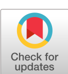

PDF Download
3650200.3656614.pdf
11 February 2026
Total Citations: 1
Total Downloads: 1130

Latest updates: https://dl.acm.org/doi/10.1145/3650200.3656614

Published: 30 May 2024

Citation in BibTeX format

ICS '24: 2024 International Conference on Supercomputing
*June 4 - 7, 2024*
*Kyoto, Japan*

Conference Sponsors:
SIGARCH

RESEARCH-ARTICLE

# Matrix-free SBP-SAT finite difference methods and the multigrid preconditioner on GPUs

**ALEXANDRE CHEN**, University of Oregon, Eugene, OR, United States

**BRITTANY A ERICKSON**, University of Oregon, Eugene, OR, United States

**JEREMY E KOZDON**, Naval Postgraduate School, Monterey, CA, United States

**JEEWHAN CHOI**, University of Oregon, Eugene, OR, United States

**Open Access Support** provided by:

**University of Oregon**

**Naval Postgraduate School**

ICS '24: Proceedings of the 38th ACM International Conference on Supercomputing (May 2024)
https://doi.org/10.1145/3650200.3656614
ISBN: 9798400706103

# Matrix-free SBP-SAT finite difference methods and the multigrid preconditioner on GPUs

Alexandre Chen
University of Oregon
Eugene, Oregon, USA
yiminc@uoregon.edu

Brittany A. Erickson
University of Oregon
Eugene, Oregon, USA
bae@uoregon.edu

Jeremy Kozdon
Naval Postgraduate School
Monterey, California, USA
jeremy@kozdon.net

Jee Choi
University of Oregon
Eugene, Oregon, USA
jeec@uoregon.edu

## ABSTRACT

Summation-by-parts (SBP) finite difference methods are widely used in scientific applications alongside a special treatment of boundary conditions through the simultaneous-approximate-term (SAT) technique which enables the valuable proof of numerical stability. Our work is motivated by multi-scale earthquake cycle simulations described by partial differential equations (PDEs) whose discretizations lead to huge systems of equations and often rely on iterative schemes and parallel implementations to make the numerical solutions tractable. In this study, we consider 2D, variable coefficient elliptic PDEs in complex geometries discretized with the SBP-SAT method. The multigrid method is a well-known, efficient solver or preconditioner for traditional numerical discretizations, but they have not been well-developed for SBP-SAT methods on HPC platforms. We propose a custom geometric-multigrid preconditioned conjugate-gradient (MGCG) method that applies SBP-preserving interpolations. We then present novel, matrix-free GPU kernels designed specifically for SBP operators whose differences from traditional methods make this task nontrivial but that perform 3× faster than SpMV while requiring only a fraction of memory. The matrix-free GPU implementation of our MGCG method performs 5× faster than the SpMV counterpart for the largest problems considered (67 million degrees of freedom). When compared to off-the-shelf solvers in the state-of-the-art libraries PETSc and AmgX, our implementation achieves superior performance in both iterations and overall runtime. The method presented in this work offers an attractive solver for simulations using the SBP-SAT method.

## CCS CONCEPTS

• **Mathematics of computing** → **Partial differential equations**; **Numerical differentiation**; • **Computing methodologies** → **Parallel algorithms**.

This work is licensed under a Creative Commons Attribution International 4.0 License.

*ICS '24, June 04–07, 2024, Kyoto, Japan*
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0610-3/24/06
https://doi.org/10.1145/3650200.3656614

## KEYWORDS

SBP-SAT, multigrid, preconditioner, matrix-free, GPU

**ACM Reference Format:**
Alexandre Chen, Brittany A. Erickson, Jeremy Kozdon, and Jee Choi. 2024. Matrix-free SBP-SAT finite difference methods and the multigrid preconditioner on GPUs. In *Proceedings of the 38th ACM International Conference on Supercomputing (ICS '24), June 04–07, 2024, Kyoto, Japan.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650200.3656614

## 1 INTRODUCTION

Computational modeling of the natural world involves pervasive material and geometric complexities that are hard to understand, incorporate, and analyze. The partial differential equations (PDEs) governing many of these systems are subject to boundary and interface conditions, and all numerical methods share the fundamental challenge of how to enforce these conditions in a stable manner. Additionally, applications involving elliptic PDEs or implicit time-stepping require efficient solution strategies for linear systems of equations.

Most applications in the natural sciences are characterized by multiscale features in both space and time which can lead to huge linear systems of equations after discretization. Our work is motivated by large-scale (~hundreds of kilometers) earthquake cycle simulations where frictional faults are idealized as geometrically complex interfaces within a 3D material volume and are characterized by much smaller-scale features (~microns) [21, 28]. In contrast to the single-event simulations, e.g. [49], where the computational work at each time step is a single matrix-vector product, earthquake cycle simulations must integrate with adaptive time-steps through the slow periods between earthquakes, and are tasked with a much more costly linear solve. For example, even with upscaled parameters so that larger grid spacing can be used, the 2D simulations in [21] generated matrices of size $\sim 10^6$, and improved resolution and 3D domains would increase the system size to $\sim 10^9$ or greater. Because iterative schemes are most often implemented for the linear solve (since direct methods require a matrix factorization that is often too large to store in memory), it is no surprise that the sparse matrix-vector product (SpMV) arises as the main computational workhorse. The matrix sparsity and condition number depend on several physical and numerical factors including the material heterogeneity of the Earth's material properties, order of accuracy, the coordinate transformation (for irregular grids), and the mesh

size. For large-scale problems, matrix-free (on-the-fly) techniques for the SpMV are fundamental when the matrix cannot be stored explicitly.

In this work, we use summation-by-parts (SBP) finite difference methods [30, 40, 52, 53], which are distinct from traditional finite difference methods in their use of specific one-sided approximations at domain boundaries that enable the highly valuable proof of stability, a necessity for numerical convergence. Weak enforcement of boundary conditions has additional superior properties over traditional methods, for example, the simultaneous-approximation-term (SAT) technique, which relaxes continuity requirements (of the grid and the solution) across physical or geometrical interfaces, with low communication overhead for efficient parallel algorithms [18]. For these reasons SBP-SAT methods are widely used in many areas of scientific computing, from the flow over airplane wings to biological membranes, to earthquakes and tsunamigenesis [20, 36, 43, 46, 54, 60]; these studies, however, have not been developed for linear solves or were limited to small-scale simulations.

With this work, we contribute a novel iterative scheme for linear systems based on SBP-SAT discretizations where nontrivial computations arise due to boundary treatment. These methods are integrated into our existing, public software for simulations of earthquake sequences. Specifically, we make the following contributions:

- Since preconditioning of iterative methods is a hugely consequential step towards improving convergence rates, we develop a custom geometric multigrid preconditioned conjugate gradient (MGCG) algorithm which shows a near-constant number of iterations with increasing system size. The required iterations (and time-to-solution) are much lower compared to several off-the-shelf preconditioners offered by the PETSc library [4], a state-of-the-art library for scientific computing.
- We develop custom, matrix-free GPU kernels (specifically for SBP-SAT methods) for computations in the volume and boundaries, which show improved performance as compared to the native, matrix-explicit implementation, while requiring only a fraction of memory.
- GPU-acceleration of our resulting matrix-free, preconditioned iterative scheme shows superior performance compared to state-of-the-art methods offered by NVIDIA.

Furthermore, the ubiquity of SBP-SAT methods in modern scientific computing applications means our work has the propensity to advance scientific studies currently limited to small-scale problems.

The paper is organized as follows: First, we present related (albeit limited) work on preconditioning and GPU acceleration of iterative schemes for SBP-SAT methods. Next, we provide a detailed description of our model problem, followed by the corresponding SBP-SAT discretization which generates the linear system we focus on in this work. We include details of our code development in the Julia programming language and its HPC capabilities and verify our solutions through rigorous spatial convergence tests to ascertain correctness. We next discuss a two-pronged approach for accelerating time-to-solution, with developed methods customized for SBP-SAT methods. The first is through a preconditioning approach, which can be thought of as a mathematical means for reducing the number of overall iterations required for the iterative scheme to converge. Second, individual iterations can be accelerated by GPU parallelism and the development of matrix-free operations. Preconditioning and parallel performance are compared against options from several state-of-the-art libraries (PETSc and NVIDIA AmgX). We conclude with a summary of our results and proposed future work.

## 2 RELATED WORK

Conjugate gradient (CG) is a standard iterative method for solving linear systems involving symmetric positive-definite (SPD) matrices. But the performance of CG often relies fundamentally on specialized preconditioning techniques specific to the application, e.g. [1, 25, 44, 59]. To our knowledge, no systematic studies of preconditioning performance for SBP-SAT methods have been done until this study. Multigrid (MG) is a technique for both solving and preconditioning linear systems using a hierarchy of successively coarser grids. Three key ingredients define multigrid methods, namely, interpolation operators (prolongation and restriction), smoothers, and (if used) a direct solve on a coarse grid. MG has inherent high-parallelism, smooths low-frequency error components and has proven to be a highly successful preconditioner for improving convergence rates in a wide range of application problems [55]. In [50], MG methods were explored for SBP-SAT methods, and the development of SBP-preserving interpolation operators showed improved performance by modifying standard interpolation operators near boundaries. However, these studies only explored MG as a solver, and not its effectiveness as a preconditioner. In addition, they applied Galerkin coarsening to produce the coarse grid operators rather than through a rediscretization of the original PDE. Because part of the focus of our work is the development of matrix-free methods, defining the coarse grid operators in this fashion would require writing an entirely different kernel for every grid level, as well as more memory for data storage [11]. Therefore, to fully utilize the efficiency of our matrix-free methods, as well as to reduce complexity in and number of kernels needed, our focus is on rediscretization approaches when using MG as a preconditioner for CG (denoted MGCG).

The CG iteration itself involves three components: the sparse matrix data structure (if matrix-explicit methods are used), the reduction operations to compute inner products, and the matrix-vector multiply. For such matrix-explicit methods, loading the matrix from device memory often dominates SpMV performance. Even though GPUs are bandwidth rich (an order of magnitude or higher than CPUs), making them ideally suited for SpMV [16], the SpMV constitutes the majority of the computational load. Specifically, performance can be limited by low computational intensity, irregular memory access, and sparsity [33], and though not the focus of this work, much effort has been devoted to sparse matrix data structures such as CSR (our baseline comparison), ELL, COO, and BELLPACK (GPU only) [3, 5, 7, 8, 16, 56, 58].

To our knowledge, the only previous work on SBP-SAT methods on the GPU considered a matrix-explicit SpMV [32], or used stencil computations in seismic wave propagations but did not solve linear systems [45]. Here we develop a matrix-free implementation of the SpMV and corresponding MGCG method to solve our model problem. Matrix-free methods include the additional benefits of

increased arithmetic intensity and reduced memory footprint, and do not require matrix assembly (which can be cumbersome for complex applications), all of which can improve performance on modern, bandwidth-limited processors [13, 24, 35, 42].

# 3 MODEL PROBLEM AND METHODS

## 3.1 Partial Differential Equations for the Solid Earth

Our work is motivated by the study of quasi-static deformation of the solid Earth over the time-scales of earthquake cycles. Motion is governed by the equilibrium equation and a constitutive relationship describing the material properties. The standard assumption is that the Earth is linear elastic, defined on a sub-domain of $\mathbb{R}^3$. While solutions to the 3D elasticity equations are the eventual target, 2D problems are considered in this work in order to sort out implementation details with reduced computational costs. The 2D anti-plane shear problem [37] is particularly ubiquitous in earthquake cycle benchmark problems [22], where a vertical cross-section of a 3D problem (assuming invariance in one-direction) gives rise to an elliptic equation at each time-step, where only one non-zero component of the displacement vector exists and depends on two spatial variables, namely,

$$-\nabla \cdot (\mu \nabla u) = f \quad \text{for } (x, y) \in \Omega, \tag{1}$$

where $\mu(x, y)$ is the spatially-varying shear modulus, $u$ is Earth's material displacement in the $z$-direction, and $f$ comprises body forces. In order to enable complex fault geometries and topography, we assume that $\Omega$ is a curved quadrilateral in $\mathbb{R}^2$, which enables extensions to arbitrary domains partitioned into computational blocks, e.g. [29]. As Figure 1(a) illustrates, the boundary can be partitioned into four curves $\partial\Omega_i,\ i = 1, \ldots, 4$, where (for example), $\partial\Omega_3$ represents Earth's surface and the shear modulus $\mu(x, y)$ can vary in order to represent heterogeneities in the crust, for example, a shallow sedimentary basin, as illustrated.

For generality we consider both Dirichlet and Neumann boundary conditions, namely

$$u = g_1, \quad \partial\Omega_1, \tag{2a}$$
$$u = g_2, \quad \partial\Omega_2, \tag{2b}$$
$$\boldsymbol{n} \cdot \mu \nabla u = g_3, \quad \partial\Omega_3, \tag{2c}$$
$$\boldsymbol{n} \cdot \mu \nabla u = g_4, \quad \partial\Omega_4, \tag{2d}$$

where vector $\boldsymbol{n}$ is the outward pointing normal to the domain boundary and $g_i,\ i = 1, \ldots, 4$ represent boundary data.

## 3.2 Coordinate Transformation

In order to solve (1)-(2) with SBP-SAT methods, the domain $\Omega$ is transformed (via a conformal mapping) to the regular, square domain $(r, s) \in \bar{\Omega} = -1 \leq (r, s) \leq 1$, as in Figure 1(b). The transformed equations are given by

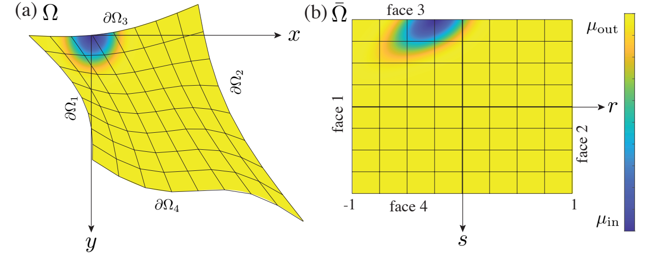


**Figure 1: (a) Geometrically complex physical domain $\Omega$ with material stiffness that increases from $\mu_{\text{in}}$ within a shallow, ellipsoidal sedimentary basin, to stiffer host rock given by $\mu_{\text{out}}$. (b) $\Omega$ is transformed to the regular, square domain $\bar{\Omega}$ via conformal mapping.**

$$-\bar{\nabla} \cdot \left(\mathbf{c}\bar{\nabla} u\right) = Jf, \quad \text{for } (r, s) \in \bar{\Omega}, \tag{3a}$$
$$u = g_1, \quad \text{face 1}, \tag{3b}$$
$$u = g_2, \quad \text{face 2}, \tag{3c}$$
$$\hat{\boldsymbol{n}}^3 \cdot \mathbf{c}\bar{\nabla} u = S_J^3 g_3, \quad \text{face 3}, \tag{3d}$$
$$\hat{\boldsymbol{n}}^4 \cdot \mathbf{c}\bar{\nabla} u = S_J^4 g_4, \quad \text{face 4}, \tag{3e}$$

where $\bar{\nabla} u = \left[\frac{\partial u}{\partial r}, \frac{\partial u}{\partial s}\right]^T$, face $k$ for $k = 1, \ldots 4$ define the domain boundaries of $\bar{\Omega}$, given in Figure 1(b). $J > 0$ is the Jacobian and $S_J^k$ is the surface Jacobian on face $k$. Vector $\hat{\boldsymbol{n}}^k$ is the outward pointing normal to the face and $2 \times 2$ matrix $\mathbf{c}$ is symmetric positive definite (SPD) and encodes the variable material properties $\mu(x, y)$ and the coordinate transform, see [23, 29] for more details.

## 3.3 SBP-SAT Finite Difference Methods

SBP methods approximate partial derivatives using one-sided differences at all points close to the boundary node, generating a matrix approximating a partial derivative operator. In this work we focus on second-order derivatives that appear in (3), however, the matrix-free methods we derive are applicable to any second-order PDE. We consider SBP finite-difference approximations to boundary-value problem (3), i.e. on the square computational domain $\bar{\Omega}$; solutions on the physical domain $\Omega$ are obtained by the inverse coordinate transformation.

In this work, we focus on SBP operators with second-order accuracy which contains abundant complexity at domain boundaries to enable insight into implementation design extendable to higher-order methods. To provide background on the SBP methods we first describe the 1D operators, as Kronecker products are used to form their multi-dimensional counterparts.

*3.3.1 1D Operators.* We discretize the spatial domain $-1 \leq r \leq 1$ with $N + 1$ evenly spaced grid points $r_i = -1 + ih, i = 0, \ldots, N$ with grid spacing $h = 2/N$. A function $u$ projected onto the computational grid is denoted by $\boldsymbol{u} = [u_0, u_1, \ldots, u_N]^T$ and is often taken to be the interpolant of $u$ at the grid points. We define the grid basis

vector $\vec{e}_j$ to be a vector with value 1 at grid point $j$ and 0 for the rest, which allows us to extract the jth component: $u_j = \vec{e}_j^T \vec{u}$.

*Definition 3.1 (First Derivative).* A matrix $\boldsymbol{D}_r$ is an SBP approximation to the first derivative operator $\partial/\partial r$ if it can be decomposed as $\boldsymbol{H}\boldsymbol{D}_r = \boldsymbol{Q}$ with $\boldsymbol{H}$ being SPD and $\boldsymbol{Q}$ satisfying $\vec{u}^T(\boldsymbol{Q}+\boldsymbol{Q}^T)\vec{v} = u_N v_N - u_0 v_0$.

Here, $\boldsymbol{H}$ is a diagonal quadrature matrix and $\boldsymbol{D}_r$ is the standard central finite difference operator in the interior which transitions to one-sided at boundaries.

*Definition 3.2 (Second Derivative).* Letting $c = c(r)$ denote a material coefficient, we define matrix $\boldsymbol{D}_{rr}^{(c)}$ to be an SBP approximation to $\frac{\partial}{\partial r}\left(c\frac{\partial}{\partial r}\right)$ if it can be decomposed as $\boldsymbol{D}_{rr}^{(c)} = \boldsymbol{H}^{-1}(-\boldsymbol{M}^{(c)} + c_N\vec{e}_N\vec{d}_N^T - c_0\vec{e}_0\vec{d}_0^T)$ where $\boldsymbol{M}^{(c)}$ is SPD and $\vec{d}_0^T\vec{u}$ and $\vec{d}_N^T\vec{u}$ are approximations of the first derivative of $u$ at the boundaries.

With these properties, both $\boldsymbol{D}_r$ and $\boldsymbol{D}_{rr}^{(c)}$ mimic integration-by-parts in a discrete form which enables the proof of discrete stability [39, 40].

$\boldsymbol{D}_{rr}^{(c)}$ is a centered difference approximation within the interior of the domain, but includes approximations at boundary points as well. For illustrative purposes alone, if $c = 1$ (e. g. a constant coefficient case), the matrix is given by

$$\boldsymbol{D}_{rr}^{(c)} = \frac{1}{h^2}\begin{bmatrix} 1 & -2 & 1 & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & 1 & -2 & 1 \end{bmatrix},$$

which, as highlighted in red, resembles the traditional (second-order-accurate) Laplacian operator in the domain interior.

### 3.3.2 2D SBP Operators.

The 2D domain $\bar{\Omega}$ is discretized using $N+1$ grid points in each direction, resulting in an $(N+1)\times(N+1)$ grid of points where grid point $(i, j)$ is at $(x_i, y_j) = (-1+ih, -1+jh)$ for $0 \le i, j \le N$ with $h = 2/N$. Here we have assumed equal grid spacing in each direction, only for notational ease; the generalization to different numbers of grid points in each dimension does not impact the construction of the method and is implemented in our code. A 2D grid function $\boldsymbol{u}$ is ordered lexicographically and we let $\boldsymbol{C}_{ij} = \text{diag}(\boldsymbol{c}_{ij})$ define the diagonal matrix of coefficients, see [29].

In this work we imply summation notation whenever indices are repeated. Multi-dimensional SBP operators are obtained by applying the Kronecker product to 1D operators, for example, the 2D second derivative operators are given by

$$\begin{aligned}\frac{\partial}{\partial i}c_{ij}\frac{\partial}{\partial j} &\approx \tilde{\boldsymbol{D}}_{ij}^{c_{ij}} \\ &= (\boldsymbol{H}\otimes\boldsymbol{H})^{-1}\left[-\tilde{\boldsymbol{M}}_{ij}^{(c_{ij})} + \boldsymbol{T}\right],\end{aligned} \tag{4}$$

for $i, j \in \{r, s\}$. Here $\tilde{\boldsymbol{M}}_{ij}^{(c_{ij})}$ is the sum of SPD matrices approximating integrated second derivatives (i.e. sum over repeated indices $i, j$) for example $\int_{\bar{\Omega}} \frac{\partial}{\partial r}c_{rr}\frac{\partial}{\partial r} \approx \tilde{\boldsymbol{M}}_{rr}^{(c_{rr})}$ and matrix $\boldsymbol{T}$ involves the boundary derivative computations, see [23] for complete details.

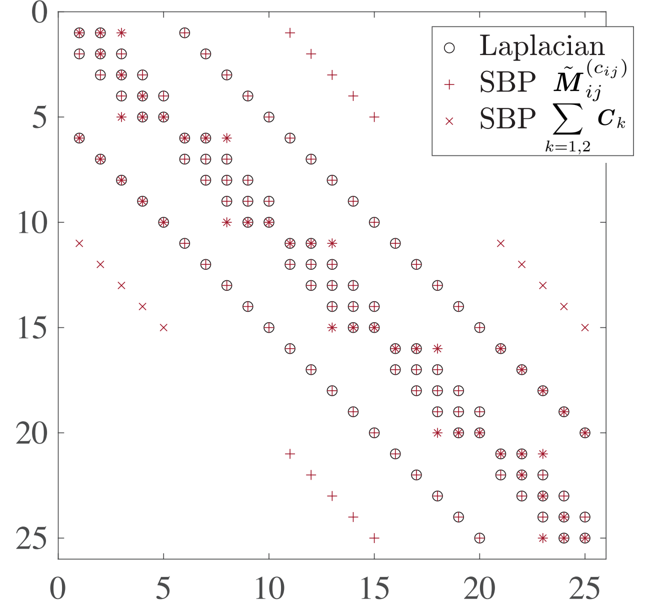


**Figure 2: Sparsity pattern for matrix $A$ with $N = 5$ grid points in each direction. Traditional 5-point Laplacian stencil in black circles. Contributions to $A$ are separated into contributions from the volume (red +) and from the boundary enforcement (red ×), so that contributions from both (red ∗) cancel deviations from symmetry and render $A$ SPD.**

### 3.3.3 SAT Penalty Terms.

SBP methods are designed to work with various impositions of boundary conditions that lead to provably stable methods, for example through weak enforcement via the simultaneous-approximation-term (SAT) [14] which we adopt here. As opposed to traditional finite difference methods that "inject" boundary data by overwriting grid points with the given data, the SAT technique imposes boundary conditions weakly (through penalization), so that all grid points approximate both the PDE and the boundary conditions up to a certain level of accuracy. The combined approach is known as SBP-SAT. Where traditional methods that use injection or strong enforcement of boundary/interface conditions destroy the discrete integration-by-parts property, using SAT terms enables proof of the method's stability (a necessary property for numerical convergence) [38].

## 4 PROBLEM DISCRETIZATION

The SBP-SAT discretization to (3) is given by

$$-\tilde{\boldsymbol{D}}_{ij}^{(c_{ij})}\boldsymbol{u} = \boldsymbol{J}\boldsymbol{f} + \sum_k \boldsymbol{p}_k \tag{5}$$

where $\boldsymbol{p}_k$ are SAT vectors formed from the boundary condition on face $k$. To illustrate the structure of the SAT vectors, enforcing Dirichlet conditions on faces $k = 1, 2$ generates

$$\boldsymbol{p}_k = (\boldsymbol{H}\otimes\boldsymbol{H})^{-1}\left(\boldsymbol{G}_k^T - \boldsymbol{L}_k^T\boldsymbol{H}_k\boldsymbol{\tau}_k\right)\left(\boldsymbol{L}_k\boldsymbol{u} - \boldsymbol{g}_k\right), \tag{6}$$

where matrix $\mathbf{G}_k$ computes the weighted derivative on face $k$, $\mathbf{L}_k$ is the face extraction operator and $\boldsymbol{H}_k$ is the 1D $\boldsymbol{H}$ matrix in the direction parallel to face $k$. Matrix $\boldsymbol{\tau}_k = \hat{n}_i^k C_{ij}^k \hat{n}_j^k \boldsymbol{\Gamma}_k$, where $C_{ij}^k$ is the diagonal matrix of coefficients restricted face $k$ and $\boldsymbol{\Gamma}_k$ is the diagonal penalty matrix on face $k$ with sufficiently large components to ensure discrete stability, according to

$$\boldsymbol{\Gamma}_k \geq \frac{4}{h_\perp^k}\mathbf{I} + \frac{1}{h_\perp^k}\mathbf{P}_k, \tag{7}$$

where

$$\mathbf{P}_k = \begin{cases} C_{rr}^k (C_{rr}^{k,min})^{-1}, & \text{k = 1, 2,} \\ C_{ss}^k (C_{ss}^{k,min})^{-1}, & \text{k = 3, 4.} \end{cases} \tag{8}$$

Here $\boldsymbol{C}^{k,min}$ is the minimum value of $c$ in the two points orthogonal to the boundary and $h_\perp^k$ is the grid spacing orthogonal to face $k$ [23].

In order to render the linear system (5) SPD, we multiply by $(\boldsymbol{H} \otimes \boldsymbol{H})$ (the discrete equivalent of integrating over $\bar{\Omega}$ and discretizing the weak form). This process yields the final linear system

$$\boldsymbol{A}\boldsymbol{u} = \boldsymbol{b} \tag{9}$$

where

$$\boldsymbol{A} = \tilde{\boldsymbol{M}}_{ij}^{(c_{ij})} + \sum_{k=1,2} \boldsymbol{C}_k, \tag{10}$$

is SPD [23], and matrices

$$\boldsymbol{C}_k = -\mathbf{L}_k^T \mathbf{G}_k - \mathbf{G}_k^T \mathbf{L}_k + \mathbf{L}_k^T \boldsymbol{H}_k \boldsymbol{\tau}_k \mathbf{L}_k. \tag{11}$$

Right-hand side vector $\boldsymbol{b}$ is given by

$$\boldsymbol{b} = (\boldsymbol{H} \otimes \boldsymbol{H}) \left[ \boldsymbol{J}\boldsymbol{f} + \sum_k \boldsymbol{K}_k \boldsymbol{g}_k \right] \tag{12}$$

which encodes the source term and boundary data. Here matrices $\boldsymbol{K}$ depend on boundary conditions; for those given in (3) they are

$$\boldsymbol{K}_1 = \boldsymbol{L}_1^T \boldsymbol{H}_1 \boldsymbol{\tau}_1 - \boldsymbol{G}_1^T \tag{13}$$

$$\boldsymbol{K}_2 = \boldsymbol{L}_2^T \boldsymbol{H}_2 \boldsymbol{\tau}_2 - \boldsymbol{G}_2^T \tag{14}$$

$$\boldsymbol{K}_3 = \boldsymbol{L}_3^T \boldsymbol{H}_3 \tag{15}$$

$$\boldsymbol{K}_4 = \boldsymbol{L}_4^T \boldsymbol{H}_4. \tag{16}$$

Note that $\boldsymbol{A}$ includes contributions from both volume operators ($\tilde{\boldsymbol{M}}_{ij}^{c_{ij}}$) and from the SAT enforcement of boundary terms ($\boldsymbol{C}_k$), and differs from the traditional discrete Laplacian near all domain boundaries; see Figure 2. At Dirichlet boundaries (faces 1 and 2), $\boldsymbol{C}_k$ modifies the layer of three points normal to the face (i.e. the SAT imposition penalizes all points used in the computation of the derivative normal to the face).

## 5 DEVELOPMENT AND VERIFICATION

In this work all code is written in the Julia programming language. Julia is a dynamically typed language for scientific computing designed with high performance in mind [10]. Julia supports general-purpose GPU computing with the package CUDA.jl. Through communications in LLVM intermediate representations with NVIDIA's compiler, it is claimed that CUDA.jl achieves the same level of performance as CUDA C according to previous research[9]. Aimed to address the "two-language" problem, Julia enables implementation ease of complex mathematical algorithms while achieving high performance, an ideal match for computational scientists without expertise in low-level language-based HPC. Julia has gained attention among the HPC community, with notable examples including The Climate Machine [51], a new Earth System model written purely in Julia that is capable of running on both CPUs and GPUs by utilizing KernelAbstractions.jl [17], a pure Julia device abstraction similar to Raja, Kokkos, and OCCA [6, 15, 41]. In addition, because a large body of researchers studying SBP methods use Julia in serial, e.g. [29, 47], our developments will enable these users to gain HPC capabilities in their code with minimal overhead.

**Table 1: Discretization error and convergence rate for the SBP-SAT method. The use of more digits illustrates the convergence rate approaching the theoretical rate 2.**

| $N$ | Discretization error | Convergence rate |
|---|---|---|
| $2^9$ | 1.353609E-05 | 1.998341E+00 |
| $2^{10}$ | 3.385302E-06 | 1.999455E+00 |
| $2^{11}$ | 8.464360E-07 | 1.999811E+00 |
| $2^{12}$ | 2.116192E-07 | 1.999930E+00 |
| $2^{13}$ | 5.290578E-08 | 1.999973E+00 |

The SBP-SAT discretization used to form system (9) is rigorously verified for correctness by confirming numerical convergence towards a known, analytical solution [48]. For the exact solution (denoted with an asterisk) we take

$$u^*(x, y) = \sin(\pi x)\sinh(\pi y), \tag{17}$$

on the domain $\Omega$ with corners $(x, y) = \{(-0.3, 0), (0.5, -.25), (0, 1), (1, 1.5)\}$, illustrated in Figure 1(a), with curved edges defined by sinusoids; the grid is formed with transfinite interpolation. The spatially variable shear modulus is taken to be

$$\mu(x, y) = \frac{\mu_{\text{out}} - \mu_{\text{in}}}{2} \tanh((x^2 + c^2 y^2 - \bar{r})/r_w) + 1) + \mu_{\text{in}} \tag{18}$$

which forms a semi-ellipse representing a shallow, sedimentary basin. The basin contains compliant material with $\mu_{\text{in}} = 20$, surrounded by stiffer host rock, with $\mu_{\text{out}} = 32$. The basin parameters $c = 0.5$, $\bar{r} = 6.25\text{E-}4$ and $r_w = 0.015$, which implicitly define a basin depth and width of 0.05. The discrete $L^2$-error is defined by the $H$−norm, namely

$$\begin{aligned} E_h &= ||\boldsymbol{u}^* - \boldsymbol{u}||_H \\ &= \sqrt{(\boldsymbol{u}^* - \boldsymbol{u})^T \boldsymbol{J} \,(\boldsymbol{H} \otimes \boldsymbol{H})\, (\boldsymbol{u}^* - \boldsymbol{u})}, \end{aligned} \tag{19}$$

where $\boldsymbol{u}^*$ is the exact solution (17) evaluated on the grid. $E_h$ is reported in Table 1 using a direct solve (i.e. we compute the discretization error), which shows that we have achieved convergence at the theoretical rate.

## 6 PERFORMANCE: PRECONDITIONING

Iterative methods enable the solution to larger problems of the form (9) compared to a direct solve that requires storing a matrix factorization. However, the convergence of CG depends predominantly on the condition number of the matrix $\boldsymbol{A}$, which can be reduced (to accelerate convergence) through preconditioning techniques. The

preconditioning matrix $\boldsymbol{M}$ has to be SPD and fixed, and although it need not be explicitly assembled nor inverted, good preconditioners should satisfy $\boldsymbol{M} \approx \boldsymbol{A}^{-1}$.

In this work we develop a custom MG preconditioner by first adopting the second-order SBP-preserving prolongation/restriction operators from[50], which maintain accuracy at domain boundaries and correctly transfer residual vectors to the coarser grids. The 2D restriction operator is given by

$$\boldsymbol{I}_h^{2h} = \boldsymbol{H}_{2h}^{-1} \left(\boldsymbol{I}_{2h}^{h}\right)^T \boldsymbol{H}_h \tag{20}$$

where $\boldsymbol{H}_h$ and $\boldsymbol{H}_{2h}$ denote $\boldsymbol{H} \otimes \boldsymbol{H}$ with grid spacing $h$ and $2h$, respectively. The 2D prolongation operator $\boldsymbol{I}_{2h}^h$ is defined by $\boldsymbol{I}_{2h}^h = I_{2h}^h \otimes I_{2h}^h$, where $I_{2h}^h$ is the standard 1D prolongation operator [12], see Appendix A in [50].

One feature that differentiates our problem formulation from those in [50] is that our matrix in (9) is rendered SPD only after the multiplication of (4) on the left by $(\boldsymbol{H} \otimes \boldsymbol{H})$, which introduces additional grid information when calculating the associated residual vector. To properly transfer this grid information we found improved performance of our preconditioner when further modifying the restriction operator to account for grid spacing. This is achieved by excluding the $(\boldsymbol{H} \otimes \boldsymbol{H})$ term when computing the residual on the fine grid, then restricting using $\boldsymbol{I}_r$, and then re-introducing the grid spacing on the coarse grid. In addition, we explored both Galerkin coarsening and rediscretization to form the coarse-grid operators and found the latter to give superior performance. The pseudo-code for the custom MG method (which we will apply as a preconditioner) is given in Algorithm 1.

We first compare the preconditioning performance of our custom geometric multigrid against the multigrid using Galerkin's condition from [50]. To avoid the influence of hyperparameters in smoothers, we use the Gauss-Seidel method as smoother. This method is not suitable for GPU parallelization, so we only focus on the preconditioning performance in terms of iteration reduction here. For a 2D computational grid of $N = 2^k$ points in each direction, we use a single V-cycle multigrid of $k - 1$ levels with 5 smoothing steps on each level including the coarsest grid (i.e. taking $\nu_1 = \nu_2 = \nu_3 = 5$ and $N_{maxiter} = 1$ in Algorithm 1) as the preconditioner. The MGCG stops when the relative residual is reduced to less than $10^{-8}$ times the initial residual. For all tests in this work we initialize the iterative scheme with the zero vector. We report results in Table 2. Our custom geometric multigrid method achieves comparable preconditioning results with the multigrid method using Galerkin's condition from [50], both having near-constant iteration counts for different problem sizes. The iteration count for non-preconditioned CG almost doubles when $N$ is doubled, indicating worse condition numbers on finer grids.

Many types of smoothers for multigrid methods can be explored for best performance. For the rest of the work, we focus on Richardson iteration given by $\mathbf{x}_{k+1} = \mathbf{x}_k + \omega(\mathbf{b} - \mathbf{A}\mathbf{x}_k)$ because it can be easily accommodated by our subsequent development of matrix-free kernels for $\boldsymbol{A}$. Here $\omega$ is chosen to satisfy the convergence criteria and its optimal value depends on the eigenvalues of $\boldsymbol{A}$ as $\omega_{opt} = \frac{2}{\lambda_{max}+\lambda_{min}}$, where $\lambda_{max}$ and $\lambda_{min}$ are the largest and smallest eigenvalues of $\boldsymbol{A}$ respectively. We use Arpack.jl, which is a Julia wrapper of ARPACK that uses the Implicitly Restarted Arnoldi Method to calculate eigenvalues for sparse matrices. For small $N$, we can compute $\lambda_{max}$ and $\lambda_{min}$ directly, but for large $N$ values, these become computationally intractable. We use extrapolation to approximate values for $\lambda_{max}$ and $\lambda_{min}$ for $N \geq 32$ based on observation of eigenvalues for $N \leq 32$, namely,

$$\lambda_{min,2N} = \lambda_{min,N}/4,$$
$$\lambda_{max,2N} = \lambda_{max,N} + 0.6 * (\lambda_{max,N} - \lambda_{max,N/2}),$$

**Algorithm 1** $(k+1)$-level MG for $\boldsymbol{A}_h\boldsymbol{u}_h = \boldsymbol{f}_h$, with smoothing $S_{h_k}^{\nu}$ applied $\nu$ times. SBP-preserving restriction and interpolation operators are applied. Grid coarsening $(k \rightarrow k+1)$ is done through successive doubling of grid spacing until reaching the coarsest grid. The multigrid cycle can be performed $N_{maxiter}$ times. $\boldsymbol{r}$ represents the residual, and $\boldsymbol{v}$ represents the solution to the residual equation used during the correction step. This algorithm is adapted from [34].

**function** MG($\boldsymbol{f}_{h_k}$, $\boldsymbol{A}_{h_k}$, $\boldsymbol{u}_{h_k}^{(0)}$, $k$, $N_{maxiter}$)
 **for** $n = 0, 1, 2, \ldots, N_{maxiter}$ **do**
  $\boldsymbol{u}_{h_k}^{(n)} \xleftarrow{S_{h_k}^{\nu_1}} \boldsymbol{u}_{h_k}^{(n)}$ ▷ Pre-smoothing $\nu_1$ times
  $\boldsymbol{r}_{h_k}^{(n)} = \boldsymbol{f}_{h_k}^{(n)} - \boldsymbol{A}_{h_k}^{(n)}\boldsymbol{u}_{h_k}^{(n)}$ ▷ Calculating residual
  $\tilde{r}_{h_k} = (\boldsymbol{H}_k \otimes \boldsymbol{H}_k)^{-1}\boldsymbol{r}_{h_k}^{(n)}$ ▷ Removing grid info
  $\boldsymbol{r}_{h_{k+1}} = (\boldsymbol{H}_{k+1} \otimes \boldsymbol{H}_{k+1})\boldsymbol{I}_{h_k}^{h_{k+1}}\tilde{\boldsymbol{r}}_{h_k}$ ▷ Restriction
  **if** $k + 1 = k_{\max}$ **then**
   $\boldsymbol{v}_{h_{k+1}}^{(n)} \xleftarrow{S_{h_{k+1}}^{\nu_2}} \boldsymbol{0}_{h_{k+1}}$ ▷ Smoothing on coarsest grid
  **else**
   $\boldsymbol{v}_{h_{k+1}}^{(n)} = \text{MG}(\boldsymbol{r}_{h_{k+1}}, \boldsymbol{A}_{h_{k+1}}, \boldsymbol{0}_{h_{k+1}}, k+1, 1)$
   ▷ Recursive definition of MG
  **end if**
  $\boldsymbol{v}_k^n = \boldsymbol{I}_{h_{k+1}}^{h_k}\boldsymbol{v}_{k+1}^{(n)}$ ▷ Interpolation
  $\boldsymbol{u}_k^{(n+1)} = \boldsymbol{u}_k^{(n)} + \boldsymbol{v}_k^n$ ▷ Correction
  $\boldsymbol{u}_k^{(n+1)} \xleftarrow{S_{h_k}^{\nu_3}} \boldsymbol{u}_{h_k}^{(n+1)}$ ▷ Post-smoothing $\nu_3$ times
 **end for**
**end function**

**Table 2: Performance of MGCG using our custom preconditioner (denoted MGCG) against the multigrid using Galerkin's condition (denoted MGCG-Galerkin) from [50] and non-preconditioned CG. We report the total iterations to converge and the final relative residual for three different algorithms.**

| N | MGCG-Galerkin | MGCG | CG |
|---|---|---|---|
| 64 | 6 / 2.33 E-9 | 6 / 2.25 E-09 | 221 / 9.98 E-9 |
| 128 | 6 / 8.39 E-9 | 6 / 2.42 E-9 | 431 / 9.61 E-9 |
| 256 | 7 / 2.44 E-9 | 6 / 2.16 E-9 | 839 / 9.79 E-9 |
| 512 | 7 / 2.63 E-9 | 6 / 1.89 E-9 | 1643 / 9.70 E-9 |
| 1024 | 7 / 2.72 E-9 | 6 / 1.73 E-9 | 3208 / 9.93 E-9 |

**Table 3: Iterations and time to converge for $N = 2^{10}$ using 1 smoothing step in PETSc PAMGCG with V cycle (first three rows) vs. our MGCG using Richardson iteration as smoother (last row)**

| mg_levels_ksp_type | mg_levels_pc_type | iters | time |
|---|---|---|---|
| chebyshev | sor | 18 | 4.105 s |
| | jacobi | 22 | 3.382 s |
| | bjacobi | 17 | 3.945 s |
| richardson | sor | 18 | 3.581 s |
| | jacobi | 49 | 3.729 s |
| | bjacobi | 16 | 3.729 s |
| cg | sor | 17 | 4.081 s |
| | jacobi | 23 | 3.849 s |
| | bjacobi | 16 | 3.971 s |
| richardson | none | 11 | 0.086 s |

**Table 4: Iterations and time to converge for $N = 2^{10}$ using 5 smoothing steps in PETSc PAMGCG with V cycle (first three rows) vs. our MGCG using Richardson iteration as smoother (last row)**

| mg_levels_ksp_type | mg_levels_pc_type | iters | time |
|---|---|---|---|
| chebyshev | sor | 10 | 10.76 s |
| | jacobi | 14 | 10.20 s |
| | bjacobi | 9 | 10.58 s |
| richardson | sor | 9 | 10.13 s |
| | jacobi | DV | 9.24 s |
| | bjacobi | 8 | 10.28 s |
| cg | sor | 9 | 10.47 s |
| | jacobi | 13 | 10.54 s |
| | bjacobi | 8 | 10.45 s |
| richardson | none | 8 | 0.069s |

where $\lambda_{min,N}$ represents the minimal eigenvalue of a linear system formed for our 2D problem with $N+1$ grid points in each direction and $\lambda_{max,N}$ is the corresponding maximum value. There are many configurations for the multigrid method, but we found that the total number of iterations required by MGCG is largely determined by the number of grid levels and smoothing steps and is less impacted by the choice of smoother itself.

MG methods have many tunable parameters. For this initial study, we explored the MGCG performance varying the number of Richardson pre- and post-smoothing steps on every level between 1 and 5. We considered a single V-cycle (i.e. taking $\nu_1 = \nu_2 = \nu_3 = 5$ and $N_{maxiter} = 1$ in Algorithm 1), including on the coarsest grid (5 grid points in each direction). For all tests in this work we initialize the iterative scheme with the zero vector. All algorithms stop when the relative residual is reduced to less than $10^{-6}$ times the initial residual.

Comparisons against an unpreconditioned CG are not generally appropriate as most real-world applications require preconditioning to make any solution tractable. The Portable, Extensible Toolkit for Scientific Computing (PETSc) [4] is one of the most widely used parallel numerical software libraries, featuring extensive preconditioning methods, many of which can be tested by users via relatively simple command-line options. We experimented with several of PETSc's off-the-shelf algebraic multigrid preconditioned CG solvers (denoted PAMGCG). PETSc's PAMGCG is similar to our MGCG and only requires loading PETSc formatted **A** and **b** (from which it forms the coarse grid operators).

We tested PAMGCG with various configurations against our custom MGCG, applying the same stopping criterion (here based on the relative norm of the residual vector reduced to 1E-6), with results provided in Tables 3 and 4. The `mg_levels_ksp_type` and `mg_levels_pc_type` in the tables stand for Krylov subspace method types and preconditioner types used at each level of the multigrid in PAMGCG. When classical iterative methods are used as smoothers, `mg_levels_ksp_type` is set as `richardson` and the particular smoother (e.g. Jacobi) is set by `mg_levels_pc_type`. Since our MGCG uses Richardson iteration as the smoother for multigrid, we report `mg_levels_ksp_type` as `richardson` and `mg_levels_pc_type` as `none` to maintain coherence across the columns. Iterations and total time to converge are reported. We found that the Jacobi iteration is not a good choice as smoother in PAMGCG. When using 1 smoothing step, it takes more iterations than other configurations. It does not converge (denoted as DV) when using 5 smoothing steps. Aside from this configuration, other PETSc configurations in the table exhibit comparable performance in both the number of iterations and the convergence time. We found that additional options within PAMGCG play relatively minor roles in performance. Our MGCG results (reported in the last rows), however, show superior performance in terms of both iteration counts and overall time.

## 7 ENVIRONMENT AND IMPLEMENTATIONS

Preconditioning reduces the total number of iterations required by CG, but time-to-solution can be further accelerated by reducing individual time-per-iteration. We achieve these improvements by adding GPU-capabilities to our MGCG method, specifically through the development of matrix-free kernels for computing the SpMV (the action of matrix $\boldsymbol{A}$).

### 7.1 Environment Configuration

For the GPU implementation, the matrix-vector product is tested using an NVIDIA V100-SXM2 GPU with 32 GB of memory and an A100-PCIE with 40 GB of memory. Tests are carried out using CUDA 12.0. In addition, we have a dual E5-2690v4 (28 cores) CPU with 256 GB of system memory. The results for CG and MGCG are conducted on the A100 GPU.

### 7.2 Kernel Design

Stencil computations have proven efficient in utilizing GPU resources to achieve optimal performance [31, 57]. In this work we implement a similar GPU kernel for our 2D problem by matching each spatial node to a GPU thread, however, our work requires

specialized treatment for domain boundaries. The most computationally expensive operator is the volume operator $\tilde{\boldsymbol{M}}_{ij}^{c_{ij}}$, which differs from traditional finite difference operators in that it involves derivative approximations at domain boundaries. However, the use of else statements in GPU kernels tends to lead to warp divergence and should be avoided. We construct the matrix-free action of $\boldsymbol{A}$, referred to as `mfA!()` based on node location. Algorithm 2 provides the partial pseudocode, i.e. it includes pseudocode for the $\tilde{\boldsymbol{M}}_{ij}^{c_{ij}}$ calculation; boundary condition calculations are further detailed in Algorithm 3. At interior nodes the action of $\tilde{\boldsymbol{M}}_{ij}^{c_{ij}}$ is defined by a single stencil (with spatially varying coefficients). The action of $\tilde{\boldsymbol{M}}_{ij}^{c_{ij}}$ on boundary nodes, however, has a different stencil depending on the face number and whether the node is at a corner of the domain. To avoid race conditions at corners (while minimizing conditional statements), only normal components of $\tilde{\boldsymbol{M}}_{ij}^{c_{ij}}$ are computed (as they correspond to the same stencil). For example on face 1 only the action of $\tilde{\boldsymbol{M}}_{rr}^{c_{rr}}$ and $\tilde{\boldsymbol{M}}_{rs}^{c_{rs}}$ are computed at the corners, see Figure 3. The action of the remaining components of $\tilde{\boldsymbol{M}}_{ij}^{c_{ij}}$ on the corner nodes are computed in computations associated with adjacent faces (faces 3 and 4).

At boundary nodes we must also compute boundary condition operators $\boldsymbol{C}_k$, with differing stencils depending on face number and whether a node is an interior node, an interior boundary node (i.e. not a corner), or a corner node. Algorithm 3 provides the pseudocode for nodes on face 1; stencils are differentiated with superscripts $int, sw, nw$, corresponding to the interior boundary, northwest, and southwest corner nodes, respectively. Figure 3 further illustrates the nodes involved in each computation: black dots correspond to nodes within the 2D domain. Black circles correspond to the interior nodes that are modified by the action of $\tilde{\boldsymbol{M}}_{ij}^{c_{ij}}$. On the western boundary (face 1), the three-node layer adjacent to face 1 is used to compute the actions of the volume and boundary operators. Blue diamonds and red stars correspond to nodes that are modified by the different components of $\tilde{\boldsymbol{M}}_{ij}^{c_{ij}}$. Green squares correspond to the nodes that are modified by the boundary operator $\boldsymbol{C}_1$ in order to impose the Dirichlet condition (in this case a layer of three nodes normal to the face).

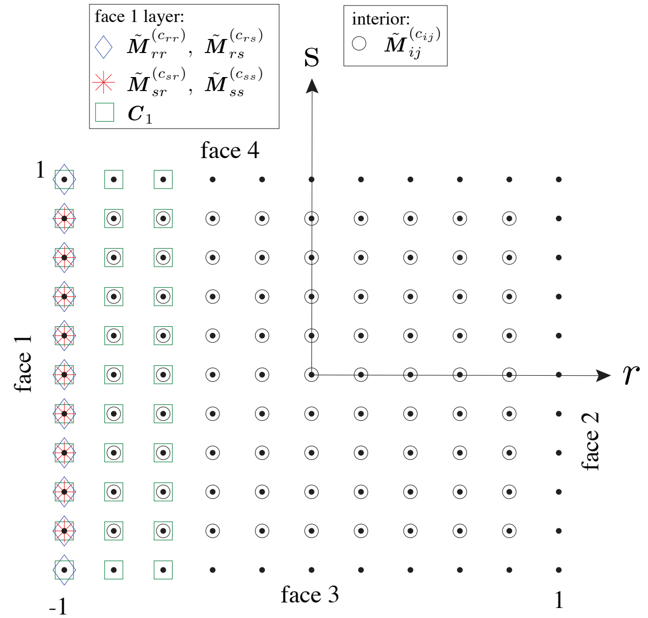


**Figure 3: Schematic of 2D computational domain; nodes denoted with solid black dots. `mfA!()` modifies interior nodes, denoted with circles. For face 1, contributions to `mfA!()` from coordinate transformation matrices modify nodes corresponding to different shapes. Calculations by boundary operator $C_1$ modify nodes in green squares.**

## 8 PERFORMANCE: MATRIX-FREE GPU KERNELS

### 8.1 Performance Comparison

With `mfA!()` we can carry out the matrix-vector product without explicitly storing the matrix. In this section we compare its performance against the matrix-explicit cuSPARSE SpMV implementation available through CUDA.jl. We note that this is not an exhaustive comparison against all possible sparse matrix data structures. Our goal is to establish a baseline comparison of our matrix-free implementation against the standard sparse matrix format CSR in CUDA.jl, with a focus on integration with preconditioning for improving CG performance.

We set up our benchmark as follows: We discretize the domain $\bar{\Omega}$ in each direction using $N{+}1$ grid points, varying $N$ from $2^4$ to $2^{13}$, so the matrix $\boldsymbol{A}$ is of size $(N{+}1)^2{\times}(N{+}1)^2$. Figures 4 and 5 compare the performance of the matrix-free implementation against the matrix-explicit SpMV provided with cuSPARSE using the CSR format on both the A100 GPU and V100 GPU. The performance is measured by profiling 10,000 SpMV calculations with NVIDIA Nsight Systems, and the time results shown in the figures represent the time to perform one SpMV calculation. For problem sizes large enough for GPUs with $N$ greater than $2^{10}$, we see consistent speedup from `mfA!()` kernel with higher speedup achieved for larger problem sizes. On the A100 GPU, our speedup ranges from 3.0× to 3.1×, . On the V100 GPU, we see a similar trend, with our speedup ranging from 3.1× to 3.6×.

The `mfA!()` kernel has a low arithmetic intensity of 0.28 based on the computation of the interior points (which accounts for more than 99% of the total computation and data access). This puts the `mfA!()` kernel in the bandwidth-limited regime [19]. If we plot this on the Roofline model, as shown in Figure 6 as the left red dot, we see that our kernel achieves performance that is *higher* than what is possible for the given arithmetic intensity. If we calculate the arithmetic intensity based on the assumption that the data is read from the DRAM only *once* (i.e., the ideal case when the kernel only incurs compulsory cache misses), as shown in Figure 6 as the right red dot, we see a higher arithmetic intensity of 1.85 and our achieved performance falls below the Roofline. This suggests that a large portion of our data is coming from the fast memory (e.g., L1 or L2 caches), leading to performance that is better than what can be achieved if the data is only coming from the DRAM.

**Algorithm 2** Matrix-Free GPU kernel Action of matrix-free A for interior nodes.

**function** mfA!(odata, idata, $c_{rr}, c_{rs}, c_{ss}, h_r, h_s$)
  $i, j$ =get_global_thread_IDs()
  $g = (i-1) * (N+1) + j$ ▷ compute global index
  **if** $2 \le i, j \le N$ **then** ▷ interior nodes
    odata[g] = $(hs/hr)$(- (0.5$c_{rr}[g$-1] + 0.5$c_{rr}[g]$)idata[$g$-1] +
      + (0.5$c_{rr}[g$-1] + $c_{rr}[g]$ - 0.5$c_{rr}[g$+1])idata[$g$] +
      - (0.5$c_{rr}[g]$ + 0.5$c_{rr}[g$+1])idata[$g$+1]) + ▷ compute $M_{rr}$ stencil

      + 0.5$c_{rs}[g$-1](-0.5idata[$g$-$N$-2] + 0.5idata[$g$+$N$]) +
      - 0.5$c_{rs}[g$+1](-0.5idata[$g$-$N$] + 0.5idata[$g$+$N$+1]) + ▷ compute $M_{rs}$ stencil

      + 0.5$c_{rs}[g$-$N$-1](-0.5idata[$g$-$N$-2] + 0.5idata[$g$-$N$]) +
      - 0.5$c_{rs}[g$+$N$+1](-0.5idata[$g$-$N$] + 0.5idata[$g$+$N$+2]) + ▷ compute $M_{sr}$ stencil

      - (0.5$c_{ss}[g$-$N$-1] + 0.5$c_{ss}[g]$)idata[$g$-$N$-1] +
      + (0.5$c_{ss}[g$-$N$-1] + $c_{ss}[g]$ + 0.5$c_{ss}[g$+$N$+1])idata[$g$] -
      - (0.5$c_{ss}[g]$ + 0.5$c_{ss}[g$+$N$+1])idata[$g$+$N$+1])) ▷ compute $M_{ss}$ stencil
  **end if**
  ... ▷ boundary nodes, e.g. Algorithm 3
  **return** *nothing*
**end function**

**Algorithm 3** Matrix-Free GPU kernel Action of matrix-free A for west boundary (face 1).

**if** $2 \le i \le N$ and $j = 1$ **then** ▷ interior west nodes
  odata[$g$] = $\left(M_{rr}^{int} + M_{rs}^{int} + M_{sr}^{int} + M_{sr}^{int} + C_1^{int}\right)$ (idata) ▷ apply boundary $M$ and $C$ stencils
  odata[$g$+1] = $C_1^{int}$(idata) ▷ apply interior $C$ stencil
  odata[$g$+2] = $C_1^{int}$(idata) ▷ apply interior $C$ stencil
**end if**
**if** $i = 1$ and $j = 1$ **then** ▷ southwest corner node
  odata[$g$] = $\left(M_{rr}^{sw} + M_{rs}^{sw} + C_1^{sw}\right)$ (idata) ▷ apply southwest partial $M$ and $C$ stencils
  odata[$g$+1] = $C_1^{sw}$(idata) ▷ apply southwest interior boundary $C$ stencil
  odata[$g$+2] = $C_1^{sw}$(idata) ▷ apply southwest interior boundary $C$ stencil
**end if**
**if** $i = N + 1$ and $j = 1$ **then** ▷ northwest corner node
  odata[$g$] = $\left(M_{rr}^{nw} + M_{rs}^{nw} + C^{nw}\right)$ (idata) ▷ apply northwest partial $M$ and $C$ stencils
  odata[$g$+1] = $C^{nw}$(idata) ▷ apply northwest interior boundary $C$ stencil
  odata[$g$+2] = $C^{nw}$(idata) ▷ apply northwest interior boundary $C$ stencil
**end if**

To confirm our hypothesis, we use NVIDIA Nsight Compute to profile our code for the problem size of N=$2^{13}$. The profile shows that we achieve 72% L1 cache hit rate and 57% L2 cache hit rate, which indicates that the majority of our data is coming from the L1 and L2 caches (approximately 88%), and that our DRAM reads are due mostly to compulsory cache misses (i.e., when the input data is read for the first time). This explains why our code performs better than the DRAM-bounded performance. Figure 7 shows the Roofline model generated by Nsight Compute, based on performance counter measurements of how much of the overall data is coming from different levels of the memory hierarchy. Figure 7 confirms that the majority of our data comes from the L1 cache, followed by L2 and DRAM. It also suggests that we can further improve the performance of our `mfA!()` kernel by improving data reuse in the L1 cache, which will yield up to 3.8× speedup.

In future work, we will target improved performance of `mfA!()`, for example through additional memory optimization techniques to improve L1 cache hit rate, especially with respect to its performance on newer architectures. In the present work, however, we focus on utilizing `mfA!()` to solve the linear system with preconditioning.

## 8.2 Memory Usage Comparison

Next we compare the memory usage of `mfA!()` against the SpMV kernel via the built-in memory status function in CUDA.jl. CUDA.jl currently has good support for only three different sparse matrices: CSR, CSC, and COO. In Julia, the default sparse matrix format is CSC, but in CUDA.jl, the default sparse matrix format is CSR, and thus,

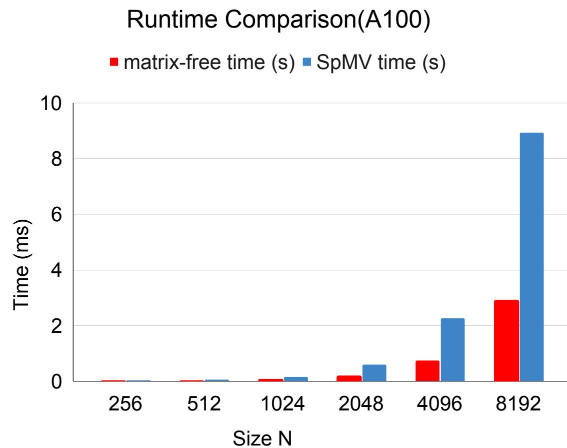


**Figure 4: Performance of SpMV vs matrix-free `mfA!()` on A100 GPU. Total time for matrix-free (red) and matrix-explicit CSR (blue) formats are shown in charts plotted against $N$, where the matrix is size $(N+1)^2 \times (N+1)^2$.**

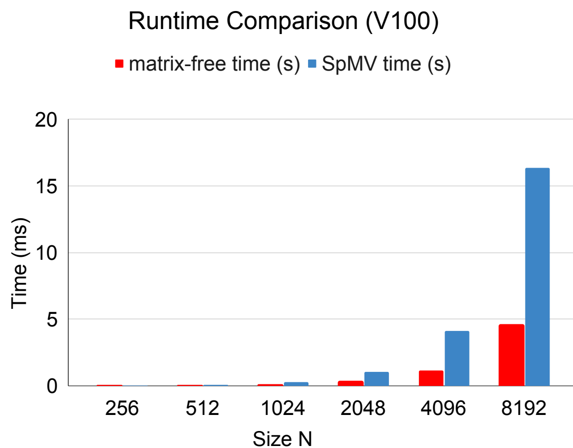


**Figure 5: Performance of SpMV vs matrix-free `mfA!()` on V100 GPU. Total time for matrix-free (red) and matrix-explicit CSR (blue) formats are shown in charts plotted against $N$, where the matrix is size $(N+1)^2 \times (N+1)^2$.**

there is a necessary conversion between these two formats when converting the CPU arrays to GPU arrays in Julia. However, for our problem, where the matrix is SPD, both CSR and CSC formats use exactly the same amount of memory; the only difference is in the use of row pointer `rowptr` values (for CSR) instead of column pointer values `colptr` (for CSC), and the order of nonzero values `nzval`. To avoid redundancy, we merge key results in memory consumption for CSC and CSR formats into three different numbers for each $N$. The collected data is given in Table 5.

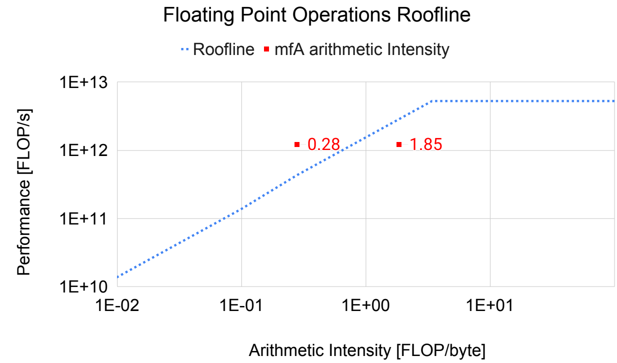


**Figure 6: Roofline model analysis for our matrix-free `mfA!()` on the A100 GPU. The red dot on the left represents the performance achieved by our kernel and its arithmetic intensity (0.28). The red dot on the right represents the same but assuming data is loaded only once from DRAM (i.e., compulsory misses), which yields a higher arithmetic intensity (1.85). The fact that our kernel (red dot) achieves higher performance than what is predicted by the Roofline model suggests that a large portion of our data is coming from the caches.**

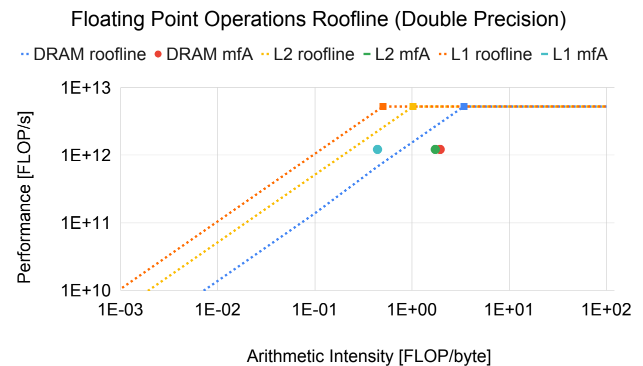


**Figure 7: Roofline model generated by Nsight Compute, based on performance counter measurements of how much of the overall data is coming from different levels of the memory hierarchy. This confirms our hypothesis that the majority of our data is coming from the L1 cache, and that further improving data reuse in L1 will yield up to $3.8\times$ speedup.**

For the matrix-free method, memory consumption is reported in Table 6. In order to perform the matrix-vector product, we need to allocate memory to store the coefficients $c_{rr}$, $c_{ss}$ and $c_{rs}$; each requires the same size of memory as the numerical solution and must be stored on each grid level when using geometric multigrid as a preconditioner. In addition, we must compute and store the minimum coefficient values $C_{rr}^{k,min}$ on faces 1 and 2, as specified in (8), which we denote $\Psi_1$ = and $\Psi_2$, respectively.

These are associated with Dirichlet boundary conditions and are significantly smaller in size, and thus reported in KB. Adding up

| $N$ | m | nzval | memory size |
|---|---|---|---|
| $2^{10}$ | 1050625 | 9447429 | 0.1596 GB |
| $2^{11}$ | 4198401 | 37769221 | 0.6379 GB |
| $2^{12}$ | 16785409 | 151035909 | 2.5509 GB |
| $2^{13}$ | 67125249 | 604061701 | 10.2020 GB |

**Table 5: Number of nonzero values (nzval) for CSC or CSR sparse matrices with different $N$, where matrix size is $(N+1)^2 \times (N+1)^2$. The matrices are SPD. Here, m represents the number of rows, and nzval represents the number of nonzero values. The total memory size (last column) is calculated using previous columns.**

| $N$ | crr/css/crs | $\Psi_1/\Psi_2$ | total memory size |
|---|---|---|---|
| $2^{10}$ | 0.008405 GB | 8 KB | 0.02523 GB |
| $2^{11}$ | 0.03359 GB | 16 KB | 0.1008 GB |
| $2^{12}$ | 0.1343 GB | 32 KB | 0.4029 GB |
| $2^{13}$ | 0.5370 GB | 65 KB | 1.6111 GB |

**Table 6: Memory allocation for matrix-free methods where matrix size is $(N{+}1)^2{\times}(N{+}1)^2$. Here crr, css, and csr correspond to coefficient matrices of size $(N+1)^2$. $\Psi_1$ and $\Psi_2$ are used in Dirichlet boundary conditions and are vectors of length $N+1$. Total memory allocated (last column) is calculated using previous columns.**

these contributions, we can compute the total memory size, which we provide in the last columns of Tables 5 and 6: We can see that there is a significant reduction in additional required memory for the matrix-free method than the memory to store sparse matrices in CSC or CSR format. When calculating the total memory used for an SpMV operation (including writing results into output vectors), we need to add additional memory allocated for the input data and output data, which require the same memory as the coefficients (the first column of Table 6). A simple calculation can show that the total memory required when using an SpMV kernel is a constant 4.2× of that required for the matrix-free method.

## 9 PERFORMANCE: MATRIX-FREE MGCG ON GPUS

With the matrix-free action of $\boldsymbol{A}$ established, we can solve system (9) with a matrix-free version of our custom MGCG method (MF-MGCG). Other than low-level GPU kernels, Julia also supports high-level vectorization for GPU computing, which we utilize extensively in our MGCG code for convenience. In this section, we compare its performance against MGCG using the cuSPARSE (matrix-explicit) SpMV (SpMV-MGCG) and also against the state-of-the-art off-the-shelf methods offered by NVIDIA, namely, AmgX - the GPU accelerated algebraic multigrid. The solvers and preconditioners used by AmgX are stored as JSON files. We explored different sample JSON configuration files for AmgX in the source code and found that CG preconditioned by classical AMG performed best for our problem. To maintain a multigrid setup comparable to our MGCG, we modified the `PCG_CLASSICAL_V_JACOBI.json` to use 1 and 5 smoothing steps with block Jacobi as the smoother. All algorithms stop when the relative residual is reduced to less than $10^{-6}$ times the initial residual. We report our results in Table 7. Also included in the table are results using a direct solve (using LU factorization in LAPACK in Julia) only because it is so often used in the earthquake cycle community for volume based codes [22] and our developed methods offer promising alternatives. As illustrated, the GPU-accelerated iteratives schemes achieve much better performance for the problem sizes tested.

Table 7 illustrates that our MGCG method uses fewer iterations to converge compared to AmgX, while iterations for both remain generally constant with increasing problem size. When we increase smoothing steps from 1 to 5, the AmgX sees reduced iterations, but the time to solve also increases by roughly 3×. Because we apply rediscretization (rather than Galerkin coarsening) for MGCG, the setup time is negligible. The setup time in the AmgX is comparable to the solve time however, which adds additional cost to use AmgX as a solver. Our SpMV-MGCG is roughly 2× slower than the AmgX using 1 smoothing step, but our MF-MGCG is faster than AmgX, up to 2× speedup for $N = 2^{13}$. Compared to our SpMV-MGCG, our MF-MGCG achieves more than 2× speedup, and the speedup is more obvious at $N = 2^{13}$, indicating that the MF-MGCG is suitable for large problems.

## 10 SUMMARY AND FUTURE WORK

In this work we present a matrix-free implementation of multigrid preconditioned conjugate gradient in order to solve 2D, variable coefficient elliptic problems discretized with an SBP-SAT method. Our custom multigrid preconditioner achieves similar preconditioning performance against the multigrid using Galerkin's condition from previous work, and it is more suitable for GPU code. The MGCG algorithm requires a nearly constant number of iterations to converge for various problem sizes. We explored several comparable solvers and preconditioners in PETSc and found that MGCG requires fewer iterations for the same convergence criteria. We developed matrix-free kernels that outperform the cuSPARSE SpMV kernels from NVIDIA (i.e. using CUDA.jl) in both runtime and memory usage. We used Nsight Compute to analyze the performance of our matrix-free kernel. This offers us more insights into the achieved computation and memory performance, which points directions for future kernel-level optimizations on newer GPU architectures. The resulting matrix-free MGCG method achieves better performance than several off-the-shelf solvers offered by NVIDIA's AmgX when tested on the same GPU. We also demonstrate Julia's ability to leverage both high-level vectorization and low-level GPU kernels for GPU computing, achieving comparable performance to packages in native CUDA C. This facilitates seamless integration of GPU-accelerated MGCG solvers into existing Julia code without external reliance on C solvers.

This work is a fundamental first step towards high-performance implementations to solve linear systems using SBP-SAT methods. Future work will target SBP-SAT methods with higher-order accuracy in 3D, as well as explorations of additional GPU kernel

**Table 7: Time to perform a direct solve after LU factorization on CPUs vs PCG on GPUs. We report time in seconds and iterations to converge. For AmgX, we report setup + solve time. For our MGCG, setup time is negligible. "ns" is short for the number of smoothing steps. GPU results are tested on A100.**

| $N$ | Direct Solve | AmgX (ns = 1) | AmgX (ns = 5) | SpMV-MGCG (ns = 5) | MF-MGCG (ns = 5) |
|---|---|---|---|---|---|
| $2^{10}$ | 0.912 s | (0.0319 s + 0.0243 s) / 25 | (0.0321 s + 0.0435 s) / 17 | 7.019E-2 s / 8 | 2.851E-2 s / 8 |
| $2^{11}$ | 6.007 s | (0.086 s + 0.161 s) / 55 | (0.086 s + 0.311 s) / 38 | 0.158 s / 7 | 0.0605 s / 7 |
| $2^{12}$ | 22.382 s | (0.310 s + 0.235 s) / 24 | (0.323 s + 0.488 s) / 15 | 0.564 s / 7 | 0.207 s / 7 |
| $2^{13}$ | 134.697 s | (1.334 s + 1.643 s) / 24 | (1.217 s + 1.865 s) / 16 | 5.028 s / 7 | 0.865 s / 7 |

optimization and multi-GPU implementation. We also plan to improve the performance of the preconditioner by systematic experiments with different preconditioner configurations using PETSc and applying second-order smoothers that have exhibited improved performance in the multigrid method as well as the mixed-precision techniques [2, 26, 27].

## ACKNOWLEDGMENTS

This work benefited from access to the University of Oregon high performance computing cluster, Talapas and Oregon Advanced Computing Institute for Science and Society (OACISS). This work is supported by NSF award #2053372 and #2036980.

## REFERENCES

[1] B. T. Aagaard, M. G. Knepley, and C. A. Williams. 2013. A domain decomposition approach to implementing fault slip in finite-element models of quasi-static and dynamic crustal deformation. *Journal of Geophysical Research: Solid Earth* 118, 6 (2013), 3059–3079.

[2] Ahmad Abdelfattah, Hartwig Anzt, Erik G Boman, Erin Carson, Terry Cojean, Jack Dongarra, Alyson Fox, Mark Gates, Nicholas J Higham, Xiaoye S Li, et al. 2021. A survey of numerical linear algebra methods utilizing mixed-precision arithmetic. *The International Journal of High Performance Computing Applications* 35, 4 (2021), 344–369.

[3] Sarah AlAhmadi, Thaha Mohammed, Aiiad Albeshri, Iyad Katib, and Rashid Mehmood. 2020. Performance analysis of sparse matrix-vector multiplication (SpMV) on graphics processing units (GPUs). *Electronics* 9, 10 (2020), 1675.

[4] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Steven Benson, Jed Brown, Peter Brune, Kris Buschelman, Emil M. Constantinescu, Lisandro Dalcin, Alp Dener, Victor Eijkhout, Jacob Faibussowitsch, William D. Gropp, Václav Hapla, Tobin Isaac, Pierre Jolivet, Dmitry Karpeev, Dinesh Kaushik, Matthew G. Knepley, Fande Kong, Scott Kruger, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Lawrence Mitchell, Todd Munson, Jose E. Roman, Karl Rupp, Patrick Sanan, Jason Sarich, Barry F. Smith, Stefano Zampini, Hong Zhang, Hong Zhang, and Junchao Zhang. 2023. PETSc Web page. https://petsc.org/. https://petsc.org/

[5] M. Baskaran and R. Bordawekar. 2009. Optimizing Sparse Matrix-Vector Multiplications on GPUs. *Computer Science* 8 (2009), 812–47.

[6] David A. Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J. Kunen, Olga Pearce, Peter Robinson, Brian S. Ryujin, and Thomas RW Scogland. 2019. RAJA: Portable Performance for Large-Scale Scientific Applications. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 71–81. https://doi.org/10.1109/P3HPC49587.2019.00012

[7] N. Bell and M. Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 1–11. https://doi.org/10.1145/1654059.1654078

[8] Akrem Benatia, Weixing Ji, Yizhuo Wang, and Feng Shi. 2018. BestSF: A sparse meta-format for optimizing SpMV on GPU. *ACM Transactions on Architecture and Code Optimization (TACO)* 15, 3 (2018), 1–27.

[9] T. Besard, C. Foket, and B. De Sutter. 2018. Effective extensible programming: unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 30, 4 (2018), 827–841. https://doi.org/10.1109/TPDS.2018.2872064

[10] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. 2017. Julia: A fresh approach to numerical computing. *SIAM review* 59, 1 (2017), 65–98. https://doi.org/10.1137/141000671

[11] Achi Brandt. 2006. Guide to multigrid development. In *Multigrid Methods: Proceedings of the Conference Held at Köln-Porz, November 23–27, 1981*. Springer, 220–312.

[12] William L. Briggs, Van Emden Henson, and Steve F. McCormick. 2000. *A Multigrid Tutorial, Second Edition* (second ed.). Society for Industrial and Applied Mathematics.

[13] J. Brown. 2010. Efficient Nonlinear Solvers for Nodal High-Order Finite Elements in 3D. *Journal of Scientific Computing* 45 (2010), 48–63. https://doi.org/10.1007/s10915-010-9396-8

[14] M. H. Carpenter, D. Gottlieb, and S. Abarbanel. 1994. Time-stable Boundary Conditions for finite-difference Schemes Solving Hyperbolic Systems: Methodology and Application to high-order Compact Schemes. *J. Comput. Phys.* 111, 2 (1994), 220–236. https://doi.org/10.1006/jcph.1994.1057

[15] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202–3216.

[16] J. W. Choi, A. Singh, and R. W. Vuduc. 2010. Model-Driven Autotuning of Sparse Matrix-Vector Multiply on GPUs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10)*. Association for Computing Machinery, New York, NY, USA, 115–126. https://doi.org/10.1145/1693453.1693471

[17] Valentin Churavy, Dilum Aluthge, Anton Smirnov, James Schloss, Julian Samaroo, Lucas C Wilcox, Simon Byrne, Tim Besard, Ali Ramadhan, Maciej Waruszewski, Simeon David Schaub, Meredith, William Moses, Jake Bolewski, Navid C. Constantinou, Max Ng, Carsten Bauer, Michel Schanen, johnbcoughlin, Viral B. Shah, Vaibhav Kumar Dixit, Tomas Chor, Tim Holy, Takafumi Arakaki, Sharan Yalburgi, Ruibin Liu, and Páll Haraldsson. 2024. *JuliaGPU/KernelAbstractions.jl: v0.9.18*. https://doi.org/10.5281/zenodo.10780635

[18] David C. Del Rey Fernández, Jason E. Hicken, and David W. Zingg. 2014. Review of summation-by-parts operators with simultaneous approximation terms for the numerical solution of partial differential equations. *Computers & Fluids* 95 (2014), 171–196.

[19] Nan Ding and Samuel Williams. 2019. *An instruction roofline model for gpus*. IEEE.

[20] B. A. Erickson and S. M. Day. 2016. Bimaterial effects in an earthquake cycle model using rate-and-state friction. *Journal of Geophysical Research: Solid Earth* 121 (2016), 2480–2506. https://doi.org/10.1002/2015JB012470

[21] B. A. Erickson and E. M. Dunham. 2014. 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. https://doi.org/10.1002/2013JB010614

[22] Brittany A Erickson, Junle Jiang, Michael Barall, Nadia Lapusta, Eric M Dunham, Ruth Harris, Lauren S Abrahams, Kali L Allison, Jean-Paul Ampuero, Sylvain Barbot, et al. 2020. The community code verification exercise for simulating sequences of earthquakes and aseismic slip (SEAS). *Seismological Research Letters* 91, 2A (2020), 874–890.

[23] Brittany A. Erickson, Jeremy E. Kozdon, and Tobias Harvey. 2022. A Non-Stiff Summation-By-Parts Finite Difference Method for the Scalar Wave Equation in Second Order Form: Characteristic Boundary Conditions and Nonlinear Interfaces. *Journal of Scientific Computing* 93, 1 (2022), 17.

[24] M. Fabien, M. G. Knepley, R. T. Mills, and B. M. Rivière. 2017. Heterogeneous computing for a hybridizable discontinuous Galerkin geometric multigrid method. *arXiv: Numerical Analysis* (2017).

[25] Robert D. Falgout and Ulrike Meier Yang. 2002. hypre: A Library of High Performance Preconditioners. In *Computational Science — ICCS 2002*, Peter M. A. Sloot, Alfons G. Hoekstra, C. J. Kenneth Tan, and Jack J. Dongarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 632–641.

[26] Gene H Golub and Richard S Varga. 1961. Chebyshev semi-iterative methods, successive overrelaxation iterative methods, and second order Richardson iterative methods. *Numer. Math.* 3, 1 (1961), 157–168.

[27] Martin H Gutknecht and Stefan Röllin. 2002. The Chebyshev iteration revisited. *Parallel Comput.* 28, 2 (2002), 263–283.

[28] J. E. Kozdon, E. M. Dunham, and J. Nordström. 2012. Interaction of Waves with Frictional Interfaces Using Summation-by-Parts Difference Operators: Weak

Enforcement of Nonlinear Boundary Conditions. *Journal of Scientific Computing* 50 (2012), 341–367. https://doi.org/10.1007/s10915-011-9485-3

[29] J. E. Kozdon, B. A. Erickson, and L. C. Wilcox. 2020. Hybridized Summation-By-Parts Finite Difference Methods. *Journal of Scientific Computing* (2020). https://doi.org/10.1007/s10915-021-01448-5

[30] H.-O. Kreiss and G. Scherer. 1974. Finite element and finite difference methods for hyperbolic partial differential equations. In *Mathematical aspects of finite elements in partial differential equations; Proceedings of the Symposium*. Madison, WI, 195–212. https://doi.org/10.1016/b978-0-12-208350-1.50012-1

[31] Marcin Krotkiewski and Marcin Dabrowski. 2013. Efficient 3D stencil computations using CUDA. *Parallel Comput.* 39, 10 (2013), 533–548.

[32] M. Kupiainen, J. Gong, L. Axner, E. Laure, and J. Nordström. 2020. GPU-Acceleration of A High Order Finite Difference Code Using Curvilinear Coordinates. In *Proceedings of the 2020 International Conference on Computing, Networks and Internet of Things (CNIOT2020)*. 41–47. https://doi.org/10.1145/3398329.3398336

[33] J. Kurzak, D. A. Bader, and J. J. Dongarra. 2010. Scientific Computing with Multicore and Accelerators. In *Chapman and Hall / CRC computational science series*. https://doi.org/10.1201/B10376

[34] Chang Liu and William Henshaw. 2023. Multigrid with Nonstandard Coarse-Level Operators and Coarsening Factors. *Journal of Scientific Computing* 94, 3 (2023), 58.

[35] Karl Ljungkvist. 2017. Matrix-Free Finite-Element Computations on Graphics Processors with Adaptively Refined Unstructured Meshes. In *Proceedings of the 25th High Performance Computing Symposium* (Virginia Beach, Virginia) *(HPC '17)*. Society for Computer Simulation International, San Diego, CA, USA, Article 1, 12 pages.

[36] G. C. Lotto and E. M. Dunham. 2015. High-order finite difference modeling of tsunami generation in a compressible ocean from offshore earthquakes. *Computational Geosciences* 19, 2 (2015), 327–340. https://doi.org/10.1007/s10596-015-9472-0

[37] M. Lubarda and V. Lubarda. 2019. Intermediate Solid Mechanics. , 228–258 pages. https://doi.org/10.1017/9781108589000.011

[38] Ken Mattsson. 2003. Boundary Procedures for Summation-by-Parts Operators. *Journal of Scientific Computing* 18, 1 (01 Feb 2003), 133–153.

[39] K. Mattsson, F. Ham, and G. Iaccarino. 2009. Stable Boundary Treatment for the Wave Equation on Second-Order Form. *Journal of Scientific Computing* 41, 3 (2009), 366–383. https://doi.org/10.1007/s10915-009-9305-1

[40] K. Mattsson and J. Nordström. 2004. Summation by parts operators for finite difference approximations of second derivatives. *J. Comput. Phys.* 199, 2 (2004), 503–540. https://doi.org/10.1016/j.jcp.2004.03.001

[41] Medina, DS and St-Cyr, A. and Warburton, T. 2014. OCCA: A unified approach to multithreading languages. *arXiv preprint arXiv:1403.0968* (2014).

[42] E. H. Müller, X. Guo, R. Scheichl, and S. Shi. 2013. Matrix-free GPU implementation of a preconditioned conjugate gradient solver for anisotropic elliptic PDEs. *Computing and Visualization in Science* 16 (2013), 41–58. https://doi.org/10.1007/s00791-014-0223-x

[43] J. Nordström and S Eriksson. 2010. Fluid Structure Interaction Problems: The Necessity of a Well Posed, Stable and Accurate Formulation. *Communications in Computational Physics* 8, 5 (2010), 1111–1138. https://doi.org/10.4208/cicp.260409.120210a

[44] Michal Osusky. 2013. *A parallel Newton-Krylov-Schur algorithm for the Reynolds-averaged Navier-Stokes equations.* Ph. D. Dissertation. University of Toronto.

[45] Ramesh Pankajakshan, P-H Lin, and Björn Sjögreen. 2019. Porting a 3D seismic modeling code (SW4) to CORAL machines. *IBM Journal of Research and Development* 64, 3/4 (2019), 17–1.

[46] N. Anders Petersson and Björn Sjögreen. 2012. Stable and Efficient Modeling of Anelastic Attenuation in Seismic Wave Propagation. *Communications in Computational Physics* 12, 1 (2012), 193–225. https://doi.org/10.4208/cicp.201010.090611a

[47] Hendrik Ranocha and Jan Nordström. 2021. A new class of A stable summation by parts time integration schemes with strong initial conditions. *Journal of Scientific Computing* 87, 1 (2021), 1–25.

[48] P. Roache. 1998. *Verification and Validation in Computational Science and Engineering* (1 ed.). Hermosa Publishers, Albuquerque, NM.

[49] Daniel Roten, Yifeng Cui, Kim B. Olsen, Steven M. Day, Kyle Withers, William H. Savran, Peng Wang, and Dawei Mu. 2016. High-Frequency Nonlinear Earthquake Simulations on Petascale Heterogeneous Supercomputers. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 957–968. https://doi.org/10.1109/SC.2016.81

[50] A. A. Ruggiu, P. Weinerfelt, and J. Norström. 2018. A new multigrid formulation for high order finite difference methods on summation-by-parts form. *J. Comput. Phys.* 359 (2018), 216–238. https://doi.org/10.1016/j.jcp.2018.01.011

[51] A. Sridhar, Y. Tissaoui, S. Marras, Z. Shen, C. Kawczynski, S. Byrne, K. Pamnany, M. Waruszewski, T. H. Gibson, J. E. Kozdon, V. Churavy, L. C. Wilcox, F. X. Giraldo, and T. Schneider. 2022. Large-eddy simulations with ClimateMachine v0.2.0: a new open-source code for atmospheric simulations on GPUs and CPUs. *Geoscientific Model Development* 15, 15 (2022), 6259–6284.

[52] B. Strand. 1994. Summation by parts for finite difference approximations for $d/dx$. *J. Comput. Phys.* 110, 1 (1994), 47–67. https://doi.org/10.1006/jcph.1994.1005

[53] M. Svärd and J. Nordström. 2014. Review of summation-by-parts schemes for initial-boundary-value problems. *J. Comput. Phys.* 268 (2014), 17–38. https://doi.org/10.1016/j.jcp.2014.02.031

[54] E. Swim, F.-K. Benra, H. J. Dohmen, J. Pei, S. Schuster, and B. Wan. 2011. A Comparison of One-Way and Two-Way Coupling Methods for Numerical Analysis of Fluid-Structure Interactions. *Journal of Applied Mathematics* 2011 (2011), 1–16. https://doi.org/10.1155/2011/853560

[55] O. Tatebe. 1993. The multigrid preconditioned conjugate gradient method. In *The Sixth Copper Mountain Conference on Multigrid Methods*. NASA. Langley Research Center.

[56] F. Vázquez, J. J. Fernández, and E. M. Garzón. 2011. A new approach for sparse matrix vector product on NVIDIA GPUs. *Concurrency and Computation: Practice and Experience* 23, 8 (2011), 815–826.

[57] Anamaria Vizitiu, Lucian Itu, Cosmin Niţă, and Constantin Suciu. 2014. Optimized three-dimensional stencil computation on Fermi and Kepler GPUs. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–6.

[58] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. 2014. YaSpMV: Yet Another SpMV Framework on GPUs. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Orlando, Florida, USA) *(PPoPP '14)*. Association for Computing Machinery, New York, NY, USA, 107–118. https://doi.org/10.1145/2555243.2555255

[59] Ulrike Meier Yang et al. 2002. BoomerAMG: A parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics* 41, 1 (2002), 155–177.

[60] W. Ying and C.S. Henriquez. 2007. Hybrid finite element method for describing the electrical response of biological cells to applied fields. *IEEE Transactions on Bio-medical Engineering* 54, 4 (2007), 611–620. https://doi.org/10.1109/TBME.2006.889172