Georgia Tech College of Computing
School of Computer Science

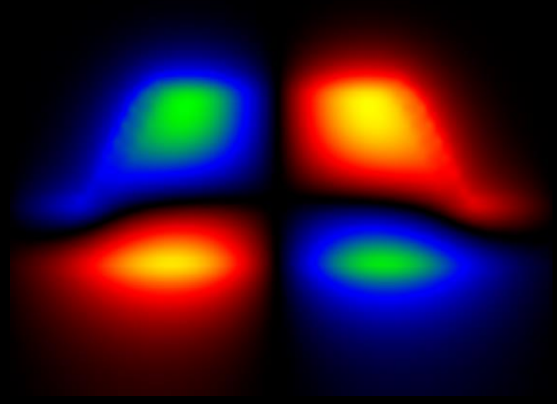
Georgia Tech College of Engineering
School of Electrical and Computer Engineering

GPU-Accelerated Electromagnetic Mode Solvers for High Performance Computing Applications

Joel Slaby

Conner Yurkon

John Moxley



April 25, 2023

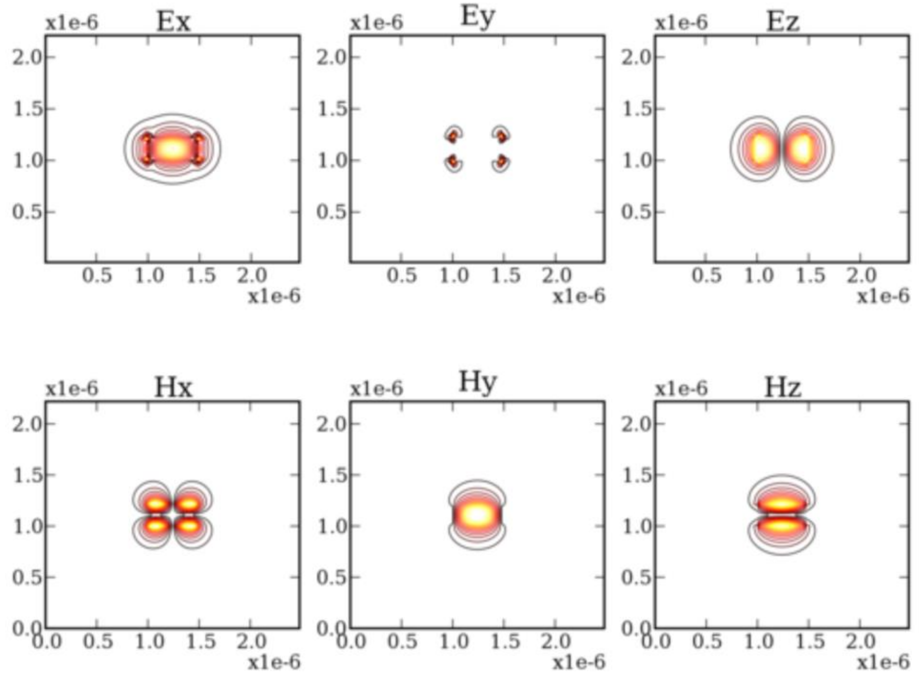
Dielectric Waveguide Mode Solver

- **Goal:** Find solution to Maxwell’s equations for a given geometry
  - Iterative eigenvalue problem with sparse matrices

$$(\mu^{-1}\nabla \times \epsilon^{-1}\nabla \times)\mathbf{H} = \omega^2\mathbf{H}$$

- **Current challenges:**
  - Finding eigenvalues *and* eigenvectors is an expensive calculation
  - Limited to smaller structures at lower resolutions
  - Speed-up can enable:
    - faster device design and optimizations
    - higher resolution calculations (more accurate)

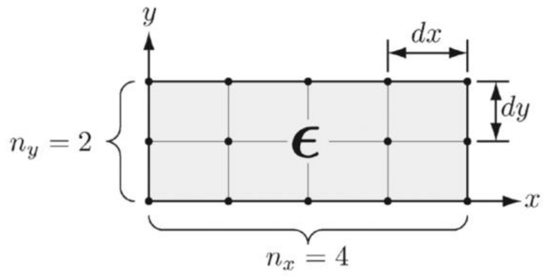
- **Existing work:**
  - A GPU Solver for Sparse Generalized Eigenvalue Problems With Symmetric Complex-Valued Matrices Obtained Using Higher-Order FEM: https://ieeexplore.ieee.org/iel7/6287639/8274985/08468163.pdf
  - This work uses the finite-element method (FEM), which isn't scalable to larger geometries. We will use the finite-difference method (FD)



https://ieeexplore.ieee.org/iel7/6287639/8274985/08468163.pdf

Mode Solver Problem Breakdown

(1) Define your geometry



(3) Solve eigenpair problem

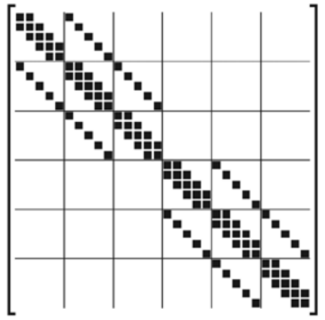
This is where HPC comes into play

$$\begin{bmatrix} A_{xx} & A_{xy} \\ A_{yx} & A_{yy} \end{bmatrix} \begin{bmatrix} H_x \\ H_y \end{bmatrix} = \beta^2 \begin{bmatrix} H_x \\ H_y \end{bmatrix}$$

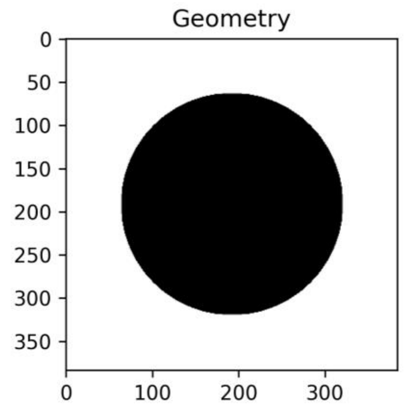
(2) Create the maxwell operator matrix $\boldsymbol{A}$

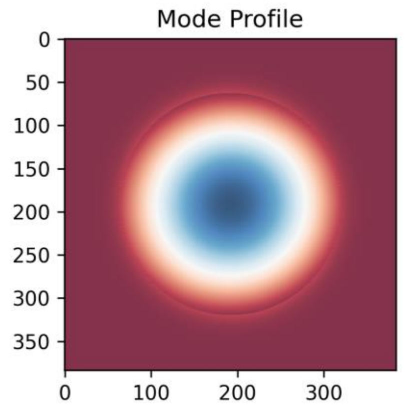
sparse matrix

$$\underbrace{(\mu^{-1}\nabla \times \epsilon^{-1}\nabla \times)}_{\boldsymbol{A} =}\mathbf{H} = \omega^2 \mathbf{H}$$



(4) Compute the other mode fields





A. B. Fallahkhair, K. S. Li and T. E. Murphy, "Vector Finite Difference Modesolver for Anisotropic Dielectric Waveguides", J. Lightwave Technol. 26(11), 1423-1431, (2008).

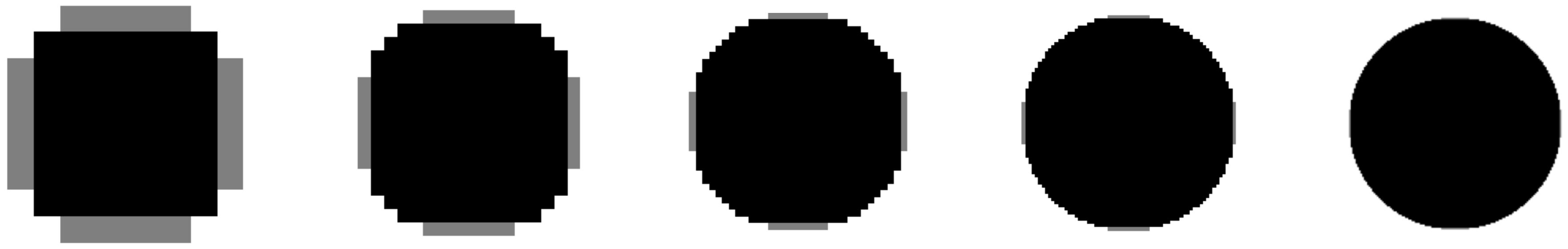
Performance Metrics

Accuracy Metrics

- Eigenvalue error vs. resolution
  - Ensure eigenvalues converge to correct value from baseline
- Visual normalized mode-field accuracy
  - Should also converge to baseline with higher resolution

Timing Metrics

- Total time at a given resolution
- Time per task
  - Task 1: Build matrix A
  - Task 2: Solve eigenpair problem
  - Task 3: Compute other mode fields
- Total time vs. resolution
  - Subsequently calculate GPU speed-up

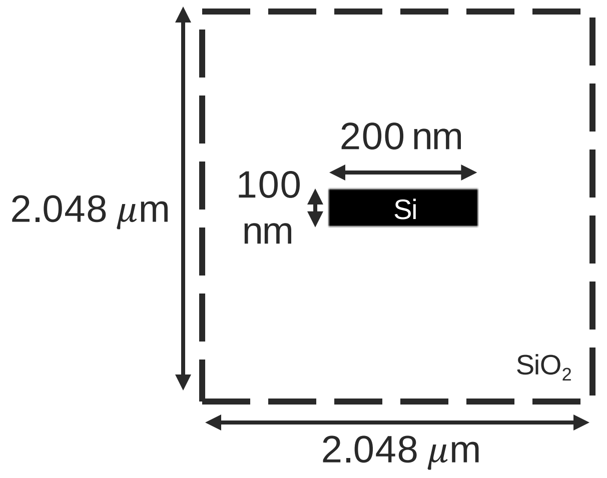


Baseline Testing

- **EmPy:** Open-source, Python-based, fully vectorial finite difference mode solver
  - Implements algorithm found in "Vector Finite Difference Modesolver for Anisotropic Dielectric Waveguides"
  - Uses *scipy.sparse.linalg.eigs*, a wrapper to the ARPACK SNEUPD, DNEUPD, CNEUPD, ZNEUPD, functions which use the Implicitly Restarted Arnoldi Method to compute eigenvalues and eigenvectors
  - Sparse matrices stored in compressed sparse row (CSR) format

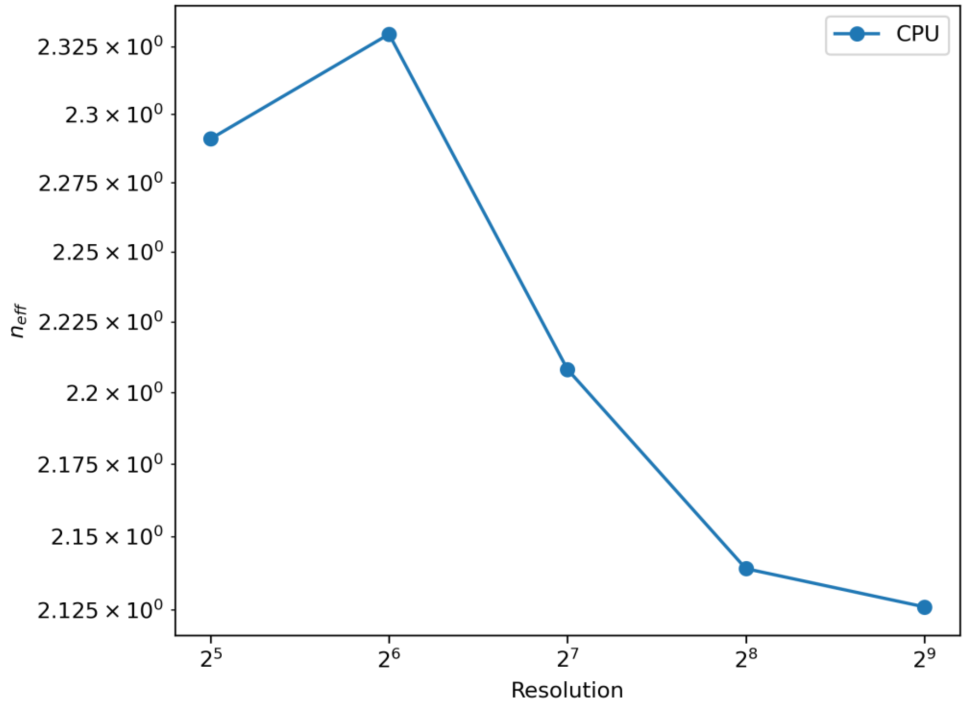
- Test case: Standard silicon rectangular waveguide
  - **Resolution:** 512 pixels
  - **Spatial Pixel Size:** 4nm/pixel
  - **Eigenvector Matrix Size:** 512 x 512 (262,144 elements)

Rectangular Waveguide



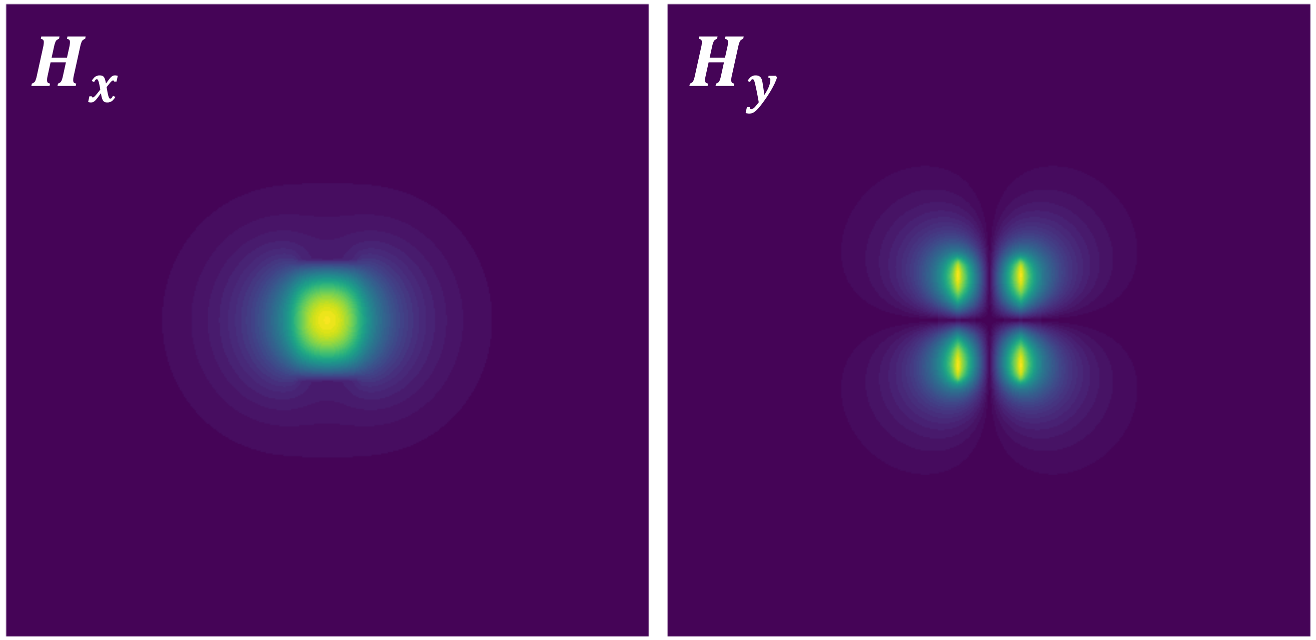
Baseline Testing: Accuracy

n_{eff} Convergence
(Eigenvalue)

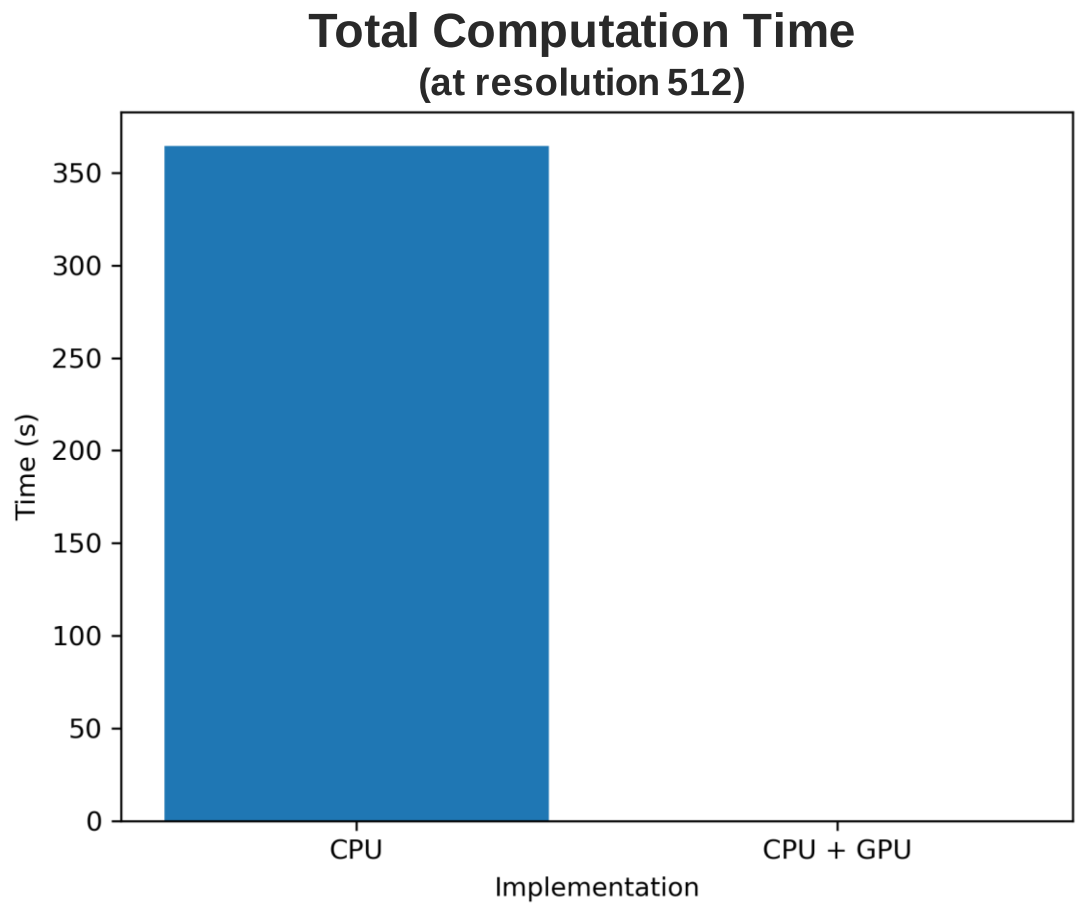


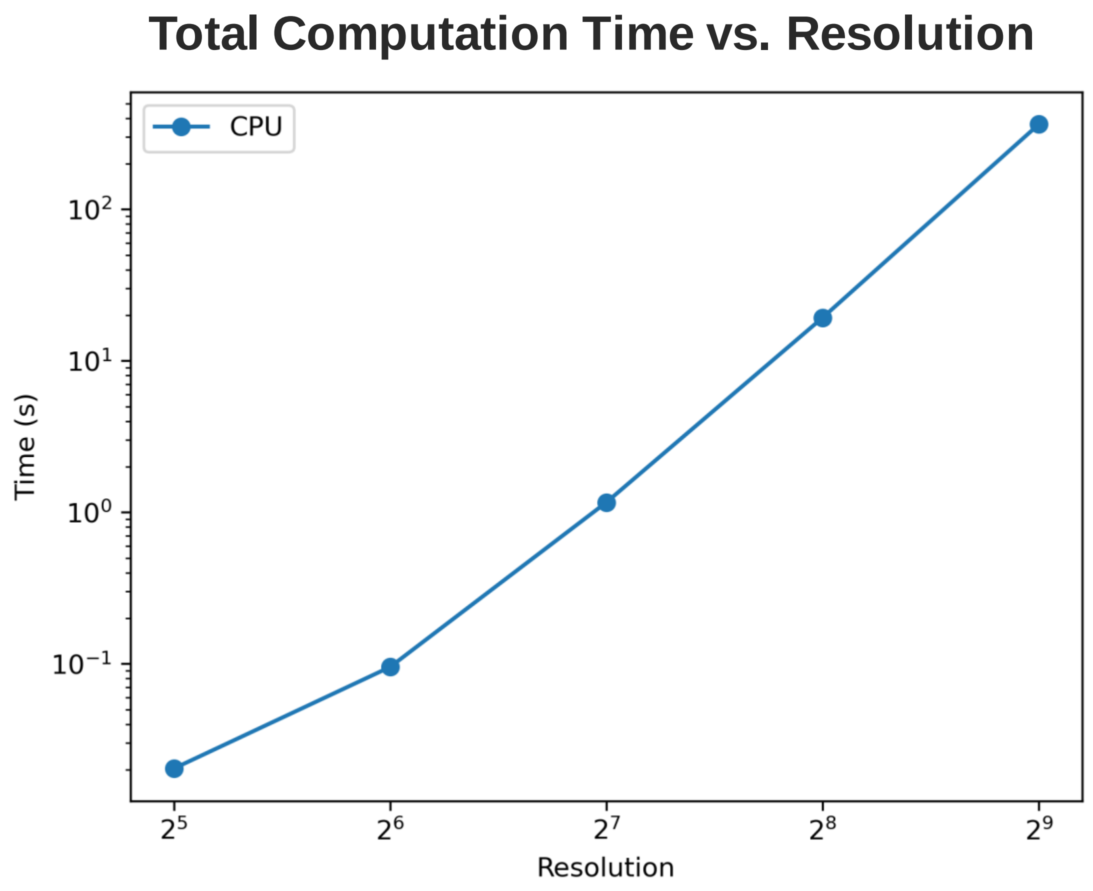
$$n_{eff} = \frac{\lambda\sqrt{\beta}}{2\pi}$$

Mode Fields
(Eigenvectors)



Baseline Testing: Timing

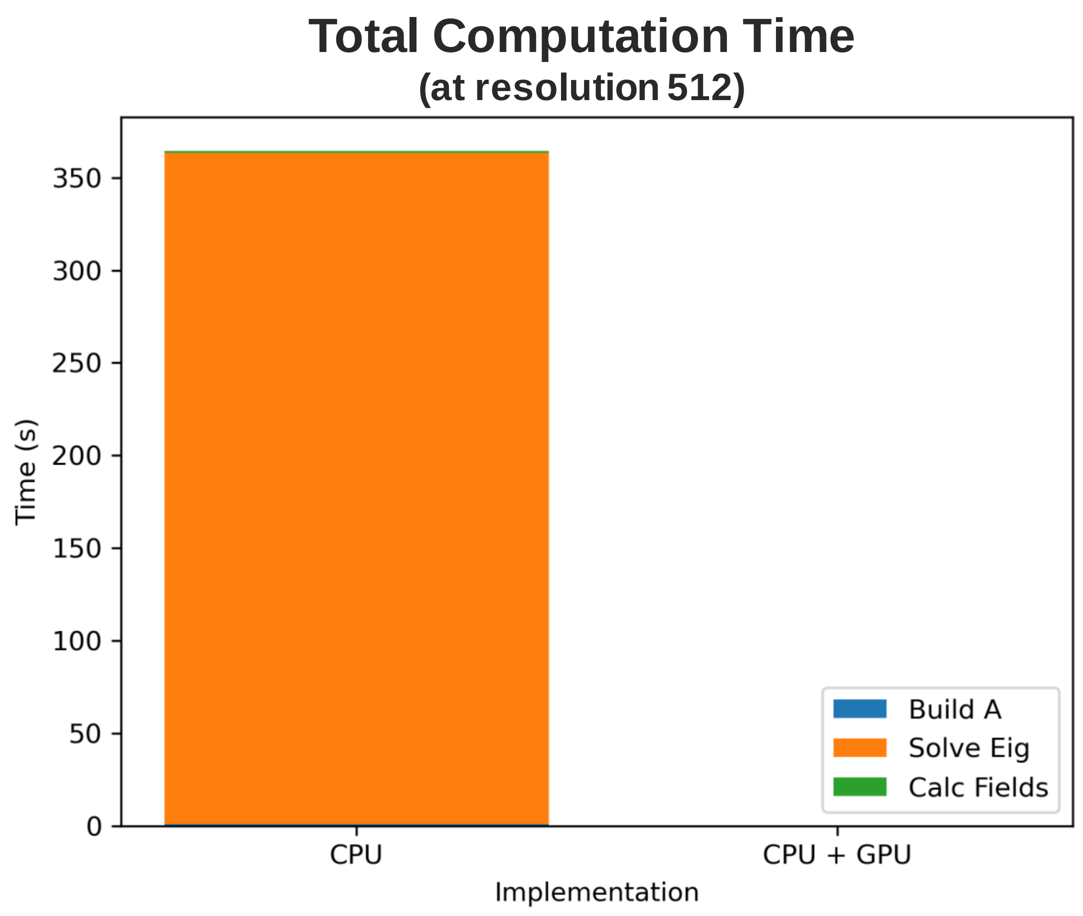


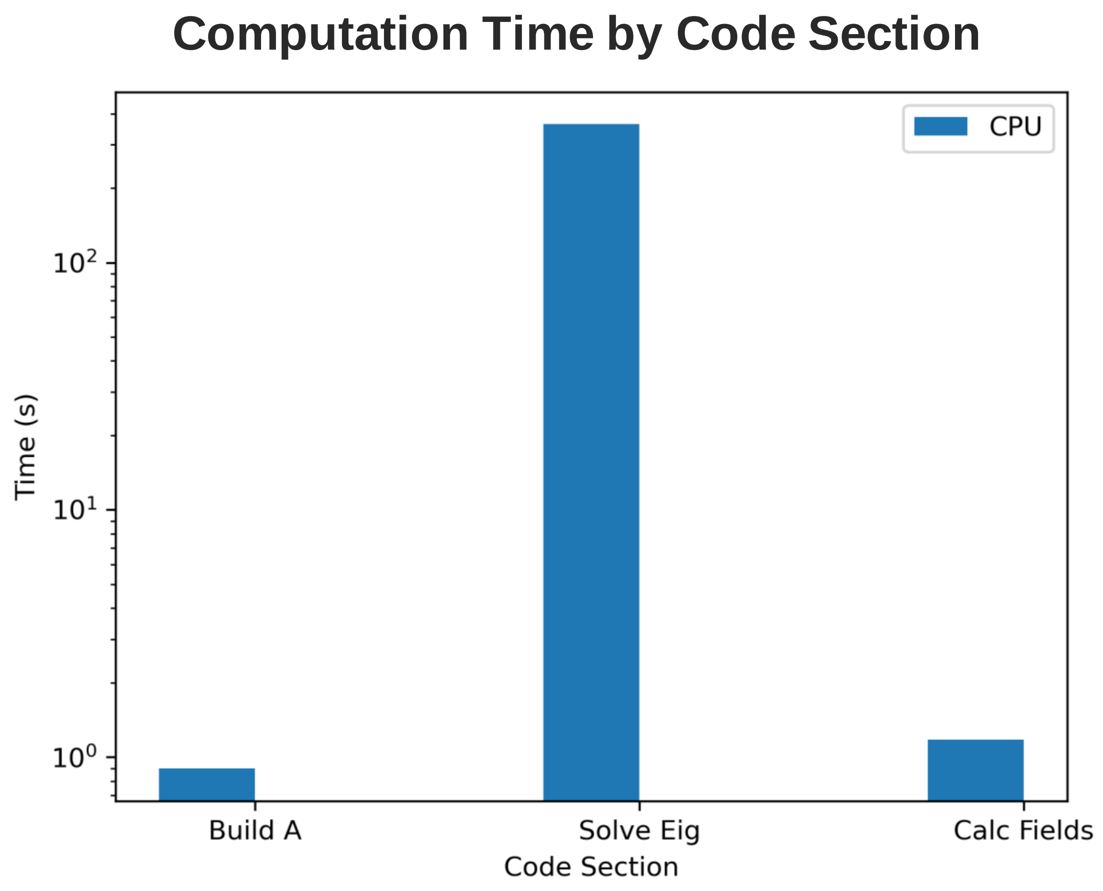


Georgia Tech Terabit Optical Networking Center

Baseline Testing: Timing

Georgia Institute of Technology





Solution: GPU Accelerated Mode Solver

Georgia Institute of Technology

- **Goal:** Accelerate a presently implemented mode solver on a GPU

- **Design:**
  - Start with existing solution (serial), open-source such as EmPy
  - Convert the eigenmode problem to be GPU compatible
    - Choose an appropriate algorithm: Arnoldi iteration and QR Algorithm
  - Validate accuracy and determine GPU speed-up

- **Challenges we faced:**
  - The matrices we're working with are *very* large (dimensions up to 500,000) which makes it difficult to apply simpler algorithms. Everything must work be tailored to sparse matrices (copies, computation, etc.)
  - The sped-up version isn't always as accurate as the serial version. Balancing speed with accuracy is an important trade-off

How do we solve an eigenvalue problem?

- Many methods!
- Which one do we want?
  - Dependent on matrix form and which eigenvalues you need

Iterative Eigenvalue Algorithms

Power Iteration
Rayleigh Quotient Iteration
Inverse Iteration
Preconditioned Inverse Iteration
Bisection Method
Laguerre Iteration
QR Algorithm
Jacobi Eigenvalue Algorithm
Divide-and-conquer
Homotopy Method
Folded Spectrum Method
MRRR Algorithm

Dense or Sparse?

Symmetric (Hermitian)?

Smallest or Largest Eigenvalues?

Eigenvalues or Eigenpairs?

Real or Complex?

Parallelizable?

How do we solve an eigenvalue problem?

- Many methods!
- Which one do we want?
  - Dependent on matrix form and which eigenvalues you need

Iterative Eigenvalue Algorithms

Power Iteration
Rayleigh Quotient Iteration
Inverse Iteration
Preconditioned Inverse Iteration
Bisection Method
Laguerre Iteration
QR Algorithm
Jacobi Eigenvalue Algorithm
Divide-and-conquer
Homotopy Method
Folded Spectrum Method
MRRR Algorithm

Dense or **Sparse**?

Symmetric (Hermitian)?

Smallest or **Largest Eigenvalues**?

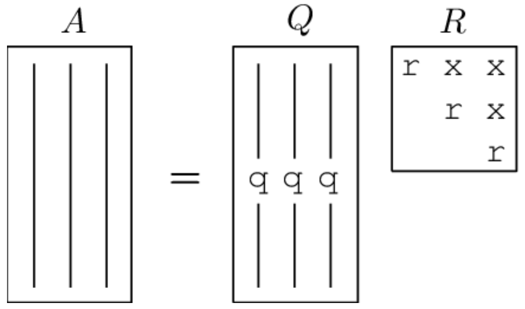
Eigenvalues or **Eigenpairs**?

Real or Complex?

Parallelizable?

QR Algorithm

- Iterative method based on the QR decomposition, which factors a matrix A into the product of two matrices Q and R
  - Q is orthonormal (with the property that $Q^T = Q^{-1}$) and R is upper triangular



- The iteration is straightforward:

$$\boxed{A = Q_1 R_1, \qquad R_k Q_k = A_{k+1} = Q_{k+1} R_{k+1}, \qquad k = 1, 2, 3, \ldots}$$

  - All A_k are similar and so share the same eigenvalues. Proof: $A_{k+1} = R_k Q_k = Q_k^{-1} Q_k R_k Q_k = Q_k^{-1} A_k Q_k = Q_k^T A_k Q_k$.
  - $A_k \to U$, an upper triangular matrix called the Schur form of A, whose diagonal entries are the eigenvalues of A.
- To compute the associated eigenvectors, we simultaneously apply this iteration:

$$\boxed{S_1 = Q_1, \qquad S_k = S_{k-1} Q_k = Q_1 Q_2 \cdots Q_{k-1} Q_k, \quad k > 1}$$

  - $S_k \to S$, a matrix whose columns contain the orthonormal eigenvector basis of A

Addressing Sparsity

- **Goal:** Convert a large, sparse matrix A to small, dense matrix H ("condensed form"), where the eigenvalues of H are a subset of the (approximate) eigenvalues of A
  - The QR Algorithm as described has complexity $O(n^4)$, making it impractical for our input sizes
  - We will show that first forming H and then applying the QR Algorithm to H is much less expensive

- **Method:** Arnoldi Iteration
  - Projects A onto a subspace of the original vector space, producing the desired matrix H
  - More precisely, this algorithm produces a sequence of orthonormal vectors (called Arnoldi vectors) that span the order-r Krylov subspace $K_r(A,b)$. H can be interpreted as the representation in the basis of Arnoldi vectors of the orthogonal projection of A onto $K_r(A,b)$

$$\mathcal{K}_r(A,b) = \operatorname{span}\{b, Ab, A^2b, \ldots, A^{r-1}b\}$$

- **Benefit:** Avoids *expensive* matrix-matrix operations
  - Instead, we rely on a series of matrix-vector operations
  - The algorithm uses the modified Gram-Schmidt procedure, a method of orthonormalizing a set of vectors in an inner product space
  - We choose r empirically, but we always have that $r \ll n$, and so the QR complexity becomes $O(r^3)$ (we also lose a factor of r because of the special form of H)

Arnoldi Iteration

- Produces an upper Hessenberg matrix H and an orthonormal matrix Q_n (containing the Arnoldi vectors) such that $A = Q_n H_n Q_n^*$
  - A matrix is upper Hessenberg if it is square and has zero entries below the first subdiagonal:

$$\mathbf{H} = \begin{pmatrix} * & * & \cdots & * & * \\ * & * & \cdots & * & * \\ 0 & * & \cdots & * & * \\ 0 & 0 & * & \cdots & * \\ & & \vdots & & \\ 0 & \cdots & 0 & * & * \end{pmatrix}$$

Upper Hessenberg Structure

- The eigenvalues of H are called the Ritz eigenvalues, and they typically converge to the *largest* eigenvalues of A first

```
$b =$ arbitrary, $q_1 = b/\|b\|$
for $n = 1, 2, 3, \ldots$
    $v = Aq_n$
    for $j = 1$ to $n$
        $h_{jn} = q_j^* v$
        $v = v - h_{jn} q_j$
    $h_{n+1,n} = \|v\|$
    $q_{n+1} = v / h_{n+1,n}$
```

Arnoldi Iteration

Eigenvectors from Arnoldi Iteration

Georgia Institute of Technology

- An eigenvector of A can be formed by multiplying:
  - Q_n (from the Arnoldi iteration)
  - An eigenvector v of H_n (computed from the QR algorithm)
- Proof:

$$H_n v = \lambda v \qquad (v \text{ is an eigenvector})$$

$$H_n Q_n^* Q_n v = \lambda v \qquad (\text{insert } Q_n^* Q_n = 1)$$

$$Q_n H_n Q_n^* Q_n v = \lambda Q_n v \qquad (\text{left multiply by } Q_n)$$

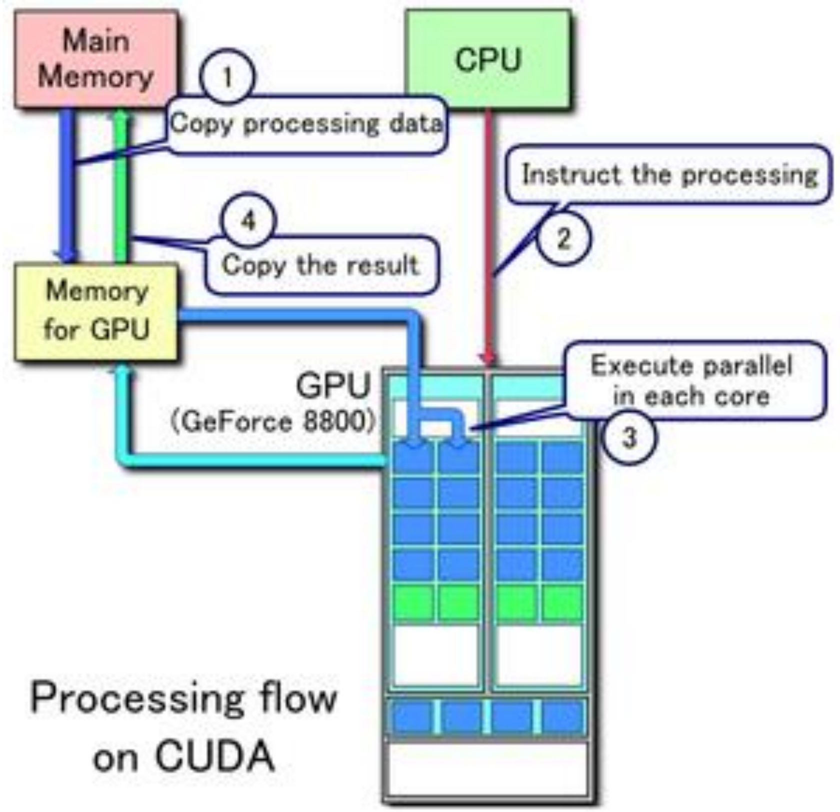
$$\mathrm{A} Q_n v = \lambda Q_n v \qquad (A = Q_n H_n Q_n^*)$$

$$\mathrm{A}(Q_n v) = \lambda (Q_n v) \qquad (Q_n v \text{ is an eigenvector of } A)$$

Georgia Tech Terabit Optical Networking Center

Parallelization: CuPy

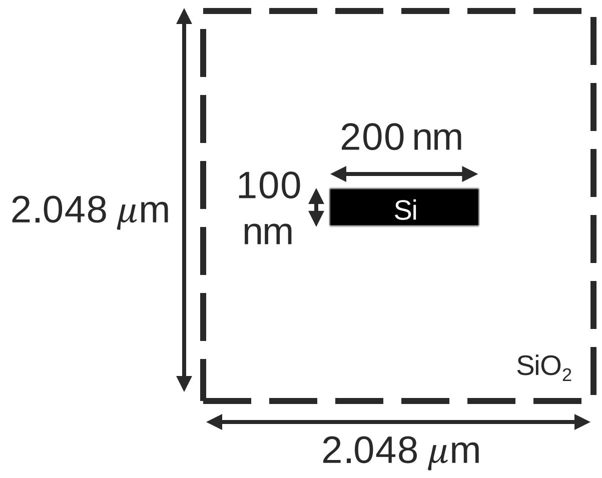
- Drop-in replacement to run existing NumPy/SciPy code using NVIDIA CUDA
- Access to low-level CUDA features straight from Python, making it ideal for extending EmPy
- Utilizes CUDA Toolkit libraries including cuBLAS, cuRAND, cuSOLVER, cuSPARSE, cuFFT, cuDNN and NCCL to make full use of the GPU architecture
- These libraries include routines for the matrix operations we employ (e.g. QR decomposition) on sparse matrices



Datasets / Testbed

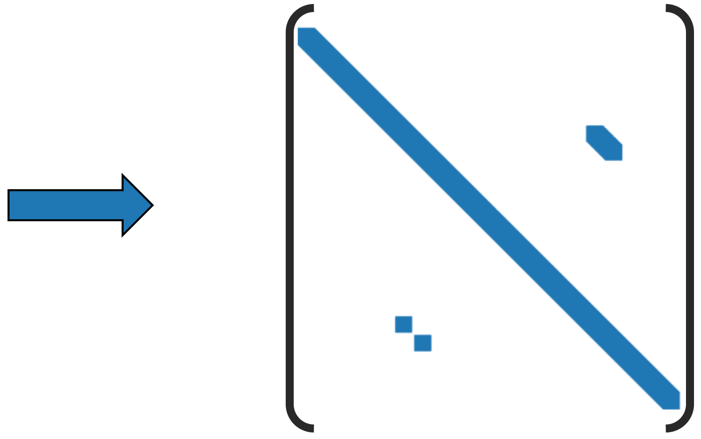
- **Dataset:** Standard silicon rectangular waveguide
  - Resolutions scaled from 32 to 512 (by factors of 2)

Rectangular Waveguide



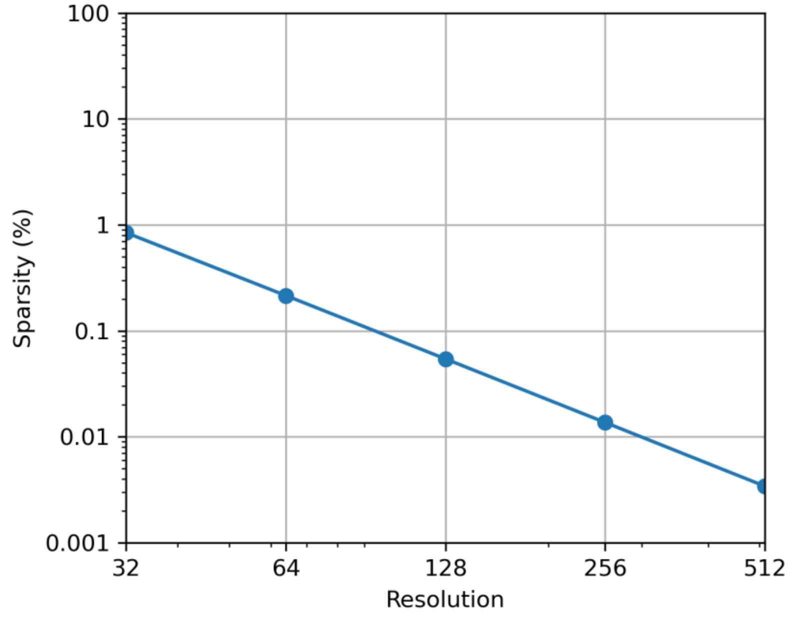
Resolution: 512
Spatial Pixel Size: 4nm/pixel
Eigenvector Matrix Size: 512 x 512 (262,144 elements)

Structure of Generated *A*



Matrix Size: $\sim 5 \cdot 10^5 \times 5 \cdot 10^5$
Number of nonzeros (nnz): $\sim 10^7$
Sparsity: ~ 0.003 %

Matrix Sparsity

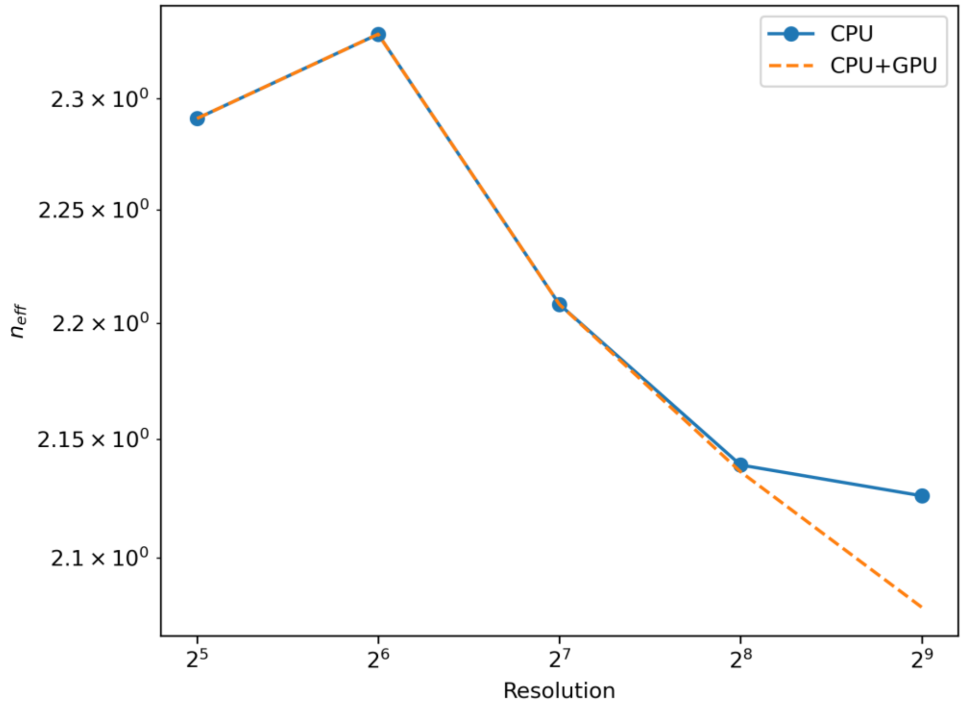


- **Test Bed:**
  - PACE Cluster (coc-ice-gpu)
  - Tesla v100 GPU
  - Conda environment enabled with development version of EmPy

Validation Testing: Accuracy

Georgia Institute of Technology

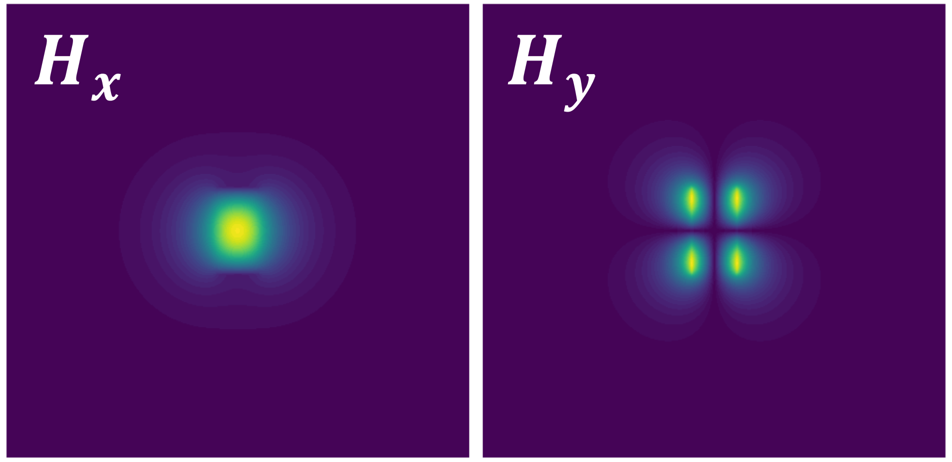
n_{eff} Convergence
(Eigenvalue)



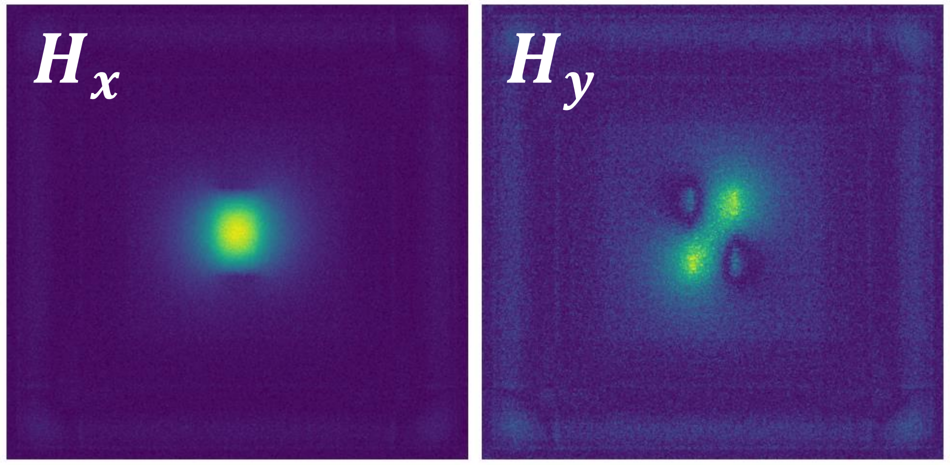
$$n_{eff} = \frac{\lambda\sqrt{\beta}}{2\pi}$$

Mode Fields
(Eigenvectors)

CPU
(baseline)



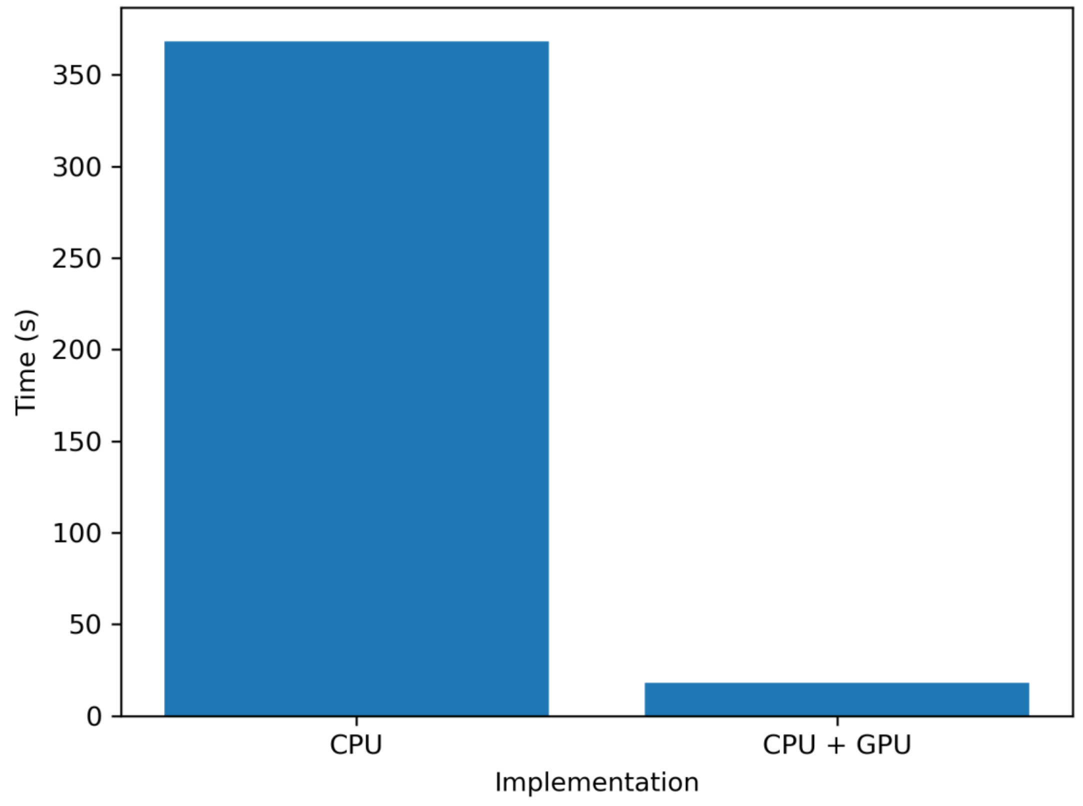
CPU + GPU
(validation)



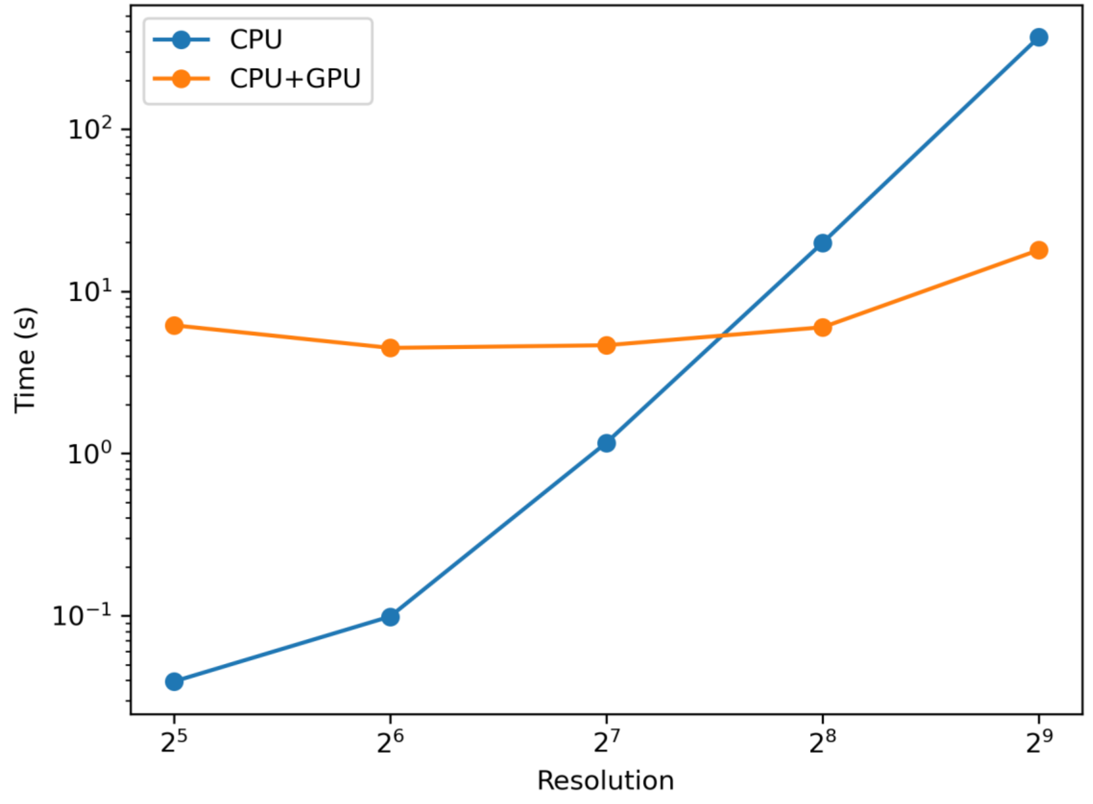
Performance Testing: Timing

Georgia Institute of Technology

Total Computation Time
(at resolution 512)

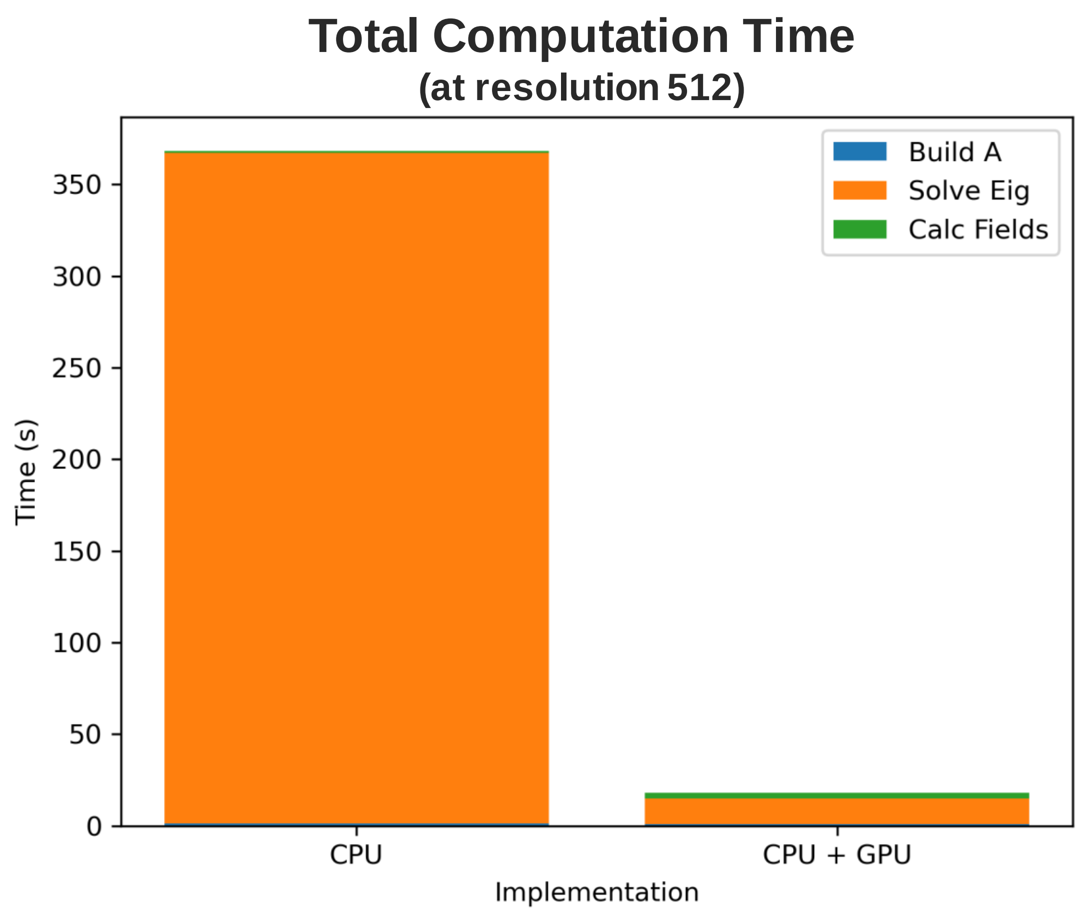


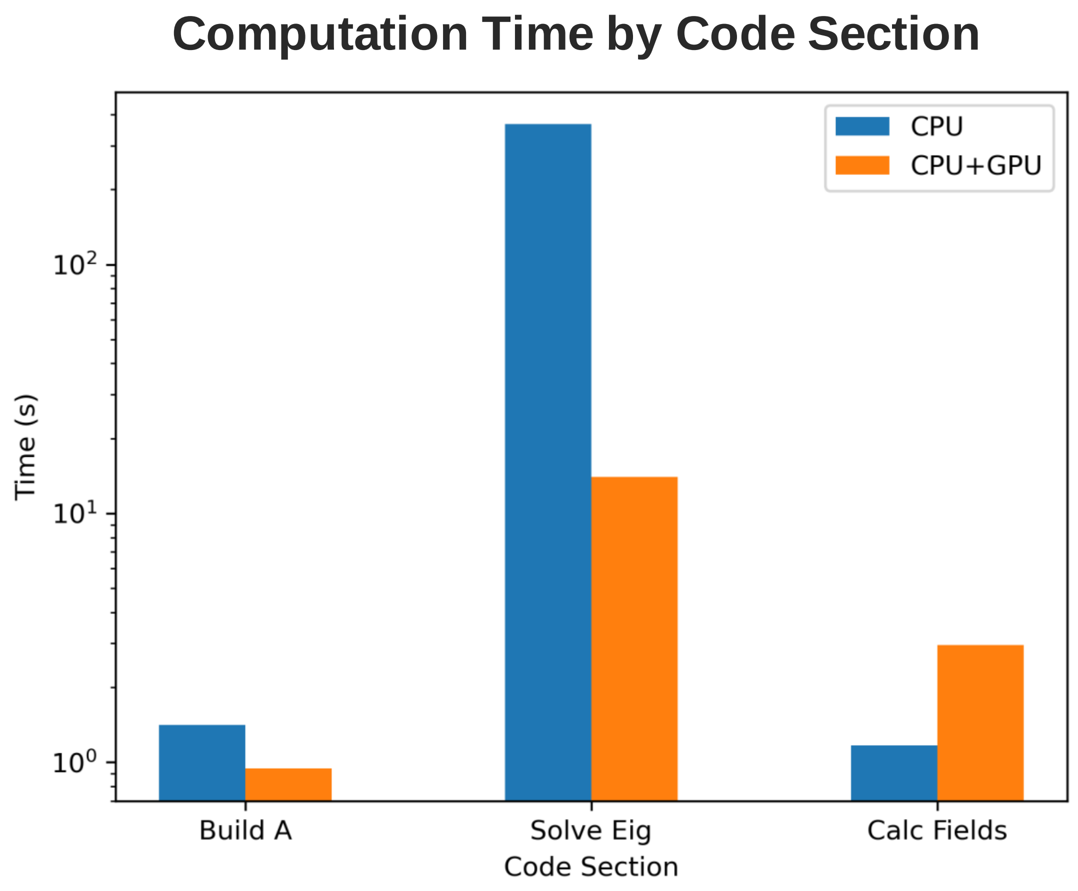
Total Computation Time vs. Resolution



Performance Testing: Timing

Georgia Institute of Technology





Conclusion and Future Work

- **Implemented a GPU-accelerated eigenmode solver**
  - Utilized Arnoldi iteration for sparse matrices
  - Completed QR iteration to converge eigenvalues and eigenvectors simultaneously
  - GPU-accelerated implementation enabled by CuPy
  - At a resolution of 512, observed a speedup of 18.8x

- Challenges
  - Our implementation was faster at higher resolutions, but less accurate
    - Requires more sophisticated eigensolver algorithm
  - Large noise present in the eigenvectors
    - Likely due to initialized vector for linear solve in Arnoldi iteration
    - Causes denser eigenvectors and longer computation time for *Task 3: compute other fields*
    - Attempted to optimize by initializing with ones, created artifacts in other parts of the eigenvector

- Future Work
  - Implement more robust eigenvalue solver using implicitly restarted Arnoldi iteration and a QR algorithm using shifts and deflation
  - Cohesively include our work in open-source library EmPy

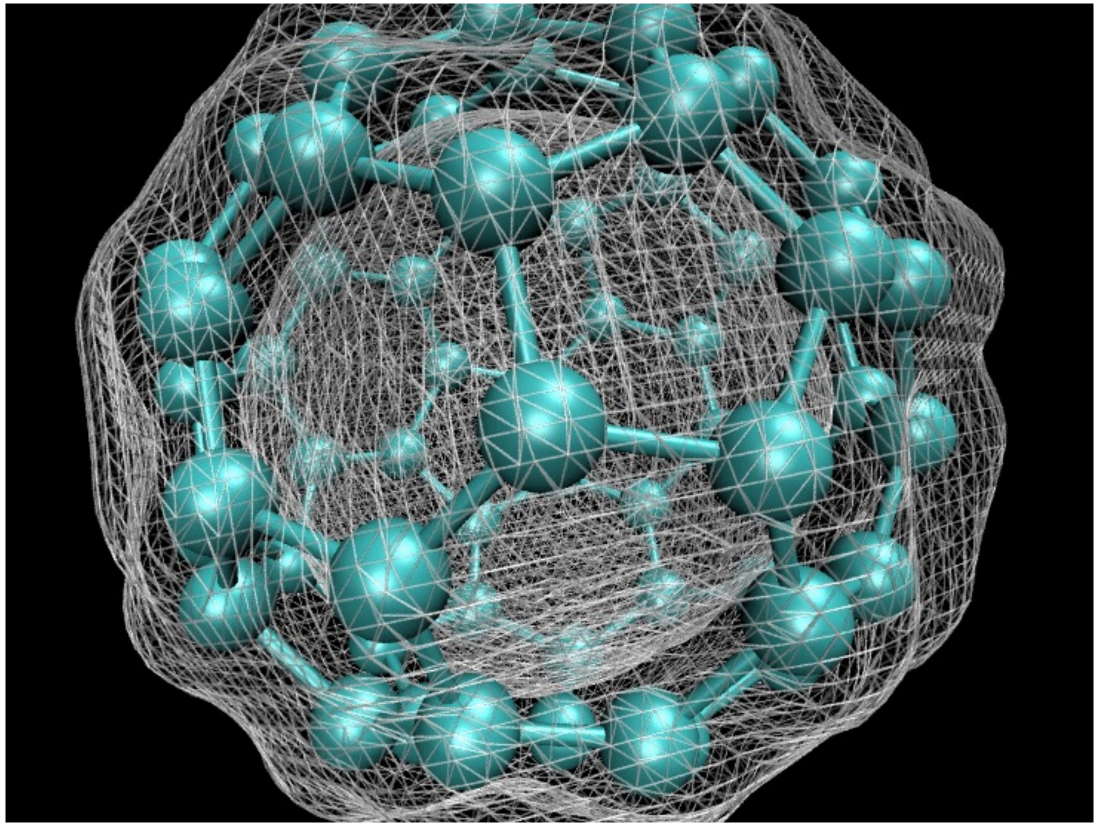
CSE 6230 Project Presentation: Performance Analysis of Proxy-apps for Computational Chemistry Methods

Shehan Parmar, Austin Wallace, Blair Johnson

Spring 2023

Project Category & Problem Definition

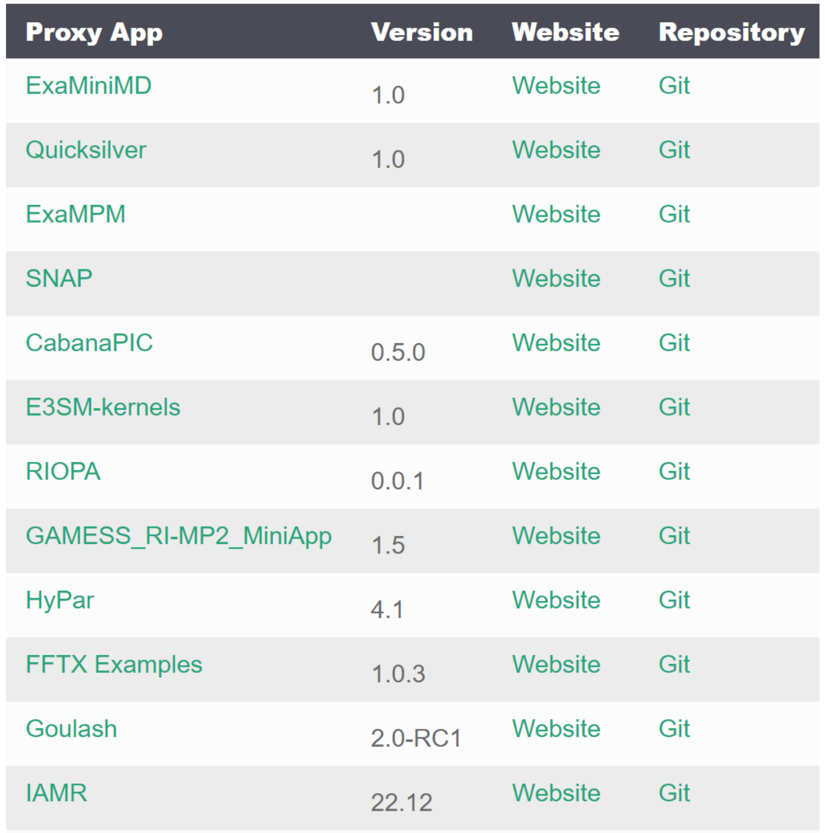
- **Category**: Scientific Application & Reproducibility
- **Problem Definition**:
  - Density Functional Theory (DFT) is used to investigate properties of molecular systems.
  - DFT relies on solving the Kohn-Sham equations, which require computationally expensive orthogonalization of large matrices.
  - Proxy apps can be used to reduce development workload and yet draw conclusions on code performance on heterogeneous architectures.



https://en.wikipedia.org/wiki/Density_functional_theory#/media/File:C60_isosurface.png

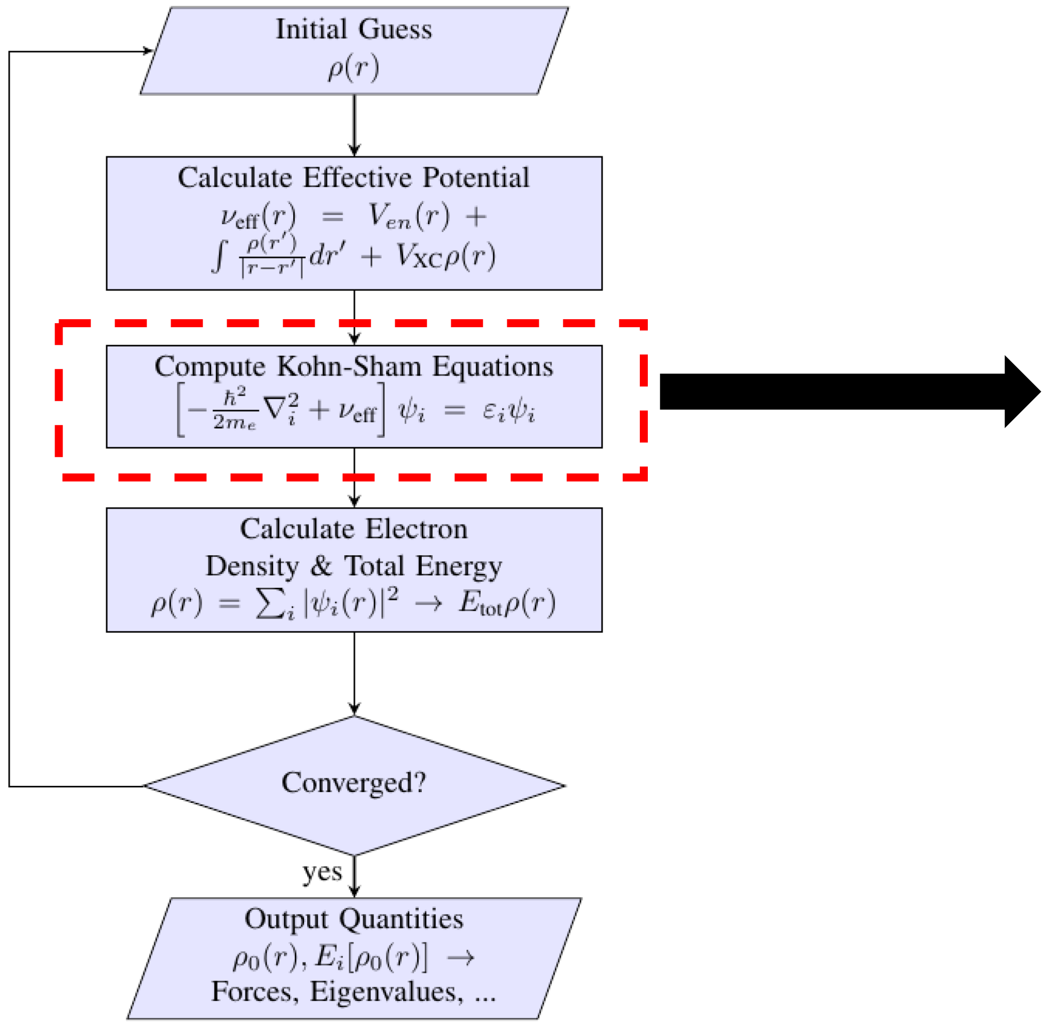
What are proxy apps?

- Proxy apps reduce the problem to essential components to understand performance-critical aspects of an algorithm.
- In this work, we will employ the **Löwdin orthonormalization of a tall-skinny matrix** as a proxy app for solving the KS equations.
- Other possible proxy-apps: https://proxyapps.exascaleproject.org/app/



Proxy App	Version	Website	Repository
ExaMiniMD	1.0	Website	Git
Quicksilver	1.0	Website	Git
ExaMPM		Website	Git
SNAP		Website	Git
CabanaPIC	0.5.0	Website	Git
E3SM-kernels	1.0	Website	Git
RIOPA	0.0.1	Website	Git
GAMESS_RI-MP2_MiniApp	1.5	Website	Git
HyPar	4.1	Website	Git
FFTX Examples	1.0.3	Website	Git
Goulash	2.0-RC1	Website	Git
IAMR	22.12	Website	Git

Procedure for Solving Kohn-Sham Equations



Initial Guess $\rho(r)$
Calculate Effective Potential $\nu_{\text{eff}}(r) = V_{en}(r) + \int \frac{\rho(r')}{|r-r'|} dr' + V_{XC}\rho(r)$
Compute Kohn-Sham Equations $\left[-\frac{\hbar^2}{2m_e}\nabla_i^2 + \nu_{\text{eff}}\right]\psi_i = \varepsilon_i \psi_i$
Calculate Electron Density & Total Energy $\rho(r) = \sum_i |\psi_i(r)|^2 \rightarrow E_{\text{tot}}\rho(r)$
Converged?
yes
Output Quantities $\rho_0(r), E_i[\rho_0(r)] \rightarrow$ Forces, Eigenvalues, ...

1. Compute the Gram matrix $S = A^T A$
2. Compute $C = S^{1/2}$
3. Update A: $A_{new} = AC$

Majid, M.F.; Mohd Zaid, H.F.; Kait, C.F.; Ahmad, A.; Jumbri, K. Ionic Liquid@Metal-Organic Framework as a Solid Electrolyte in a Lithium-Ion Battery: Current Performance and Perspective at Molecular Level. *Nanomaterials* **2022**, *12*, 1076. https://doi.org/10.3390/nano12071076

Georgia Tech

Goals

1. Reproduce and benchmark method from [1] on GT clusters
2. Introduce and benchmark mixed precision scheme

Parallel Computing 100 (2020) 102703

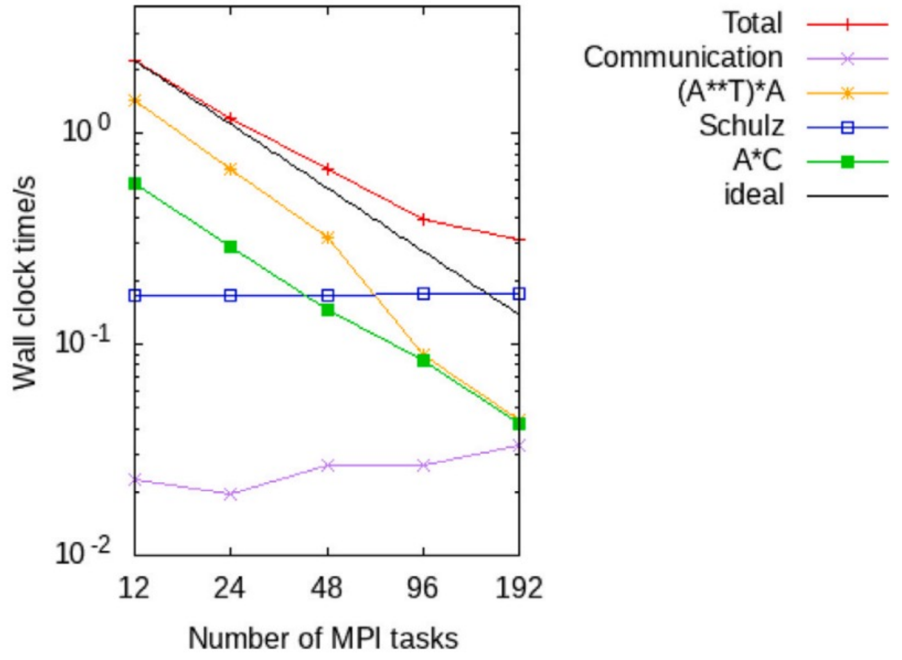


Fig. 7. Strong scaling of the Löwdin orthonormalization for a 3,000,000×3000 tall and skinny matrix.

Strong scaling benchmark method [1]

[1] M. Lupo Pasini, B. Turcksin, W. Ge, and J.-L. Fattebert, "A parallel strategy for density functional theory computations on accelerated nodes," *Parallel Computing*, vol. 100, p. 102703, Dec. 2020, doi: 10.1016/j.parco.2020.102703.

Georgia Tech

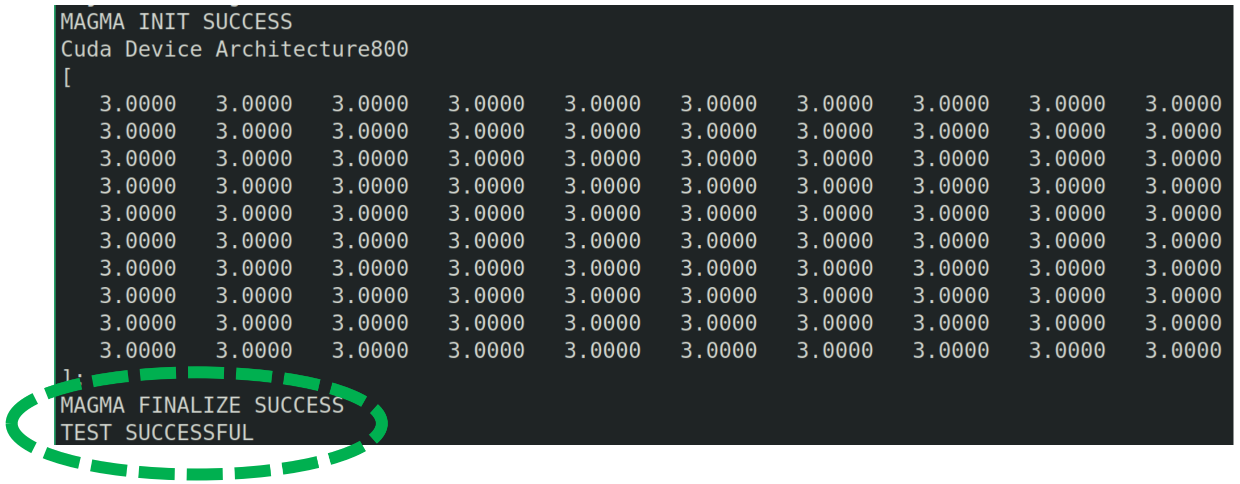
Datasets & Obtained Performance Metrics

- INPUTS:
  1. Testbeds
     - Hive
     - COC ICE
     - ICE HAMMER
  2. Matrix Sizes
     - Tall Skinny Matrices
       - 3,000,000 x 300
       - 3,000,000 x 3,000 did not fit into memory on our Hive test bed
     - N X N Square Matrices
       - N = 1092, 2096, 3140, 4146, 5156, 6210, 7210, 8256, 9260, 10304
  3. Gaussian Wavefunctions standard deviations, σ
     - $\sigma = 0.25, 0.5, 0.8$

- METRICS:
  1. Time breakdown
     - Total
     - AllReduce
     - Host to Device (H to D)
     - Memory Initialization (Mem Init)
     - Schulz Iterations
     - $A^T A$
  2. No. of Iterations to Convergence

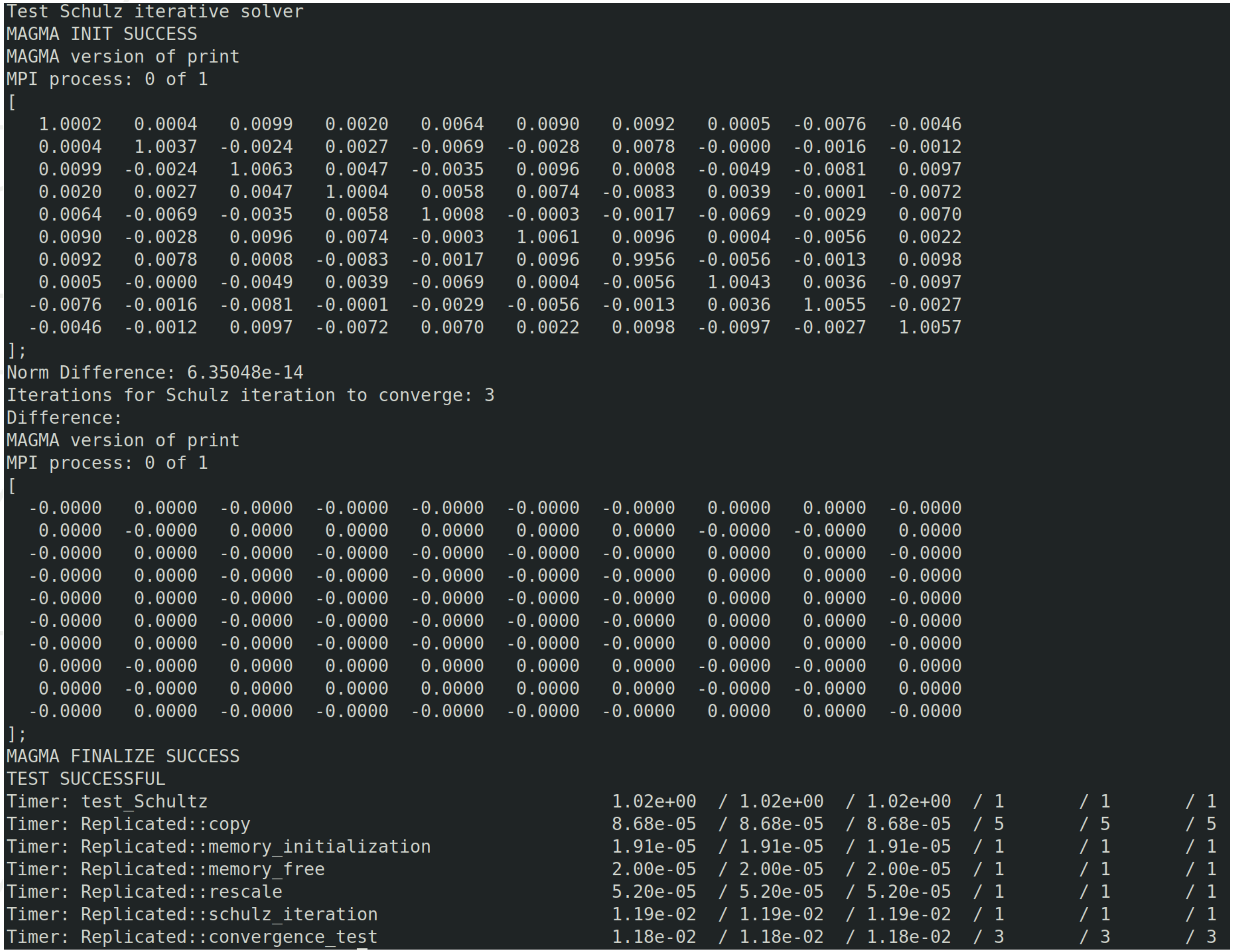
Validation – Tests

- Main program broken down into 9 tests:
  - ✓testAllreduce
  - ✓testMAGMA
  - ✓testMatMul
  - ✓testMaxNormReplicated
  - ✓testMPI
  - ✓testOrtho
  - ✓testSchulz
  - ✓testSchulzSingle

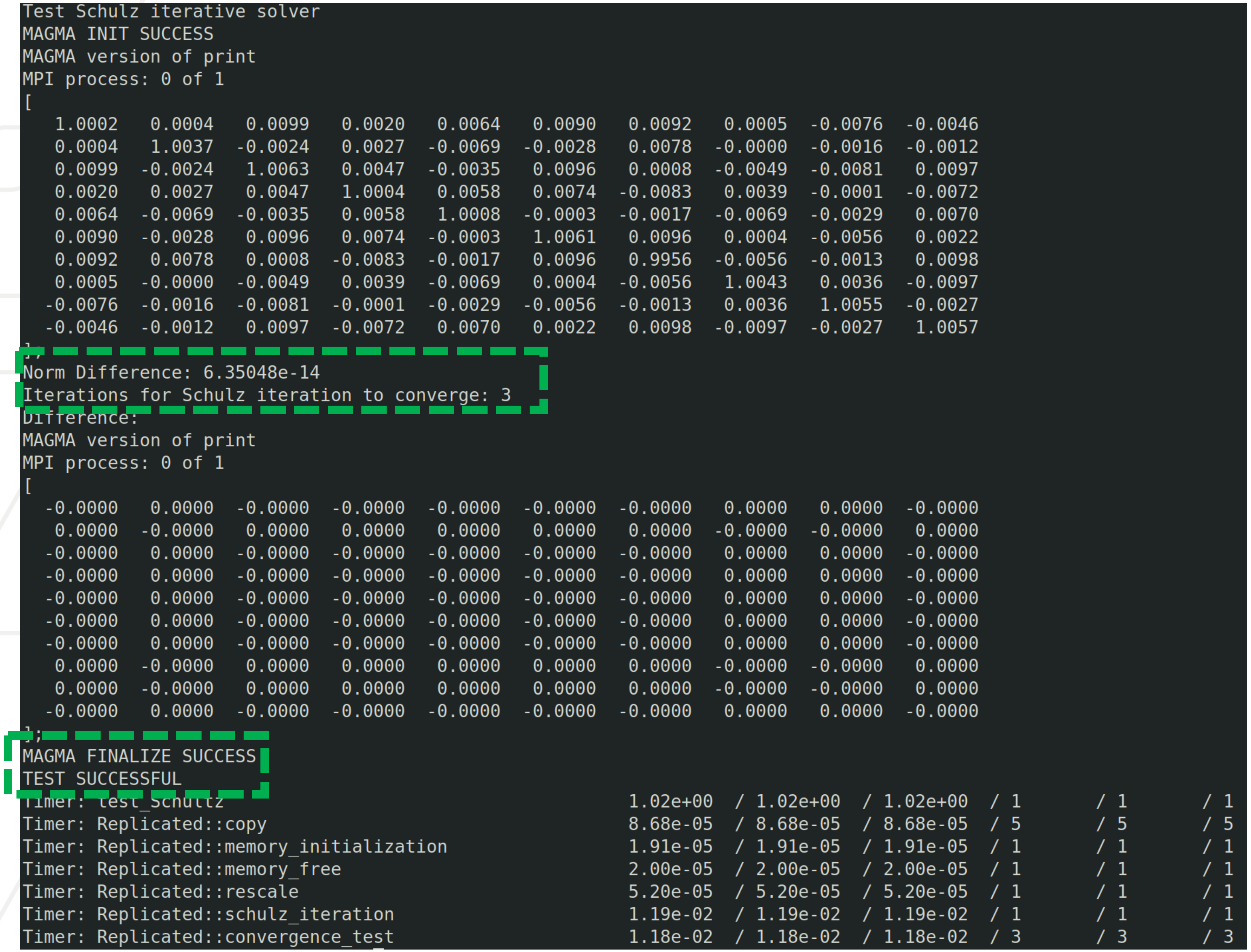


```
MAGMA INIT SUCCESS
Cuda Device Architecture800
[
   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000
   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000
   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000
   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000
   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000
   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000
   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000
   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000
   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000
   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000   3.0000
];
MAGMA FINALIZE SUCCESS
TEST SUCCESSFUL
```

Georgia Tech

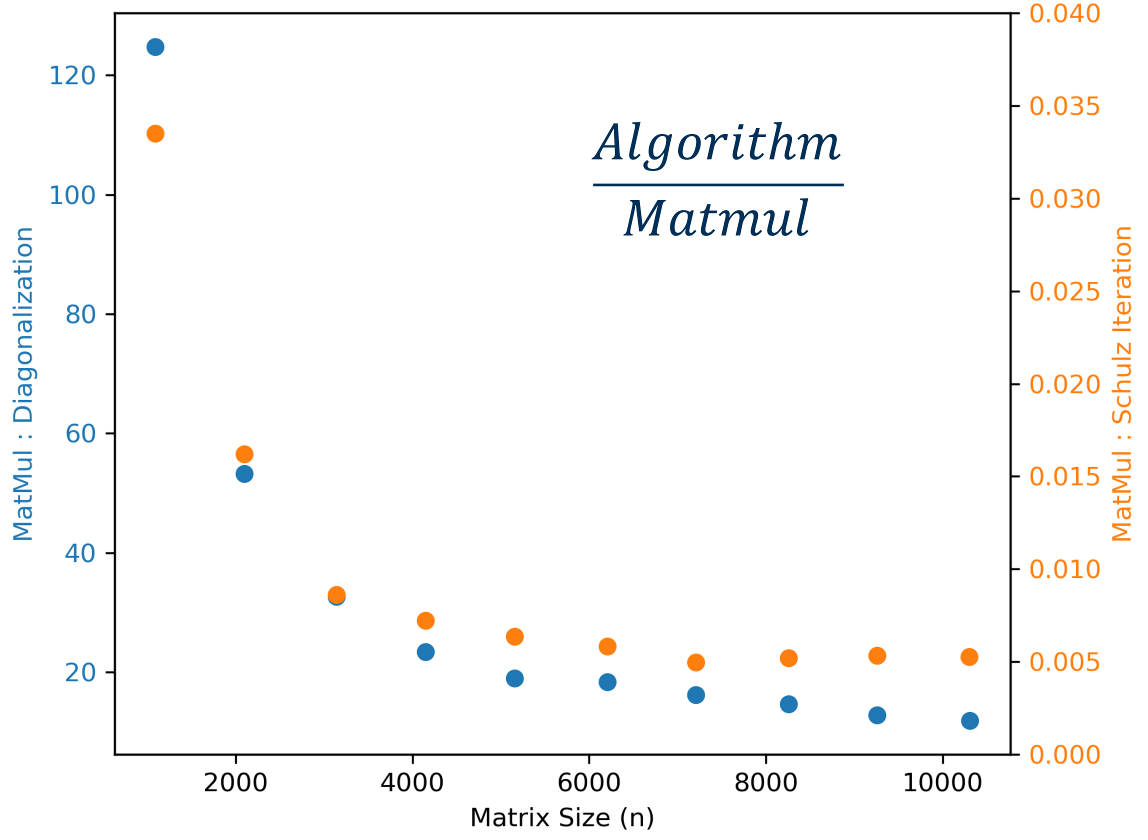


```
Test Schulz iterative solver
MAGMA INIT SUCCESS
MAGMA version of print
MPI process: 0 of 1
[
   1.0002   0.0004   0.0099   0.0020   0.0064   0.0090   0.0092   0.0005  -0.0076  -0.0046
   0.0004   1.0037  -0.0024   0.0027  -0.0069  -0.0028   0.0078  -0.0000  -0.0016  -0.0012
   0.0099  -0.0024   1.0063   0.0047  -0.0035   0.0096   0.0008  -0.0049  -0.0081   0.0097
   0.0020   0.0027   0.0047   1.0004   0.0058   0.0074  -0.0083   0.0039  -0.0001  -0.0072
   0.0064  -0.0069  -0.0035   0.0058   1.0008  -0.0003  -0.0017  -0.0069  -0.0029   0.0070
   0.0090  -0.0028   0.0096   0.0074  -0.0003   1.0061   0.0096   0.0004  -0.0056   0.0022
   0.0092   0.0078   0.0008  -0.0083  -0.0017   0.0096   0.9956  -0.0056  -0.0013   0.0098
   0.0005  -0.0000  -0.0049   0.0039  -0.0069   0.0004  -0.0056   1.0043   0.0036  -0.0097
  -0.0076  -0.0016  -0.0081  -0.0001  -0.0029  -0.0056  -0.0013   0.0036   1.0055  -0.0027
  -0.0046  -0.0012   0.0097  -0.0072   0.0070   0.0022   0.0098  -0.0097  -0.0027   1.0057
];
Norm Difference: 6.35048e-14
Iterations for Schulz iteration to converge: 3
Difference:
MAGMA version of print
MPI process: 0 of 1
[
  -0.0000   0.0000  -0.0000  -0.0000  -0.0000  -0.0000  -0.0000   0.0000   0.0000  -0.0000
   0.0000  -0.0000   0.0000   0.0000   0.0000   0.0000   0.0000  -0.0000  -0.0000   0.0000
  -0.0000   0.0000  -0.0000  -0.0000  -0.0000  -0.0000  -0.0000   0.0000   0.0000  -0.0000
  -0.0000   0.0000  -0.0000  -0.0000  -0.0000  -0.0000  -0.0000   0.0000   0.0000  -0.0000
  -0.0000   0.0000  -0.0000  -0.0000  -0.0000  -0.0000  -0.0000   0.0000   0.0000  -0.0000
  -0.0000   0.0000  -0.0000  -0.0000  -0.0000  -0.0000  -0.0000   0.0000   0.0000  -0.0000
  -0.0000   0.0000  -0.0000  -0.0000  -0.0000  -0.0000  -0.0000   0.0000   0.0000  -0.0000
   0.0000  -0.0000   0.0000   0.0000   0.0000   0.0000   0.0000  -0.0000  -0.0000   0.0000
   0.0000  -0.0000   0.0000   0.0000   0.0000   0.0000   0.0000  -0.0000  -0.0000   0.0000
  -0.0000   0.0000  -0.0000  -0.0000  -0.0000  -0.0000  -0.0000   0.0000   0.0000  -0.0000
];
MAGMA FINALIZE SUCCESS
TEST SUCCESSFUL
Timer: test_Schultz                                  1.02e+00  / 1.02e+00  / 1.02e+00  / 1       / 1       / 1
Timer: Replicated::copy                              8.68e-05  / 8.68e-05  / 8.68e-05  / 5       / 5       / 5
Timer: Replicated::memory_initialization             1.91e-05  / 1.91e-05  / 1.91e-05  / 1       / 1       / 1
Timer: Replicated::memory_free                       2.00e-05  / 2.00e-05  / 2.00e-05  / 1       / 1       / 1
Timer: Replicated::rescale                           5.20e-05  / 5.20e-05  / 5.20e-05  / 1       / 1       / 1
Timer: Replicated::schulz_iteration                  1.19e-02  / 1.19e-02  / 1.19e-02  / 1       / 1       / 1
Timer: Replicated::convergence_test                  1.18e-02  / 1.18e-02  / 1.18e-02  / 3       / 3       / 3
```



```
Test Schulz iterative solver
MAGMA INIT SUCCESS
MAGMA version of print
MPI process: 0 of 1
[
   1.0002   0.0004   0.0099   0.0020   0.0064   0.0090   0.0092   0.0005  -0.0076  -0.0046
   0.0004   1.0037  -0.0024   0.0027  -0.0069  -0.0028   0.0078  -0.0000  -0.0016  -0.0012
   0.0099  -0.0024   1.0063   0.0047  -0.0035   0.0096   0.0008  -0.0049  -0.0081   0.0097
   0.0020   0.0027   0.0047   1.0004   0.0058   0.0074  -0.0083   0.0039  -0.0001  -0.0072
   0.0064  -0.0069  -0.0035   0.0058   1.0008  -0.0003  -0.0017  -0.0069  -0.0029   0.0070
   0.0090  -0.0028   0.0096   0.0074  -0.0003   1.0061   0.0096   0.0004  -0.0056   0.0022
   0.0092   0.0078   0.0008  -0.0083  -0.0017   0.0096   0.9956  -0.0056  -0.0013   0.0098
   0.0005  -0.0000  -0.0049   0.0039  -0.0069   0.0004  -0.0056   1.0043   0.0036  -0.0097
  -0.0076  -0.0016  -0.0081  -0.0001  -0.0029  -0.0056  -0.0013   0.0036   1.0055  -0.0027
  -0.0046  -0.0012   0.0097  -0.0072   0.0070   0.0022   0.0098  -0.0097  -0.0027   1.0057
];
Norm Difference: 6.35048e-14
Iterations for Schulz iteration to converge: 3
Difference:
MAGMA version of print
MPI process: 0 of 1
[
  -0.0000   0.0000  -0.0000  -0.0000  -0.0000  -0.0000  -0.0000   0.0000   0.0000  -0.0000
   0.0000  -0.0000   0.0000   0.0000   0.0000   0.0000   0.0000  -0.0000  -0.0000   0.0000
  -0.0000   0.0000  -0.0000  -0.0000  -0.0000  -0.0000  -0.0000   0.0000   0.0000  -0.0000
  -0.0000   0.0000  -0.0000  -0.0000  -0.0000  -0.0000  -0.0000   0.0000   0.0000  -0.0000
  -0.0000   0.0000  -0.0000  -0.0000  -0.0000  -0.0000  -0.0000   0.0000   0.0000  -0.0000
  -0.0000   0.0000  -0.0000  -0.0000  -0.0000  -0.0000  -0.0000   0.0000   0.0000  -0.0000
  -0.0000   0.0000  -0.0000  -0.0000  -0.0000  -0.0000  -0.0000   0.0000   0.0000  -0.0000
   0.0000  -0.0000   0.0000   0.0000   0.0000   0.0000   0.0000  -0.0000  -0.0000   0.0000
   0.0000  -0.0000   0.0000   0.0000   0.0000   0.0000   0.0000  -0.0000  -0.0000   0.0000
  -0.0000   0.0000  -0.0000  -0.0000  -0.0000  -0.0000  -0.0000   0.0000   0.0000  -0.0000
];
MAGMA FINALIZE SUCCESS
TEST SUCCESSFUL
Timer: test_Schultz                                      1.02e+00  / 1.02e+00  / 1.02e+00  / 1       / 1       / 1
Timer: Replicated::copy                                  8.68e-05  / 8.68e-05  / 8.68e-05  / 5       / 5       / 5
Timer: Replicated::memory_initialization                 1.91e-05  / 1.91e-05  / 1.91e-05  / 1       / 1       / 1
Timer: Replicated::memory_free                           2.00e-05  / 2.00e-05  / 2.00e-05  / 1       / 1       / 1
Timer: Replicated::rescale                               5.20e-05  / 5.20e-05  / 5.20e-05  / 1       / 1       / 1
Timer: Replicated::schulz_iteration                      1.19e-02  / 1.19e-02  / 1.19e-02  / 1       / 1       / 1
Timer: Replicated::convergence_test                      1.18e-02  / 1.18e-02  / 1.18e-02  / 3       / 3       / 3
```

Solution – Matrix Size



- Computing the inverse square root of a matrix through diagonalization (**blue**) on a GPU

$$\boldsymbol{S}^{-\frac{1}{2}} = VD^{-\frac{1}{2}}V^T$$

is inefficient and memory-bound on a GPU.

- The Schulz iterative method (**orange**) parallelizes much more efficiently,

$$Y_{k+1} = 0.5Y_k(3I - Z_kY_k)$$
$$Z_{k+1} = 0.5(3I - Z_kY_k)Z_k$$

where $Y_k \to S^{\frac{1}{2}}$ and $Z_k \to \boldsymbol{S}^{-\frac{1}{2}}$ as $k \to \infty$.

[1] M. Lupo Pasini, B. Turcksin, W. Ge, and J.-L. Fattebert, "A parallel strategy for density functional theory computations on accelerated nodes," *Parallel Computing*, vol. 100, p. 102703, Dec. 2020, doi: 10.1016/j.parco.2020.102703.

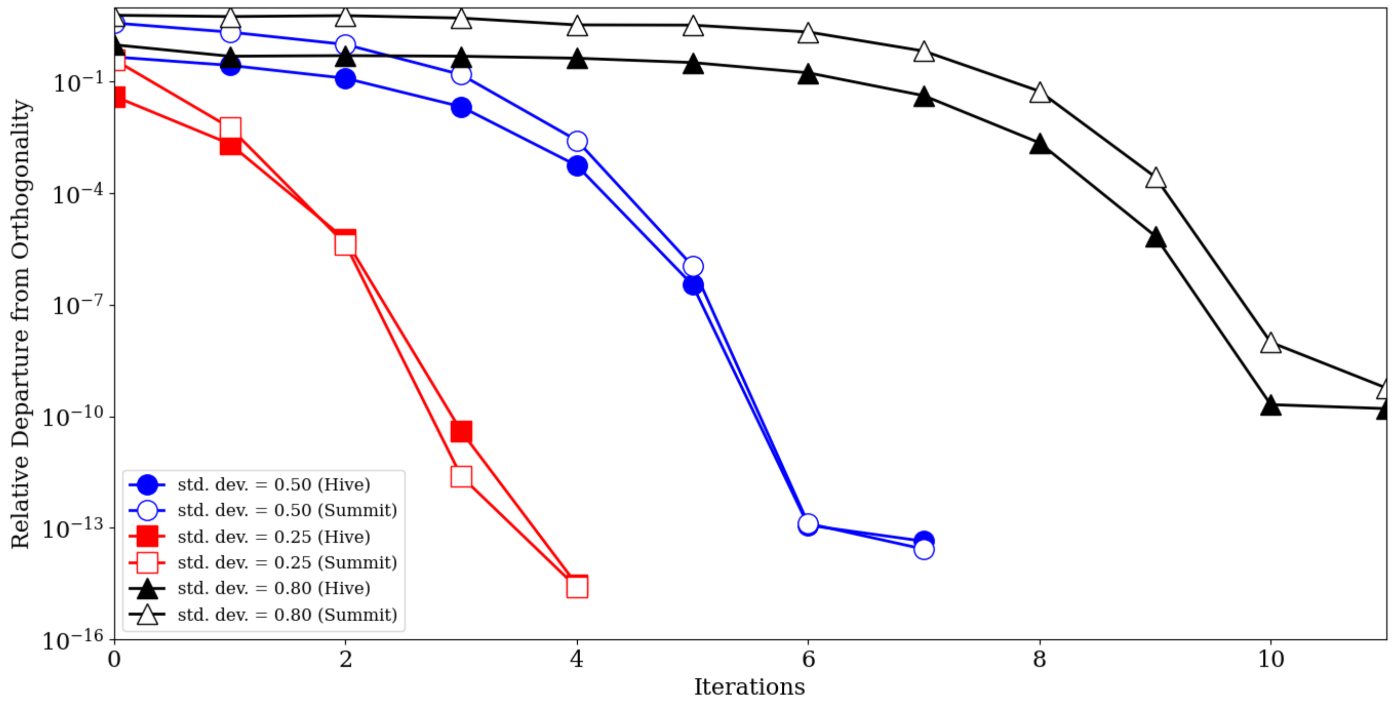
Georgia Tech

Solution – Convergence of Schulz Iteration

- Investigated number of Schulz iterations needed to restore orthogonality of columns of A matrix
- **Case**: tall-skinny, 300,000 X 3000 matrix with standard deviations of $\sigma = 0.25, 0.5, 0.8$
- Relative departure from orthogonality defined as

$$\frac{||DA^T AD - I||}{||DA^T AD||}$$

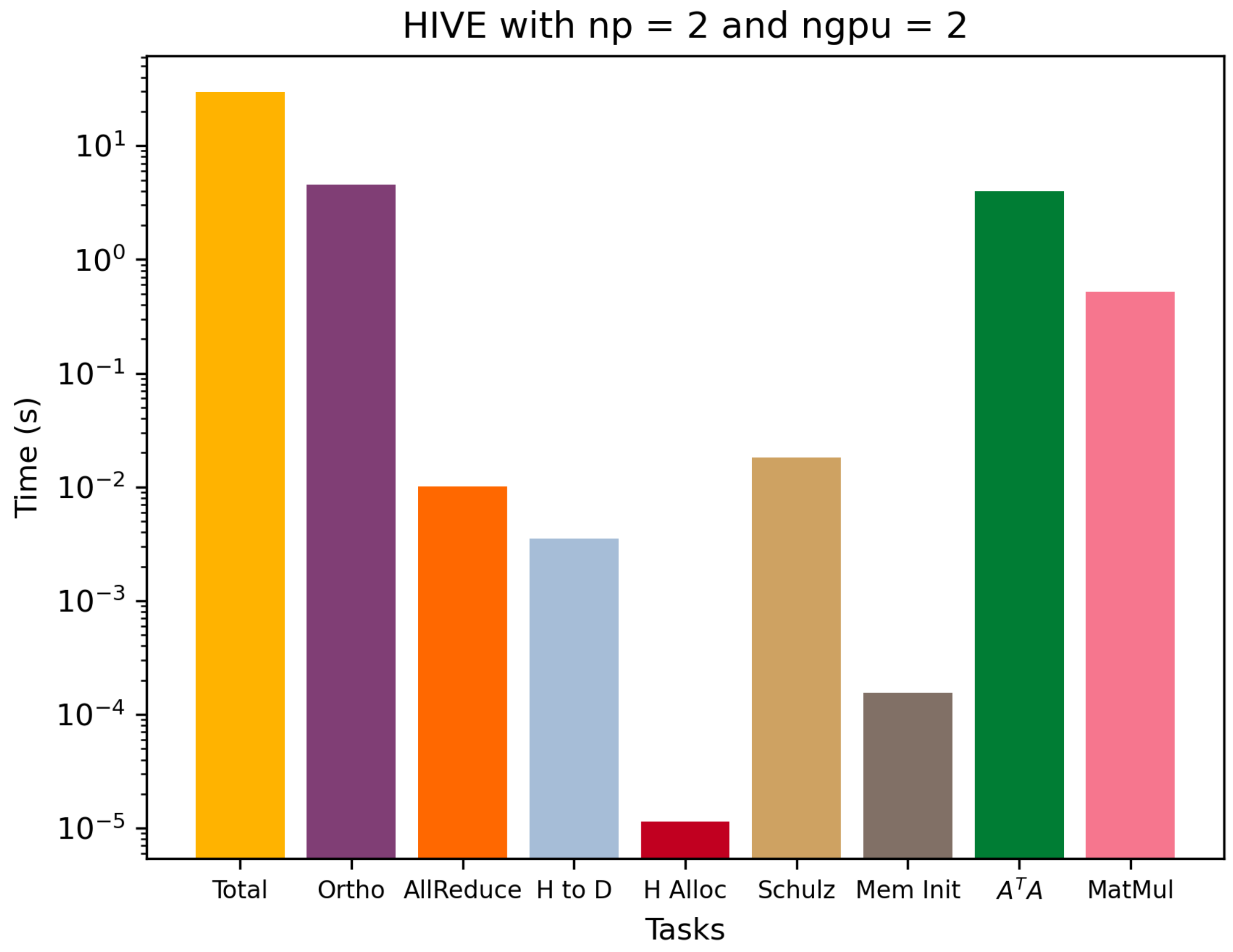
where $D_{ii} = S_{ii}^{-\frac{1}{2}}$ is diagonal scaling matrix.



Solution – Strong Scaling (Hive vs. Summit)

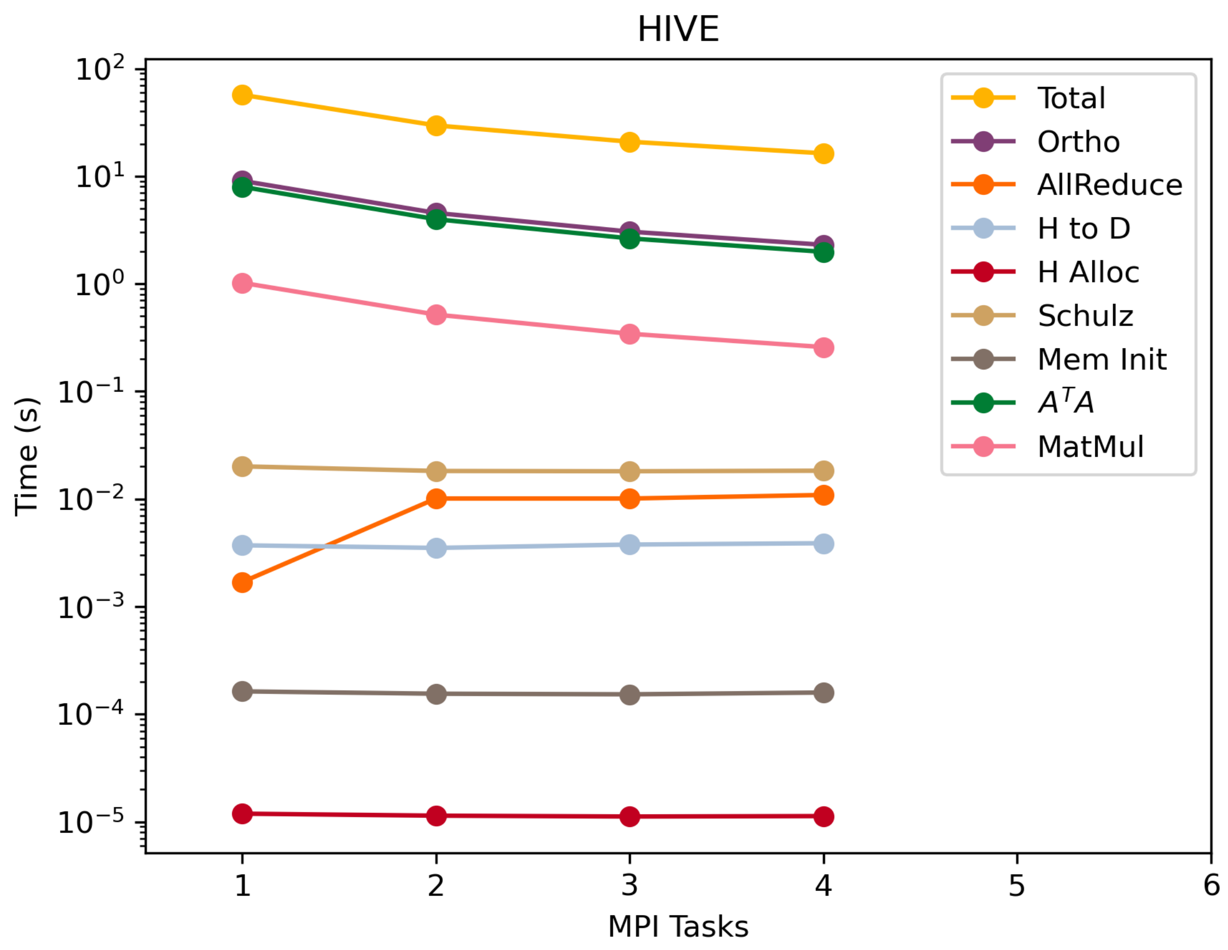
- We plan to reproduce strong scaling tests from Summit on Hive and ICEHAMMER
  - Currently the benchmark is not making use of multiple GPUs when MPI is managed by SLURM (the scheduler on both Hive and ICEHAMMER)
- We plan to address this issue and reproduce strong scaling tests for:
  - Hive: 1, 2, 4 x Tesla V100s
  - ICEHAMMER: 1, 2, 4, 8, 16 x Tesla A100s
  - COC-ICE: 1, 2, 4, 8, 16 x Tesla V100s

Solution – Time Breakdown at one case



- Runtime breakdown of Schulz iterative method on 3,000,000 x 300 tall skinny matrix
- These timing results were obtained with a two MPI tasks
- Each MPI task:
  - 1 process
  - One V100 GPU

Solution – Strong Scaling



- Currently the program is not taking advantage of additional MPI tasks and GPUs when SLURM manages MPI on ICEHAMMER
- Hive can only request up to 4 GPUs per job, so each MPI Task has a single V100 GPU and one process
- We are working to address issues on COC-ICE and ICEHAMMER for accurate strong scaling test results

Challenges

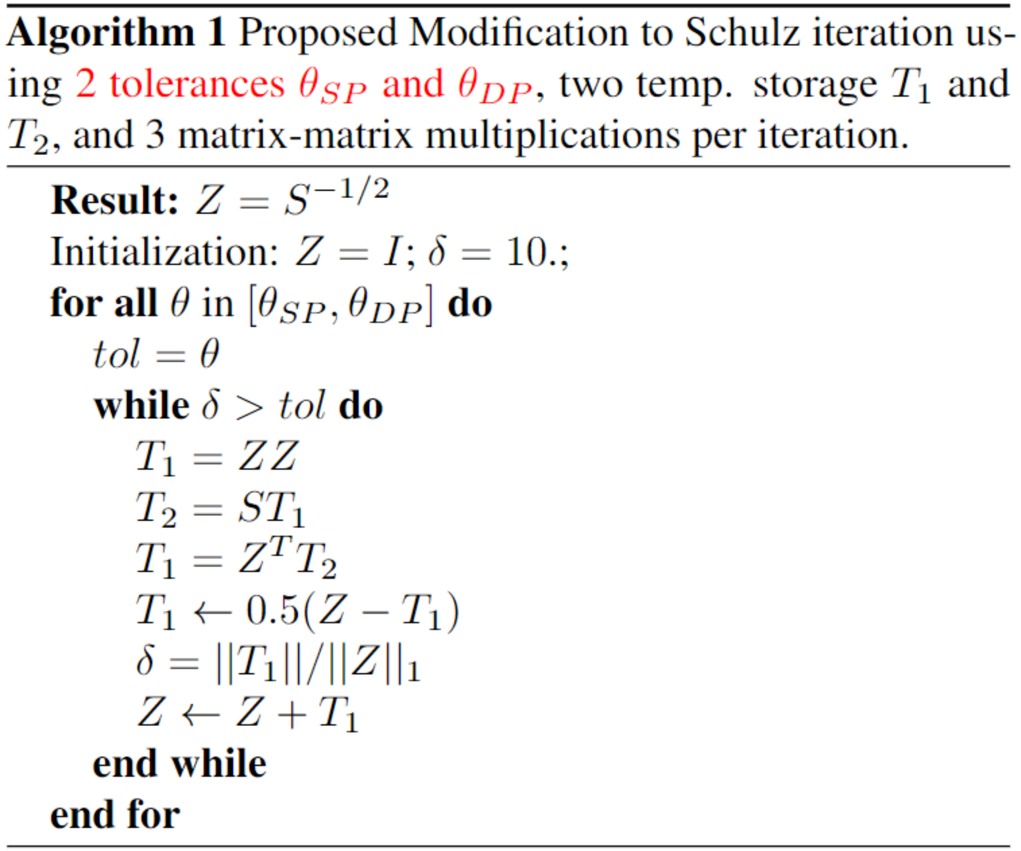
- Building Magma (GPU Numerical Linear Algebra Library)
  - The build scripts contained a bug that produced an incomplete pkgconfig file
  - We had to manually modify the autogenerated magma.pc to add the library's include directories and cflags
- Building ParrLO
  - We had never used pkgconfig before, so we initially made manual modifications to the CMakeLists.txt to include and link against Magma
  - We later figured out that we could prepend our $PKG_CONFIG_PATH var with the magma.pc file which simplified the build process
  - We encountered errors linking against the version of boost available on PACE, which we were able to fix by manually specifying the missing libraries with linker flags and modifying our CMakeLists.txt to set the CXX ABI to the old C++98 standard for compatibility
- With these changes we were able to get most tests running

Ongoing Challenges

- Most tests running on PACE, HIVE, and ICEHAMMER
  - Schulz Iteration, MPI, Orthonormalization, Replicated MaxNorm, MAGMA
- MatMul test and main benchmark fail on PACE and ICEHAMMER
  - We suspect it may have something to do with CUDA-aware MPI
  - Several parts of the program return invalid pointer errors on freeing memory
- We do not have sufficient resources to run the full benchmark on HIVE
  - The paper's benchmark matrix is 3,000,000 x 3,000 = ~72GB at double precision
  - With only 4 x 16GB Tesla V100 GPUs on HIVE, we must reduce the matrix size to run the program
  - We hope to get the full-sized project running on ICEHAMMER where we have access to 16 x 80GB Tesla A100 GPUs

Future Work – Mixed Precision

- Magma single, half, & double precision APIs for BLAS operations
  - magma_sgemm
  - magma_hgemm
  - magma_dgemm
- We plan to modify the benchmark with these and investigate the impact on communication time and convergence



Algorithm 1 Proposed Modification to Schulz iteration using 2 tolerances θ_{SP} and θ_{DP}, two temp. storage T_1 and T_2, and 3 matrix-matrix multiplications per iteration.

```
Result: $Z = S^{-1/2}$
Initialization: $Z = I; \delta = 10.;$
for all $\theta$ in $[\theta_{SP}, \theta_{DP}]$ do
    $tol = \theta$
    while $\delta > tol$ do
        $T_1 = ZZ$
        $T_2 = ST_1$
        $T_1 = Z^T T_2$
        $T_1 \leftarrow 0.5(Z - T_1)$
        $\delta = ||T_1||/||Z||_1$
        $Z \leftarrow Z + T_1$
    end while
end for
```

Georgia Tech

Accelerate Tensor Computation Leveraging CodeGen and SIMD on ARM CPU

Fan Qu, Peidi Song

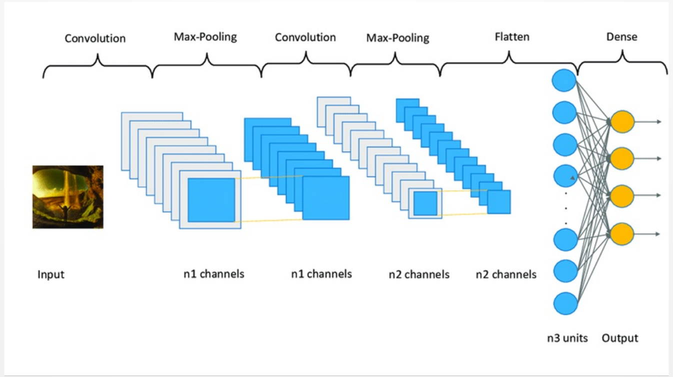
CONTENTS

Category

- Application
  - In detail, our focus is on developing a practical solution to accelerate tensor computation

Problem

- Tensor computation
  - Widely used in neural networks
  - E.g., GEMM, Convolution2D, Convolution1D…
  - Computationally expensive and time-consuming, particularly on ARM CPUs



Problem

- Machine learning library
  - Example
    - PyTorch, Tensorflow
  - Advantages
    - High performance
    - User friendly APIs
  - Challenges
    - Contributed by experts only
    - Long and complex handwritten kernels
    - Limited number of operators

- Code generation technology
  - Main idea
    - User-defined computation
    - User-defined or auto optimization
  - Advantages
    - Highly customizable operators
    - Convenient for optimization and tuning
  - Challenges
    - Worse performance than libraries

Problem

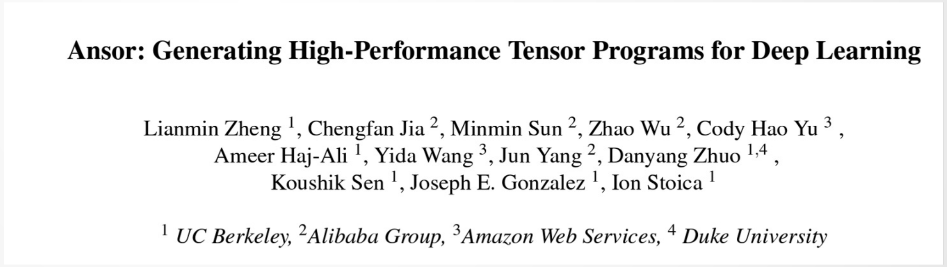
- Goal
  - Accelerate tensor computation on ARM CPUs
  - Leverage CodeGen and SIMD technology
  - Demonstrate how to implement customized operators
  - Achieve reasonable performance

Performance Metrics

- Runtime for data of different of size
- Evaluate different tensor operations

Baselines

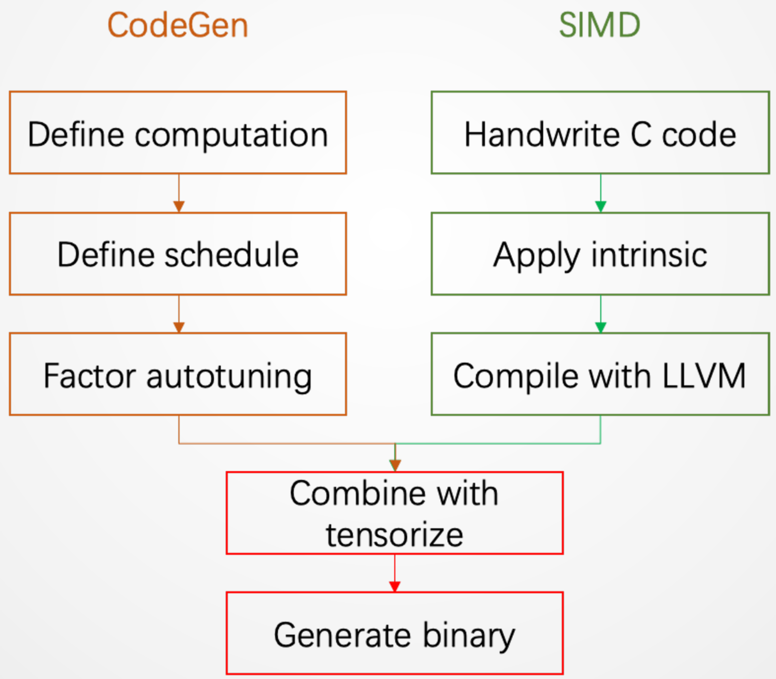
- PyTorch
- Ansor
- TensorFlow
- AutoTVM
- …


Ansor: Generating High-Performance Tensor Programs for Deep Learning

Lianmin Zheng [1], Chengfan Jia [2], Minmin Sun [2], Zhao Wu [2], Cody Hao Yu [3] ,
Ameer Haj-Ali [1], Yida Wang [3], Jun Yang [2], Danyang Zhuo [1,4] ,
Koushik Sen [1], Joseph E. Gonzalez [1], Ion Stoica [1]

[1] *UC Berkeley,* [2]*Alibaba Group,* [3]*Amazon Web Services,* [4] *Duke University*

Solution - Workflow



Solution - CodeGen

TVM

- A Python interface end-to-end compiler framework for CPU, GPU, and accelerators
- Seperate computation and optimization
  - Define computation
    - `C = sum(A[i,k]*B[k,j], reduce=k)`
  - Define schedule primitive
    - `io, ii = split(i, factor=8)`
    - `reorder(io, jo, ko, ki, ii, ji)`

Solution - CodeGen (GEMM as example)

- Define computation
  - `C = sum(A[b,i,k]*B[b,k,j], reduce=k)`
- Define schedule
  - Cache read
    ```
    A_local = sch.cache_read(A, "local", C)
    ```
  - Split axes
    ```
    cfg.define_split("tile_i", i, num_outputs=4)
    i1, i2, i3, i4 = cfg["tile_i"].apply(s, O_pad, i)
    ```
  - Reorder axes
    ```
    sch[C].reorder(b,i1,j1,...)
    ```
  - Parallel
    ```
    sch[C].parallel(sch[C].fuse(b,i1,j1))
    ```

Solution - CodeGen

- Factor autotuning
  - Define configs when splitting axes
  - Apply `autotvm.tuner.GATuner` as the tuner
    - This tuner applys genetic algorithm
    - Tune for multiple trials
    - Early stop when no better schedule found in several trials
  - Apply best schedule

```
task = autotvm.task.create("gemm", args, target)
measure_option = autotvm.measure_option(LocalBuilder, LocalRunner)
tuner = autotvm.tuner.GATuner(task)
tuner.tune(n_trials, early_stopping)
with autotvm.apply_history_best("gemm.log"):
        func = tvm.build(s, args)
```

Solution - SIMD

NEON SIMD Instructions

- NEON is a technology that enables parallel processing on ARM CPUs.
- Vectorization: processing multiple elements of the tensors at the same time
- Low-Level Optimization: loop unrolling and memory alignment to maximize the performance

Solution - SIMD

- Apply NEON intrinsics
  - Vector load
    - `vld1q_f32`
  - FMA
    - `vfmaq_n_f32`
  - Vector store
    - `vst1q_f32`
- Compile with LLVM

```
ll_code = clang.create_llvm(c_code, options=["-O2", ], cc="clang")
```

Solution - Tensorize and Generate binary

- Use `tensorize` to combine TVM Python and NEON C code

  -
    ```
    sch[C].tensorize(intrin_micro_kernel())
    sch[C].pragma("import_llvm", gemm_kernel())
    ```
- Generate binary
  - `gemm = tvm.build(sch, arg_bufs)`

Validation

We use the calculation result of PyTorch as the ground truth.

Datasets

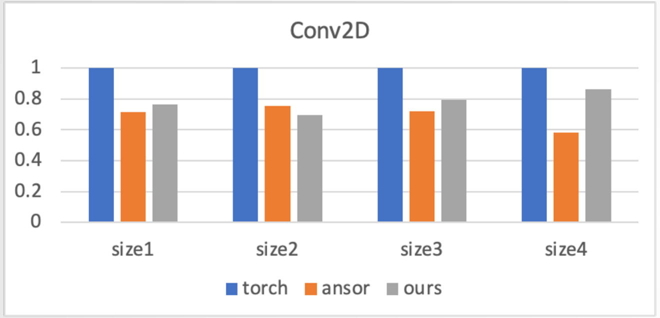
- Different operators
  - GEMM
  - Convolution 2D
  - Convolution 1D
- Different tensor sizes
  - Refer to sizes in Transformer, U-nets
- Generate tensors randomly

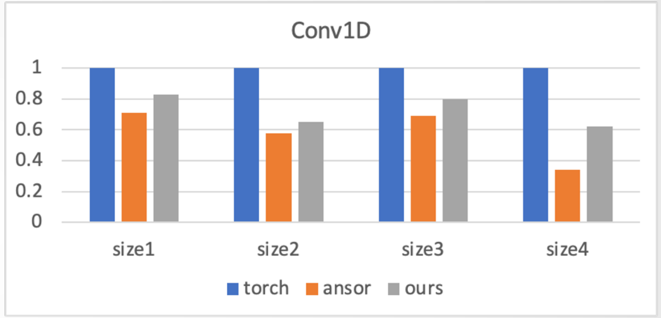
Platforms

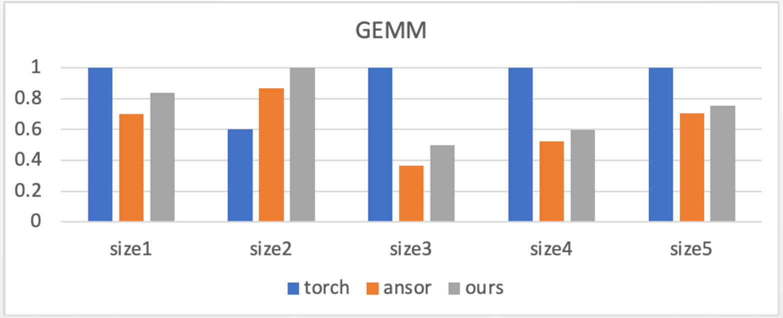
We optimize on common ARM CPUs for consumers.

- Apple Silicon
  - M2 chip, Avalanche
  - 8-Core CPU
  - 16 GB Unified Memory

Plots


Conv2D
1
0.8
0.6
0.4
0.2
0
size1
size2
size3
size4
torch
ansor
ours


Conv1D
1
0.8
0.6
0.4
0.2
0
size1
size2
size3
size4
torch
ansor
ours


GEMM
1
0.8
0.6
0.4
0.2
0
size1
size2
size3
size4
size5
torch
ansor
ours

References

- Paszke, Adam, et al. "Pytorch: An imperative style, high-performance deep learning library." Advances in neural information processing systems 32 (2019).
- Abadi, Martín, et al. "Tensorflow: a system for large-scale machine learning." Osdi. Vol. 16. No. 2016. 2016.
- Chen, Tianqi, et al. "TVM: An automated end-to-end optimizing compiler for deep learning." arXiv preprint arXiv:1802.04799 (2018).
- Chen, Tianqi, et al. "Learning to optimize tensor programs." Advances in Neural Information Processing Systems 31 (2018).
- Zheng, Lianmin, et al. "Ansor: Generating high-performance tensor programs for deep learning." Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation. 2020.

Thanks

CSE 6230 Project Presentation

Parallel Framework for Particle Dynamics Simulation

Category: Application

Yu Du
yudu@gatech.edu

Changhai Man
cman8@gatech.edu

Qiang Wu
qwu350@gatech.edu

Georgia Institute of Technology

April 2023

Problem Introduction

Lots of applications calls for simulation frameworks about large-scale particle systems!

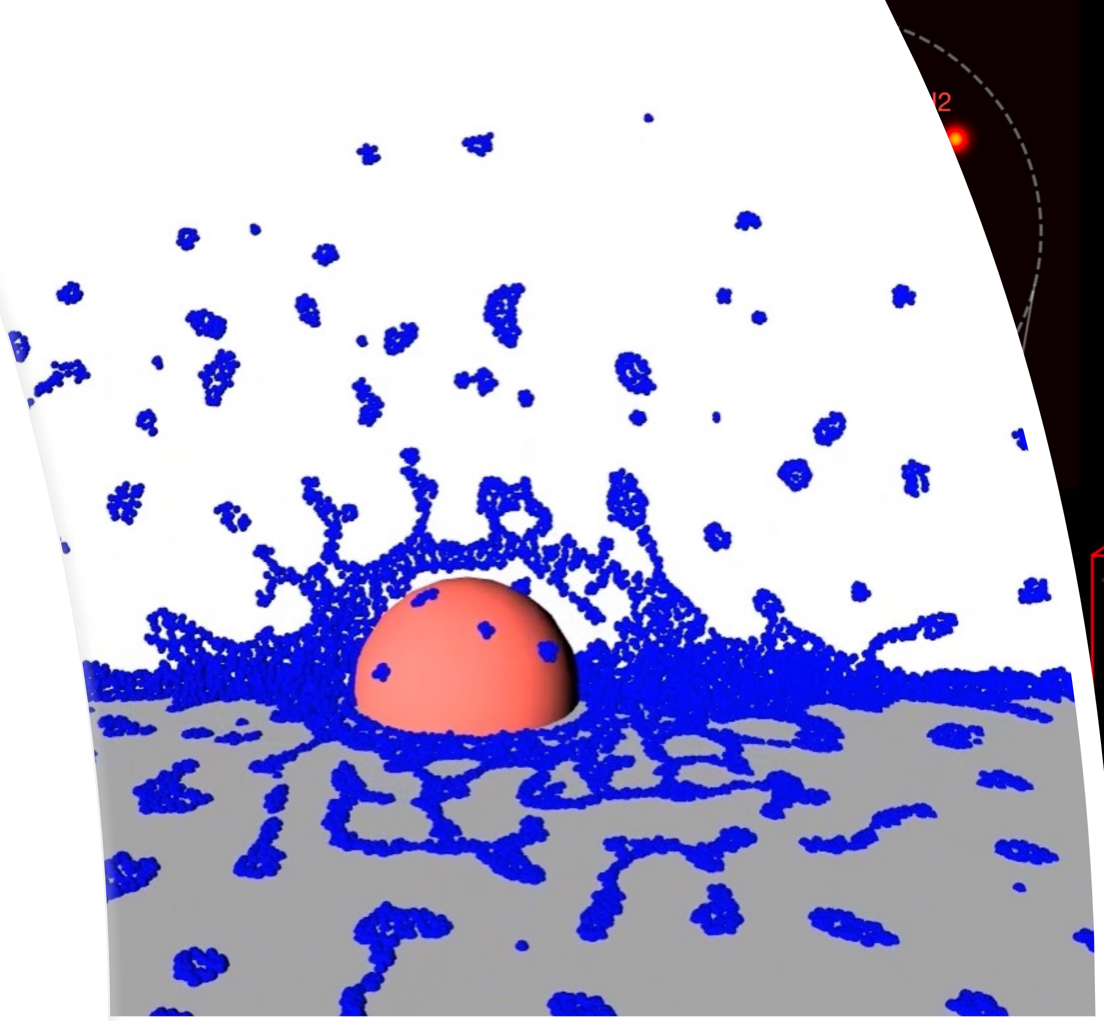
- Astronomy:

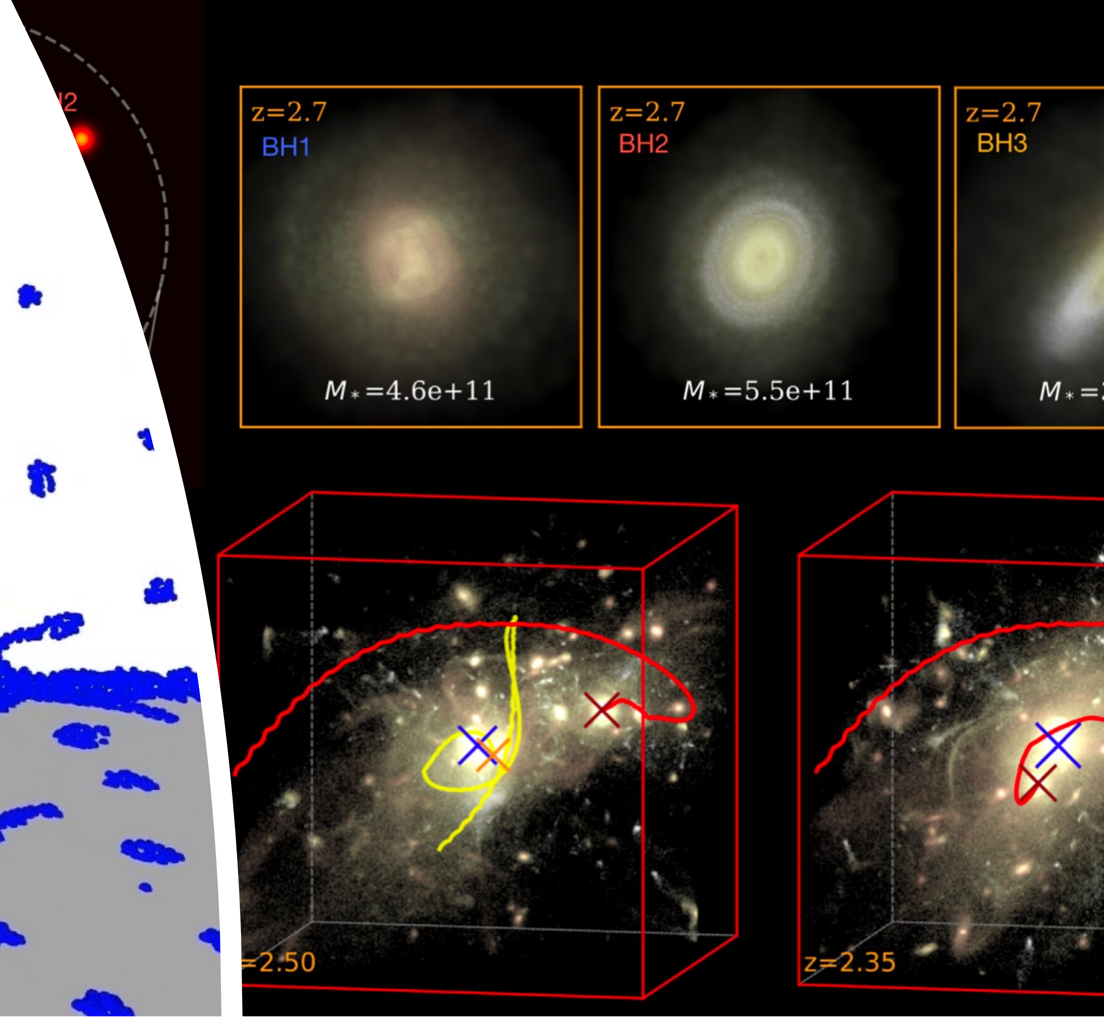
  Gravity simulations for galaxy systems.
- Chemistry & Biochemistry:

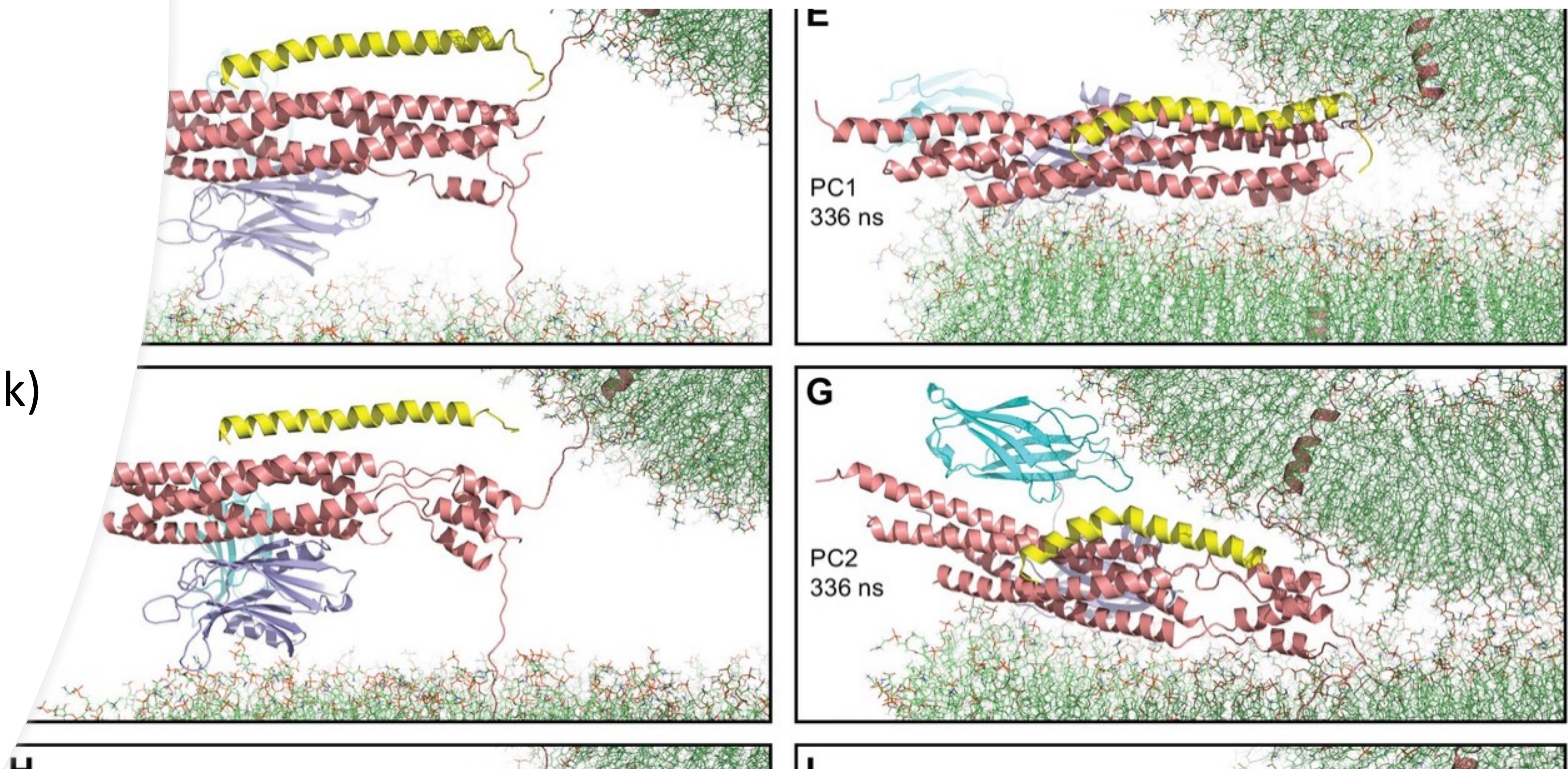
  Molecular dynamics
- Electro-Dynamics:

  Multiple particles moving under electromagnetic fields (Accelerator, Tokamak)
- Fluid-Dynamics:

  Smoothed Particle Hydrodynamics
- Many others...







Problem Abstraction

For particle dynamic systems, we have:

A set of particles $\mathbf{p} \in \mathbf{\Omega}$.
For each particle $\boldsymbol{p}$, it has the position(assume in 3d) $\boldsymbol{r}$, velocity $\boldsymbol{v}$, and some other attributes $\boldsymbol{s}$ (for example, massive, charge, temperature, box volume, etc.)

$$\boldsymbol{p} = (\boldsymbol{r}, \boldsymbol{v}, \boldsymbol{s}), \boldsymbol{r} = (x, y, z), \boldsymbol{v} = (v_x, v_y, v_z), \boldsymbol{s} = (m, q, \dots)$$

Between each particles, there is some interactions, for example, gravity, Coulomb, Van der waals force.

$$\begin{bmatrix} f(\boldsymbol{p}_1, \boldsymbol{p}_2) \cdots f(\boldsymbol{p}_1, \boldsymbol{p}_k) \\ \vdots \quad\quad \cdots \quad\quad \vdots \\ f(\boldsymbol{p}_k, \boldsymbol{p}_1) \cdots f(\boldsymbol{p}_k, \boldsymbol{p}_k) \end{bmatrix}$$

These interactions to one particle can be merged:

$$f(\boldsymbol{p_k}, \mathbf{\Omega}) = \sum_i f(\boldsymbol{p}_k, \boldsymbol{p}_i)$$

And the interactions will affect the position of each particles along the time under certain timestep Δt:

$$\boldsymbol{v_k}(t + \Delta t) = \boldsymbol{v}_k(t) + \frac{f(\boldsymbol{p}_k, \mathbf{\Omega})}{m} \Delta t$$

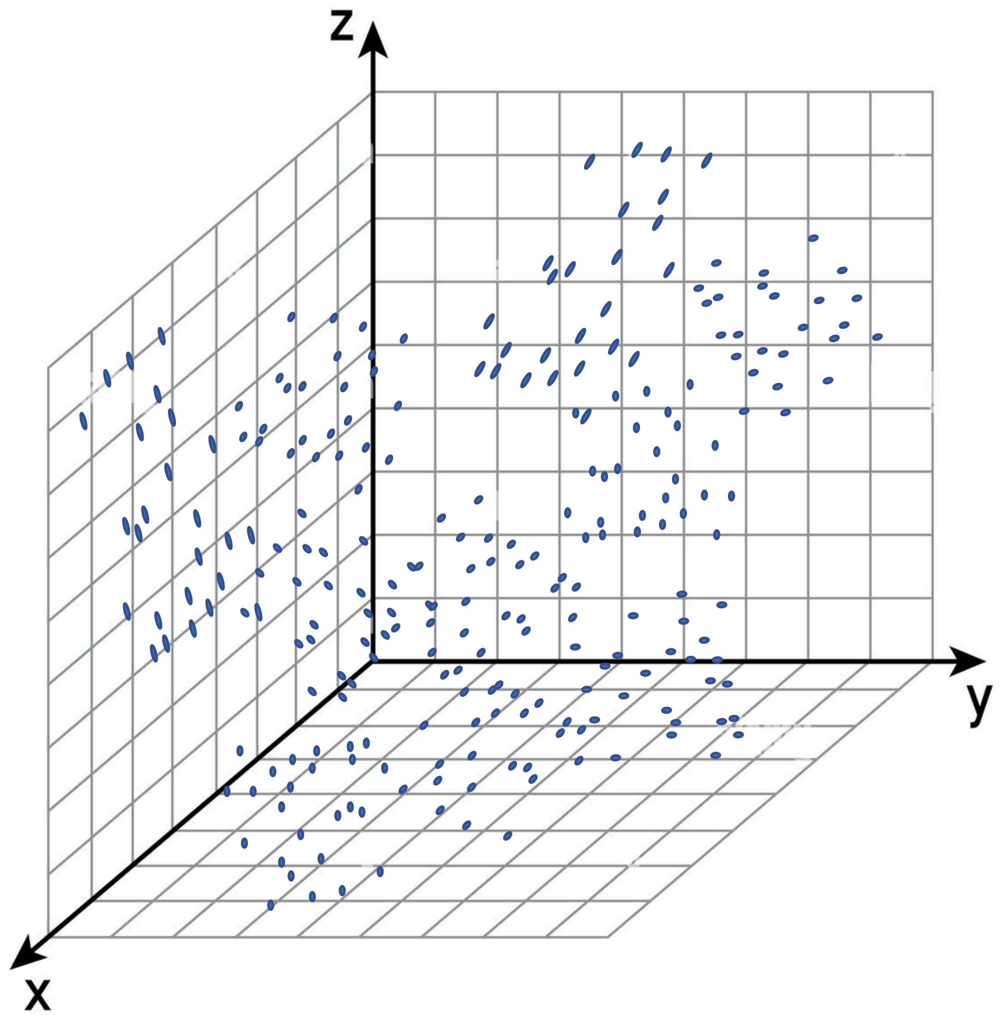
$$\boldsymbol{r_k}(t + \Delta t) = 2\boldsymbol{r}_k(t) - \boldsymbol{r}_k(t - \Delta t) + \frac{f(\boldsymbol{p}_k, \mathbf{\Omega})}{m} \Delta t^2$$

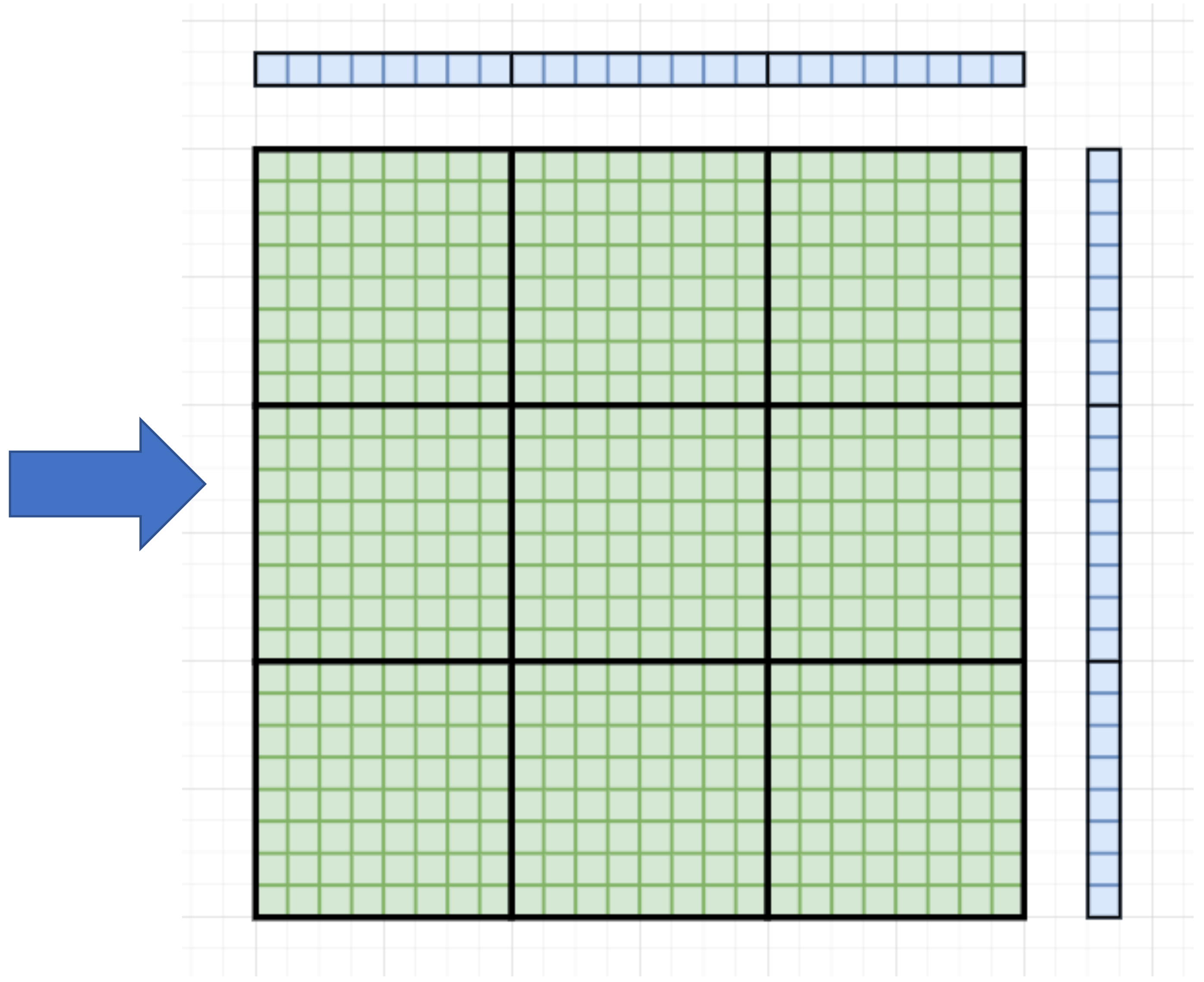
Solution I: MPI & OpenMP

For N particles, each particles would be interactive to other N-1 particles, which build up a NxN matrix of interactions.

In our implementation

- We implement a OpenMP based program for each node,
- Then we use MPI to dispatch workloads across nodes.



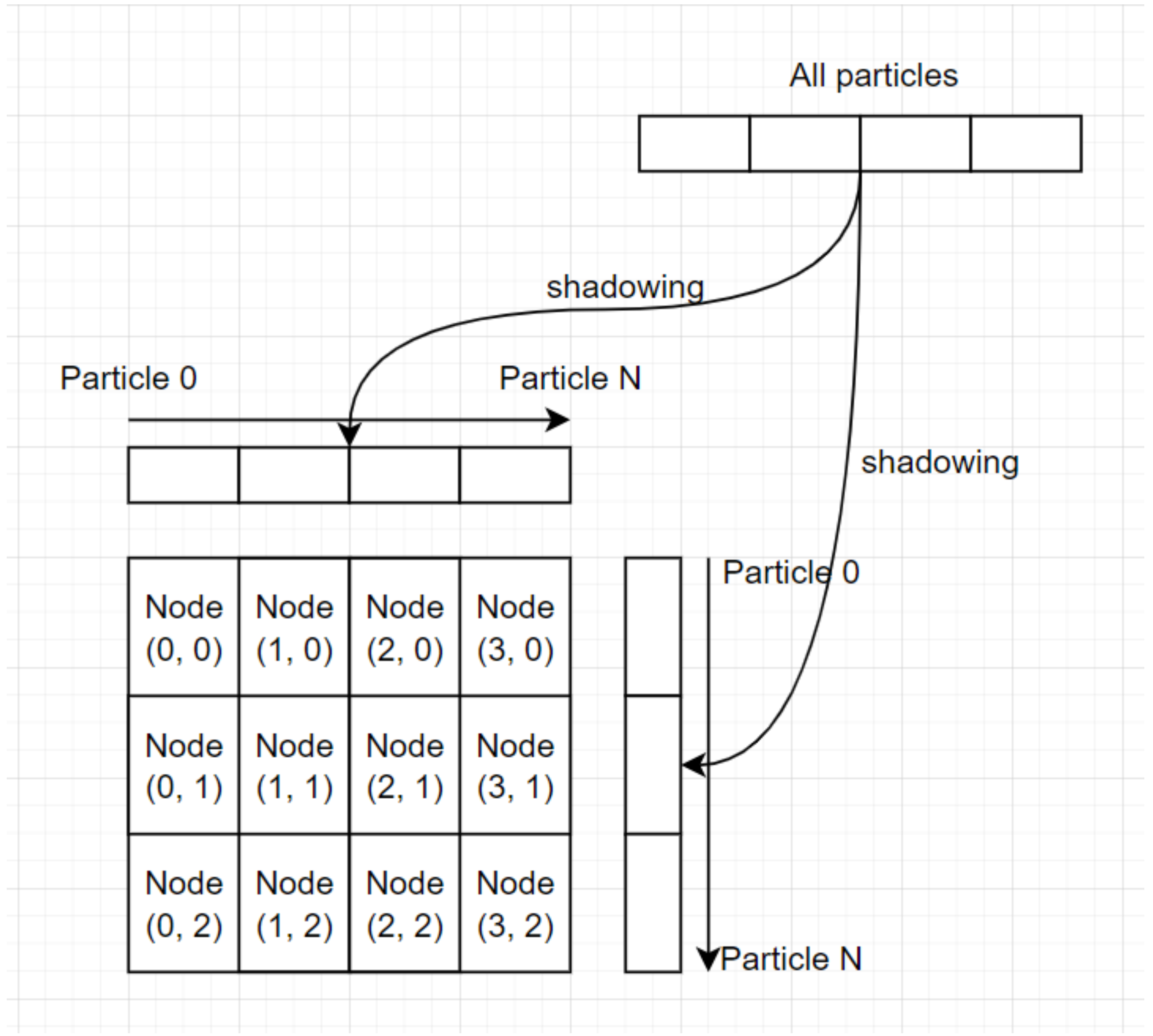


Solution I: MPI & OpenMP

MPI Implementation: Dense Scheduler

To implemented the distributed computing, here we have the parallel strategy as follows:

1. Shadowing all particles in X and Y edges,
2. Scatter all particles to edge nodes,
3. Broadcast from the edge nodes to inner nodes,
4. Calculate accelerations at each node (locally)
5. Allreduce accelerations across cols/rows(globally)
6. Update velocity && position at edge nodes(in one direction)
7. Gather and Re-scatter particles to all edge nodes(in the other direction)



Solution I: MPI & OpenMP

MPI Implementation: Cutoff Scheduler

In real problems, the interaction between particles might decay rapidly:

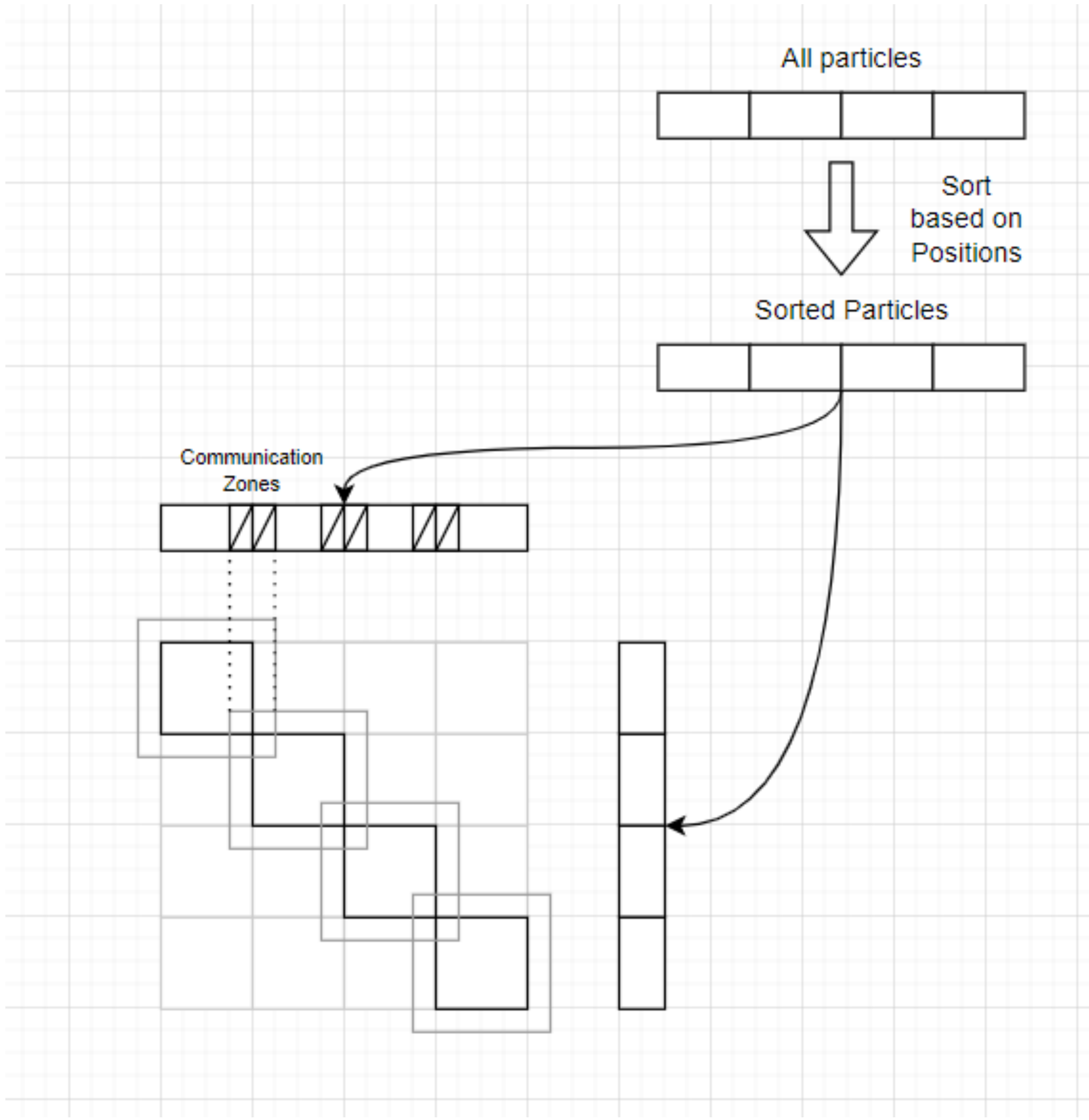
- Van der Waals forces: $f = Ar^{-12} - Br^{-6}$

Which makes it possible to speed-up the calculation by using cutoff approximation, which assume:

- Interaction beyond a certain range can be ignored.

Based on the assumption, we have:

- Sort particles before scattering.
- For each node, it only compute interaction between same sets of particles.
- To avoid the approximation error, setup a slightly large neighbor zone which can only be observer, but not calculate the accelerations.
- The computation complexity shrinks from N^2 to N
- Also, the communication is greatly reduced.



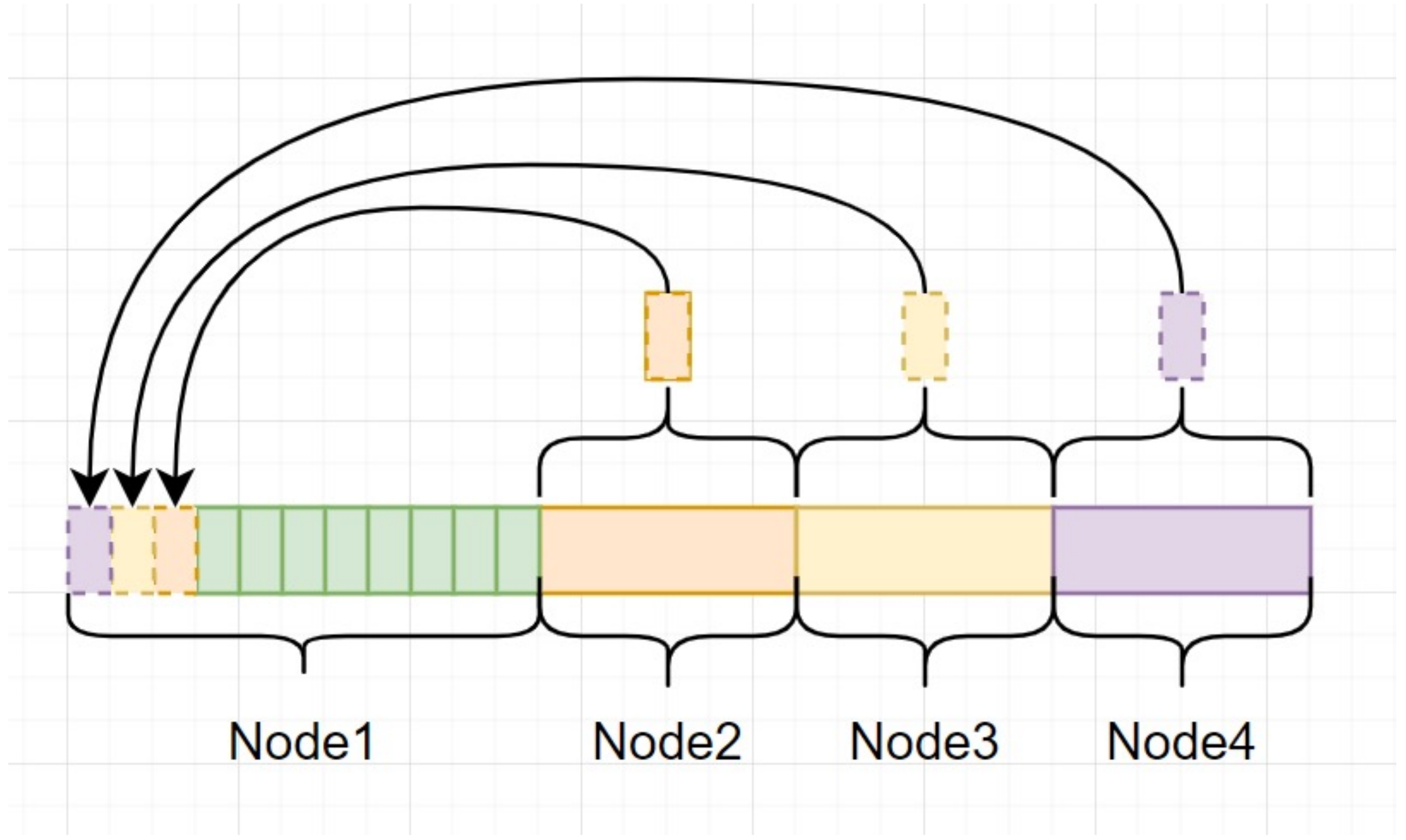
Solution I: MPI & OpenMP

MPI Implementation: “Centroid” Scheduler

For some interactions, the cut-off assumption is wrong, for example:

- Gravity: Sun has a long distant from us, however, we and the earth feel the gravity from sun.

For these kind of interactions, we approximate the distant particles with their “centroid”, instead of force cut-off.



Solution II: CUDA

Naive Implementation: Two nested for-loop

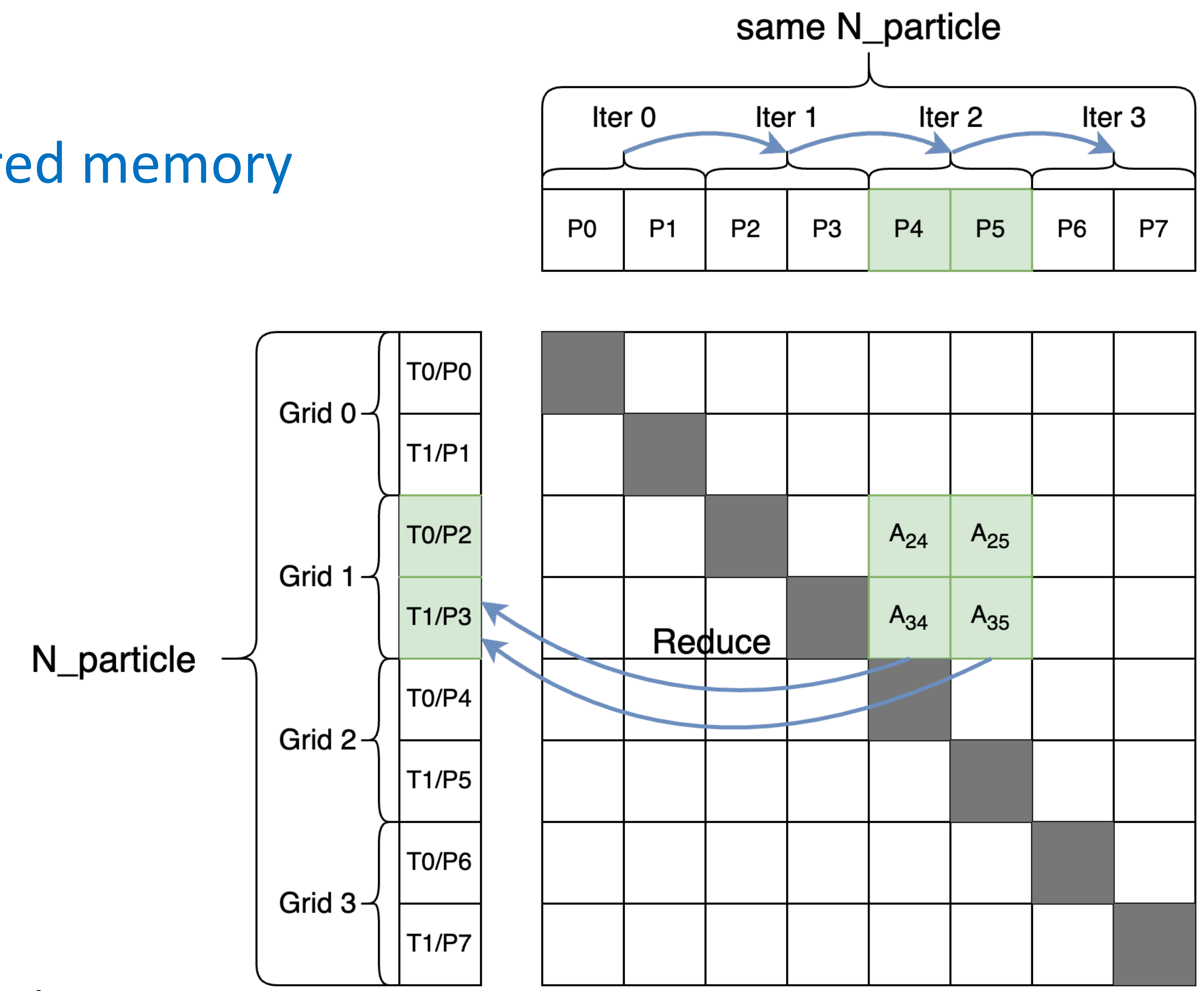
--> costs much more time! (around 50x OpenMP)

Improvement: lift computational intensity with shared memory

- N = # of particles = # of threads;
- One kernel call for each timestep;

For each kernel call:

- Load corresponding block of particles to share memory once;
- Iteratively load other blocks of particles to share memory;
- Compute interaction (acceleration) for each pair;
- Reduce to update the responsible particle;
- No need to physically allocate the $n \times n$ matrix.



Result: Much faster than that of OpenMP! (around 1/60)

Performance Metric, Baselines, Validation, Datasets & Testbed

Performance Metric:

1. Time (Strong Scaling Plot, Run-time Breakdown, Roofline Model);
2. Memory (Complexity Analysis)

Baseline: Sequential version of our algorithm

Validations:

1. (Main method) Parallel version result == sequential version result. (Will show in demo)
2. (Optional) Intuitive visualization

Animation on JavaScript: Visualization (Refer to [1])

Datasets: Randomly distributed initial position, velocity and mass

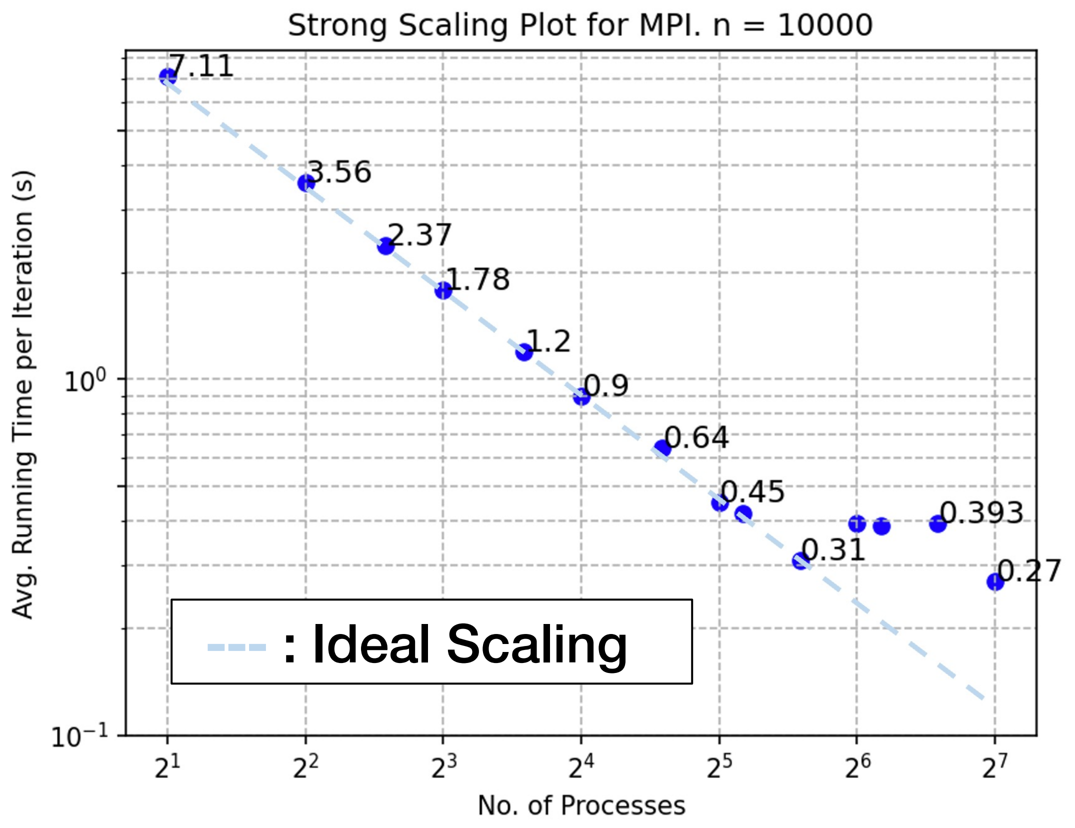
Testbed:

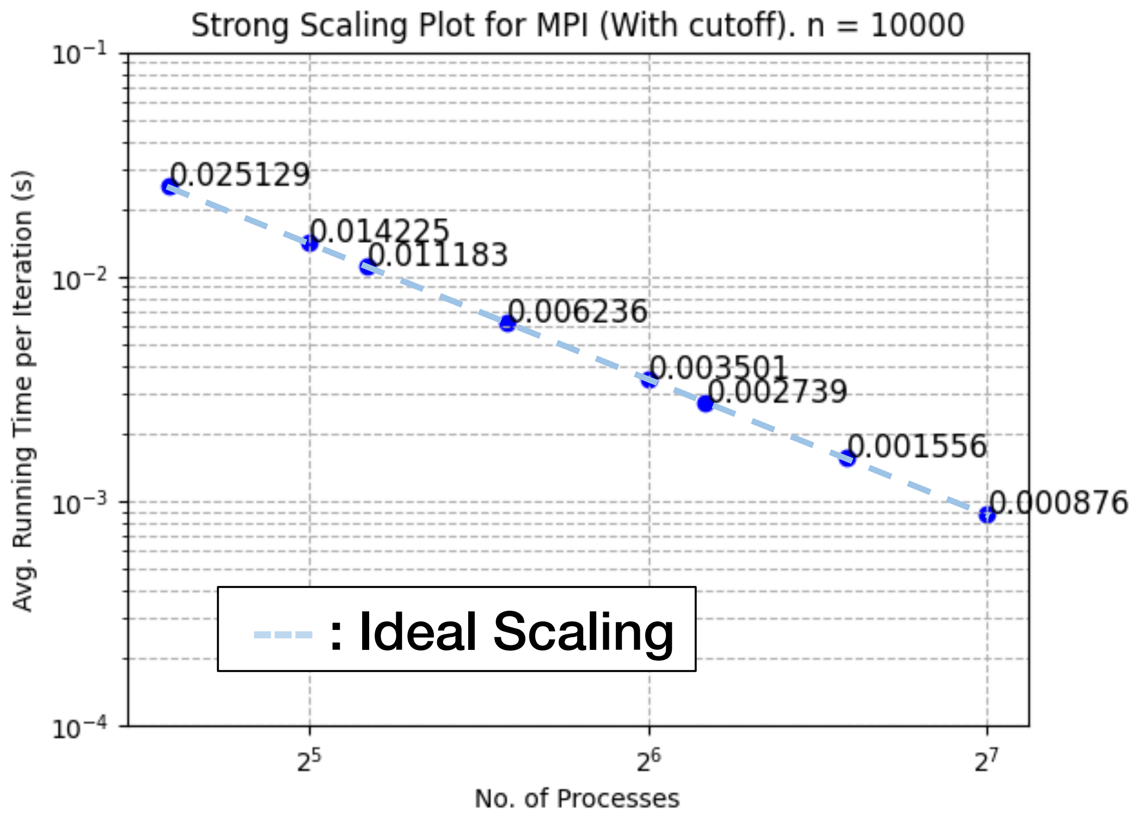
OpenMP and MPI solution: coc-ice-multi, 16 nodes, 8 processes per node;

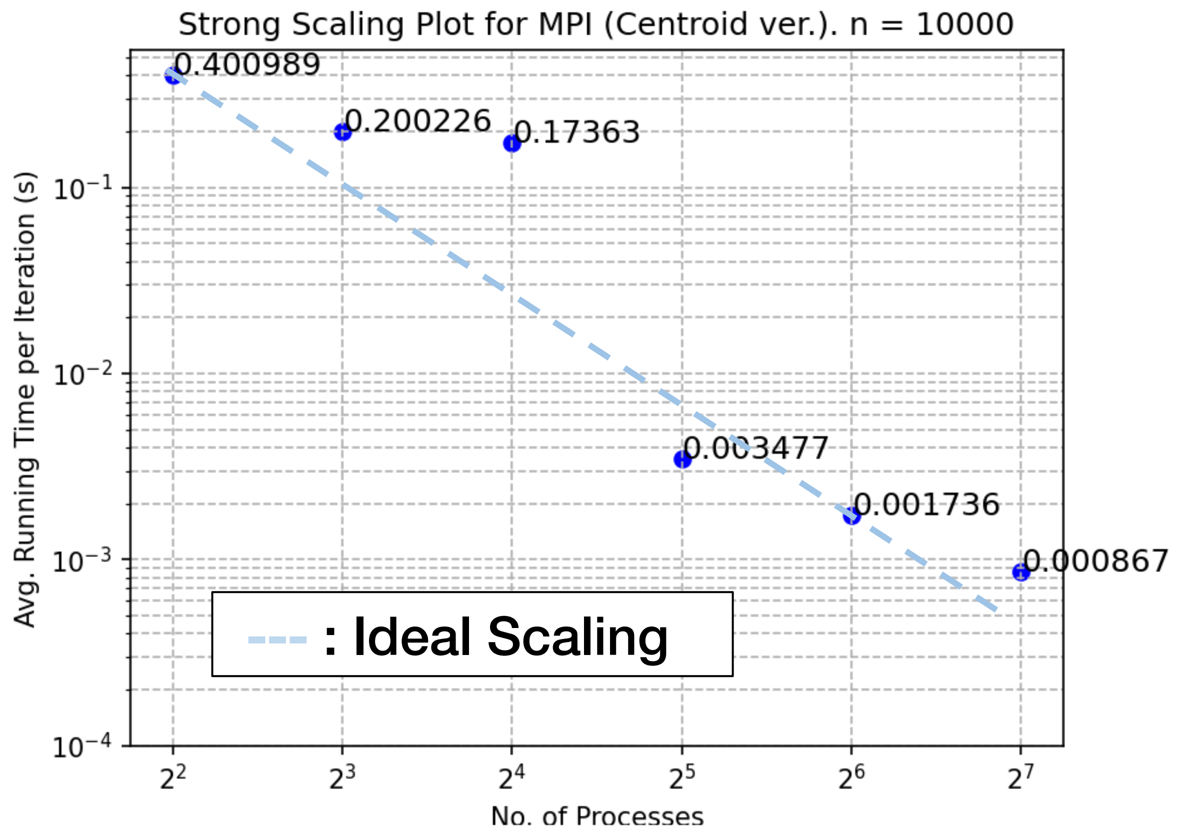
CUDA solution: coc-ice-gpu, 1 Tesla V100 GPU, 1 node

[1] https://hunar4321.github.io/particle-life/particle_life_3d.html

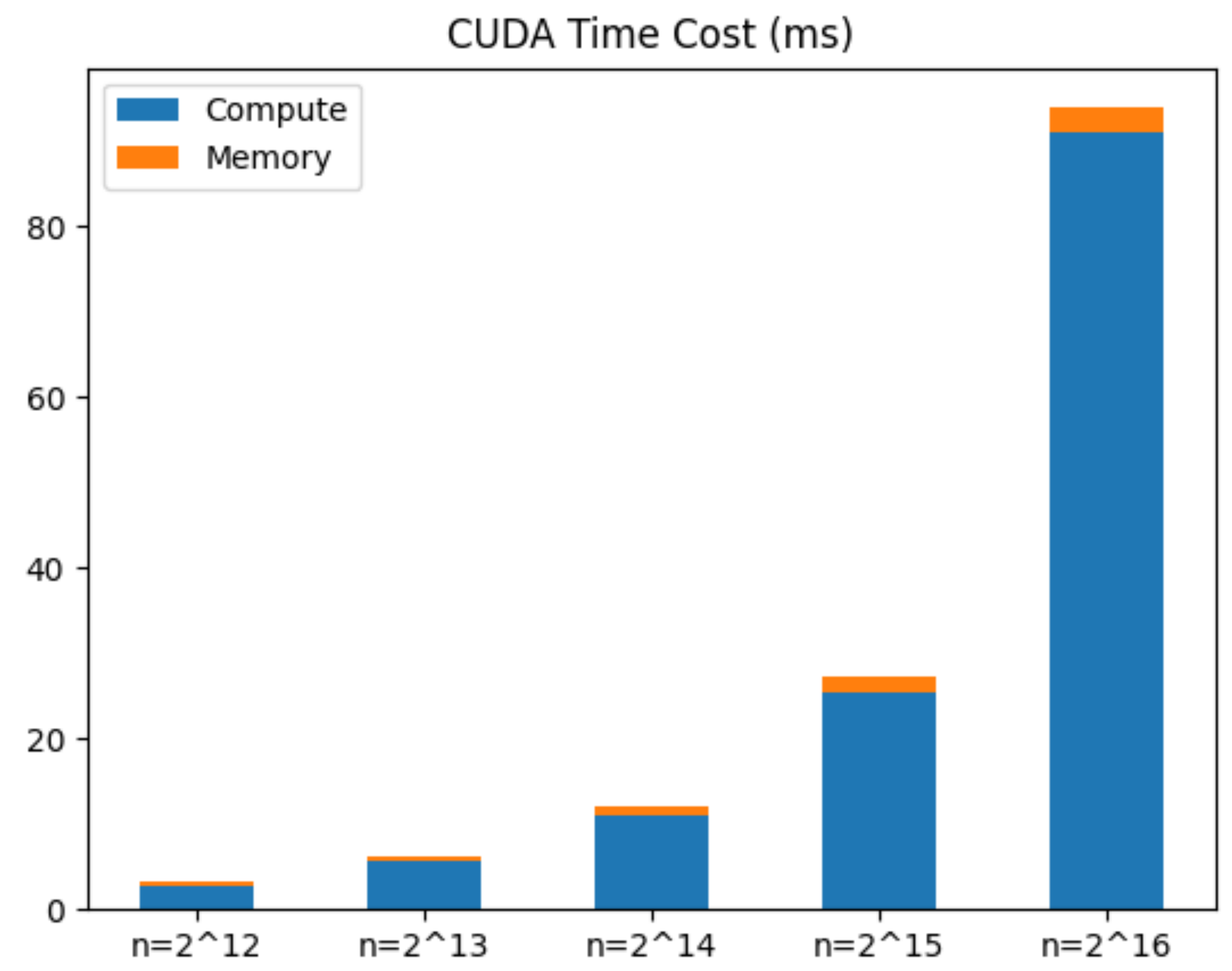
Plots for MPI Solution



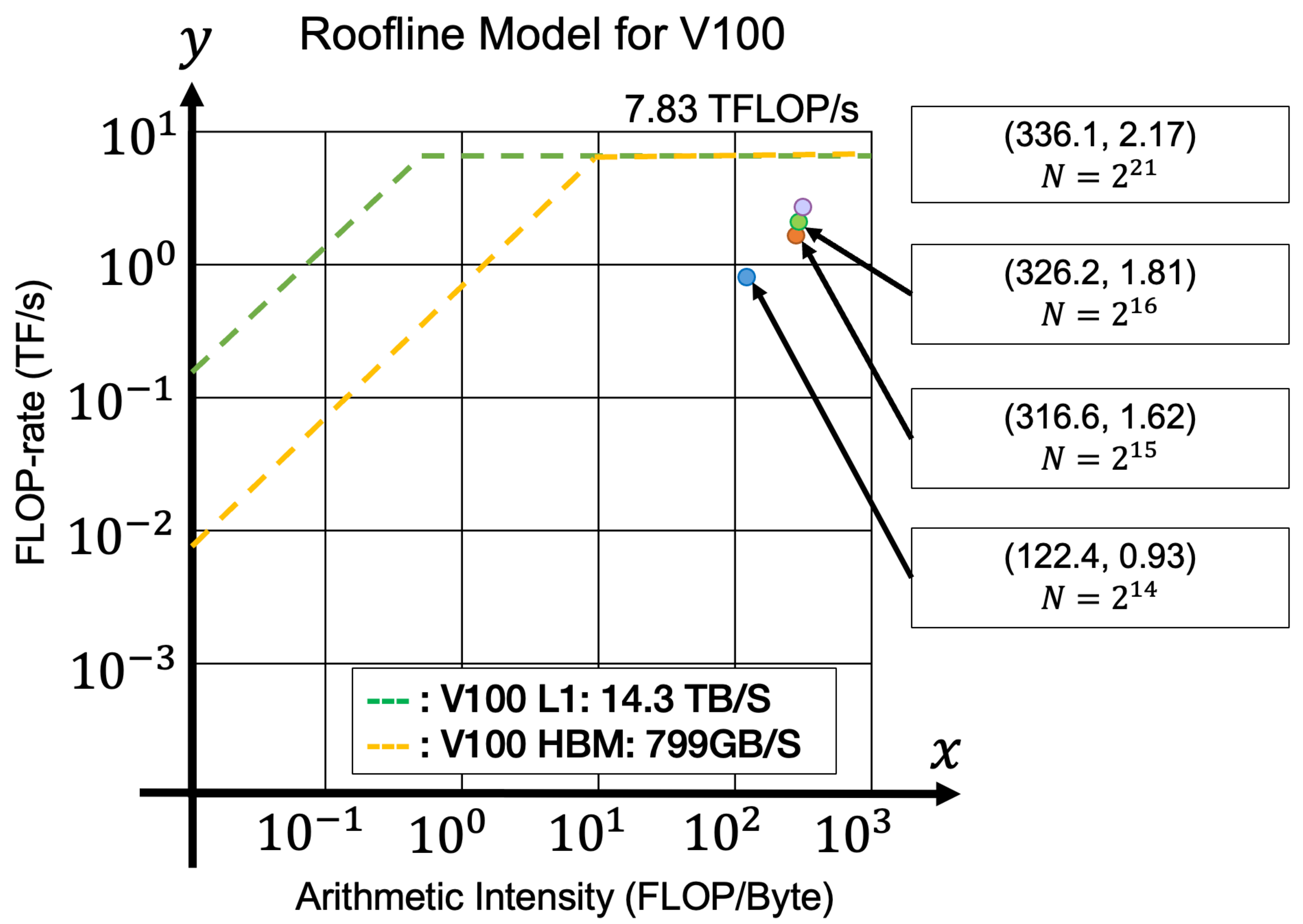




Plots for CUDA Solution



Compute Time = 77526 ms (n=2^17)



Proposed Future Work

- Experiment on cut-off implementation;
- Combine MPI + CUDA and compare performance;
- Use software to measure program memory usage;
- Develop solution for multiple types of interaction forces;
- Try OOP so that user can easily define their own particle;
- The CUDA kernel optimization strongly related to the interaction rule.

Summary

- Category: Application
- Problem: Develop a parallel framework for the general particle dynamics problem
- Performance Metric: Time (strong scaling, time breakdown, roofline model); Memory (theoretical analysis)
- Baseline: Sequential version of our solution
- Solution: (1) MPI version; (2) CUDA version
- Validation: Same input, same output; intuitive visualization
- Datasets: Randomly generated particles (w.r.t. position, velocity, acceleration, mass)
- Test bed: PACE (coc-ice-multi & coc-ice-gpu)
- Plots: (1) Strong Scaling for MPI; (2) Time breakdown for CUDA; (3) Roofline Model for CUDA

Q&A

Yu Du
yudu@gatech.edu

Changhai Man
cman8@gatech.edu

Qiang Wu
qwu350@gatech.edu

Georgia Institute of Technology

April 2023