM	✓	✓	✓	✓
	SpGeMM	✓	✓	✓	✓
Krylov solvers	BiCG	✓	✓	✓	✓
	BiCGSTAB	✓	✓	✓	✓
	CG	✓	✓	✓	✓
	CGS	✓	✓	✓	✓
	GMRES	✓	✓	✓	✓
	IDR	✓	✓	✓	✓
Preconditioners	(Block-)Jacobi	✓	✓	✓	✓
	ILU/IC		✓	✓	✓
	Parallel ILU/IC	✓	✓	✓	✓
	Parallel ILUT/ICT	✓	✓	✓	✓
	Sparse Approximate Inverse	✓	✓	✓	✓
Batched	Batched BiCGSTAB	✓	✓	✓	
	Batched CG	✓	✓	✓	
	Batched GMRES	✓	✓	✓	
	Batched ILU	✓	✓	✓	
	Batched ISAI	✓	✓	✓	
	Batched Jacobi	✓	✓	✓	
AMG	AMG preconditioner	✓	✓	✓	✓
	AMG solver	✓	✓	✓	✓
	Parallel Graph Match	✓	✓	✓	✓
Sparse direct	Symbolic Cholesky	✓	✓	✓	✓
	Numeric Cholesky	UNDER DEVELOPMENT			
	Symbolic LU	✓	✓	✓	✓
	Numeric LU	✓	✓	✓	
	Sparse TRSV	✓	✓	✓	
Utilities	On-Device Matrix Assembly	✓	✓	✓	✓
	MC64/RCM reordering	✓			
	Wrapping user data	✓			
	Logging	✓			
	PAPI counters	✓			

- Individual system scheduling handled by GPU runtime.
  - How can we tell the runtime to schedule harder problems first?
  - How do we identify the harder problems?

- Extensions to monolithic problems by maximizing the cache usage and aiming to cache the matrix in the L2/L3 cache.
  - Has shown promise for medium size problems.