Noname manuscript No.
(will be inserted by the editor)

Matrix Free GPU Acclerated Solutions to 2D Poisson Equation Using SBP-SAT Finite Difference Operators

Alexandre Chen · Toby Harvey

the date of receipt and acceptance should be inserted later

Abstract The Poisson Equation is an elliptic partial differential equation (PDE) that is widely used in physics, fluid dynamics, and other fields to study steady-state problems. In this paper, we present our research on solving Poisson's Equation with Summation-by-Parts (SBP) operators which incorporate Simultaneous-Approximation Terms (SAT). Particularly, we explore high performance computing techniques to solve this problem for the large linear system that arises from discretization. To solve this large linear problem, we adopted iterative methods, particularly the conjugate gradient (CG) method. We use multi-threading with CPU and GPU parallelization with CUDA. We first formed our problem with sparse matrices , and then compared it with a matrix-free method. We have implemented a GPU matrix-free function with performance comparable to sparse-matrix vector multiplication (SpMV), and we did a performance analysis with results supporting that there is further optimization to be done in future work.

1 Problem statement

Poisson's equation is commonly written as:

$$-\nabla \cdot (\mathbf{c}\nabla u) = f$$

Where u is the desired function to be solved for, $\mathbf{c}$ is a matrix valued function, ∇ denotes the gradient, and f is a source function over the entire domain. We take our boundary value problem to be in 2D with diagonal and constant $\mathbf{c}$, on the unit square, with Dirichlet boundary conditions on the vertical edges and Neumann boundary conditions on the horizontal edges. Which reduces too:

$$-\left(c_{11}\frac{\partial^2 u}{\partial x^2} + c_{22}\frac{\partial^2 u}{\partial y^2}\right) + f(x,y) = 0 \qquad (x,y) \in [0,1]\times[0,1] \tag{1a}$$

$$u = g_E = \sin(\pi y) \qquad x = 0 \tag{1b}$$

$$u = g_W - \sin(\pi y) \qquad x = 1 \tag{1c}$$

$$\frac{\partial u}{\partial y} = g_S = -\cos(\pi x) \qquad y = 0 \tag{1d}$$

$$\frac{\partial u}{\partial y} = g_N = -\cos(\pi x) \qquad y = 1 \tag{1e}$$

This equation is the steady state version of the wave equation in two dimensions and models displacement u for elastic physical systems in which acceleration is negligible, and in which stress is prescribed to the north and south boundaries, and displacement is fixed on the east and west boundaries.

2 SBP-SAT Theory and Discretization

Summation by parts (SBP) finite difference operators are a high order accurate framework for solving PDEs. SBP operators have several advantages over standard second order difference operators traditionally used in science. First, in many problems of fluid dynamics and seismology high resolution solutions are required for

Address(es) of author(s) should be given

scientific application (1). Increasing the accuracy of a finite difference solution can be done by either mesh refining or increasing stencil computation widths (raising the order of the operators). It can be shown that higher order accurate difference operators with a coarser mesh are computationally more efficient than lower order methods with a fine mesh (5). Since SBP operators can be derived for arbitrary accuracy they benefit from this efficiency. Secondly, SBP operators produce provably stable spatial discretizations, which means that when implemented one does not need to be concerned with numerical instability arising from the spatial discretization. As described below the energy method is used to derive conditions on the boundary that insure stability of the PDE in the continuous problem. SBP operators mimic this derivation which always involves the use of integration by parts. The operators are required to have a summation by parts property that can be used analogously to prove discrete spatial stability. Lastly, in the SBP-SAT framework boundary conditions are enforced weakly through simultaneous approximation terms. SATs allow for discontinuities in both material properties, and within grid positioning by splitting a problem into multiple blocks and enforcing boundary conditions between blocks. Since the solution on the edge of one block is only dependent on the boundary nodes of the other block, the amount of information that needs to be pasted between blocks is also minimized, which is good for parallelizing SBP-SAT schemes over multiple computational domains.

2.1 Second Derivative SBP Operators

2.2 1D Operators

To demonstrate how SBP Operators bound the energy of the system of interest we consider the dynamic 1D analog of (1) with homogeneous (Dirichlet and Neumann) boundary data on both sides for simplicity:

$$\rho\frac{\partial^2 u}{\partial t^2} = c\frac{\partial^2 u}{\partial x^2} \qquad 0 \le x \le 1 \quad 0 \le t \le T \tag{2a}$$

$$u(x,0) = f(x) \tag{2b}$$

$$\frac{\partial u(0,t)}{\partial x} = u(1,t) = 0 \tag{2c}$$

To bound the total amount of mechanical energy within the system we can use the energy method which with no source terms and zero boundary data reveals conservation of energy. Since (2) is a linear PDE this implies stability, and well posedness assuming existence of a solution(2). Multiplying (2a) by the velocity $\frac{\partial u}{\partial t}$, and integrating by parts results in:

$$\rho\int_0^1 \frac{\partial^2 u}{\partial t^2}\frac{\partial u}{\partial t}dx = c\int_0^1 \frac{\partial^2 u}{\partial x^2}\frac{\partial u}{\partial t}dx \tag{3}$$

$$\frac{\rho}{2}\frac{\partial}{\partial t}\int_0^1 \frac{\partial u^2}{\partial t}dx + c\int_0^1 \frac{\partial^2 u}{\partial x t}\frac{\partial u}{\partial x}dx = c\frac{\partial u}{\partial x}\frac{\partial u}{\partial t}\Big|_0^1 \tag{4}$$

$$\frac{\rho}{2}\frac{\partial}{\partial t}\int_0^1 \frac{\partial u^2}{\partial t}dx + \frac{\partial}{\partial t}\frac{c}{2}\int_0^1 \frac{\partial u^2}{\partial x}dx = c\frac{\partial u}{\partial x}\frac{\partial u}{\partial t}\Big|_0^1 \tag{5}$$

$$\frac{\partial}{\partial t}\left(\frac{\rho}{2}\|v\| + \frac{c}{2}\|\sigma\|\right) = cv\sigma|_0^1 = 0 \tag{6}$$

$$\tag{7}$$

Where in the last equation we wrote the time derivative as velocity and the spatial derivative as stress denoted by σ. We found that the change in energy of the system was zero because σ is 0 on the left, and u is fixed on the right. In this problem we assume homogeneous boundary data which gives us a finite energy estimate, but problem (1) has inhomogenous boundary data. Using Duhamel's principle stability can also be shown for the inhomogenous problem, but we do not go over this (**?**).

SBP operators are designed to mimic this energy estimate discretely to insure spatial numerical stability in a semi-discretization. We discretize the domain $0 \le x \le 1$ with N evenly spaced grid points $h/N - 1$ distance apart, and keep time continuous. We let $x_i = ih, i = 0...N$, and estimate solution to u as a vector $\mathbf{u} = \begin{bmatrix} u_0 & u_1 & \cdots & u_N \end{bmatrix}^T$, where u_i denotes the approximate solution at grid point x_i. Using SBP operators the semi discretization of (2a) is:

$$\rho\frac{\partial^2 \mathbf{u}}{\partial t^2} = \mathbf{H}^{-1}(-\mathbf{M}^c + c\mathbf{BS})\mathbf{u} \tag{8}$$

Where $\mathbf{H}$ is a numerical quadrature matrix, $\mathbf{M}$ is an approximation of the second derivative on the interior and $\mathbf{BS}$ is the approximation of first derivatives on the boundary that insures the summation by parts property(7). We omit the exact deals on the constraints of these matrices such that the semi-discrete system is stable for the sake of time, but one can refer to(1). We also note that problem (1) does not have a time derivative in it, but a similar analysis in 2D follows and upon setting the time derivative to 0 a stable system is achieved. In the SBP-SAT community the exact form of each matrix is usually solved for using the symbolic algebra system Maple in which the constraints on the operators for stability can be input. In our discretization we use operators obtained in (3).

2.3 SAT enforcement of Boundary Conditions

SAT terms are used to in force boundary conditions on the system. Typically injection is used to enforce boundary conditions in finite difference methods. Injection means that boundary nodes are just set to the exact value of boundary function, but this method usually destroys the SBP property of derivative matrices. SAT terms weakly in-force boundary conditions and for our above problem the left boundary vector $\mathbf{b_0}$ and the right boundary vector $\mathbf{b_1}$ are:

$$\mathbf{b}_0 = \alpha_0 \mathbf{H}^{-1}\mathbf{e}_0 \left(\mathbf{e}_0^T \mathbf{cBSu} - u(0,t)_x\right) \tag{9}$$

$$\mathbf{b}_1 = \alpha_1 \mathbf{H}^{-1}\mathbf{e}_N(\mathbf{e}_N^T \mathbf{cu} - u(1,t)) + \beta \mathbf{H}^{-1}(\mathbf{cBS})^T \mathbf{e}_N(\mathbf{e}_N^T \mathbf{u} - u(1,t)) \tag{10}$$

Where e_0, and e_N are used to extract a single node from the discretization, and are vectors of all zeros except a 1 at their subscripted index. We again omit the details of why the boundary conditions take this form to ensure stability, but an explanation can be found in (1).

3 Discretization of the Continuous Problem

We now present a SBP-SAT discretization of the problem defined by (1). We discretize the unit square with a $N \times N$ grid of points, where $h = 1/(N-1)$ is the distance between each point in either dimension or the grid spacing, and the location of a point is $(x_i, y_j) = (ih, jh)$. For our solution vector $\mathbf{u}$, we stack vectors for each vertical set of points in the domain on top of each other so that $\mathbf{u} = \begin{bmatrix} u_{0,0} & u_{0,1} & ... & u_{N,N-1} & u_{N,N} \end{bmatrix}^T$. The corresponding 2D operators to the 1D problem can be constructed using the Kronecker product $\otimes$ in the following way assuming that c_{11}, and c_{22} are constant in space, and the grid spacing h is the same in both directions:

$$\mathbf{D}_x^2 = \mathbf{H}^{-1}(-\mathbf{M}^c + c\mathbf{BS}) \otimes I_N \tag{11a}$$

$$\mathbf{D}_y^2 = I_N \otimes \mathbf{H}^{-1}(-\mathbf{M}^c + c\mathbf{BS}) \tag{11b}$$

$$\mathbf{b}_N = \alpha_n (I_N \otimes \mathbf{H}^{-1})\mathbf{E}_N \left(I_N \otimes \mathbf{cBSu}_N - g_N\right) \tag{11c}$$

$$\mathbf{b}_S = \alpha_s (I_N \otimes \mathbf{H}^{-1})\mathbf{E}_S \left(I_N \otimes \mathbf{cBS0u}_S - g_S\right) \tag{11d}$$

$$\mathbf{b}_E = \alpha_E (\mathbf{H}^{-1} \otimes I_N)\mathbf{E}_E(\mathbf{cu}_E - g_E) + \beta(\mathbf{H}^{-1} \otimes I_N)(\mathbf{cBS} \otimes I_N)^T(\mathbf{u}_E - g_E) \tag{11e}$$

$$\mathbf{b}_W = \alpha_W (\mathbf{H}^{-1} \otimes I_N)\mathbf{E}_W(\mathbf{cu}_W - g_W) + \beta(\mathbf{H}^{-1} \otimes I_N)(\mathbf{cBS} \otimes I_N)^T(\mathbf{u}_W - g_W) \tag{11f}$$

$$\tag{11g}$$

Where $\mathbf{E}$ is a $N^2 \times N$ matrix that projects face values of the sub-scripted face back into a full N^2 length vector, and I_N is a $N \times N$ identity matrix. using these operators problem (1) can be rewritten as:

$$(H^{-1} \otimes H^{-1})((\mathbf{D}_x^2 + \mathbf{D}_y^2)\mathbf{u} + \mathbf{b}_N + \mathbf{b}_S + \mathbf{b}_E + \mathbf{b}_W + \mathbf{F}) = 0 \tag{12}$$

Where F is a vectorized sampling of the source function f in (1a). Extracting terms with u in them from the boundary terms in (18) and moving all other terms without u to the RHS of (18) results in a system of equations $\mathbf{Au} = \mathbf{b}$, where A is positive definite. For a proof refer to (4). To solve (1) we use the conjugate gradient method on this resulting system.

4 Conjugate Gradient Method

We use the following vanilla version of the conjugate gradient algorithm without using preconditioners for any speed up. The SpMV operation occurs when we calculate Au_0 and Ap_k. In the matrix-explicit method these terms are calculated with SpMV functions that we implemented, and in the matrix-free method these terms were calculated with composite matrix-free functions.

Algorithm 1: Conjugate Gradient Algorithm

CG(A,b,u_0, ϵ)
$r_0 := b - Au_0$
$p_0 = r_0$
$k = 0$
while $r_{k+1} < \epsilon$ **do**
  $\alpha_k = \frac{r_k^T r_k}{p_k^T A p_k}$
  $x_{k+1} = x_k + \alpha_k p_k$
  $r_{k+1} = r_k - \alpha_k A p_k$
  $\beta_k = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
  $p_{k+1} = r_{k+1} + \beta_k p_k$
  $k = k + 1$
end
return u_0

5 OpenMP Implementation

We implemented both sparse CSR and matrix-free Parallel CPU codes using OpenMP to solve the system in (18). In what follows we describe both implementations.

5.1 Sparse CSR

In order to run a CG implementation with a CSR sparse matrix-vector product(SpMV), and parllelized dot products and vector adds we first needed to construct the matrices for the system. We do not know of any public SBP-SAT libraries in C, so we constructed the matrices ourselves from scratch. Instead of directly constructing $\mathbf{A}$ and $\mathbf{b}$ we constructed each $N \times N$ matrix in (11) that makes up all the sub operators. Obviously if we constructed $\mathbf{A}$ and $\mathbf{b}$ directly we would save time, but if we want to solve other similar problems in a SBP-SAT scheme making each allows for some modularity. We constructed the operators in COO format by iterating over the number of spatial nodes, and directly inserting entries into the matrix. The Kronecker products can be directly computed using the modulo operator within this loop to enforce where each block in the resulting block matrix begins and ends. There are two serious draw backs to this method. First, the conversion from COO to CSR to pass the matrix to the CG solver takes time. If we directly constructed the $\mathbf{A}$ matrix in a CSR format we wouldn't need to convert it. Secondly by iterating over spatial nodes, and adding entries into the matrices, we only know the index within the row and column and values arrays as we iterate over nodes, which makes parallelization is impossible because each thread might write to the wrong index as another thread may have already incremented the COO index and moved on. Alternatively iterating over the number of non-zeros in the matrix and calculating what the spatial node is for that iteration would allow for parallization, because the indices in to COO format are already know. In the future we should construct them the latter way. While this is a non-optimal implementation it only has to be done once since we can just write matrices to file. The resulting matrix is then passed to the CG kernel, which has a parallel SpMV, add-vecs, and dot-product kernel in it.

5.2 Matrix-Free Implementation

The matrix free calculation of $\mathbf{A}$ was divided into the computation of the $\mathbf{D}$ matrices, and the pieces of $\mathbf{b_N}$, $\mathbf{b_S}$, $\mathbf{b_E}$, $\mathbf{b_W}$ which are relevant to $\mathbf{A}$. The $\mathbf{D}$ matrices both in turn individually compute their quadrature operators, interior operators, and boundary operators and add the resulting vectors together. The boundary SAT matrices do not compute their sub-operators but instead compute themselves all in a single function. Finally the result of each vector needs to be added to get the result of the $\mathbf{A}$ operator.

Due to the method of spatial discretization most of the operators are pattered so that reoccurring weights appear at intervals of N apart where N is the number of physical nodes in a single direction, with different weights filling sections in between these intervals. We elected to deal with this patterning by using modulo operators like:

```
for(int i = 0 ; i < n*n ; i++)
    if (i % n == 0 || i % n-1 == 1)
```

An alternative implementation would be to write loops that run exactly over each patterned section without any if statements, but would result in more loop declarations. We are unsure which method would be better for performance or if it would be negligible.

6 CUDA Implementation

Modern multicore CPUs contain up to several tens of cores and have a peak floating point performance of $\mathcal{O}(100)$ GFLOPs. GPUs have up to 10 thousand cores making use of massively parallel SIMD architecture. The latest Nvidia RTX 3090 delivers a stunning 936 Gbps memory bandwidth with 35.58 TFLOPS (single precision). For comparison, Intel Xeon Platinum 8284 (priced at $15460) delivers a maximum bandwidth of 131.13 Gbps with roughly 1.5 TFLOPS. The ratio of peak FLOP rate with memory bandwidth also means that computations on GPU is essentially "free" and the performance is limited by the speed with which data can be read from global memory and utilized on the GPU device.

We describe a matrix-free GPU implementation of the Conjugate Gradient (CG) method to solve problem (1). The most computationally expensive part of traditional CG iteration is the evaluation of a large sparse matrix-vector product (SpMV). Reading the sparse matrix and vector could be the bottle-neck for SpMV computation on the GPU. In our implementation here, the matrix is not stored explicitly but recalculated to represent the effect on the input data. This reduces access to global memory significantly compared to the matrix-explicit code. The downside of using a matrix-free implementation for our SBP-SAT method is that this method would involve lots of independent components, requiring more intermediate results for the matrix-free method. Thus the need to store matrices in matrix-explicit methods is compensated for by having significantly fewer intermediate memory allocations. In finite difference methods, for lower order accuracy such as the p = 2 (p denotes the order of accuracy and determines the stencil width) case, the size of the sparse matrix A is roughly 3 to 4 times of that of the vector b. The intermediate results of matrix-free methods require significantly more memory space than the sparse-matrix method. However when the order of accuracy increases to p = 6 or p = 8, the sparse matrix representation would require significantly more memory space to store all the data, while the matrix-free method will require the same amount of the memory for intermediate results. For this project, we only got to finish the lower order of accuracy case of p = 2, thus the advantage of using matrix-free might not be noticeable.

6.1 Sparse Matrix Implementation

The compressed sparse row storage format (CSR) is a very popular and general format which has been widely used in GPU implementations of Krylov subspace algorithms. In our conjugate gradient method, we implemented the SpMV function on the GPU for the CSR format to work with a CG framework. This allows us to compare the GPU performance with the CPU performance in terms of speed-up. We could also compare this implementation with the GPU matrix-free method to see the performance difference of the two schemes. For structured matrices such as the matrix A in our problem, the ELLPACK format might be more suitable. The details of the implementations will not be covered in this section since they are widely and commonly used. The performance of CG with sparse matrices on the GPU will be discussed in the Performance Results section.

6.2 Matrix-free Implementation

In the matrix-free method, the matrix A doesn't need to be stored explicitly in the algorithm. It is sufficient to evaluate the sparse matrix-vector result $y \leftrightarrow Ax$. In our sparse-matrix formulation, A is a composition of the operators on the interior and boundary data.

We start our matrix-free implementation by implementing separate SBP-SAT operators. In this section, we show second-order SBP finite difference operators in x direction D2x_shared_kernel() as an example.

Listing 1: D2x_shared_kernel() in CUDA Using Shared Memory

```
template <class T>
__global__ void
D2x_shared_kernel(T* idata, T* odata, int Nx, int Ny, double h){
    const int H_W = 2; // for halo width
    const int TD_1= 4; // for tile dimension in x direction
    const int TD_2 = 4; // for tile dimension in y direction
    __shared__ double tile[TD_1][TD_2 + 2 * H_W];

    unsigned tidx = threadIdx.x;
```

```
    unsigned tidy = threadIdx.y;

    unsigned i = blockIdx.x * blockDim.x + tidx;
    unsigned j = blockIdx.y * blockDim.y + tidy;
    unsigned global_index = i + j * Ny;

    // for tile indexing
    unsigned int k = tidx;
    unsigned int l = tidy;
    // Writing global memory to shared memory
    // for tile itself

    if ((k<TD_1) && (l<TD_2) && (i<Ny) && (j<Nx)){
        tile[k][l+H_W] = idata[global_index];
    }
    // for left halo
    if ((k<TD_1) && (l<H_W) && (i<Ny) && (j>=H_W) && (j<H_W+Nx)){
        tile[k][l] = idata[global_index - H_W * Ny];
    }
    // for right halo
    if ((k<TD_1) && (l>=TD_2-H_W) && (l<TD_2) && (i<Ny) && (j<Nx-H_W)){
        tile[k][l+2*H_W] = idata[global_index + H_W*Ny];
    }
    // Starting finite difference calculation and writing data to output.
    // left boundary for second order
    if ((k<TD_1) && (l + H_W < TD_2 + 2*H_W - 2) && (i < Ny) && (j == 0)){
        odata[global_index] = (tile[k][l + H_W]
                              -2*tile[k][l+H_W+1] + tile[k][l+H_W+2]) / (h*h);
    }
    // center data for second order
    if ((k<TD_1) && (l+H_W<TD_2+2*H_W-1) && (i<Ny) && (j>=1) && (j<Nx - 1)){
        odata[global_index] = (tile[k][l + H_W - 1]
                              - 2*tile[k][l+H_W] + tile[k][l+H_W + 1]) / (h*h);
    }
    // right boundary for second order
    if ((k < TD_1) && (2 <= l+H_W) && (l+H_W < TD_2+2*H_W) && (i < Ny) && (j == Nx-1)){
        odata[global_index] = (tile[k][l+H_W - 2]
                              - 2*tile[k][l + H_W - 1] + tile[k][l+H_W])/ (h*h);
    }
    __syncthreads();
}
```

We used shared memory to load data from global memory and re-use the loaded data in order to reduce the memory I/O between the host and the device. When commuting finite difference stencils a halo region at the boundary of a block must be loaded so that boundary threads can still do computations. For second-order SBP operators, we need to set the halo width to be the value of p=2. We map global_index of grid points in the real domain to indices on the GPU. We use the thread index for writing data from global memory to a shared memory tile on the GPU and compute the resulting stencils using these loaded data. Because our finite difference operators involve both data indices of globa_index and local indices on shared memory, we use up to five Boolean values in if condition loops. The halo width and tile dimension are hard-coded into our kernels, however, they should be better used as input variables for better testing and optimization of kernel performances. Following the same pattern, we've implemented all SBP kernels for p=2. We've done extensive test on our kernels to make sure they output the same results as corresponding sparse-matrix operators, however, there can be still indexing bugs in our code that we failed to capture in our test. Once all SBP operators are completed, we can assemble our matrix-free function similar to how we assemble sparse-matrix operators. The code is in listing 2

Listing 2: Matrix_free_GPU() in CUDA

```
void Matrix_free_GPU(double* idata,
    double* d_out_d2x, double*d_out_d2y,
    double* d_out_bysy1,double* d_out_v2f_1,double* d_out_hyinv1,
    double* d_out_bysy2,double* d_out_v2f_2,double* d_out_hyinv2,
    double* d_out_bxsx3,double* d_out_v2f_3,double* d_out_hxinv3_a,double* d_out_hxinv3_b,
    double* d_out_bxsx4,double* d_out_v2f_4,double* d_out_hxinv4_a,double* d_out_hxinv4_b,
    double* d_out_all, double* d_out_hy, double* d_out_hx,
    int Nx, int Ny, int N, double h, int TILE_DIM_1, int TILE_DIM_2,
    double alpha1, double alpha2, double alpha3, double alpha4, double beta)
{
```

```
    dim3 dB2dx(TILE_DIM_1, TILE_DIM_2, 1); // 2d GPU block for operators in x direction
    dim3 dG2dx(Nx/TILE_DIM_1 + 1, Ny/TILE_DIM_2 + 1,1); // 2d GPU grid for x direction
    dim3 dB2dy(TILE_DIM_2, TILE_DIM_1, 1); // 2d GPU block for operators in y direction
    dim3 dG2dy(Nx/TILE_DIM_2 + 1, Ny/TILE_DIM_1 + 1,1); // 2d GPU grid for y direction

    dim3 dB1d(256,1,1); // 1d GPU block
    dim3 dG1d(N/256+1,1,1); // 1d GPU Grid

    dim3 dB_2d(16,16); // 2d block for kernels without shared memory
    dim3 dG_2d(Nx/16+1, Ny/16+1); // 2d grid for kernels without shared memory

    D2x_shared_kernel<double><<<dG2dx, dB2dx>>>(idata,d_out_d2x,Nx,Ny,h);
    D2y_shared_kernel<double><<<dG2dy, dB2dy>>>(idata,d_out_d2y,Nx,Ny,h);

    // for face 1: North
    D2y_shared_kernel<double><<<dG2dy,dB2dy>>>(idata,d_out_bysy1,Nx,Ny,h);
    BySy_shared_kernel<double><<<dG2dy,dB2dy>>>(idata,d_out_bysy1,Nx,Ny,h);
    VOLtoFACE_shared_kernel<double><<<dG1d,dB1d>>>(d_out_bysy1,d_out_v2f_1,Nx,Ny,h,1);
    Hyinv_shared_kernel<double><<<dG2dy,dB2dy>>>(d_out_v2f_1,d_out_hyinv1,Nx,Ny,h);

    // for face 2: South
    VOLtoFACE_shared_kernel<double><<<dG1d, dB1d>>>(d_out_bysy1,d_out_v2f_2,Nx,Ny,h,2);
    Hyinv_shared_kernel<double><<<dG2dy, dB2dy>>>(d_out_v2f_2,d_out_hyinv2,Nx,Ny,h);

    // for face 3: West
    VOLtoFACE_shared_kernel<double><<<dG1d,dB1d>>>(idata,d_out_v2f_3,Nx,Ny,h,3);
    BxSx_tran_shared_kernel<double><<<dG2dx, dB2dx>>>(d_out_v2f_3,d_out_bxsx3,Nx,Ny,h);
    Hxinv_shared_kernel<double><<<dG2dx, dB2dx>>>(d_out_bxsx3,d_out_hxinv3_a,Nx,Ny,h);
    Hxinv_shared_kernel<double><<<dG2dx, dB2dx>>>(d_out_v2f_3,d_out_hxinv3_b,Nx,Ny,h);


    // for face 4: East
    VOLtoFACE_shared_kernel<double><<<dG1d, dB1d>>>(idata,d_out_v2f_4,Nx,Ny,h,4);
    BxSx_tran_shared_kernel<double><<<dG2dx, dB2dx>>>(d_out_v2f_4,d_out_bxsx4,Nx,Ny,h);
    Hxinv_shared_kernel<double><<<dG2dx, dB2dx>>>(d_out_bxsx4,d_out_hxinv4_a,Nx,Ny,h);
    Hxinv_shared_kernel<double><<<dG2dx, dB2dx>>>(d_out_v2f_4,d_out_hxinv4_b,Nx,Ny,h);

    // assemble them using vec_add_gpu
    vec_add_gpu(N,1.0,d_out_d2x,d_out_d2y,d_out_all);
    vec_add_gpu(N,alpha1,d_out_hyinv1, d_out_all, d_out_all);
    vec_add_gpu(N,alpha2,d_out_hyinv2 , d_out_all, d_out_all);
    vec_add_gpu(N,beta, d_out_hxinv3_a , d_out_all, d_out_all);
    vec_add_gpu(N,alpha3, d_out_hxinv3_b , d_out_all, d_out_all);
    vec_add_gpu(N,beta, d_out_hxinv4_a , d_out_all, d_out_all);
    vec_add_gpu(N,alpha4, d_out_hxinv4_b , d_out_all, d_out_all);
    vec_add_gpu(N,-2.0, d_out_all, d_out_all, d_out_all);
    Hy_shared_kernel<double><<<dimGrid_2d_y, dimBlock_2d_y>>>(d_out_all,d_out_hy,Nx,Ny,h);
    Hx_shared_kernel<double><<<dimGrid_2d_x, dimBlock_2d_x>>>(d_out_hy,d_out_hx,Nx,Ny,h);
}
```

In this function, Nx, Ny, N, h, alpha1, alpha2, alpha3, alpha3, beta are values associated with our method (11). We call each kernel with information containing grid numbers in Nx and Ny to perform related finite difference operations. This is equivalent to generating sparse-matrix representations of these finite difference operators with grid spacing information and then use these sparse-matrices for the SpMV calculation. After all finite difference results were calculated, we use vec_add_gpu() function to assemble them together. In the final step, we apply the Hy and Hx. As can be easily seen, this implementation is not efficient and is mainly for debugging purposes to make sure our matrix-free function gives the same output with our sparse-matrix function in each step. The issue with this implementation is that there are too many intermediate memory allocations and non-efficient GPU kernel calls. We will further discuss this issue in 7.3. The final d_out_hx is the equivalent output of a SpMV function with sparse-matrix A. With this implementation, we were able to form a matrix-free function that achieves the same effect as the sparse-matrix formulation on given input u. We can develop an associated cg_gpu_matrix_free() function using our implementation here. We achieve the same level of accuracy with our sparse-matrix GPU results and comparable performance. The slight difference indicates there might be rounding error in floating point arithmetic or bugs in our matrix-free implementations.

# of elements in single dimension	SpMV error	SpMV rate	matrix-free error	matrix-free rate
4	3.109916e-02	NA	3.109916e-02	NA
8	7.086568e-03	2.065776	7.086568e-03	2.065776
16	1.735038e-03	2.015004	1.735038e-03	2.015004
32	4.318985e-04	2.003097	4.318985e-04	2.003097
64	1.078903e-04	2.00055	1.078903e-04	2.00055
128	2.698288e-05	1.99971	2.698288e-05	1.99971
256	6.748052e-06	1.99989	6.748052e-06	1.99989

Table 1: Convergence rates for CPU implementations

7 Performance Results

7.1 Verification of Correct Solution

In order to verify that our code is in fact calculating the correct solution to (1) we use the Method of Manufactured Solution (MMS). MMS is a technique in which a guess is made to the solution of (1), we refer to it as $u_e(x, y)$. u_e's derivatives are then plugged into (1a), and the source term f is solved for. In other words we show that u_e is a solution to (1) if the source term is in fact the one we just solved for. We then insert a discrete source function at each grid point into the system $\mathbf{Au} = \mathbf{b}$. This is equivalent to adding $f(x_i, y_i)$ to every corresponding stacked point in $\mathbf{b}$. This scheme allows us to find the error between a vertorized version of u_e, and the numerical solution $\boldsymbol{u}$ with a source function added to the numerical scheme. Our guess for u_e and the corresponding source function were:

$$u_e(x, y) = \sin(\pi x + \pi y) \tag{13}$$

$$f(x, y) = -2\pi^2 \sin(\pi x + \pi y) \tag{14}$$

We calculate the error between $\mathbf{u}$ and $\mathbf{ue}$ with the norm induced by $H \otimes H$, which approximates the volume integral over the domain, as :

$$error = \sqrt{(\mathbf{u} - \mathbf{ue})(H \otimes H)(\mathbf{u} - \mathbf{ue})} \tag{15}$$

We calculate the convergence rate for second order operators with:

$$rate = \log_2 \left(\frac{error_h}{error_{h/2}} \right) \tag{16}$$

Theoretically this rate should asymptotically converge to 2 for second order operators (6). We check our results for the CPU implementations in Table 1. For the GPU implementation, we get a CG solution accuracy of 1e-11, which is far below the errors shown in the table. The error should be similar to the numbers in the table with the same convergence rate and we won't be listing them here.

7.2 CPU Results Analysis

To test performance for the CPU implementations we timed the CG solver doubling the number of physical nodes in each direction (see Figure 1). The SpMV CG method significantly out performs the matrix-free implementation. While there is no time complexity reason why matrix-free methods should or should not out perform SpMV (both are $O(n^2)$ where n is the dimension of the matrix) we have some hypothesis as to why the matrix free implementation is so much slower and a similar analysis for the GPU kernels is also done. With every intermediate kernel called to get the operator of a single matrix we must iterate over the size of the matrix, and potentially do an add_vecs() to combine the operators. For our current CPU implementation to calculate the $\mathbf{D}$ matrices 4 iterations of the matrix length are done each for a total of 8. Another 3 iterations must be done for add_vec()'s for both operators for a total of 12 iterations. Each boundary operation is done in one iteration for a total of 16, and then must be added back together in the end for a total of 22 iterations. On the other hand SpMv must only do one iteration, and only 1 function call. Ideally all the matrix-free functions would be merged into one, and in future work this should be done.

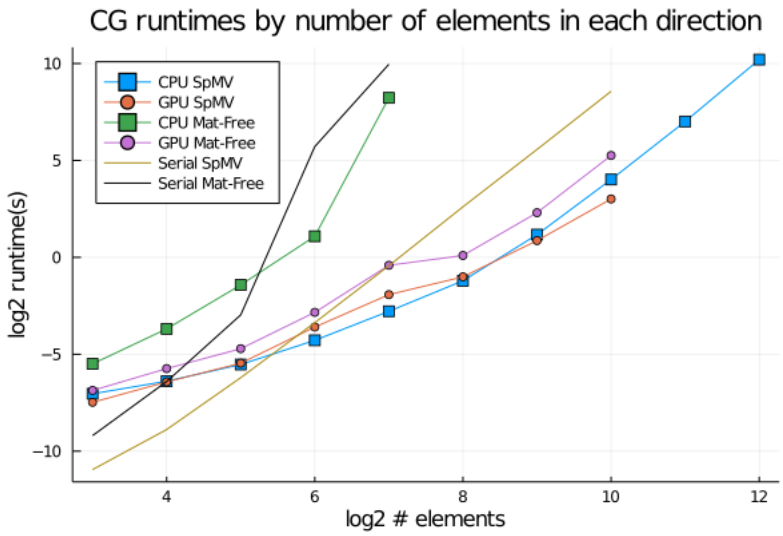

Fig. 1: CPU vs unoptimized GPU run-times with increasing of elements

7.3 GPU Results Analysis and Optimization

Our unoptimized matrix-free CG code achieves comparable performance with our sparse-matrix CG code, with a performance difference of 3 to 4 times. It appears as though our matrix-free CG code could be further optimized to a level that could exceed our sparse-matrix CG code for even the case of p=2 case. Given the characteristic of our matrix-free method, it has the potential of having more significant advantages over the sparse-matrix method in future implementations. We will present these analysis and work in this subsection.

For the CG algorithm, the SpMV calculation or our matrix-free function is the most computationally expensive part. The total time for CG should almost be equivalent to the time for these operations if the rest of the code were implemented optimally. Whether CG converges or not, the total cost of the computation should be the same once the problem size is given and the actual iteration time is fixed. To study the effect of different operators in our matrix_free_gpu() function, we need to choose a problem that is large enough so we will have enough iterations before the convergence is reached. For our problem, we choose Nx = Ny = 1025. Our sparse matrix has the size of 1050625 by 1050625 and our RHS of the equation is an array of length 1050625. In general, the maximum number of the iterations is set to be the dimension of the matrix, hence, 1050625. The tolerance is usually set to be the machine epsilon for given floating point types, so we set it to be 1e-16 here. For performance analysis we want to set our max_iter to be much lower than the matrix dimension to ensure we've run the same number of iterations before CG quits. In this test, we set it to be 2,000. We choose 123 to be our random seed.

7.3.1 Tile Size in Kernels with Shared Memory

The first important parameters in the GPU kernels using shared memory are the shared memory tile dimension. In this problem, we hard-coded them to be 4 by 4. This size is too small to fully utilize the resources available on our V100 GPU. However, when we test different tile size, the total time of CG didn't change. While we were only testing the performance of individual kernels, the tile dimension matters. Ideally, for operators in the x direction, we should launch tiles that could store more data in the x direction for reducing the percentage of the halo region compared to the total size of the shared memory. In this way, we could do more useful data loading and finite difference calculations. But in the test, there is no significant difference between tile shape tile itself having size 4 by 16 vs 16 by 4 vs 8 by 8. This could be due to memory coalescing issues since we stack our points in the y direction and reading neighboring data in the x direction isn't efficient. Compared to the shape, it's the total size of the tile that matters because this is directly associated with the GPU architecture. In our test, when the total area of the shared memory is 64, the performance of our kernels is the most optimal. Further increasing or decreasing the total area of shared memory would decrease the performance. Combined with the fact that the performance of the CG function almost doesn't change, we could guess that the computation itself doesn't account for the majority of the runtime. This is confirmed with further tests.

7.3.2 Reducing Kernels and Intermediate allocations

In our current implementation, we have 2 kernels for doing second derivative operations in x and y: D2x_shared_kernel and D2y_shared_kernel. Then in final steps, we use vec_add_gpu() function to add these two and write it into

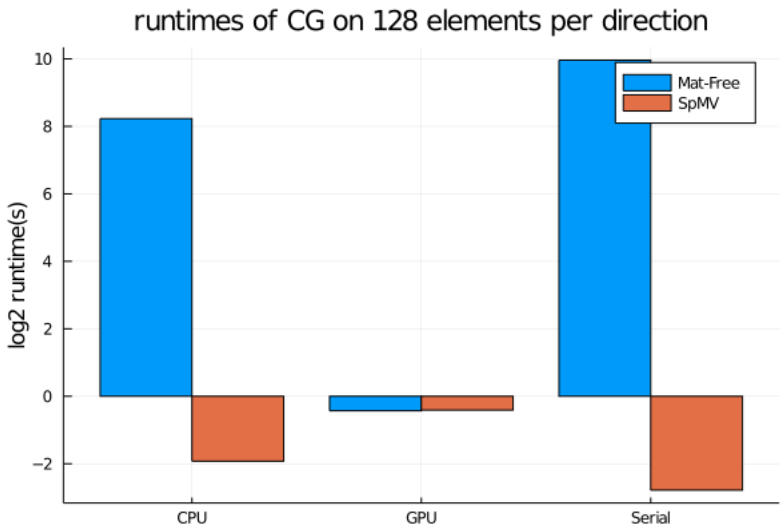

Fig. 2: Differences in run-time for 16641 node simulation where CPU implementations were run on a 28 core system against unoptimized GPU code.

d_out_all. These two can be easily merged into one kernel D2_kernel with direct output to d_out_all. Given that rewriting a shared memory version would require significantly more work and is prone to indexing errors, here we only implemented a naive version without using shared memory on GPU. When we use this one GPU kernel call to replace three GPU kernel calls we see a significant performance boost. In the original code, for our selected matrix and vectors, CG would cost 1.128 s for sparse-matrix SpMV on GPU and 4.029 for matrix-free on GPU. After replacing this, we observe 1.125 s for sparse-matrix SpMV on GPU and 3.645 s for matrix-free on GPU. We save 9.5% of time by doing simply this merging. Wealso measure the total time for this D2_kernel as only 0.071 s. This is surprisingly fast because the D2 operators would involve all data for a given grid size, and the complexity of D2 is $\mathcal{O}(N^2)$, compared to the rest of the operators only working with boundary data with the complexity of $\mathcal{O}(N)$.

Another observation is that in the final stage of merging kernels, to take the minus value of the given function, we use the vec_add_gpu to add -2.0 times of d_out_gpu to itself. Removing this line would change the matrix-free effect on a given input data by a minus sign. It would not affect the convergence of the CG itself. The only difference is that the CG output has an offset of a minus sign. Removing this line would further reduce the total time from 3.645 to 3.598 second. We already achieves more than 10% speedup from our previous implementation with very little changes to our code.

To further explore the different components for our CG code and their cost to see how we could optimize our future implementation, we gradually comment out difference sections of the matrix_free_gpu() code. We divide these operators and functions into four categories:

1. D2_kernel: second order differential operator on all boundary interior data, complexity: $\mathcal{O}(N^2)$
2. Other operators involving only boundary data, complexity: $\mathcal{O}(N)$
3. vec_add_gpu: Merging all GPU kernel outputs, complexity: $\mathcal{O}(N)$
4. Rest of cg_gpu_matrix_free(): Other parts of the CG code, with ddot_gpu() using a reduction kernel being the most expensive, complexity: $\mathcal{O}(N)$

Notice that in our test here, we only care about the cost of these operations. Notice that we could not achieve convergence by simply commenting them out without proper replacements. Since we make sure in previous convergence tests, we did all 10,000 iterations. The performance result should be independent from the convergence result. We list the time in the following table:

	D2	Other Operators	Vec_add_gpu	Other Parts of CG	CG Matrix Free	CG Sparse Matrix
Time	0.075	2.366	0.416364	0.73815	3.596	1.126
Percentage	2.09%	65.80%	11.58%	20.53%	100.00%	31.31%

Table 2: Performance of different components of the matrix-free CG code

From the table, we could observe that in our matrix-free implementation D2, which should the be most expensive operation, but only accounts for 2 % of the total time. This indicates our code is far from optimized. The rest of the $\mathcal{O}(N)$ operations account for the majority of the time. We could further improve this part by

merging kernels. Operations on four boundaries could be merged into four separate kernels, and the final stage of using Hx and Hy to provide grid space info can be merged into previous kernel calls. And all vec_add_gpu() kernel can be merged into one big vec_nadd_gpu() function that does vector addition for variable numbers of input data. Higher-level observation of this problem shows that each boundary operations are independent. Thus we could merge them all into one function call that directly, and output the results of the combination of these separate function calls. Thus the total amount of kernel calls would be reduced to only 3, down from the current 23 kernel calls. With these implementations, we should be able to observe that our matrix-free implementation on GPU outperforms the sparse-matrix implementation on GPU. For matrix-explicit formulations, the majority of the entries are associated with D2 operator. If we assume that the rest part of the sparse-matrix CG code excluding SpMV operation costs the same amount of the time with the rest part of matrix-free CG code excluding matrix_free_gpu() function, then the SpMV in Sparse-Matrix representation would require roughly 0.388s, which is five times more than our naive D2 kernel here. This part could be further improved by using sparse matrix representations such as ELL for structured sparse arrays such as ours from finite difference formulation.

7.3.3 Other Parts of the CG Algorithm

We've mostly focused on optimizing and analyzing our matrix-free kernels and function for CG. It would be also worthwhile to look into the rest of the parts of the CG code excluding the SpMV function or equivalent matrix-free. While the cost of the rest of the parts should be trivial given their complexity, the actual percentage of the total time is not. They are far exceeding the cost of D2 which theoretically speaking should be much more expensive. There could be several issues here:

1. The implementation of ddot_gpu() function is not optimized. Since this function would repeatedly call reduction kernel, and our implementation could be not optimal, and also there is data copying from GPU to CPU in this function, which could be the bottleneck
2. The conditional loops could be the bottle neck. There is no way we could optimize this part. But since our SBP-SAT method has proven stability, we could predict the total iteration steps we need for the given problem size. Also the CG accuracy is not equivalent to the solution accuracy compared to the analytical solution, we do not actually need CG to achieve 1e-12 level accuracy while the direct solve for small amount of grid points could only give 1e-4 level of accuracy as shown in table 1. While this part is more empirical than analytical, requiring more pure HPC knowledge.

7.4 Results After Partial Optimzation

After we finished doing the optimizations explained above, we retested our matrix-free operations against our sparse-matrix representations. Since removing the awkward vec_add_gpu() function call to take the inverse, our floating point precision has been improved. We now have much better accuracy and require fewer steps to converge compared with our sparse-matrix CG code. As a result, our matrix-free CG code on the GPU now requires less time compared to the SpMV CG code.

Nx - 1	Max_iter	CSR Time (s)	Matrix-Free Time (s)	Sparse accuracy	Matrix-free accuracy	Sparse Iter	Matrix-free Iter
8	81	0.005728	0.002812	1.38E-14	6.87E-15	66	15
16	289	0.011589	0.009104	1.22E-11	3.44E-12	135	55
32	1089	0.023284	0.018817	1.26E-10	2.33E-11	266	119
64	4225	0.044117	0.040855	8.63E-10	1.43E-10	525	265
128	16641	0.159109	0.099452	8.83E-09	1.16E-09	1714	589
256	66049	1.049618	0.442895	1.18E-07	1.01E-08	9857	1836
512	263169	1.801014	1.790442	8.44E-07	1.03E-07	9029	3396
1024	1050625	8.07943	5.432265	1.46E-11	3.55E-12	14434	2996

Table 3: Results after Partial Optimization for Matrix-free CG Code on GPU

8 Future Work and Conclusions

Matrix-free representations can utilize the architectural aspects of the GPU where the computation on the GPU is almost "free" when you look at its peak FLOP and Bandwidth. Even though the GPU has a huge bandwidth, but it's not optimized for latency, hence why we should mind doing repeated calculations. And this is the motivation for exploring matrix-free method on GPU in this paper. Although we didn't have enough time to finish our matrix-free operators for p=4, it would be exciting to their performances compared to sparse-matrix methods. Because at p=4, sparse-matrix representation would require two times more data storage, and thus increase the I/O pressure, while for matrix-free methods the only increase is the number of computations

for each kernel and the computation is almost "free" on GPU. We already have sparse matrix CG code for ELL format, however we do not have enough time to test its performance. It would be also interesting to see how this performed against other implementations. Even though our existing implementations show that the matrix-free second-order differential operator has the a potential huge advantage over the sparse-matrix second-order differential operator in terms of the performance, we couldn't ignore the following drawbacks of the matrix-free methods:

8.1 More Conditional Loops in Higher Order Operators

Although for higher order finite difference operators, matrix-free methods are promising given their characteristics, in our problem, the loop conditions are already very complex for p = 2. If we wants to have different operations for each different types, we would have many different loop conditions with different combination of Boolean values based on the global index and local thread index, and the GPU is not efficient for such loop conditions. With recent improvements in memory performance for new GPU architectures, we could get away without using shared memory. However, simply ditching this might not be enough. Since in our D2_kernel, we have nine different loop conditions already. If we use s similar approach, we would be facing tens of different loop conditions in p=4. One way that we could reduce the number of loop conditions is to do more calculations for one loop type. For example, once the loop condition determines the global_index = 0, we know this point is the beginning point, and we know the operations on its neighboring points. We could do operations on these points within the current loop condition. Determining how many neighboring points for which we should calculate their output requires more research and implementations, however this should be able to help decrease the difficulty for writing matrix-free operators for higher-order operators.

8.2 Matrix-free Methods are Harder to Implement

For sparse-matrix methods, once we implement the CG method on GPU, all we need to do is to generate sparse-matrix representations of our data, and use the same CG code to do calculations on these data. While for matrix-free method, if you want to implement optimized code, you need to do lots of manual optimization of the code to achieve optimal performance. This has increased not only the difficulty in development, but also difficulty in debugging. This has been a serious difficulty in this project, and to make full use of the matrix-free method, a better abstraction of the problem and writing code with higher-levels of abstraction would be helpful.

Another argument against matrix-free methods is that there already exist lots of pre-conditioners for sparse-matrix CG, while in matrix-free methods, there aren't available pre-conditioners to our knowledge. It's possible to implement your own pre-conditioners for your own CG, which has been done previously. (8) However, it certainly adds complexity to the already much more complex implementation of matrix-free methods.

References

1. Del Rey Fernández, D. C., Hicken, J. E., and Zingg, D. W. Review of summation-by-parts operators with simultaneous approximation terms for the numerical solution of partial differential equations. *Computers Fluids 95* (2014), 171 – 196.
2. Erickson, B. A., and Dunham, E. M. An efficient numerical method for earthquake cycles in heterogeneous media: Alternating subbasin and surface-rupturing events on faults crossing a sedimentary basin. *Journal of Geophysical Research: Solid Earth 119*, 4 (2014), 3290–3316.
3. Gustafsson, B. *High order difference methods for time dependent PDE*, vol. 38 of *Springer Series in Computational Mathematics*. Springer, 2008.
4. Kozdon, J. E., Erickson, B. A., and Wilcox, L. C. Hybridized summation-by-parts finite difference methods.
5. KREISS, H.-O., and OLIGER, J. Comparison of accurate methods for the integration of hyperbolic equations. *Tellus 24*, 3 (1972), 199–215.
6. LeVeque, R. J. *Finite Difference Methods for Ordinary and Partial Differential Equations*. Society for Industrial and Applied Mathematics, 2007.
7. Mattsson, K. Summation by parts operators for finite difference approximations of second-derivatives with variable coefficients. *J. Sci. Comput. 51*, 3 (June 2012), 650–682.
8. Müller, E., Guo, X., Scheichl, R., and Shi, S. Matrix-free gpu implementation of a preconditioned conjugate gradient solver for anisotropic elliptic pdes. *Comput. Vis. Sci. 16*, 2 (Apr. 2013), 41–58.