Matrix-free Methods for Summation-by-Parts Finite Difference Operators on GPUs

Alexandre Chen, Brittany A. Erickson, Jee Whan Choi, University of Oregon
Jeremy E. Kozdon, Naval Postgraduate School (now NextSilicon)

Outline of the Talk

1. Problem description and motivation
2. SBP operators and the SBP-SAT method
3. General purpose GPU computing
4. Our matrix-free kernels for SBP operators
5. Multi-grid preconditioned conjugate gradient (CG) methods
6. Improved Initial Guess from Interpolation
7. Conclusions and Future Work

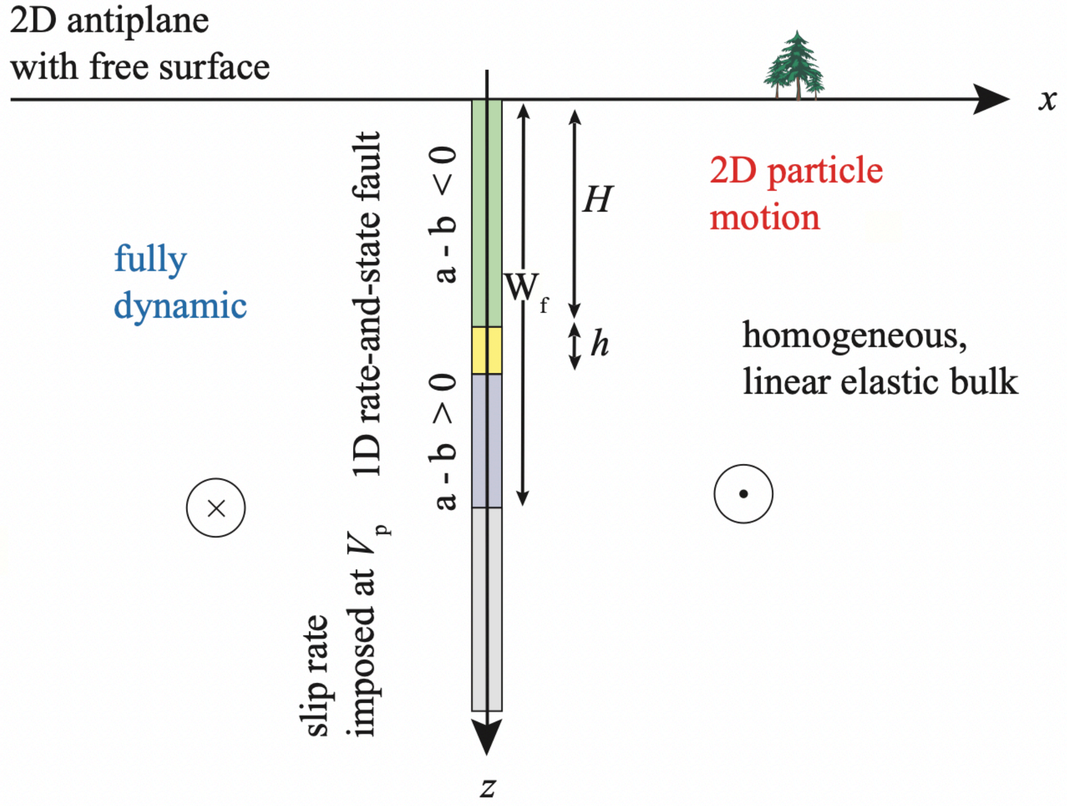
*Results shared in this talk based on Chen, A., Erickson, B. A., Kozdon, J. E., and Choi, J.-W. (2022), in revision.

Problem Description

We are solving the 2D Poisson equation motivated by large scale earthquake cycle simulations.

Simplification: Anti-plane strain, coordinate transformation for the domain

$$
\begin{aligned}
-\Delta u &= f, && \text{for } (x,y) \in \Omega,\\
u &= g_{\mathrm{W}}, && x = 0,\\
u &= g_{\mathrm{E}}, && x = 1,\\
\boldsymbol{n} \cdot \nabla u &= g_{\mathrm{S}}, && y = 0,\\
\boldsymbol{n} \cdot \nabla u &= g_{\mathrm{N}}, && y = 1,
\end{aligned}
$$



2D problem illustration. From benchmark problem 1 (BP1) description
https://strike.scec.org/cvws/seas/download/SEAS_BP1_FD.pdf

Motivations

What we want to achieve with GPU computing

- 1. How big a problem we can solve?
  - Computational Domain Size: Order of hundreds of kms with frictional length scale on the order millimeters
  - Numerical solution size (memory): ~ 1GB (~100 million unknowns)
  - Sparse matrix size (memory): ~ 10 GB
  - Memory for the factorization of the matrix: > 100 GB
- 2. Can we solve a large problem faster as well?

SBP Operators and SBP-SAT Methods

1. Summation-by-parts (SBP) operators are finite-difference operators that preserve the integration-by-parts property in a discrete form.

2. Simultaneous-approximation-terms (SAT) use weak enforcement for boundary conditions that preserve the SBP property.

Key features:

a. SBP operators are identical to central finite difference for interior points for the second derivative.

b. SBP operators mainly differ from central finite difference operators for the boundary points (both first derivative and second derivative).

c. SBP operators incorporate all boundary points into the calculation, while central finite difference methods usually work with interior points only.

$\mathbf{D}_{xx}$ is a centered difference approximation within the interior of the domain, but includes approximations at boundary points as well. For $p = 2$ it is given by the matrix

$$\mathbf{D}_{xx} = \frac{1}{h^2}\begin{bmatrix} 1 & -2 & 1 & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & 1 & -2 & 1 \end{bmatrix},$$

which, as highlighted in red, resembles the traditional Laplacian operator in the domain interior.

SBP-SAT Discretization in Matrix-explicit Form

The SBP-SAT discretization is given by

$$-\mathbf{D}_2 \boldsymbol{u} = \boldsymbol{f} + \mathbf{b}^N + \mathbf{b}^S + \mathbf{b}^W + \mathbf{b}^E,$$

where

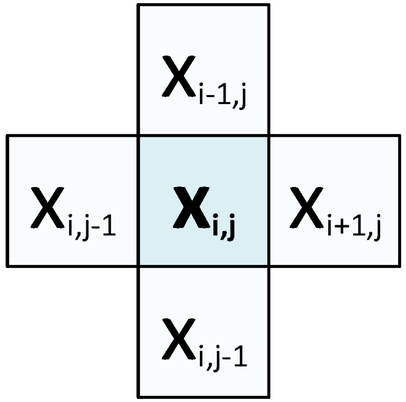
$$\mathbf{D}_2 = \left(\mathbf{I} \otimes \mathbf{D}_{xx}\right) + \left(\mathbf{D}_{yy} \otimes \mathbf{I}\right)$$

is the discrete Laplacian operator and $\mathbf{u}$ is the grid function approximating the solution, formed as a stacked vector of vectors.

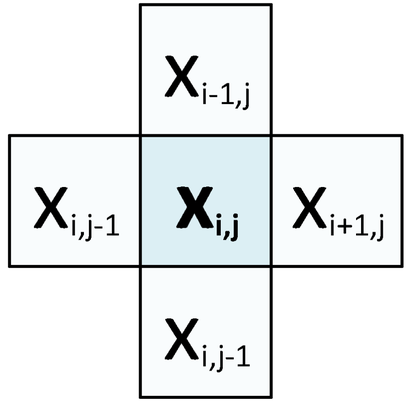
The SAT terms $\mathbf{b}^N, \mathbf{b}^S, \mathbf{b}^W, \mathbf{b}^E$ enforce all boundary conditions weakly. To illustrate the structure of these vectors, the SAT term enforcing Dirichlet data on the west boundary is given by

$$\mathbf{b}^W = \alpha \left(\mathbf{H}^{-1} \otimes \mathbf{I}\right)(\mathbf{E}_\mathrm{W}\mathbf{u} - \mathbf{e}_\mathrm{W}^T\mathbf{g}_\mathrm{W}) - \left(\mathbf{H}^{-1}\mathbf{e_0}\mathbf{d_0}^T \otimes \mathbf{I}\right)(\mathbf{E}_\mathrm{W}\mathbf{u} - \mathbf{e}_\mathrm{W}^T\mathbf{g}_\mathrm{W})$$

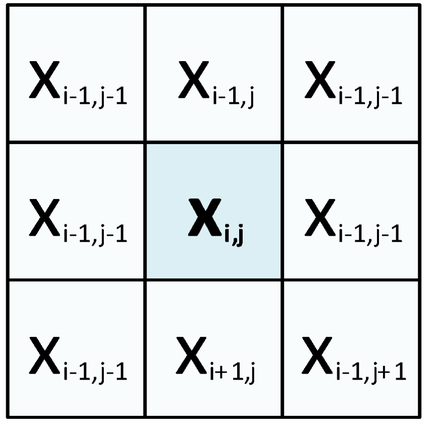
- The stencil computation is identical for interior points (similar to central finite difference)
  - Single-instruction-multiple-data (SIMD)
- The SBP operators are mainly unique for their handling of the boundary data
- Idea: split the domain and write different functions (GPU kernels)



(a) Laplace



(b) 5-point

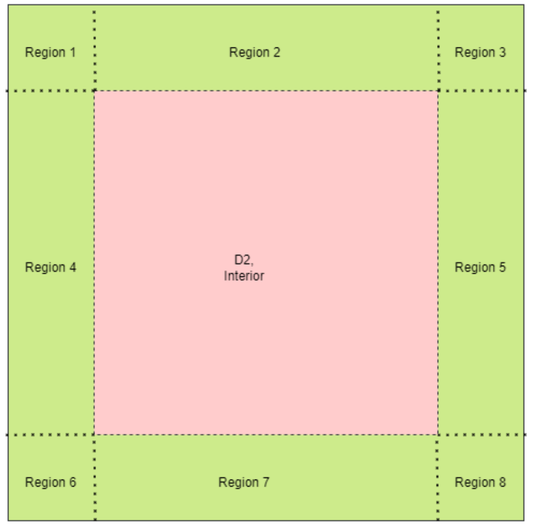


(c) 9-Point

Laplace: $x_{i,j}^{t+1} = 0.25 \cdot (x_{i,j-1}^{t} + x_{i-1,j}^{t} + x_{i+1,j}^{t} + x_{i,j+1}^{t})$

5-Point: $x_{i,j}^{t+1} = c_0 \cdot x_{i,j-1}^{t} + c_1 \cdot x_{i-1,j}^{t} + c_2 \cdot x_{i,j}^{t} + c_3 \cdot x_{i+1,j}^{t} + c_4 \cdot x_{i,j+1}^{t}$

9-Point: $x_{i,j}^{t+1} = c_0 \cdot x_{i-1,j-1}^{t} + c_1 \cdot x_{i,j-1}^{t} + c_2 \cdot x_{i+1,j-1}^{t} + c_3 \cdot x_{i-1,j}^{t} + c_4 \cdot x_{i,j}^{t}$
$+ c_5 \cdot x_{i+1,j}^{t} + c_6 \cdot x_{i-1,j+1}^{t} + c_7 \cdot x_{i,j+1}^{t} + c_8 \cdot x_{i+1,j+1}^{t}$



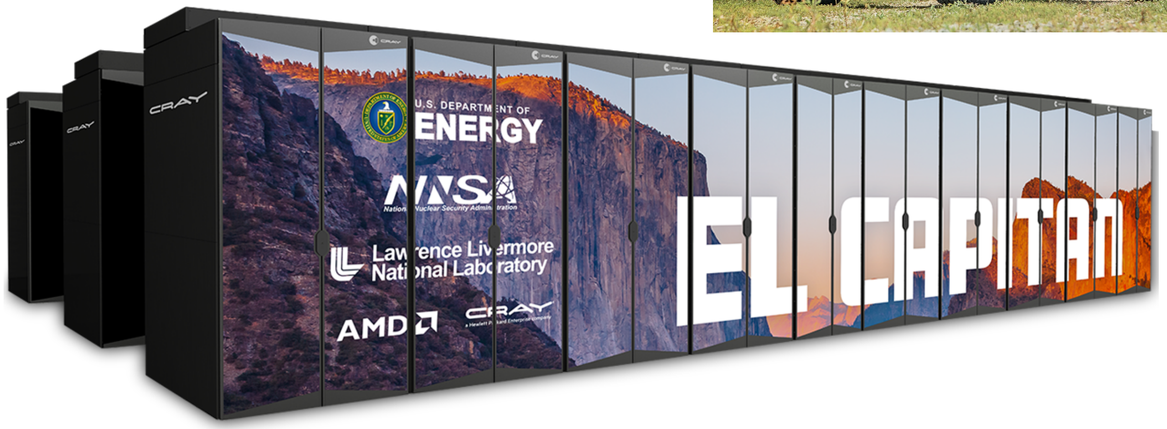
Laplacian: All regions, but only D2 interior has the same coefficients for stencil computation

Dirichlet Boundary Conditions: West (Region 1, 4, 6) and East (Region 3, 5, 8)

Neumann Boundary Conditions: North (Region 1, 2, 3) and South (Region 6, 7, 8)

General Purpose GPU Computing (GPGPU)

- GPU and CPU are optimized for different purposes
  - CPU: more versatile, low latency, small number of faster cores
  - GPU: less versatile, high latency but high throughput, large number of slower cores
- Toolkits for GPGPU:
  - CUDA (NVIDIA) is the mainstream platform
  - OpenCL (CPUs + GPUs, different vendors)
  - ROCm (AMD), oneAPI (Intel)



A Survey of General-Purpose Computation on Graphics Hardware, John D. Owens et. al

The proposed exa-scale super computer El Capitan is built using AMD CPUs and GPUs.

GPU Programming Model

- The CUDA architecture is built around a scalable array of multithreaded *Streaming Multiprocessors (SMs)* Each SM has a set of execution units, a set of registers and a chunk of shared memory.
- The basic unit of execution is the *Warp,* currently consisting of a collection of 32 threads.
- A *kernel* is a function that compiles to run on high-throughput accelerators (Eg. GPU) rather than on the host (CPU) where the main program is run on.
- Performance of a single V100 GPU (June 21, 2017)
  - 84 SMs, 5120 CUDA cores, 32 GB Memory, $10,664
  - Memory throughput: 900 GB/s for
    - Analogy: Move food from kitchen to your plate
  - Peak FLOPS: 7 teraFlops (TFOPS) for double precision
    - Analogy: Eat the food your plate

ASCI White, Fastest Supercomputer in 2000. 8192 CPUs , 6 TB memory
7.226 TFLOPS $110 million in 2000

A GPU kernel for 2D Laplacian

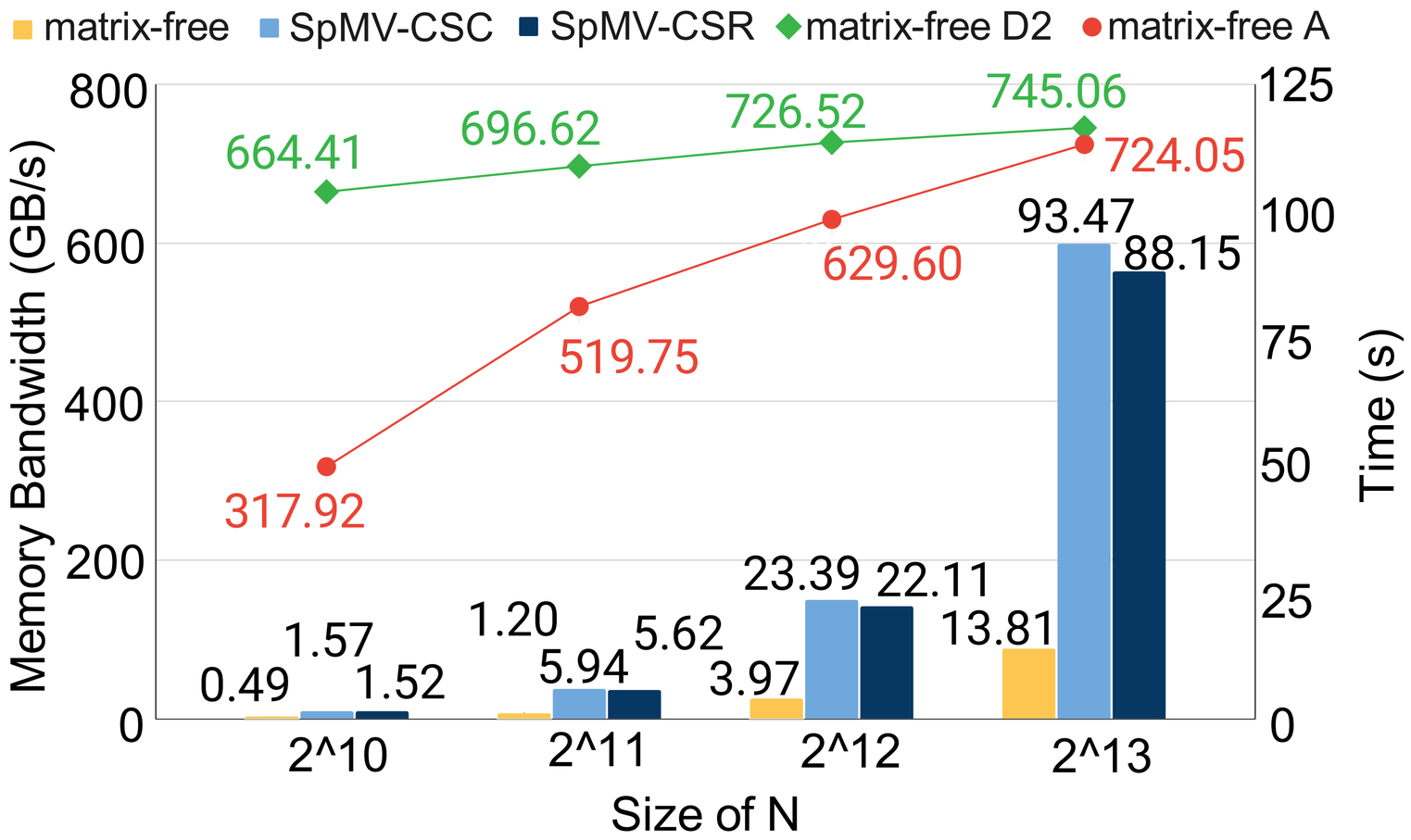
```
function D2_split_naive(idata, odata, Nx, Ny, h,::Val{TILE_DIM1}, ::Val{TILE_DIM2})
                    where {TILE_DIM1, TILE_DIM2}
    tidx = threadIdx().x
    tidy = threadIdx().y

    i = (blockIdx().x - 1) * TILE_DIM1 + tidx
    j = (blockIdx().y - 1) * TILE_DIM2 + tidy

    if i <= Nx && j <= Ny
        if i == 1 || i == Nx || j == 1 || j == Ny
            @inbounds odata[i, j] = 0
        else
            @inbounds odata[i, j] = (idata[i-1,j] + idata[i+1,j] + idata[i, j-1] + idata[i, j+1] - 4*idata[i, j])
       end
   end
   nothing
end
```

```
@cuda threads=blockdim blocks=griddim D2_split_naive(idata, odata, Nx, Ny, h, Val(TILE_DIM_1), Val(TILE_DIM_2))
```

Matrix-free GPU Kernels Performance

10000 Iterations	Matrix-free GPU Kernels					
# of grid points each direction	D2_split	SBP_N	SBP_S	SBP_W	SBP_E	Memcpy (HtoD + DtoH)
2^10+1	233.7 ms	31.4 ms	32.2 ms	23.5 ms	18.0 ms	58.8 ms
2^11+1	888.6 ms	33.8 ms	32.7 ms	23.3 ms	17.6 ms	60.0 ms
2^12+1	3.4 s	41.5 ms	33.9 ms	23.4 ms	17.4 ms	86.6 ms
2^13+1	13.4 s	54.9 ms	38.6 ms	24.0 ms	17.5 ms	171.5 ms

Matrix-free vs SpMV



Performance of SpMV kernels vs matrix-free kernels for 10000 calculations. Memory bandwidth of the matrix-free D2 kernel in green and total kernel A (the contribution of both D2 and boundary kernels) in red shown against N, where matrix is size (N+1)^2 x (N+1)^2. Also shown is SpMV total time for matrix-free (yellow), matrix-explicit CSC (light blue) and matrix-explicit CSR (dark blue).

Matrix-free Iterative Methods

Approach 1. Preconditioned Conjugate Gradient (PCG) Method

- Two-level multigrid as a preconditioner
- Direct solve on the coarsest grid
- Obtaining coarse grid operators directly rather than using Galerkin Condition
- Using standard interpolation operators with matrix-free implementations
- Using Richardson Iteration
  - $x^{k+1} = x^k + \omega(b - Ax^k)$

Two-Grid Correction Scheme

$$\mathbf{v}^h \leftarrow MG(\mathbf{v}^h, \mathbf{f}^h).$$

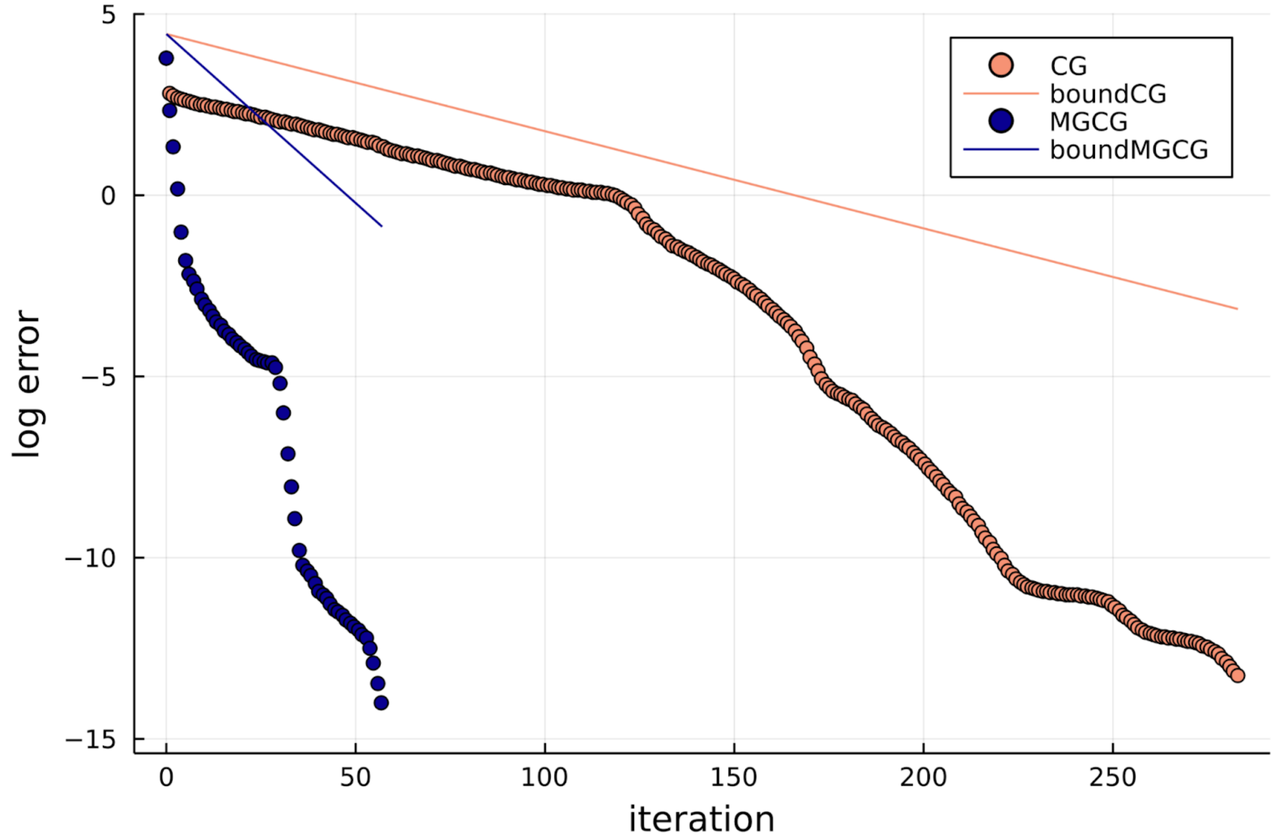
- Relax ν_1 times on $A^h\mathbf{u}^h = \mathbf{f}^h$ on Ω^h with initial guess $\mathbf{v}^h$.
- Compute the fine-grid residual $\mathbf{r}^h = \mathbf{f}^h - A^h\mathbf{v}^h$ and restrict it to the coarse grid by $\mathbf{r}^{2h} = I_h^{2h} r^h$.
- Solve $A^{2h}\mathbf{e}^{2h} = \mathbf{r}^{2h}$ on Ω^{2h}.
- Interpolate the coarse-grid error to the fine grid by $\mathbf{e}^h = I_{2h}^h \mathbf{e}^{2h}$ and correct the fine-grid approximation by $\mathbf{v}^h \leftarrow \mathbf{v}^h + \mathbf{e}^h$.
- Relax ν_2 times on $A^h\mathbf{u}^h = \mathbf{f}^h$ on Ω^h with initial guess $\mathbf{v}^h$.

A Two-level Multigrid Scheme from A Multigrid Tutorial 2nd Edition (Briggs, 2000)

Matrix-free Iterative Methods

The Convergence Analysis of PCG vs CG

- The upper bound is calculated from general convergence theory using the condition numbers of matrix A and M*A
- M is the preconditioner matrix for multigrid preconditioner



The Multigrid Preconditioned Conjugate Gradient Method, Osamu Tatebe 1993

Convergence for the CG method. Upper bound derived from Iterative Methods for Solving Linear Systems, Anne Greenbaum

Matrix-free Iterative Methods

Approach 2. Multilevel CG with Interpolated Initial Guess

- Similar to two-level multigrid, we have a coarse grid and a fine grid
- Solve the linear system on the coarse grid using the direct solve
- Interpolate the direct solve results to the fine grid as the initial guess
- Solve the linear system on the finest grid with the interpolated initial guess using matrix-free CG on GPU

N	MF-MGCG	SpMV-MGCG	BLAS-MGCG	SpMV-CG
2^{10}	2.6 s / 24	3.5 s / 24	8.0 s / 24	1.1s / 3242
2^{11}	12.6 s / 24	16.4 s / 24	39.3 s / 24	7.4 s / 6445
2^{12}	63.2 s / 23	79.0 s / 23	248.9 s / 23	53.3 s / 12303
2^{13}	323.8 s / 24	395.9 s / 24	884.5 s / 24	416.9 s / 24394

TABLE V
COMPARISON OF MGCG VS CG, TIME / CG ITERATIONS

MF-MGCG: Multigrid Preconditioned Conjugate Gradient with Matrix-Free kernels on GPU
SpMV-MGCG: Multigrid Preconditioned Conjugate Gradient using SpMV kernels on GPU
BLAS-MGCG: Multigrid Preconditioned Conjugate Gradient using BLAS on CPU
SpMV-CG: Conjugate Gradient using SpMV operators on GPU

N	MF-MLCG	SpMV-MLCG	BLAS-MLCG	Direct Solve
2^{10}	0.7 s / 1842	0.8 s / 1842	25.1 s / 1842	0.6 s
2^{11}	2.0 s / 1821	2.9 s / 1821	117.8 s / 1821	2.9 s
2^{12}	11.4 s / 3311	18.2 s / 3311	882.6 s / 3311	13.4 s
2^{13}	27.8 s / 1022	33.7 s / 1022	1141.2 s / 1022	61.1 s

TABLE VI
COMPARISON OF MULTI-LEVEL CG VS DIRECT SOLVE, TIME / EXTRA CG STEPS FOR MLCG, AND TIME FOR DIRECT SOLVE

MF-MLCG: Multilevel Conjugate Gradient using Interpolated Initial Guess with Matrix-Free kernels on GPU
SpMV-MGCG: Multilevel Conjugate Gradient using Interpolated Initial Guess with SpMV kernels on GPU
BLAS-MGCG: Multilevel Conjugate Gradient using Interpolated Initial Guess with BLAS on CPU
Direct Solve: Direct Solve excluding time for LU decomposition

Conclusions and Future Work

- Matrix-Free kernels for SBP operators not only saves memory, but can be much faster than SpMV kernels, and we achieve empirical memory bandwidth capacity
- Multigrid can be a very effective preconditioner for our problem. It can saves the number of PCG interactions significantly over CG. And it can be faster than CG. However, the speed is not ideal when compared to the direct solve
- Using interpolated initial guess can reduce the number of iterations significantly, and it can be faster than the direct solve (CPU) with parallelism on GPU. However, it is not ideal for CPU or non-parallel system
- Using machine learning to provide a better initial guess is promising