Accelerating High-Order Stencils on GPUs

Ryuichi Sai  
Department of Computer Science  
Rice University  
Houston, TX, USA  
ruyichi@rice.edu

John Mellor-Crummey  
Department of Computer Science  
Rice University  
Houston, TX, USA  
johnmc@rice.edu

Xiaozhu Meng  
Department of Computer Science  
Rice University  
Houston, TX, USA  
xm13@rice.edu

Mauricio Araya-Polo  
Computational Science and Engineering  
Total E&P Research and Technology US, LLC.  
Houston, TX, USA

Jie Meng  
Computational Science and Engineering  
Total E&P Research and Technology US, LLC.  
Houston, TX, USA

Abstract—Stencil computations are widely used in HPC applications. Today, many HPC platforms use GPUs as accelerators. As a result, understanding how to perform stencil computations fast on GPUs is important. While implementation strategies for low-order stencils on GPUs have been well-studied in the literature, not all of the techniques work well for high-order stencils, such as those used for seismic imaging. Furthermore, coping with boundary conditions often requires different computational logic, which complicates efficient exploitation of the thread-level parallelism on GPUs. In this paper, we study practical seismic imaging computations on GPUs using high-order stencils on large domains with meaningful boundary conditions. We manually crafted a collection of implementations of a 25-point seismic modeling stencil in CUDA along with code to apply the boundary conditions. We evaluated our stencil code shapes, memory hierarchy usage, data-fetching patterns, and other performance attributes. We conducted an empirical evaluation of these stencils using several mature and emerging tools and discuss our quantitative findings. Among our implementations, we achieve twice the performance of a proprietary code developed in C and mapped to GPUs using OpenACC. Additionally, several of our implementations have excellent performance portability.

Index Terms—stencil computation, high-order, boundary condition, HPC, GPU

I. INTRODUCTION

"Remember that Time is Money." [1] This is even more true in today’s competitive business environments, such as the oil and gas industry, where fast simulations enable more realizations of experiments that can reduce the uncertainty of hydrocarbon reserves location or CO₂ storage management processes. Seismic depth imaging is the main tool used to extract information from seismic field records to identify relevant subsurface structures. High-order stencil computations typically serve as the foundation for seismic depth imaging. Over the past two decades, the availability of sufficiently powerful computational resources has led to the every-day use of more complex stencil-based wave equation approximations, and the power of today’s petascale systems enables simulations based on the full-wave equation instead of simple approximations.

Today, HPC platforms often employ Graphics Processing Units (GPUs) to increase their computational power. Accordingly, using GPUs to accelerate full-wave equation simulations based on high-order stencils is a natural approach. However, the complexity of GPU architectures makes achieving top performance with high-order stencil computations surprisingly difficult. Without careful design, a stencil computation on a GPU is likely to underperform. An efficient implementation of high-order stencils with boundary conditions on a GPU requires paying careful attention to data reuse, warp utilization, work balance, and arithmetic intensity among other issues. For that reason, understanding how to develop efficient high-order stencils for GPUs is a topic of significant interest.

Since GPUs from different vendors have different characteristics and the characteristics of GPUs from a single vendor often change significantly between generations, performance portability across GPUs with varying characteristics is of significant interest. The best kernel on one GPU may not be the best on GPUs from other vendors and may not remain the best on newer generations of GPUs.

For these reasons, our current goal is to identify how to achieve excellent performance for high-order stencils on GPUs, and understand the factors that affect performance portability. To do so, we need to understand the strengths and weaknesses of various code shapes for high-order stencils. This paper describes our progress toward this goal and makes the following contributions:

- a careful comparison of existing approaches, including an assessment of their strengths and weaknesses when applied to high-order stencils with boundary conditions;
- the implementation and tuning of a collection of high-order stencil kernels with a selected set of algorithms and their variants using CUDA;
- a performance comparison of stencil implementations across multiple generations of NVIDIA GPUs along with a quantitative assessment of their performance using a Roofline performance model for GPUs; and
- an investigation of the characteristics of different stencil kernels that affect their performance.

The next section is an overview of related work. Sections
III and IV present our approaches and implementations, respectively. Section V describes our evaluation methodology, experimental results, and a discussion of our findings. Section VI summarizes our conclusions and briefly discusses our plans for future work.

II. RELATED WORK

There is a rich literature describing efforts to efficiently implement stencil computations on CPUs [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [15], [16], [12] and GPUs [13], [14], [17], [18], [19], [22], [23]. We discuss the most related efforts below.

Time skewing [7], [8] accelerates stencil computations by increasing data reuse and cache locality by skewing one or more data dimensions by the time dimension so that several time steps can be computed for a tile while values are in cache. It has been widely used on CPUs, e.g., [9], [10], and [11].

Overlapped tiling uses time skewing to increase the arithmetic intensity of parallel stencil computations by trading redundant computation along the boundaries of overlapped tiles for a reduction in memory bandwidth required [12], [13]. Overlapped tiling is effective on GPUs because loading data from a GPU’s global memory is much more costly than data-parallel computation. Furthermore, redundant computation can be overlapped with data accesses to help hide memory latency. While overlapped tiling has been shown to improve the performance of low-order stencils on GPUs, for high-order stencils, redundant computation grows quickly when skewed across multiple time steps by the width of a high-order stencil.

Split tiling [14] is an alternate approach for accelerating computation with time skewing. Rather than using overlapped tiles, which can introduce large amounts of redundant computation, split tiling computes points in two phases. The first phase computes tiles in parallel as hypertrapezoids that taper along the time dimension. Once all tiles from the first phase have been computed, a second phase back-fills the missing points in the time dimension.

The semi-stencil algorithm [15], [16], which has only been studied on CPUs, factors the computation of a stencil into two or more pieces. Rather than computing the result for a point as a single computation, the semi-stencil algorithm divides computation along one or more axes into two halves. While sweeping along a dimension, it computes the final result for one point and a partial result for another point a half-stencil width ahead. Compared to the traditional implementation of stencils, this approach changes the load/store ratio for the computation by trading half of the loads along a dimension for a store and a reload of a partial result. On a GPU, this has the potential for nearly halving the cache footprint of a thread block for a high-order stencil.

Nguyen et al. [17] introduce a 3.5D blocking algorithm as a mix of 2.5D spatial blocking with 1D temporal blocking. 2.5D spatial blocking involves blocking in a 2D plane and streaming along a third dimension. To increase data reuse, they store active 2D planes in GPU shared memory. In a 3.5D variant, they employ time skewing to advance the computation for multiple time steps before writing data back to the global memory. While the 3.5D algorithm works very well on CPUs, the 1D temporal blocking introduces two potential implementation challenges for high-order stencils with boundary conditions on GPUs: barrier synchronizations and limited parallelism. In this paper, we evaluate 2.5D spatial blocking of high-order stencils and plan to explore 3.5D blocking in future work.

Nguyen et al.’s approach [17] loads a central plane along with halo planes above and below into shared memory for faster data access while computing stencil operations for points in the central plane. While this strategy improves data reuse, the size of a data tile is limited by the GPU shared memory size. To reduce the shared memory pressure, we looked into the work that uses registers on GPUs. Micikevicius [18] also uses 2.5D blocking; however, his approach maintains data points along the third dimension in registers rather than in shared memory.

Recently, as part of their AN5D framework work, Matsumura et al. [19] apply three more refinements to 2.5D and 3.5D solutions: fixed register allocations, double buffering, and division of the streaming dimension. While these approaches work extremely well for simple single-statement kernels, neither boundary conditions nor multi-statement stencils are evaluated. In our work, we study a high-order stencil with boundary conditions, and part of our application has multiple statements, instead of simple single-statement stencil updates.

Other interesting approaches to tackle the stencil computations including auto-tuning with dynamic resource allocations [20], DAG reordering [21], diamond tiling using polyhedral model [24], [25], functional programming [26], [27], and multi-layer intermediate representations [28], [29], [30].

From a software engineering perspective, there are two strategies for developing stencils for NVIDIA GPUs: hand-written kernels in CUDA and Domain-Specific Language (DSL)-based approaches [6], [22], [31], [32], [33], [34], [35]. The DSL approach can simplify the generation of code with complex logic. While we are interested in DSL-based approaches for the future, the focus of this paper is to understand in detail the strengths and weaknesses of various algorithmic strategies for achieving high performance and performance portability for high-order stencils. To avoid limitations as we explore this space, we chose to evaluate hand-written kernels.

III. APPROACH

We developed several implementations of the acoustic isotropic approximation of the wave equation [36] used for seismic imaging by the oil and gas industry. Solving this with finite differences involves using a high-order stencil-based solver with suitable boundary conditions. Oil and gas applications use such strategies on large grids to model subsurface and generate seismic data from source perturbations. In our work, we employ different code shapes that differ principally in how they organize the computation (e.g., 2D vs. 3D tiles) and how they manage the memory hierarchy. In the rest of this section, we will briefly describe the acoustic isotropic approximation, explain the various data decomposition strategies we employ,
A high-level description of the algorithm is shown in Algorithm 1. We apply a Perfectly-Matched Layer (PML) [37] boundary condition to the regions around the physical domain. The resulting extended domain consists of an “inner” region and the PML region. These separate kernels can be launched concurrently. This strategy eliminates the need for checking whether the point is inside inner region or PML region in every kernel, thus, reducing the chance of branch divergence. Nevertheless, it leaves unbalanced work among threads, along with the CUDA data domain decomposition.

Algorithm 1: A high-level description of the algorithm for solving the acoustic isotropic approximation of the wave equation with constant density.

Data: $f$: source
Result: $u^n$: wavefield at timestep $n$, for $n \leftarrow 1$ to $T$
1 $u^0 := 0$;
2 for $n \leftarrow 1$ to $T$ do
3     for each point in wavefield $u^n$ do
4         Solve Eq. 2 (left hand side) for wavefield $u^n$;
5     end
6     $u^n = u^n + f^n$ (Eq. 2 right hand side);
7 end

A high-level description of the algorithm for solving the acoustic isotropic approximation of the wave equation with constant density.

Data: $f$: source
Result: $u^n$: wavefield at timestep $n$, for $n \leftarrow 1$ to $T$
1 $u^0 := 0$;
2 for $n \leftarrow 1$ to $T$ do
3     for each point in wavefield $u^n$ do
4         Solve Eq. 2 (left hand side) for wavefield $u^n$;
5     end
6     $u^n = u^n + f^n$ (Eq. 2 right hand side);
7 end

A high-level description of the algorithm for solving the acoustic isotropic approximation of the wave equation with constant density.

Data: $f$: source
Result: $u^n$: wavefield at timestep $n$, for $n \leftarrow 1$ to $T$
1 $u^0 := 0$;
2 for $n \leftarrow 1$ to $T$ do
3     for each point in wavefield $u^n$ do
4         Solve Eq. 2 (left hand side) for wavefield $u^n$;
5     end
6     $u^n = u^n + f^n$ (Eq. 2 right hand side);
7 end

A high-level description of the algorithm for solving the acoustic isotropic approximation of the wave equation with constant density.

Data: $f$: source
Result: $u^n$: wavefield at timestep $n$, for $n \leftarrow 1$ to $T$
1 $u^0 := 0$;
2 for $n \leftarrow 1$ to $T$ do
3     for each point in wavefield $u^n$ do
4         Solve Eq. 2 (left hand side) for wavefield $u^n$;
5     end
6     $u^n = u^n + f^n$ (Eq. 2 right hand side);
7 end

A high-level description of the algorithm for solving the acoustic isotropic approximation of the wave equation with constant density.

Data: $f$: source
Result: $u^n$: wavefield at timestep $n$, for $n \leftarrow 1$ to $T$
1 $u^0 := 0$;
2 for $n \leftarrow 1$ to $T$ do
3     for each point in wavefield $u^n$ do
4         Solve Eq. 2 (left hand side) for wavefield $u^n$;
5     end
6     $u^n = u^n + f^n$ (Eq. 2 right hand side);
7 end

A high-level description of the algorithm for solving the acoustic isotropic approximation of the wave equation with constant density.

Data: $f$: source
Result: $u^n$: wavefield at timestep $n$, for $n \leftarrow 1$ to $T$
1 $u^0 := 0$;
2 for $n \leftarrow 1$ to $T$ do
3     for each point in wavefield $u^n$ do
4         Solve Eq. 2 (left hand side) for wavefield $u^n$;
5     end
6     $u^n = u^n + f^n$ (Eq. 2 right hand side);
7 end

A high-level description of the algorithm for solving the acoustic isotropic approximation of the wave equation with constant density.

Data: $f$: source
Result: $u^n$: wavefield at timestep $n$, for $n \leftarrow 1$ to $T$
1 $u^0 := 0$;
2 for $n \leftarrow 1$ to $T$ do
3     for each point in wavefield $u^n$ do
4         Solve Eq. 2 (left hand side) for wavefield $u^n$;
5     end
6     $u^n = u^n + f^n$ (Eq. 2 right hand side);
7 end

A high-level description of the algorithm for solving the acoustic isotropic approximation of the wave equation with constant density.
the boundaries between the inner and PML regions when the size of the GPU blocks doesn’t evenly divide the extents of the PML and central regions.

Lastly, we developed the strategy as shown in Figure 1. We separate the inner region from PML region, and further divide PML region into six subregions. We slice the domain along the top and bottom of the inner region, and this gives us a top block, a bottom block, and a border of four walls. We further slice along the front and back, and it becomes four separate walls. These four walls and the two subregions from the first cuts result in total of six subregions of the PML region, namely: top, bottom, front, back, left and right subregions.

The symmetry of these subregions is a relevant characteristic that we discuss later along with our results. Next, we launch individual GPU kernels of stencil computations for each of the seven subregions: one for the inner region and six for the PML subregions. This approach does not have intrinsic branch divergence at the boundaries. While there are still work imbalances due to different grid sizes, they occur only for a few edge cases along the borders. We could further reduce unbalanced work by using automated code generation that tailors the number of threads to match the number of points at border locations.

C. Blocking Strategies

For each of the seven regions, we further slice it into smaller blocks, so that each block can fit into GPU’s resources for each kernel launch. We use two blocking strategies in our experiments: 3D Blocking and 2.5D Blocking.

1) 3D Blocking: We divide each of the data regions into axis-aligned 3D blocks. To find the best block dimensions, we use fixed values in each execution to simplify experiments with different values. To perform stencil computations on GPUs, each block maps to a kernel launch with a 3D thread block with the thread dimensions matching the block dimensions. All points inside the block and their halos are explicitly copied into the GPU on-chip memory before any kernel is launched.

2) 2.5D Blocking: We partition the data domain along the inner two X and Y data dimensions and perform a streaming computation along the outermost Z dimension. We launch kernels with 2D thread blocks with their dimensions matching the 2D planes.

D. Kernels and their variants

We implemented several kernels. Each kernel employs a different combination of strategies for blocking, managing data accesses, and traversing the data volume. To better understand the strengths and weaknesses of these implementation alternatives, every implementation has multiple variants, which employ different tile sizes. We describe the details of our kernel implementations in the next section.

IV. IMPLEMENTATIONS

In describing our implementations, we use $R$ to denote the width of the halo, which is half the spatial order of the stencil. For the acoustic isotropic simulations in our experiments, $R$ is 4. Let $N_x$, $N_y$, and $N_z$ denote the extents of the input data region along the $X$, $Y$, and $Z$ axes, respectively. For 3D blocks, we use $(x, y, z)$ to denote the 3D coordinate for both a point location in a 3D block and the thread in a kernel thread block. Similarly, for 2D planes, $(x, y)$ is used to locate a point in the 2D plane, as well as identifying the thread.

1) 3D Blocking Using Global Memory Only: This is conceptually and practically the simplest kernel to understand and implement. Let $D_x$, $D_y$, and $D_z$ denote the block dimensions in the $X$, $Y$, and $Z$ axes, respectively. Thus the block size is $D_x \times D_y \times D_z$. Because we launch each kernel with a thread block of the same size, the total number of points must be $\leq 1024$ to respect the GPU limit of at most 1024 threads per block. The GPU grid size of each data region is $\lceil N_x/D_x \rceil \times \lceil N_y/D_y \rceil \times \lceil N_z/D_z \rceil$.

During execution of the 25-point stencil kernel, each thread fetches the point for itself, as well as 4 neighboring points along each direction of each axis. For good performance, we ensure a good memory access pattern when stencil points are fetched directly from global memory. Since we store the
3D grid data as a flat 1D array, we ensure global memory coalescing for the most innermost dimension \( X \).

We refer to the family of 3D kernel implementations that fetch stencil points directly from the \( u \) array in global memory as \( gmem_{\{ Dx \}, \{ Dy \}, \{ Dz \}} \) in our experiments.

2) **3D Blocking Using Shared Memory for the \( u \) Array:**
This approach is a variant of the aforementioned 3D blocking using global memory. It uses same 3D blocking strategy for each of the data regions; however, instead of computing directly on data fetched from global memory, this implementation fetches the \( u \) array from global memory, stores it into shared memory, and performs the stencil computation on data fetched from shared memory. The total number of points we fetch in this case is \( Dx \times Dy \times Dz \) for a block and \( (Dx \times Dy + Dx \times Dz + Dy \times Dz) \times R \times 2 \) for halos around the block. For high order stencils, one must account for the halo size to ensure both the block and the halo fit in shared memory.

For high-order stencils, the halo accounts for a significant fraction of the data to fetch into shared memory. Thus, designing the right approach to minimize the fetch cost is crucial to overall performance. We describe the most general approach with good performance based on our experiments.

First, thread \((i, j, k)\) fetches the point \((i, j, k)\). Then, when we design the block size, for each thread dimension, we must have at least \( 2R \) threads, and we use the first \( 2R \) threads along each dimension to fetch the halos. Along each dimension, threads 0 to \( R - 1 \) fetch the halo on one side, and threads \( R \) to \( 2R - 1 \) fetch the halo on the other side. Fetching is perfectly balanced for each thread when \( |D| = 2R \).

To use this strategy for the acoustic isotropic model where \( R = 4 \), we need to have at least eight threads along each dimension. Considering that maximum number of threads in a block is 1024 and the total amount of shared memory per block, the only possible tile size for this case is \( 8 \times 8 \times 8 \). This results in perfectly balanced fetch work and computation for threads along each dimension.

Both the point and halo fetching need to be done in a fashion that enables global memory coalescing to the greatest extent possible.

We refer to the implementation that uses 3D blocking and shared memory as \( smem_u \) in our experiments.

3) **3D Blocking Using Shared Memory for Boundary Regions:**
This implementation also exploits shared memory and uses 3D blocking. The main difference between this approach and the previous one is that this implementation fetches the \( eta \) array into shared memory, whereas the previous approach fetches the \( u \) array into shared memory. Because \( eta \) is only used in the stencil computation inside the PML region, this strategy applies only in the PML kernel.

This approach may appear to be nothing new; however, it is interesting for two reasons. First, as previously described, computations on \( eta \) in the PML region use a low-order 7-point stencil rather than the 25-point high-order stencil of the inner region. In fact, the halo size of \( eta \) is just one. Such low-order stencils have been widely studied in the literature; however, the combination of high and low order stencils is seldom addressed. Second, this implementation gives us an opportunity to observe the performance changes by using global memory with a good access pattern for a high-order stencil, meanwhile using shared memory for a lower-order stencil.

In terms of fetching \( eta \) into shared memory, we have two implementations that differ in the number of conditionals. In our experiments, we refer to the shared memory kernel implementation that uses three conditionals as \( smem_eta_3 \) and the implementation that uses one conditional as \( smem_eta_1 \). We let \( R_\eta \) denote the width of halos for \( eta \), and for the acoustic isotropic PML layer, \( R_\eta \) is 1.

\( smem_eta_3 \) uses an approach similar to \( smem_u \), where the first \( 2R_\eta \) threads from each dimension fetch the halos. Since we have three dimensions, we need three conditionals, one for each dimension, respectively. Because \( R_\eta \) is just 1, we only need two threads fetching halos along each thread dimension. This could introduce unbalanced fetch work because for a 3D block of \( 8 \times 8 \times 8 \), if only two threads in each dimension perform halo fetching, that is one fourth of the threads. Because we have three dimensions, only 1/64 of the threads perform fetching, while others are idle.

![Algorithm 2: Shared Memory Fetching Strategy for \( eta \)-array Using Only One Conditional](image)

To address the work imbalance, we propose \( smem_eta_1 \) with only one condition, where we choose to use the first six threads from the \( X \) dimension to perform halo fetching. Algorithm 2 describes how we tilt the six planes of threads to identify the halo point that the each thread is responsible for the fetching. For \( 8 \times 8 \times 8 \) 3D blocks, because 6/8 threads perform the fetching, in theory, we reduce thread idleness to just 20%. However, this algorithm has relatively complex arithmetic to tilt each thread to its proper halo position, so an
Before we can start the streaming computation, points from the top halos are pre-loaded into buffer $B[0..R]$ and the first $R$ XY-subplanes are pre-loaded into $B[R..2R]$. Then, in our streaming loop, for each $z \leftarrow [0..N_z]$, we first load the $((z+R)\ mod\ 2(R+1))$ points, next, we perform the stencil computation for the $z$-th XY-subplane with the stencil points read from $B$ in shared memory; finally, we store the result back to global memory.

While we explain this strategy using a modulus operator, in practice, we avoid using it for speed. Since the $z$ index always increases by one inside the streaming loop, we use loop unrolling and index rotation to achieve the desired effect without modulus computations.

We refer to the family of kernel implementations of this strategy as $st_{smem}(Dx)(Dy)$ in our experiments.

The shared memory buffer used by this approach is limited by the GPU shared memory size. Alternatively, one can store data for stencil points along the streaming dimension in registers. We discuss two approaches that use registers to store points along the streaming dimension in the following sections.

6) 2.5D Streaming using Register Shifting: In this 2.5D streaming approach, we keep points of the current XY-subplane in shared memory. However, when we stream along the $z$-axis, we use registers for the points along the $z$-axis. In contrast to shared memory, where data loaded from one thread is accessible by other threads in the same block, registers are only accessible by the current thread. Since we are streaming along the $z$-axis, the data from $z$-axis loaded for one thread is not needed by other threads.

So we allocate a shared memory space to hold $(Dx+2R) \times (Dy+2R)$ points for the currently active plane. The shared memory footprint compared to the previous method is $1 : (2R+1)$. For high-order stencils, $R$ is large so that the shared memory usage reduction is significant. Let $S(x,y)$ denotes the shared memory with location $(x,y)$.

We also allocate $2R+1$ registers for the current point and its neighbors in each direction along the $z$-axis. Let $Reg(x,y)[i]$ denote the $i$-th register for the thread $(x,y)$.

Before we can start the streaming computation, thread $(x,y)$ fetches data values from $(x,y,z)$ for $z \leftarrow [0..R]$ and stores them into register $Reg(x,y)[0..2R]$, respectively. Then, inside the stream loop, for each $z \leftarrow [0..N_z]$, we first shift the register indices back one position on each thread, such that for $r \leftarrow (0..2R]$, $Reg(x,y)[r-1] = Reg(x,y)[r]$. Then, we load the leading point along the streaming dimension $(x,y,z+R)$ into register $Reg(x,y)[2R]$. Next, we fetch data $(x,y,z)$ from global memory into $S(x,y)$; and we finally perform the stencil computation by using the data of XY-subplanes from shared memory and data along the $z$-axis from registers. Finally, the kernel stores the stencil result for each thread back to global memory.

We refer to the family of kernel implementations using this strategy as $st_{req_shft}(Dx)(Dy)$ in our experiments.

Although the notation we use above for registers might give the impression that we are using array indexing to access

experimental evaluation is needed to see whether the strategy is profitable.

4) Semi-stencil: Semi-stencil was initially introduced for CPUs, where it separates the stencil computations into two phases, namely the forward phase and the backward phase. The algorithm reads $R+1$ points on one dimension, and the forward phases compute as the points are the left side of the stencil, and the partial result is stored to the rightmost point. Backward phases then compute as if the points are the right side of the stencil, and write out the final result to the leftmost point. By doing this, semi-stencil has the potential to improve performance by changing the ratio between load and store. For example, for a 3D stencil of halo size of $R$, with typical approach, it requires to load $6 \times R+1$ points in order to perform stencil computation for one point. Once the computation is done, a single store writes the result back. So the load-store ratio is $(6 \times R+1) : 1$. On the other hand, using semi-stencil on one dimension, we read the center point and the half points with $R+1$ loads, then forward phase writes one store, and backward phase write another store, thus total the half points with dimension in our data layout. Let $Dx$ plane through the third dimension. In our implementations, already described, the 2.5D algorithm mainly streams a 2D further parallelize the executions by running all time steps CPU implementations described in the original paper. we the stencil computation in each 3D block is close to the adopting semi-stencil algorithm on GPUs.

Our GPU implementation also uses 3D blocking. While the stencil computation in each 3D block is close to the CPU implementations described in the original paper, we further parallelize the executions by running all time steps concurrently on GPUs.

We assign the identifier semi to this implementation.

5) 2.5D Streaming with Multi-Plane using Shared Memory: Starting from this implementation, we use 2.5D blocking. As already described, the 2.5D algorithm mainly streams a 2D plane through the third dimension. In our implementations, we choose the 2D XY-subplane because $X$ is the innermost dimension in our data layout. Let $Dx$ and $Dy$ denote the dimensions of the 2D tile along the $X$ and $Y$ axes, respectively. So we launch kernels using 2D thread blocks with the size of $Dx$ by $Dy$ and a total of $Dx \times Dy$ threads. The GPU grid size of each data region is $\lceil Nx/Dx \rceil \times \lceil Ny/Dy \rceil$.

In this approach, we exploit shared memory as a buffer to store all data needed in the stencil computations for a particular XY-subplane. In addition to the current XY-subplane, we also load $R$ subplanes above the current subplane and $R$ subplanes below into the shared memory. Therefore, we allocate a buffer for $2R+1$ planes, where each has $(Dx+2R) \times (Dy+2R)$ points, thus, total of $(2R+1) \times (Dx+2R) \times (Dy+2R)$ points. Hence, the extent of each subplane must be carefully chosen so that the buffer size is as large as possible to enhance data reuse, but at the same time, it must be chosen so that the aggregate data volume of the planes doesn’t exceed the shared memory available to a block. Let $B$ denote our buffer, and $B[i]$ denote the $i$-th subplane in the buffer.

Before we can start the streaming computation, points from the top halos are pre-loaded into buffer $B[0..R]$ and the first $R$ XY-subplanes are pre-loaded into $B[R..2R]$. Then, in our streaming loop, for each $z \leftarrow [0..N_z]$, we first load the $((z+R)\ mod\ 2(R+1))$ points, next, we perform the stencil computation for the $z$-th XY-subplane with the stencil points read from $B$ in shared memory; finally, we store the result back to global memory.
register values, in our implementation, registers are expressed explicitly as $2R + 1$ scalar variables. Since our acoustic isotropic kernel has $R = 4$, which is the same as the sample code by Micikevicius [18], our implementation uses the same variable names: behind4, behind3, behind2, behind1, current, front1, front2, front3, and front4.

7) 2.5D Streaming using Fixed Registers with Loop Unrolling: Like the previous approach, this implementation uses shared memory for the current XY-subplane and registers for points along the z-axis—the streaming dimension. However, the values in the registers are fixed instead of being “shifted.”

We again allocate a shared memory of $(Dx + 2R) \times (Dy + 2R)$ points. Let $S(x, y)$ denotes the shared memory for location $(x, y)$. We allocate $2R + 1$ registers as well, and denote $Reg(x, y)[i]$ for the $i$-th register of the thread $(x, y)$. In practice, they are $2R + 1$ named variables.

Before we can start the streaming computation, thread $(x, y)$ fetches data from $(x, y, z)$ for $z \leftarrow [{-R}, R]$ and stores them into register $Reg(x, y)[0..2R]$, respectively. Then, inside the streaming loop, for each $z \in [0..Nz]$, we do not modify any value in the existing registers; we only update register $Reg(x, y)[(z + 2R) \mod (2R + 1)]$ with the value of point $(x, y, z + R)$. Then, we fetch data $(x, y, z)$ from global memory into $S(x, y)$. Next, we perform the stencil computation by using the data of XY-subplane from shared memory and $i$-th data above current point from $Reg(x, y)[(z + R - i) \mod (2R + 1)]$, and $j$-th data below the current point from $Reg(x, y)[(z + R + j) \mod (2R + 1)]$. Finally, the kernel stores the result back to global memory.

To further improve performance, we unroll the streaming loop. We introduce macros with register indices as macro placeholders. Inside the streaming loop, we expand $2R + 1$ macro calls, each with register indices shifted by one. We check and exit the loop when the stream reaches the boundary of z-axis.

We refer to the family of kernel implementations using this strategy as stencil_fixed_{Dx}_{Dy} in our experiments.

V. EVALUATION

A. Experiment Environments

We evaluate all kernel implementations on three machines with NVIDIA GPUs across several generations. Table I lists our machine specifications. And we refer to the three machines by their GPU models.

- Machine V100 is equipped with four NVIDIA V100 GPUs. We use one dedicated GPU for our experiments. We use the compiler option -arch=sm_70 to compile all kernels for this platform.
- Machine P100 is equipped with four NVIDIA P100 GPUs. We use one dedicated GPU for our experiments. We use the compiler option -arch=sm_60 to compile all kernels for this platform.
- Machine NVS510 has one NVIDIA NVS510 GPU. We use the compiler option -arch=sm_30 to compile all kernels for this platform.

On NVS510, support for some tooling is marked as deprecated. While the tools work to some extent, many have limited functionality. Also, the GPU memory available on NVS510 doesn’t support grid sizes needed for real-world use. Therefore, we only use this machine for basic comparisons across GPU generations. While we examine some metrics on this platform, we don’t discuss them in detail.

In most situations, we let the nvcc compiler figure out the register usage by itself, but we pay very close attention to the resulting register footprint. However, there are a few cases, where we specify the maximum number of registers used by a kernel using the compiler flag --maxrregcount=X to prevent register spilling.

We also use HPCToolkit [41, 42] version 20200803 and Empirical Roofline Toolkit [40] and NVIDIA Nsight Compute version 2019.5.0 during our evaluations.

B. Evaluation Methodologies

We evaluate all implementations and their variants. First, we conduct basic time measurements. Second, we use HPCToolkit’s GPU support [42] to profile the kernel details with PC sampling. Third, we run Nsight Compute for device-specific kernel characteristics. Finally, we use the Empirical Roofline Toolkit to understand memory bandwidth limits on algorithm performance. We calculate the arithmetic intensity and the performance of every kernel, and compare them with the roofline chart. We describe each of our evaluation methods below.

1) Time Measurements: For each machine, based on its device memory size, we run the kernels with a large grid size supported by the device memory. For V100, we use a grid size of 1000$^3$; for P100, we use a grid size of 893$^3$; for NVS510, we use a grid size of 300$^3$. As described in III-A, the stencil needs multiple iterations to converge. For benchmarking, we use 1000 iterations for all kernels on all machines. For each execution, we warm up the kernel by

| Machine   | GPU       | RAM (GB) | CPU       | CUDA Ver. | OS              |
|-----------|-----------|----------|-----------|-----------|-----------------|
| V100      | Tesla V100| 256      | Intel Xeon| 10.2.89   | RHEL v7.7       |
| P100      | Tesla P100| 256      | Intel Xeon| 10.2.89   | Ubuntu 18.04 LTS|
| NVS510    | NVIDIA NVS510 | 16       | Intel Xeon| 10.2.89   | RHEL v7.7       |

Table I: Machine Specifications

1We use a local fork of the tool for better Python 3 support and other minor changes needed for our environments. Our changes are made available at https://github.com/rsrice/cs-roofline-toolkit-fork under the same open-source license as the upstream repository https://bitbucket.org/berkeleylab/cs-roofline-toolkit/src/master/.
running the entire execution once, and then we repeat it five times, recording the average time for the five runs.

2) HPCToolkit: We use the August 2020 release of the HPCToolkit to collect GPU kernel metrics, such as register use, block and grid size, as well as PC sampling statistics such as exposed latencies and their kinds.

The evaluation contains four steps: Firstly, running hpcrun along with kernel executions for sampling using program counters. Thanks to the very low overhead of hpcrun, we can run our kernels with 1000 time iterations, which gives us accurate measurements that match the kernel behavior in the real world. Then, we use hpcstruct on the kernel binaries and recover the information about their relations to the source code. This is needed to contribute the performance metrics back to the source code, so that we can evaluate it later at the source code level, which makes the investigation easier. The source code structure computed by hpcstruct is then associated by hpcprof with the raw sampling data from hpcrun. Finally, a HPCToolkit performance database is generated.

HPCToolkit provides two graphic user interface to analyze the performance database, namely, HPCTviewer and HPCTraceviewer. We use HPCTviewer primarily for issues such as memory stalls, which enables us to easily spot which source lines have the most significant stalls. We also use HPCTviewer to quickly identify the performance hotspots in our kernel executions using its code-centric views. We use HPCTraceviewer to inspect the program execution over time, which enables us to quickly spot idleness and see the associated calling contexts.

We describe some of our findings using HPCToolkit in our discussion of the evaluation results.

3) Nsight Compute: We run Nsight Compute for kernel characteristics, such as theoretical and achieved occupancy. Nsight Compute provides insights when performance differences are driven by the kernel characteristics. For example, when low occupancy happens, one can easily tell from an Nsight Compute report whether or not the problem seems to be associated with the register footprint, the shared memory footprint, or the number of threads.

Nsight Compute re-plays every kernel execution multiple times to collect a complete set of measurements, which adds a huge measurement overhead. When we use Nsight Compute, we run only five iterations.

4) Roofline Performance Model: We use the GPU Roofline performance model to see how well our kernels perform relative to a machine’s practical peak based on each kernel’s arithmetic intensity and the memory bandwidth-based performance limit for that particular arithmetic intensity.

We use the Empirical Roofline Toolkit (ERT) for machine characterizations. It runs several micro-benchmarks to characterize the peak compute speed and memory bandwidth of the machine. Benchmarking directly on a machine gives us an achievable performance bound, which is substantially lower than the theoretical peak claimed by the manufacturers when a kernel is memory bound.

We characterize kernels using nvprof by measuring several kernel performance metrics, including FLOPs, L2 read and write transactions, as well as DRAM read and write transactions. Output from nvprof is then fed into the calculations of both the performance and the arithmetic intensity for each kernel. Performance is calculated by the division of the measured FLOPS by the measured execution time. Arithmetic intensities are calculated by the division of the measured FLOPS by the measured bytes accessed on DRAM and L2 cache respectively.

We compare the performance of each kernel with the peak performance of the machine it runs on. We then compare kernels by their arithmetic intensities and their relative performance.

C. Results

In this section, we first present a summary of our results in tables and plots. After presenting our findings, we discuss our kernel measurements from several perspectives.

Table II presents time measurements for the kernels. For 3D blockings, the columns $Dx$, $Dy$, and $Dz$ stand for the block dimensions along the x, y, and z axes, respectively. For 2.5D blockings, only columns for $Dx$ and $Dy$ are reported since the z-axis is unpartitioned. For $Nr$ column, only values we explicitly specified are reported, and we use ~ for the ones that compiler decide.

Table III presents the kernel characteristics for inner data region at the top, and PML regions at the bottom, respectively. As previously discussed, there are three symmetric groups for the PML subregions: top/bottom, front/back, and left/right. We group them in Table III. For shared characteristics across data regions, we further extract them into the Static column.

Table IV presents the performance characteristics of our implementations on the V100. Figure 3 visualizes these performance characteristics using roofline performance model, where Subfigures 3a and 3b showing the rooflines for L2 and DRAM, respectively, and Subfigures 3c and 3d are zoomed-in views the roofline kernel characteristics. The y-axes of these figures represent performance and x-axes show arithmetic intensity. The dots in each group of the implementations are categorized with the same color and their coordinates can be found in Table IV.

From our results, we offer the following observations.

3D Blocking using Global Memory:

The simplest implementation gmem_8x8x8 using only the global memory yields the best performance on V100. With L1 data cache and shared memory combined into a single unified memory block on the V100 [43], we have a much larger data cache available on the V100 than on previous generations of GPUs. Therefore, when retrieving data from global memory with a good access pattern, we can achieve very good performance.

Comparing the performance of the 3D kernel across GPU generations, we notice its poor performance portability. It is one of the slowest implementations on P100 and the NVS510.
Table II: Time Measurement

| Kernel Identifier | Dx | Dy | Dz | Nr | V100     | Machine |
|-------------------|----|----|----|----|----------|---------|
| gmem_4x4x4        | 4  | 4  | 4  | -  | 77.77    | 181.99  |
| gmem_8x8x4        | 8  | 8  | 4  | -  | 71.91    | 167.75  |
| gmem_8x8x8        | 8  | 8  | 8  | -  | 53.88    | 117.74  |
| gmem_16x16x4      | 16 | 16 | 4  | -  | 85.52    | 195.82  |
| gmem_32x32x1      | 32 | 32 | 1  | -  | 292.36   | 639.62  |
| smem_u            | 8  | 8  | 8  |    | 57.30    | 76.18   |
| smem_eta_1        | 8  | 8  | 8  | 54  | 54.87    | 119.15  |
| smem_eta_3        | 8  | 8  | 8  |    | 54.34    | 117.39  |
| semi              | 8  | 8  | 8  |    | 172.84   | 217.29  |
| st_smem_8x8       | 8  | 8  | -  | -  | 116.38   | 112.71  |
| st_smem_8x16      | 8  | 16 | -  | -  | 113.46   | 105.41  |
| st_smem_16x8      | 16 | 8  | -  | -  | 59.92    | 77.91   |
| st_smem_16x16     | 16 | 16 | -  | -  | 55.87    | 72.73   |
| st_reg_shift_8x8  | 8  | 8  | -  | -  | 104.36   | 144.89  |
| st_reg_shift_16x16| 16 | 16 | -  | -  | 65.79    | 80.23   |
| st_reg_shift_16x32| 16 | 32 | -  | -  | 65.61    | 82.25   |
| st_reg_shift_16x64| 16 | 64 | -  | -  | 115.54   | 98.19   |
| st_reg_shift_32x16| 32 | 16 | -  | -  | 60.83    | 70.63   |
| st_reg_shift_32x32| 32 | 32 | -  | -  | 93.92    | 76.27   |
| st_reg_shift_64x16| 64 | 16 | -  | -  | 90.98    | 80.67   |
| st_reg_shift_64x32| 64 | 32 | -  | -  | 113.88   | 152.75  |
| st_reg_shift_64x64| 64 | 64 | -  | -  | 70.24    | 84.05   |
| st_reg_shift_64x128| 128| 64 | -  | -  | 61.66   | 76.10   |
| st_reg_shift_128x16|16  | 128| -  | -  | 62.45     | 66.60   |
| st_reg_shift_256x16|16 | 256| -  | -  | 58.96    | 61.74   |

We tried several variants of the global memory implementation that differ each others in terms of block size, from smaller to larger, including gmem_4x4x4, gmem_8x8x4, gmem_8x8x8, gmem_16x16x4, and gmem_32x32x1. Our results show that, gmem_8x8x8 is the best among them. We need to load all halos before performing stencil computations. For blocks smaller than gmem_8x8x8, such as gmem_4x4x4 and gmem_8x8x4, the halo size for our 25-point stencil dominates the actual data points. Therefore, more time is spent on loading halos than points for the volume to be computed, which hurts performance. In addition, smaller block sizes also result in larger GPU grid size as we can see from Table III which means more kernel launches. For the smaller blocks, the additional overheads slow the overall execution. On the other hand, we also see performance degradation for larger blocks, gmem_16x16x4 and gmem_32x32x1. Their larger block size results in a smaller grid size. However, both have low theoretical and achieved occupancy. Table IV shows that the 3D kernels using larger blocks, especially gmem_32x32x1, incur more L2 cache misses, which increases the number of high-latency loads from global memory.

In summary, the global memory implementations are the simplest to program and need very little performance tuning. With the right tile shape and using a good global memory access pattern, on late-model GPU architectures, such as V100, one can achieve amazingly good performance with little effort. From a software engineering perspective, these implementations are easy to understand and have a low maintenance cost.

Table IV shows that using shared memory can boost performance. The yield performance gain is more significant on older generation GPUs, such as P100 and NVS510, which is consistent with results in previous research.

Recall that smem_u is a high-order stencil while smem_eta_1 and smem_eta_3 are not. From Table IV we saw smem_u runs faster than smem_eta_1 and smem_eta_3 on V100, but oppositely, it is slower on P100 and NVS510. We attribute this conflicting results to the architectural changes in V100, where it combines the L1 data cache with shared memory. As discussed previously, on V100, with good access patterns for global memory, one can achieve great performance with little effort. The overhead of using shared memory on V100 in 3D blocking erases this gain. In contrast, older generation GPUs do not have this new feature, so shared memory provides more performance benefits than its overhead. On older architectures, with high-order stencils, such as smem_u, because we load larger-size blocks into shared memory than low-order ones, such as smem_eta_1 and smem_eta_3, we see better performance.

For high-order stencils, which have a large halo size, it is not hard to reach the shared memory limit. While shared memory improves performance, the hardware limitation on shared memory limits the potential of holding all data on shared memory for high-order stencils which use large blocks.

**Semi-stencil:**

Thread synchronizations on GPUs are very expensive, and our evaluations also prove so with our semi-stencil implementation. Because of the need of storing and loading partial results, thread synchronizations are necessary to ensure the
completeness of the required computation before it can proceed to the next. As GPU runs threads concurrently in warps, we must introduce proper barriers to prevent data from being corrupted. Our choice of using 3D blocking requires thread synchronizations on all three dimensions, which exacerbates the problem. HPCToolkit also backs our reasoning with its synchronizations on all three dimensions, which exacerbates corrupted. Our choice of using 3D blocking in our future work.

Nevertheless, the methodology behind semi-stencil algorithm is still valid. Thus, to avoid the excess use of thread synchronizations, we will investigate into using double buffering. As well, we will explore using 2.5D blocking instead of 3D blocking in our future work.

**Code Shape for 2.5D-Blockings:**

For implementations using 2.5D-blocking, we observe the larger the 2D plane, the better the performance. There are

| Kernel Identifier | Block Size | Grid Size | Registers Per Thread | Achieved Active Warps | Achieved Occupancy | Theoretical Active Warps | Theoretical Occupancy |
|-------------------|------------|-----------|-----------------------|------------------------|---------------------|--------------------------|------------------------|
| gmem_4x4x4        | 64         | 13,312,053| 40                    | 37.2                   | 58.2                | 48.0                     | 75.0                   |
| gmem_8x8x4        | 256        | 3,356,157 | 40                    | 44.0                   | 68.7                | 48.0                     | 75.0                   |
| gmem_8x8x8        | 512        | 1,685,159 | 40                    | 42.5                   | 66.4                | 48.0                     | 75.0                   |
| gmem_16x16x4      | 1,024      | 853,200   | 40                    | 28.9                   | 45.2                | 32.0                     | 50.0                   |
| gmem_32x32x1      | 1,024      | 851,400   | 40                    | 29.3                   | 45.8                | 32.0                     | 50.0                   |
| smem_u            | 512        | 1,685,159 | 38                    | 44.6                   | 69.7                | 48.0                     | 75.0                   |
| smem_e1           | 512        | 1,685,159 | 40                    | 42.4                   | 66.3                | 48.0                     | 75.0                   |
| smem_e3           | 512        | 1,685,159 | 40                    | 42.4                   | 66.3                | 48.0                     | 75.0                   |
| st_smem_8x8       | 128        | 7,140     | 56                    | 27.9                   | 43.6                | 28.0                     | 43.7                   |
| st_smem_16x8      | 256        | 3,600     | 56                    | 27.9                   | 43.5                | 28.0                     | 43.7                   |
| st_reg_shift_8x8  | 64         | 14,161    | 96                    | 29.7                   | 49.7                | 32.0                     | 50.0                   |
| st_reg_shift_16x8 | 256        | 3,600     | 96                    | 15.9                   | 24.9                | 16.0                     | 25.0                   |
| st_reg_shift_32x16| 1,024      | 900       | 64                    | 32.0                   | 50.0                | 32.0                     | 50.0                   |
| st_reg_shift_32x16| 1,024      | 900       | 64                    | 32.0                   | 50.0                | 32.0                     | 50.0                   |
| st_reg_shift_32x32| 1,024      | 900       | 64                    | 32.0                   | 50.0                | 32.0                     | 50.0                   |

**TABLE III: Kernel Characteristics on V100: (top) Inner; (bottom) PML**
two main reasons for this. First, a larger 2D plane means a higher degree of concurrency. Second, with a larger plane, the percentage of halo points fetched into shared memory is smaller, which speeds up the overall performance.

In addition, our results show that \texttt{st\_reg\_shft\_16x32} runs faster than \texttt{st\_reg\_shft\_16x32}. From Table IV we see more L2 transactions with \texttt{st\_reg\_shft\_16x32}, which in turn harms performance. Therefore, one should cut the plane so that the x-dimension of the GPU’s thread block assigned to the innermost dimension has a relatively larger size.

\textbf{Register Footprint in 2.5D-Blockings:}
When we evaluate \texttt{st\_reg\_shft\_*} implementations, the variants with 2D plane size of 1024, namely \texttt{st\_reg\_shft\_16x64}, \texttt{st\_reg\_shft\_32x32}, and \texttt{st\_reg\_shft\_64x16}, show poor performance on V100. The performance degradation is caused by register spilling. The maximum registers in a blockthread is \(64 \times 1024 = 65536\). Because we have 1024 threads for these implementations, we can only have maximum 64 registers for each thread. If we do not explicitly specify the register count to \texttt{nvcc}, it assigns 80 and 96 registers to the PML and inner kernels, respectively. Running the generated binaries for these register footprints yields incorrect results. To avoid this problem, we use compiler flag \texttt{-maxrregcount=64} to limit the maximum register usage per thread. Unfortunately 64 registers are not enough to hold all of the variables at the same time, causing register spilling. The register shifting approach exacerbates register spilling due to its high frequency of register access. However, although register spilling happens to the register shifting kernels, for the \texttt{st\_fixed\_reg\_32x32} kernel, we don’t see a performance degradation because the code uses fixed registers with loop unrolling. Because the registers are fixed, the frequency of register data movement is smaller than for kernels using the register shifting approach. This allows the performance impacted by register spilling to be hidden by other thread activities.

\textbf{GPU Warp Occupancies:}
Table III shows implementations using 2.5D blocking in general have better theoretical and achieved occupancies than the ones using 3D blocking.

\textbf{Performance Portability:}
The best performing implementations on P100 and NVS510 come from 2.5D approaches. Although they are not the best kernels on V100, they are still in the fastest tier. Thus, if performance portability is a concern, implementations using 2.5D blocking, such as \texttt{st\_reg\_fixed\_32x32}, would be preferred.

\textbf{Gaps to the Roofline Ceilings:}
Our interpretation of the performance gaps are twofold:
First, although our current implementations realize a good performance for high-order stencils with boundary conditions, we see room for further performance tuning. We could improve the arithmetic intensities by designing new GPU code shapes, e.g., by employing the semi-stencil algorithm [15], [16], which reduces data movement for high-order stencils.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|}
\hline
Kernel Identifier & FLOP (\times 10^{13}) & Achieved Performance (GFLOPs) & L2 Trans. (\times 10^{12}) & L2 Arithmetic Intensity & L2 Machine Peak Performance (GFLOPs) & L2 Achieved Percentage & DRAM Trans. (\times 10^{11}) & DRAM Arithmetic Intensity & DRAM Machine Peak Performance (GFLOPs) & DRAM Achieved Percentage \\
\hline
\texttt{gmem\_4x4x4\_opt} & 4.453 & 577 & 3.38 & 0.41 & 1361 & 39.19\% & 8.42 & 1.65 & 1291 & 41.29\% \\
\texttt{gmem\_8x8x8\_opt} & 4.453 & 485 & 2.81 & 0.49 & 1635 & 35.27\% & 7.26 & 1.92 & 1498 & 38.50\% \\
\texttt{gmem\_16x16x4\_opt} & 4.453 & 142 & 2.45 & 0.57 & 1877 & 25.83\% & 6.67 & 2.08 & 1628 & 29.78\% \\
\texttt{gmem\_32x32x1\_opt} & 4.453 & 724 & 1.90 & 0.78 & 2566 & 30.00\% & 7.26 & 1.92 & 1498 & 51.39\% \\
\texttt{smem\_u\_opt} & 4.453 & 756 & 2.67 & 0.75 & 330 & 42.95\% & 6.67 & 2.12 & 1656 & 8.57\% \\
\texttt{smem\_eta\_1\_opt} & 4.453 & 345 & 1.82 & 0.77 & 2535 & 30.10\% & 7.31 & 1.90 & 1488 & 51.30\% \\
\texttt{smem\_eta\_3\_opt} & 4.453 & 356 & 1.59 & 0.87 & 2891 & 12.33\% & 12.30 & 1.13 & 885 & 40.27\% \\
\texttt{semi\_opt} & 6.400 & 366 & 1.47 & 0.95 & 3130 & 11.68\% & 13.30 & 1.05 & 820 & 44.58\% \\
\texttt{st\_smem\_8x8\_opt} & 4.453 & 356 & 2.45 & 0.57 & 1877 & 25.83\% & 6.67 & 2.08 & 1628 & 29.78\% \\
\texttt{st\_smem\_8x16\_opt} & 4.453 & 692 & 1.59 & 0.87 & 2891 & 12.33\% & 12.30 & 1.13 & 885 & 40.27\% \\
\texttt{st\_smem\_8x32\_opt} & 4.453 & 373 & 1.47 & 0.95 & 3130 & 11.68\% & 13.30 & 1.05 & 820 & 44.58\% \\
\texttt{st\_reg\_shft\_16x16\_opt} & 4.453 & 360 & 1.20 & 1.16 & 3841 & 16.41\% & 7.22 & 1.93 & 1506 & 41.86\% \\
\texttt{st\_reg\_shft\_32x32\_opt} & 4.453 & 373 & 1.15 & 1.21 & 3991 & 15.84\% & 6.76 & 2.06 & 1607 & 39.32\% \\
\texttt{st\_reg\_shft\_64x16\_opt} & 4.453 & 359 & 1.99 & 0.70 & 2317 & 15.49\% & 17.00 & 0.82 & 638 & 56.25\% \\
\texttt{st\_reg\_shft\_32x32\_opt} & 4.453 & 682 & 0.94 & 1.47 & 4861 & 14.02\% & 6.94 & 2.00 & 1566 & 43.54\% \\
\texttt{st\_reg\_shft\_64x16\_opt} & 4.453 & 456 & 1.57 & 0.89 & 2938 & 15.52\% & 14.50 & 0.96 & 752 & 60.64\% \\
\texttt{st\_reg\_fixed\_8x8\_opt} & 4.453 & 364 & 1.65 & 0.84 & 2791 & 13.05\% & 15.00 & 0.93 & 723 & 50.36\% \\
\texttt{st\_reg\_fixed\_16x8\_opt} & 4.453 & 590 & 1.27 & 1.10 & 3632 & 16.26\% & 9.59 & 1.45 & 1133 & 52.11\% \\
\texttt{st\_reg\_fixed\_16x16\_opt} & 4.453 & 673 & 1.18 & 1.18 & 3899 & 17.25\% & 7.71 & 1.80 & 1409 & 47.72\% \\
\texttt{st\_reg\_fixed\_32x16\_opt} & 4.453 & 664 & 9.12 & 1.53 & 5043 & 13.17\% & 7.14 & 1.95 & 1522 & 43.62\% \\
\texttt{st\_reg\_fixed\_32x32\_opt} & 4.453 & 703 & 1.09 & 1.27 & 4209 & 16.71\% & 9.08 & 1.53 & 1197 & 58.78\% \\
\hline
\end{tabular}
\caption{Kernel Performance Characteristics on V100}
\end{table}
or employing time-skewing to increase data reuse. While all of our implementations were manually written in CUDA, we could develop a new DSL approach or build a framework that enables us to explore more sophisticated approaches.

Second, ERT uses simple micro-benchmarks to profile the machines. In contrast, the acoustic isotropic model not only uses high-order stencils with complex boundary conditions, but also contains complicated logic with multiple statements. Thus, while the roofline ceilings provide us a guide, the logic of our complex kernels makes it difficult to hit the roofline limit.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we evaluated the performance of high-order stencils with boundary conditions. We reviewed the existing techniques for implementing 3D stencil computations and evaluated most of them suitable for high-order stencils, with the notable exception of time-skewing algorithms. We implemented multiple versions of different approaches using known techniques and combinations of them, modifying approaches as needed to better accommodate high-order stencils with boundary conditions. We evaluated our implementations with several tools on multiple platforms, quantitatively comparing stencils from various perspectives, and presented our findings and observations.

We began our evaluation by computing 25-point stencil algorithms over the entire data domain in a single kernel launch. Inefficiency caused by branch divergence led us to apply domain decomposition and compute the boundaries separate from the center region. While this improved efficiency, having the regions separate impedes our ability to apply time skewing along the streaming dimension for the 2.5D algorithm. We plan to reintegrate boundary computations with the inner region to enable us to evaluate 3.5D algorithms on high-order stencils by adding time skewing to the streaming dimension. In addition, we would like to explore whether applying the semi-stencil algorithm [15] along the streaming dimension to reduce the memory hierarchy footprint of a block’s stencil calculations and increase the arithmetic intensity of high-order kernels. We plan to experiment with the range of kernels on the NVIDIA A100 GPU to assess performance portability. In addition to NVIDIA GPUs, we plan to expand the scope of

![Graphs showing roofline performance on V100](image)

**Fig. 3: Roofline Performance on V100**
our evaluation to explore the performance of various high-order stencil implementations on leading-edge GPUs from other vendors, as soon as we can gain access to them and results on them are not embargoed.

We recognize that implementing, tuning, maintaining, and porting high-performance GPU kernels for high-order stencils is quite difficult. For the long term, we believe that a high-level representation of stencil computations in conjunction with powerful compiler technology is arguably the best strategy to improve development productivity and performance portability while also lowering maintenance costs by reducing complexity. However, a concern for DSL users is the long-term viability of their code. We are hopeful that adding DSL technology to the LLVM compiler framework will provide a path forward that will address this concern.

ACKNOWLEDGMENT

This work was supported in part by a contract from Total E&P Research & Technology USA, LLC. We thank Keren Zhou from Rice University for reviewing the drafts of this paper and helping us use his emerging GPU Performance Advisor tool, which offered insights for tuning some of the kernels we studied.

REFERENCES

[1] F. Benjamin, “Advice to a Young Tradesman,” National Archives and Records Administration/University of Virginia Press, July 1748.

[2] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, “Cache-Oblivious Algorithms,” in Proceedings of the 40th Annual Symposium on Foundations of Computer Science, USA, Oct. 1999, p. 285.

[3] M. Frigo and V. Strumpen, “Cache oblivious stencil computations,” in Proceedings of the 19th annual international conference on Supercomputing, Cambridge, Massachusetts, Jun. 2005, pp. 361-366.

[4] M. Frigo and V. Strumpen, “The cache complexity of multithreaded cache oblivious algorithms,” in Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures, Cambridge, Massachusetts, USA, Jul. 2006, pp. 271-280.

[5] R. Strzodka, M. Shaheen, D. Pajak, and H.-P. Seidel, “Cache oblivious parallelograms in iterative stencil computations,” in Proceedings of the 24th ACM International Conference on Supercomputing, Tsukuba, Ibaraki, Japan, Jun. 2010, pp. 49-59.

[6] Y. Tang, R. A. Chowdhury, B. C. Kuszmial, C.-K. Luk, and C. E. Leiserson, “The pochoir stencil compiler,” in Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures, San Jose, California, USA, Jun. 2011, pp. 117-128.

[7] D. Wonnapcott, “Using time skewing to eliminate idle time due to memory bandwidth and network limitations,” in Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000, May 2000, pp. 171-180.

[8] D. Wonnapcott, “Achieving Scalable Locality with Time Skewing,” International Journal of Parallel Programming, vol. 30, no. 3, pp. 181-221, Jun. 2002.

[9] G. Innocenti, J. Mellor-Crummey, and R. Fowler, “Increasing Temporal Locality with Skewing and Recursive Blocking,” in SC01: Proceedings of the 2001 ACM/IEEE International Symposium on Code Generation and Optimization, San Diego, CA, USA, Feb. 2020, pp. 199-211.

[10] P. S. Rawat, M. Vaidya, A. Sukumaran-Rajam, A. Rountev, L.-N. Pouget, and P. Sadayappan, “On Optimizing Complex Stencils on GPUs,” in 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), May 2019, pp. 641-652.

[11] P. S. Rawat, F. Rastello, A. Sukumaran-Rajam, L.-N. Pouget, A. Rountev, and P. Sadayappan, “Register optimizations for stencils on GPUs,” SIGPLAN Not., vol. 53, no. 1, pp. 168-182, Feb. 2018.

[12] P. Rawat et al., “SDSLc: a multi-target domain-specific compiler for stencil computations,” in Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, Austin, Texas, Nov. 2015, pp. 1-10.

[13] P. S. Rawat et al., “Domain-Specific Optimization and Generation of High-Performance GPU Code for Stencil Computations,” Proceedings of the IEEE, vol. 106, no. 11, pp. 1902-1920, Nov. 2018.

[14] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, “A practical automatic polyhedral parallelizer and locality optimizer,” SIGPLAN Not., vol. 43, no. 6, pp. 101-113, Jun. 2008.

[15] V. Bandisht, I. Pananilath, and U. Bondhugula, “Tiling stencil computations to maximize parallelism,” in SC 12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Nov. 2012, pp. 1-11.

[16] M. Steuwer, T. Remml, and C. Dubach, “LIFT: A functional data-parallel IR for high-performance GPU code generation,” in 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Feb. 2017, pp. 74-85.

[17] M. L’ccke, M. Steuwer, and A. Smith, “A functional pattern-based language in mlir,” p. 6, 2020.

[18] O. Fuhrer et al., “Towards a performance portable, architecture agnostic implementation strategy for weather and climate models,” Supercomput. Front. Innov., vol. 1, no. 1, pp. 45-62, Apr. 2014.

[19] J.-M. Gioussis and T. Grosser, “Modeling Stencils in a Multi-Level Intermediate Representation,” p. 15, 2019.

[20] T. Gysi et al., “Domain-Specific Multi-Level IR Rewriting for GPU,” arXiv:2005.13014 [cs], May 2020, Accessed: Jul. 19, 2020. [Online]. Available: http://arxiv.org/abs/2005.13014

[21] R. Baghdatli et al., “PETSs: A Python-Based Neural Compute Intermediate Language for Accelerator Programming,” in 2015 International Conference on Parallel Architecture and Compilation (PACT), Oct. 2015, pp. 138-149.

[22] R. Baghdatli et al., “Tiramisu: a polyhedral compiler for expressing fast and portable code,” in Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization, Washington, DC, USA, Feb. 2019, p. 19. Accessed: Jul. 20, 2020. [Online].

[23] M. Christen, O. Schenk, and H. Burkhart, “PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures,” in 2011 IEEE International Parallel Distributed Processing Symposium, May 2011, pp. 676-687.

[24] M. Louboutin et al., “Devito (v3.1.0): an embedded domain-specific language for finite differences and geophysical exploration,” Geoscientific Model Development, vol. 12, no. 3, pp. 1165-1187, Mar. 2019.
[35] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, “Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines,” in Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, Seattle, Washington, USA, Jun. 2013, pp. 519-530.

[36] J. Meng, A. Atle, H. Calandra, and M. Araya-Polo, “Minimod: A Finite Difference solver for Seismic Modeling,” Jul. 2020, Accessed: Aug. 15, 2020. [Online]. Available: https://arxiv.org/abs/2007.06048v1.

[37] D. Komatitsch, J. Tromp, “A perfectly matched layer absorbing boundary condition for the second-order seismic wave equation,” Geophysical Journal International, volume 154, number 1, pp. 146-153, July 2003.

[38] S. Williams, A. Waterman, and D. Patterson, “Roofline: an insightful visual performance model for multicore architectures,” Commun. ACM, vol. 52, no. 4, pp. 65-76, Apr. 2009.

[39] N. Ding and S. Williams, “An Instruction Roofline Model for GPUs,” in 2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), Nov. 2019, pp. 7-18.

[40] C. Yang, “Hierarchical Roofline Analysis on GPUs,” ECP Annual Meeting, February 2020.

[41] J. Mellor-Crummey, R. Fowler, and D. Whalley, “Tools for application-oriented performance tuning,” in Proceedings of the 15th international conference on Supercomputing, Sorrento, Italy, Jun. 2001, pp. 154-165.

[42] K. Zhou, M. Krentel, and J. Mellor-Crummey, “A tool for top-down performance analysis of GPU-accelerated applications,” in Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, Feb. 2020, pp. 415-416.

[43] NVIDIA, “NVIDIA Tesla V100 GPU Architecture,” August 2017.