Received: 28 February 2021 | Revised: 23 May 2021 | Accepted: 9 June 2021

DOI: 10.1002/cpe.6467

WILEY

SPECIAL ISSUE PAPER

Accelerating high-order stencils on GPUs

Ryuichi Sai[1] | John Mellor-Crummey[1] | Xiaozhu Meng[1] | Keren Zhou[1] | Mauricio Araya-Polo[2] | Jie Meng[2]

[1]Department of Computer Science, Rice University, Houston, Texas, USA

[2]Computational Science and Engineering, Total E&P Research and Technology US, LLC., Houston, Texas, USA

Correspondence
Ryuichi Sai, Department of Computer Science, Rice University, 6100 Main Street, MS-132, Houston, TX 77005, USA.
Email: ryuichi@rice.edu

Funding information
Total E&P Research & Technology USA, LLC

Summary

Finite-difference methods based on high-order stencils are commonly used for modeling of seismic wave propagation, weather forecasting, computational fluid dynamics, convolutional neural networks, and others. Nowadays, the community commonly employs graphics processing units (GPUs) to accelerate such stencil computations. As a result, knowing how to write efficient stencil computations for GPUs is of significant interest. While high-performance, low-order stencils on GPUs have been studied extensively in the literature, not all proposed approaches work well for high-order stencils. Furthermore, coping with boundary conditions used with stencils for seismic modeling makes it challenging to efficiently exploit thread-level parallelism on GPUs. In this article, we describe several implementations of a 25-point stencil. We evaluate our stencil code shapes, memory hierarchy usage, data access patterns, and other performance attributes on several modern GPUs and compare them with machine rooflines. On average, our top-performing kernels achieve six times the performance of a 25-point stencil code developed in C and mapped to GPUs using OpenACC. Several of our implementations have excellent performance portability across multiple generations of both NVIDIA and AMD GPUs.

KEYWORDS
boundary condition, GPU, high-order, HPC, stencil computation

1 | INTRODUCTION

To accelerate stencil computations, compute nodes in high performance computing (HPC) platforms used for seismic modeling often employ graphics processing units (GPUs) as accelerators. Understanding how to develop efficient high-order stencils for GPUs is therefore a topic of great interest. Nevertheless, accelerating high-order stencils on GPUs is surprisingly difficult due to the complexity and variety of GPU architectures. Without careful tuning, stencil implementations for GPUs are likely to deliver performance that falls far short of what is possible. An efficient implementation of a high-order stencil on a GPU requires careful attention to data reuse, warp utilization, work balance, data movement, cache alignment, and arithmetic intensity among other issues.

Performance characteristics are usually different for GPUs from different vendors, and they often change significantly between generations from a single vendor. As 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 from the same vendor, performance portability across GPUs with varying characteristics is critical.

This article explores strategies to achieve excellent performance for high-order stencils on GPUs and understand the factors that affect performance portability. To do so, we investigate the strengths and weaknesses of various code shapes for high-order stencils. This article makes the following contributions:

- a careful assessment of existing approaches, including their strengths and weaknesses when applied to high-order stencils with boundary conditions;
- a collection of implementations of high-order stencil kernels with a selected set of algorithms and their variants;
- an investigation of the characteristics of high-order stencil kernels that affect their performance, including register usage, memory footprints, exposed latencies and stalls, as well as theoretical and achieved occupancy;
- an exploration of performance optimization strategies for high-order stencils;
- a performance comparison across multiple generations of NVIDIA GPUs (NVS510, P100, V100, and A100) and AMD GPUs (MI50 and MI100); and
- a quantitative assessment of kernels we developed using the Roofline performance model on various GPUs.

Section 2 presents background about GPU performance issues and the seismic modeling problem that is the target of the stencils that we study. Section 3 reviews related work. Section 4 describes our approaches in general. Section 5 presents our implementations. Section 6 details optimizations needed to achieve excellent performance. Section 7 describes our evaluation methodology, experimental results, and a discussion of our findings. Section 8 summarizes our conclusions and plans for future work.

2 | BACKGROUND

To understand our high-order stencil implementations and evaluation, we briefly introduce necessary background about stencil computations, especially the ones used in seismic modeling, and GPU performance issues.

2.1 | Stencil computations

In stencil computations, data elements from a multi-dimensional array are iteratively updated according to a fixed pattern. The array, representing a volume of data, is often called a grid. An element in the grid is usually called a cell or a point. Calculating the next value for a cell using a stencil involves computing a weighted sum of products between values of set of neighboring cells (the set of cells used are defined by the stencil) and scaling coefficients.

Applying a stencil pattern to the points in a block requires values for points in neighboring blocks. Collectively, the points needed from neighboring blocks are known as the halo region. The thickness of the halo along each dimension is called the halo size or halo width, and it also defines the order of the stencil. When a stencil has a large halo width, it is called a high-order stencil.

A stencil computation is typically applied to a data grid over a sequence of iterations.

2.2 | Seismic modeling

In this article, we study stencil-based implementations of the acoustic isotropic approximation of the wave equation[1] used for seismic modeling. The oil and gas industry applies such imaging strategies on large grids to model subsurface and generate seismic data from source perturbations. The wave equation for an acoustic isotropic operator with constant-density has the following form:

$$\frac{1}{\mathbf{V}^2}\frac{\partial^2 \mathbf{u}}{\partial t^2} - \nabla^2 \mathbf{u} = \mathbf{f}, \tag{1}$$

where $\mathbf{u} = \mathbf{u}(x, y, z)$ is the wavefield, $\mathbf{V}$ is the Earth model (with velocity as rock property), and $\mathbf{f}$ is the source perturbation. The equation is discretized in time using a second-order centered stencil, resulting in the semi-discretized equation:

$$\mathbf{u}^{n+1} - \mathbf{Q}\mathbf{u}^n + \mathbf{u}^{n-1} = (\Delta t^2)\mathbf{V}^2\mathbf{f}^n, \text{with } \mathbf{Q} = 2 + \Delta t^2 \mathbf{V}^2 \nabla^2. \tag{2}$$

Finally, the equation is discretized in space using a 25-point stencil in 3D, with eight points in along each axis surrounding a center point, where $c_{xyz}, c_{xm}, c_{ym}, c_{zm}$ are the discretization parameters:

$$\begin{aligned} \nabla^2 \mathbf{u}(x, y, z) \approx\ & c_{xyz} \times \mathbf{u}(i, j, k) + \\ & \sum_{m=1}^{4} c_{xm} \times [\mathbf{u}(i+m, j, k) + \mathbf{u}(i-m, j, k)] + c_{ym} \times [\mathbf{u}(i, j+m, k) + \mathbf{u}(i, j-m, k)] + c_{zm} \times [\mathbf{u}(i, j, k+m) + \mathbf{u}(i, j, k-m)]. \end{aligned} \tag{3}$$

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 ← 1 to T
u^0 := 0 for n ← 1 to T do
    for each point in wavefield u^n do
        Solve Equation (2) (left-hand side) for wavefield u^n
    end
    u^n = u^n + f^n (Equation 2 right-hand side)
end
```

A high-level description of the algorithm is shown in Algorithm 1. As is common for seismic modeling, our simulations employ a perfectly matched layer (PML)[2] boundary condition around the simulation domain. The resulting extended domain consists of an “inner” region and a surrounding “PML” region.

To compute values for the inner region using the acoustic isotropic wave equation, we apply a multi-statement stencil that is eighth-order in space and second-order in time. This involves applying a star-shaped 25-point stencil to elements of a 3D array, known as the *u*-array. Computation in the PML region is more complex than that in the inner region. In the PML region, we employ the same 25-point stencil applied in the inner region and also a 7-point star-shaped stencil to another array, known as the *eta*-array, to compute boundary conditions.

The grid, representing the physical domain, tends to be substantial in production simulations. Each of its dimensions is usually large with up to 4000 points. To simulate how the waves propagate through the domain, it is necessary to apply the stencil computations iteratively for a large number of time steps.

While implementations of the acoustic isotropic kernel on GPUs are the focus of this article, the techniques we explore are useful for other high-order stencils with boundary conditions.

2.3 | GPU performance issues

High performance kernels for GPUs are typically structured differently than those for CPU as GPUs have very different architectural characteristics than CPUs.

Unlike the multiple instruction multiple data (MIMD) execution model used on CPUs, GPUs use a single-instruction-multiple-thread (SIMT) execution model. A GPU bundles a group of 32 SIMT threads known as a *warp* on NVIDIA GPUs or 64 threads known as a *wavefront* on AMD GPUs. A GPU runs all threads in a warp or a wavefront at once executing the same instruction. Computations on GPUs must exploit fine-grain data parallelism to make full use of the thread-level parallelism of the SIMT model.

On NVIDIA GPUs, a group of warps constitute a *thread block*, commonly referred to a *block*. AMD GPUs pack wavefronts into *workgroups*. To simplify the discussion in this article, we call this concept a thread block or block interchangeably regardless of the GPU vendor. Each thread block on a GPU has a limited amount of hardware resources, such as shared memory and registers. The number of blocks that can execute simultaneously on a GPU is limited by the aggregate resource quota for the active threads. To improve performance, we strive to design implementations with high occupancy[*] to use as many of the GPU functional units as possible.

GPUs also have their own memory hierarchy, which differs from that of CPUs. In addition to memory and multiple levels of cache, GPU memory hierarchies also include specialized structures such as constant memory, texture memory, and shared memory; each of these specialized structures supports limited access patterns. To exploit the GPU memory hierarchy, computations must be appropriately structured. Both data placement and data movement in the memory hierarchy need to be carefully tailored.

3 | RELATED WORK

There are many papers that describe strategies for efficient stencil implementations on CPUs[3-15] and GPUs.[16-22] We discuss the most related efforts below.

Time skewing[8,9] has been widely used on CPUs.[10-12] It increases data reuse and cache locality by skewing one or more data dimensions by the time dimension. It computes several time steps for a tile while the values are in cache to avoid costly data movement.

[*] GPU *occupancy* for a computation is the ratio of the number of warps or wavefronts that run concurrently divided by maximum number of concurrent warps or wavefronts supported by the GPU hardware.

Cache-oblivious algorithms[3-7] make optimal use of each memory level without the need of taking cache size as a parameter. They tile the domain by performing a space cut or a time cut. Cache oblivious algorithms have been effective for accelerating stencil computations on CPUs.[7]

Overlapped tiling uses time skewing to trade redundant computation along the boundaries of overlapped tiles for a reduction in memory bandwidth required.[15,16] It improves performance by increasing the arithmetic intensity of parallel stencil computations. Because loading data from a GPU's global memory is more expensive than data-parallel computation, overlapped tiling is an attractive approach for GPUs. Redundant computation can be overlapped with data accesses to help hide memory latency. While overlapped tiling has been shown to enhance 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. Additional calculations needed in cells at the boundary of the domain introduce branch divergence as boundary threads perform extra computations.

While overlapped tiling introduces a large amount of redundant computation to compute multiple time steps, split tiling[17] offers an alternate approach to time skewing. Split tiling advances points in a domain by multiple time steps with a two-phase computation. The first phase computes in parallel on hyper-trapezoidal tiles that taper along the time dimension. Once all tiles from the first phase have been computed, a second phase back-fills the missing points. Each point is only computed once, in either the hyper-trapezoid phase or the back-fill phase.

Instead of performing the whole stencil computation for a point at once, the semi-stencil algorithm[13,14] factors a stencil computation into two halves—a forward update and a backward update. Rather than loading the entire width of a stencil along a streaming dimension at once, the semi-stencil algorithm loads only half of the points along the streaming dimension, performs a backward update to complete the computation of a point to its left, and performs a forward update to partially compute a point to its right. The semi-stencil algorithm trades half of its loads of neighboring cells along the streaming dimension with a store and reload of a partial result. For high-order stencils, this approach reduces the number of loads per point in the calculation and also changes the ratio of loads to stores. The semi-stencil algorithm has been shown to improve the performance of vectorized computation on CPUs, especially for high-order stencils. The reduction in loads that the semi-stencil algorithm affords becomes more profitable as stencil width increases.

Nguyen et al.[18] introduce a 3.5D blocking algorithm that combines 2.5D spatial blocking with 1D temporal blocking. 2.5D spatial blocking involves blocking in a 2D plane and streaming along a third dimension. In a 3.5D variant, they advance the computation of each 2D tile for multiple time steps using a time skewing approach. They perform computations for each time step, and write data back to the global memory eventually. While the 3.5D algorithm works very well on CPUs, the 1D temporal blocking using time skewing shares the same challenges for high-order stencils with boundary conditions on GPUs: barrier synchronizations and limited parallelism. To increase data reuse, this approach stores active 2D planes in GPU shared memory. As a result, the tile size is limited by the shared memory available to a thread block.

Another 2.5D blocking strategy for GPUs[19] maintains data points of the currently active plane in shared memory while employing registers to store data elements along a streaming dimension. Keeping the central plane in shared memory allows faster data access and using registers reduces the shared memory pressure, enabling a larger data tile.

The AN5D framework[20] refines the 2.5D and 3.5D solutions with fixed register allocations, double buffering, and division of the streaming dimension. AN5D delivers great performance for simple single-statement kernels. However, neither boundary conditions nor multi-statement stencils are explored in this work. Our work focuses on a high-order stencil with boundary conditions, and the boundary layer of our domain requires multiple statements, instead of simple single-statement stencil updates.

Other related work tackles stencil computations with interesting approaches, including auto-tuning with dynamic resource allocations,[23] DAG reordering,[24] diamond tiling using a polyhedral model,[25,26] functional programming,[27,28] and multi-layer intermediate representations.[29-31]

In terms of software engineering, there are two common practices for developing stencil code on GPUs for evaluation: hand-written kernels and domain-specific language (DSL)-based approaches.[7,22,32-36] While we are ultimately interested in DSL-based approaches that simplify the generation of code with complex logic, in this article we study hand-written stencils to avoid limitations as we explore implementation strategies for high-order stencils to understand in detail the strengths and weaknesses of various strategies for achieving high performance and performance portability. To simplify our evaluation and enhance code reuse, we developed a framework tailored specifically to support our problem domain and the stencil implementations we study.

4 | APPROACH

We study a high-order stencil-based implementation of the acoustic isotropic wave equation approximation for seismic modeling. Solving this with finite differences essentially involves high-order stencil computations with proper handling of the boundary conditions. To explore the characteristics of various space, we evaluate different code shapes with various computational organizations (e.g., 2D vs. 3D tiles) and different strategies for memory hierarchy management. In the rest of this section, we explain data decomposition alternatives, describe blocking strategies, and discuss how we structure our implementations.

4.1 | Data domain decomposition

Our data domain contains two regions: an inner region and a surrounding PML region that implements boundary conditions for the simulation domain. At the center of the data domain sits an inner region, and the volume between it and the data domain boundaries is the PML region. Both the size of the inner region and the width of the PML region are variables in our simulations. Based on this data domain and its boundary conditions, we explored three decomposition strategies: unified domain, two-domain, and seven-domain.

The unified domain strategy handles the entire data domain as a whole. This strategy uses a single kernel that is applicable to any region of the data domain. Conditionals in the kernel determine whether a point is in the PML region or the inner region. A unified domain kernel performs a different kind of stencil computation for each region: one in the PML region with special boundary calculations and the other in the inner region for the acoustic isotropic wave function approximation. This strategy simplifies the implementation to make certain complex stencil algorithm implementations realistically possible. However, this strategy yields branch divergence for subregions that contain points in both the PML and inner regions, which hurts performance.

The two-domain strategy separates kernels for the inner region and the PML region, and launches separate kernels for the two regions concurrently. Although this strategy lowers the chance of branch divergence by avoiding conditionals that check whether a point is inside the inner region or PML region in every kernel, it leads to unbalanced work among threads along the region boundaries if the size of the GPU blocks does not evenly divide the extents of the PML and inner regions.

Figure 1 illustrates our seven-domain strategy, which is a refinement of two-domain strategy. Instead of just separating the inner and PML regions, the seven-domain strategy divides the PML region into six subregions. We slice the domain along the top and bottom of the inner region to separate top and bottom slabs of the PML region. Next, we slice around the front, back, and sides of the inner region to peel off four more slabs of the PML region. The six PML subregions obtained from these cuts are top, bottom, front, back, left, and right PML subregions. With this decomposition, we concurrently launch individual GPU kernels to perform a stencil computation on each of the seven subregions: one for the inner region and another for each of the six PML subregions. This approach eliminates the intrinsic branch divergence along the boundaries between regions and also avoids having conditional code needed to calculate points in the PML region present when calculating points in the inner region. When running different grid sizes, work imbalance only occurs in a few cases along region borders. This can be reduced with automated code generation that tailors the number of threads for the shape of each slab.

4.2 | Blocking strategies

We partition each region into thread blocks, so that each block maximizes the GPU utilization for a kernel launch without exceeding the hardware resources available to threads in a block. We use three blocking strategies in our experiments: 3D blocking, 2.5D blocking, and 3.5D blocking (Figure 2).

4.2.1 | 3D blocking

This approach divides each of the data regions into axis-aligned 3D blocks. We experiment with different block dimensions to find the best ones. We use fixed values in each execution to simplify our experiments. A GPU kernel computes a 3D data block using a 3D thread block with matching dimensions.

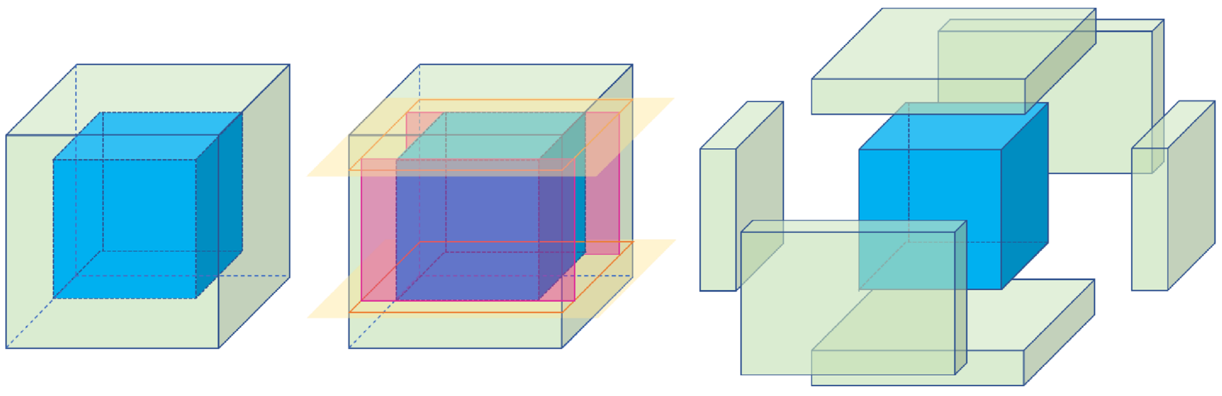

FIGURE 1 Data domain decomposition

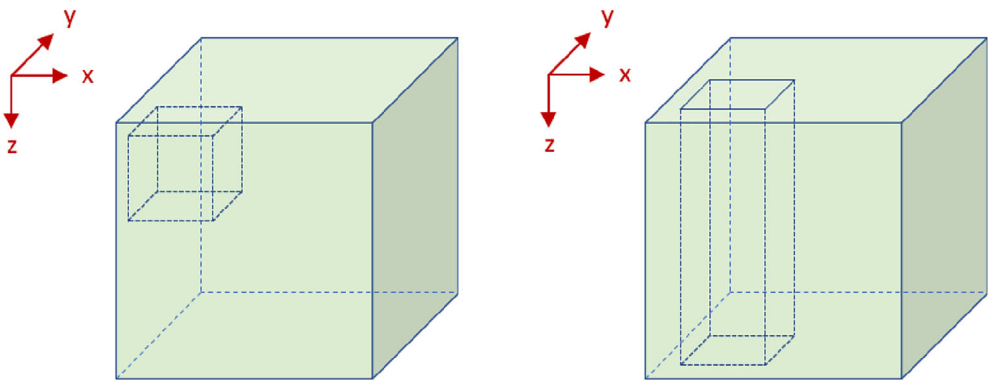

FIGURE 2 Blocking strategies: (Left) 3D blocking, (right) 2.5D and 3.5D blocking

4.2.2 | 2.5D blocking

This approach partitions the data domain along the inner two *X* and *Y* data dimensions and performs a streaming computation along the outermost *Z* dimension. A GPU kernel computes each 3D data block using a 2D thread block for each *XY* tile along with a loop that iterates over the *Z* streaming dimension.

4.2.3 | 3.5D blocking

Like 2.5D blocking, this approach also partitions the data domain along the inner *X* and *Y* dimensions. As with the 2.5D strategy, a GPU kernel computes each 3D data block using a 2D thread block for each *XY* tile and performs a streaming computation along the outermost *Z* dimension. However, a single launch of a 3.5D GPU kernel computes two time steps of the stencil computation for each 3D data block. To provide the data values needed to compute a second time step for the tile interior, each *XY* tile for the 3.5D strategy overlaps a boundary of each of its neighboring tiles by its halo width.

4.3 | Kernels and their variants

To understand the performance properties of high-order stencils and their device characteristics, we implemented several kernels with each employing an appropriate combination of strategies for padding, data decomposition, blocking, data accesses, and data volume traversals. For each implementation, we study several various block sizes to understand the strengths and weaknesses of implementation alternatives. Table 1 lists our implementation strategies, and we describe the details of these kernel implementations in the next section.

5 | KERNEL IMPLEMENTATIONS

A code repository that includes all of our kernel implementations is publicly available on Github: https://github.com/rsrice/CPE21-Artifact. In our implementations, we denote the width of halo by *R*. We use a $R = 4$ for the acoustic isotropic simulations in our experiments. Let *Nx*, *Ny*, and *Nz* 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 locate a point in a 3D block and identify its corresponding thread in a GPU thread block. Similarly, we use (x, y) to denote the coordinate for both an array element in a 2D plane and a GPU thread in a 2D thread block.

5.1 | 3D blocking using global memory only

The first kernel implementation we consider uses only global memory, which makes it conceptually easiest to understand and practically the simplest to implement. Let *Dx*, *Dy*, and *Dz* be the block dimensions of the *X*, *Y*, and *Z* axes, respectively. Thus the block size is $Dx \times Dy \times Dz$. The total number of points in a thread block needs to be $\leq$1024 to respect the GPU's limit of at most 1024 threads per block. Because we launch each kernel with thread blocks of the same size, the GPU grid size of each data region is $\lceil Nx/Dx \rceil \times \lceil Ny/Dy \rceil \times \lceil Nz/Dz \rceil$.

TABLE 1 Implementation strategies

Identifier	Description	Blocking	Domain decomposition
gmem_*	3D blocking using global memory only		
smem_u _*	3D blocking using shared memory for the u array	3D	
smem_eta_*	3D blocking using shared memory for boundary regions		
st_smem_*	2.5D streaming with multiple planes in shared memory		Seven domains
st_reg_shft_*	2.5D streaming using register shifting	2.5D	
st_reg_fixed_*	2.5D streaming using fixed registers with loop unrolling		
st_semi_*	2.5D streaming using semi-stencil		
ol_gmem_*	3.5D streaming using global memory	3.5D	Unified domain
ol_reg_shft_*	3.5D streaming using register shifting		

To compute our 25-point stencil, each thread concurrently fetches the central point for the stencil and four neighboring points along each direction of each axis. When each thread in a block concurrently fetches points from global memory, memory fetches by adjacent threads along the innermost *X* dimension coalesce, which reduces the number of memory transactions and delivers good performance. 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.

5.2 | 3D blocking using shared memory for the *u* array

This approach also uses the aforementioned 3D blocking strategy for each of the data regions. Unlike the previous approach, in which threads compute directly on data fetched from global memory, threads in this kernel read values of the *u* array from global memory, store them into shared memory, synchronize, and then perform the stencil computation on data from shared memory. The total number of points we fetch in this case is the sum of $Dx \times Dy \times Dz$ 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.

Designing the right approach to minimize the cost of data fetches is critical to overall performance for high-order stencils, because the halos account for a significant fraction of the data to fetch into shared memory. From our evaluations, we found the following approach yields good performance: first, thread (i, j, k) fetches the point (i, j, k); then, we use the first R threads along each dimension to fetch the halos from both sides. We refer to the implementation that uses 3D blocking and shared memory as $smem\_u\_\{Dx\}\_\{Dy\}\_\{Dz\}$ in our experiments.

5.3 | 3D blocking using shared memory for boundary regions

This implementation also utilizes shared memory and 3D blocking but differs from the previous approach by fetching the *eta* array into shared memory instead of fetching the *u* array into shared memory. This strategy only applies in the PML kernel because *eta* is only used in the stencil computation inside the PML region.

Although this implementation may not seem new, it is interesting for two reasons. First, unlike low-order stencils widely studied in the literature, we address the combination of high- and low-order stencils. 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. Second, by accessing the *u* array in global memory with a good access pattern for a high-order stencil, meanwhile using shared memory for the *eta* array with a lower-order stencil, it provides us an opportunity to observe the performance changes.

We use two implementations that differ in the number of conditionals when fetching *eta* into shared memory. 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*, which is one for the PML layer of the acoustic isotropic kernel.

smem_eta_3 uses the first $2 \times R\_eta$ threads from each dimension to fetch the halos. We need three conditionals for three dimensions, that is, one for each dimension. Along each dimension, threads 0 to $R\_eta - 1$ fetch the halo on one side, and threads R_eta to $2R\_eta - 1$ fetch the halo on

Algorithm 2. Shared memory halo fetching strategy for *eta* using only one conditional

```
Data: xidz, yidx, zidx: thread index of x, y, and z dimension, respectively
Data: nt: number of threads per block dimension
Result: g: coordinate for global memory
Result: s: coordinate for shared memory
if zidx <6 then
    z ← zidx & 1;
    sz ← z * 9;
    gz ← z * (bt + 1) − 1;
    xzswap ← zidx < = 1;
    yzswap ← (zidx & 2) == 2;
    si ← xzswap ? sz : (xidx+1);
    sj ← yzswap ? sz : (yidx+1);
    si ← xzswap ? (xidx+1) : (yzswap ? (yidx+1) : sz );
    gi ← xzswap ? gz : xidx;
    gj ← yzswap ? gz : yidx;
    gi ← xzswap ? xidx : (yzswap ? yidx: gz );
    s ← (si, sj, sk);
    g ← (gi, gj, gk);
end
```

the other side. Because *R_eta* is just 1 for acoustic isotropic kernel, we only need two threads fetching halos along each thread dimension, one for each side. However, this introduces unbalanced work during data fetches for a 3D block of size `8x8x8`.

*smem_eta_*1 with only one condition is designed to address the work imbalance issue. We use the first six threads from the `X`dimension to fetch halo points. Algorithm 2 shows how six planes of threads are tilted to identify the halo point each thread is responsible for fetching. However, this algorithm has relatively complex arithmetic to compute the halo position to be fetched by a thread, so an evaluation is needed to determine whether the strategy is profitable.

5.4 | 2.5D streaming with multiple planes in shared memory

This is the first implementation we consider that uses 2.5D blocking, which streams a 2D plane along the third dimension. As X is the innermost dimension in our data layout, to exploit cache, we choose the XY-subplane as the 2D plane and stream along the Z dimension. Let Dx and Dy denote the dimensions of the 2D tile along the X and Y axes, respectively. We launch kernels using 2D thread blocks of size $Dx \times Dy$. The GPU grid size of each data region is $\lceil Nx/Dx \rceil \times \lceil Ny/Dy \rceil$.

We use shared memory as a buffer to store all data needed in the stencil computations for a particular XY-subplane. We load the current XY-subplane into the shared memory buffer, and store R subplanes above the current subplane and R subplanes below in the shared memory. Therefore, we allocate a buffer for total of $(2R + 1) \times (Dx + 2R) \times (Dy + 2R)$ points in which each of the $2R + 1$ planes has $(Dx + 2R) \times (Dy + 2R)$ points. We carefully choose the extent of each subplane so that the buffer size is large enough to enhance data reuse, while making sure that the aggregated data volume of the planes is still within the shared memory quota available to a block. Let B denote our buffer, and $B[i]$ denote the ith subplane in the buffer.

Before we can start the streaming computation, points from the top halos are preloaded into buffer $B[0 \dots R)$ and the first R XY-subplanes are preloaded into $B[R \dots 2R)$. Then, for each $z \leftarrow [0 \dots Nz)$ in our streaming loop, we first load the `(z+R)` th XY-subplane into $B[(z + R) \ mod \ (2R + 1)]$; 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.

Our initial implementation of this strategy used a modulus operator, as described above. However, our evaluation showed that the modulus operator is particularly costly on a GPU. Since the `z` index always increases by one inside the streaming loop, we refined our implementation using 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.

In this approach, the GPU per-block shared memory size limits the shared memory buffer size. To avoid this limitation, alternative GPU hardware, such as registers, can be exploited for storing stencil points along the streaming dimension. We discuss a few approaches that use registers to store points along the streaming dimension in the following sections.

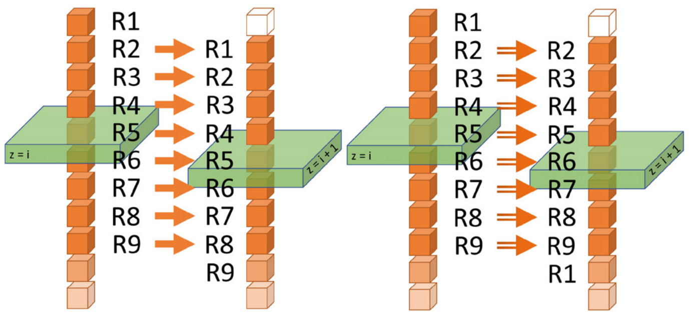

FIGURE 3 Register: (Left) shifting, (right) fixed

5.5 | 2.5D streaming using register shifting

The 2.5D streaming approach described in the previous section keeps all points in shared memory. However, the shared memory limit for a thread block limits the size of the tile. Here and in the next few sections, we keep points in the halo regions along the *z* dimension in registers, which enables us to use larger tiles.

In this 2.5D streaming approach, while we keep points of an active *XY*-subplane in shared memory, we use registers for halo points along the *z*-axis as we stream. While data loaded by any thread in a block into shared memory is accessible to all threads in the block, each register is only accessible to a single thread. We can use registers for halo data along the *z* axis because the halo data is not needed by other threads for their stencil computations.

For this approach, we allocate shared memory to hold $(Dx + 2R) \times (Dy + 2R)$ points for the currently active plane. The shared memory footprint compared to the previous method is 1:$(2R + 1)$. Since R is large for high-order stencils, its reduction in shared memory usage is significant. Let $S(x, y)$ denote the data point (x, y) in the shared memory. We 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 the streaming computation begins, each thread (x, y) fetches data values from (x, y, z) for $z \leftarrow [-R \dots R)$ and stores them into registers $Reg(x, y)(0 \dots 2R]$, respectively. Then, inside the streaming loop, for each $z \leftarrow [0 \dots Nz)$, as shown in Figure 3, we first shift the register indices back one position on each thread, such that for $r \leftarrow (0 \dots 2R]$, $Reg(x, y)[r - 1] = Reg(x, y)[r]$; next, we load the leading point along the streaming dimension $(x, y, z + R)$ into register $Reg(x, y)[2R]$; then, 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*-subplane 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_reg_shft*_{*Dx*}_{*Dy*} in our experiments.

We use the notation of array indexing to illustrate the register value accesses, however, in our implementation, registers are realized explicitly as $2R + 1$ scalar variables. We use the same variable names, `behind4`, `behind3`, `behind2`, `behind1`, `current`, `front1`, `front2`, `front3`, and `front4`, as the original implementation of 2.5D register shifting by Micikevicius,[19] since both work on stencils with halo size of 4.

5.6 | 2.5D streaming using fixed registers with loop unrolling

Like the previous approach, this implementation uses shared memory for an active *XY*-subplane and registers for points along the *z*-axis, the streaming dimension. This time, values in the registers remain fixed instead of being "shifted."

We again allocate a shared memory of $(Dx + 2R) \times (Dy + 2R)$ points. Let $S(x, y)$ denote the data point (x, y) in the shared memory. We allocate $2R + 1$ registers as well, and while they are $2R + 1$ named variables in practice, for ease of presentation, we use $Reg(x, y)[i]$ to denote the `i`th register of thread (x, y).

Before streaming starts, each thread (x, y) fetches data from (x, y, z) for $z \leftarrow [-R \dots R)$ and stores them into register $Reg(x, y)[0 \dots 2R)$, respectively. Then, inside the stream loop, for each $z \leftarrow [0 \dots Nz)$, we do not shift values between registers; instead, we update register $Reg(x, y)[(z + 2R) \text{ } mod \text{ } (2R + 1)]$ with the value of point $(x, y, z + R)$ as shown in Figure 3. 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 $Reg(x, y)[(z + R - i) \text{ } mod \text{ } (2R + 1)]$, and `j`th data below the current point from $Reg(x, y)[(z + R + j) \text{ } mod \text{ } (2R + 1)]$. Finally, the kernel stores the result back to global memory. We use *st_reg_fixed*_{*Dx*}_{*Dy*} to refer to the family of kernel implementations using this strategy in our experiments.

We improve express one iteration of the computation using a macro 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 the *z*-axis.

5.7 | 2.5D streaming using semi-stencil

Like the previous approach, this implementation continues to use shared memory for an active XY-subplane and avoids moving values in registers as much as possible for points along the streaming dimension. Rather than loading both forward and backward halos along the z-axis, only one side of the halos is loaded into registers. While streaming along the z-axis, the semi-stencil algorithm[13,14] first computes a forward partial result at the leading edge of the halo along the streaming dimension and a few iterations later completes the stencil for that point by combining that partial result with a backward contribution at the trailing edge of the halo.

We again allocate a tile in shared memory with $(Dx + 2R) \times (Dy + 2R)$ points. Let $S(x, y)$ denote the shared memory for location (x, y). We allocate only $R + 1$ registers, and denote $Reg(x, y)[i]$ for the `i`th register of the thread (x, y). We additionally allocate $R + 1$ registers for storing the partial results, and denote the `i`th partial result of the thread (x, y) by $Part(x, y)[i]$. Registers for both the x-axis points and the partial results are realized as named variables.

To prepare for streaming, each thread (x, y) fetches data from (x, y, z) for $z \leftarrow [-R \dots 0)$ and stores them into $Reg(x, y)(0 \dots R]$, respectively. As semi-stencil computes partial results, we manually iterate R steps before entering the streaming loop. For the `i`th step, each thread (x, y) fetches the x, y, i data and stores it into $Reg(x, y)[i]$, and computes the partial result $Part(x, y)[i]$ with $Reg(x, y)[((i - R)\ mod\ (R + 1)) \dots ((i - 1)\ mod\ (R + 1))]$ using the forward computation described in semi-stencil algorithm. The forward computation for a point always happens R steps ahead of its backward computation.

Inside the streaming loop, for each $z \leftarrow [0 \dots Nz)$, we update register $Reg(x, y)[(z + R)\ mod\ (R + 1)]$ with the value of point $(x, y, z + R)$, and keep the other register values unmodified. Then, we perform both the forward computation for a later point $z + R$ in the stream loop and the backward computation for the current point z. Forward computation computes the partial result for point $z + R$, R ahead of the current point z, with $Reg(x, y)[(z\ mod\ (R + 1)) \dots ((z + R - 1)\ mod\ (R + 1))]$ and stores in $Part(x, y)[z + R]$. The backward computation loads the partial result $Part(x, y)[z]$ that has been computed in either an earlier stream loop or in the setup, and completes the stencil computation together with values from `i`th data below the current point from $Reg(x, y)[(z + i)\ mod\ (R + 1)]$ where $i \leftarrow (0 \dots R]$ and from shared memory $S(x, y)$ for the currently active 2D plane. The kernel stores the final result back to global memory. We refer to the family of kernel implementations using this strategy as *st_semi_{Dx}_{Dy}* in our experiments.

5.8 | 3.5D streaming using global memory

In this section and the next, we describe two 3.5D blocking approaches. Building on the 2.5D blocking approaches described previously, 3.5D blocking implements a 1D overlapped tiling[16] for the 2D active plane. In our implementations, we perform stencil computations for two time steps with one round of global memory loads per block. With overlapped tiling, we redundantly load and compute points from adjacent blocks for the overlapped region. To simplify the 3.5D implementations, we compute with a unified domain.

We denote the two time-steps in our implementation as T' and T'', respectively. As already described, 2.5D spatial blocking mainly streams a 2D plane through the third dimension. Let Dx and Dy denote the dimensions of the 2D tile along the X and Y axes, respectively. A thread block computes a Dx by Dy data tile with $Dx \times Dy$ threads. These threads compute $Dx \times Dy$ points in T', but to accommodate the data dependencies for one additional time-step, we back away from the tile boundary, known as the overlapped region. Thus in T'', only $(Dx - 2 \times R) \times (Dy - 2 \times R)$ points are computed based on the $Dx \times Dy$ points already computed in T'. The results of time step T'' are then stored back to global memory, so only $(Dx - 2 \times R) \times (Dy - 2 \times R)$ points have the two time-step results with a 2D block of $Dx \times Dy$ threads. The GPU grid size of each data region is $\lceil Nx/(Dx - 2 \times R) \rceil \times \lceil Ny/(Dy - 2 \times R) \rceil$. Let Nt be number of total time steps in computation, instead of iterating it Nt times as in the previous implementations, with 3.5D blocking, we only iterate $Nt/2$ times.

Inside the streaming loop, for each $z \leftarrow [0 \dots Nz)$, we perform a normal 25-point stencil computation for time step T'. In this global memory only implementation, each thread fetches its own point as well as R points along each axis directly from global memory. The results of T' are stored in shared memory $S(x, y, z)$ for faster access in T'' computations. In time step T'', however, not all threads are contributing to the computation, because to implement temporal blocking, we use shared memory to store first time-step's result instead of global memory, and shared memory data cannot be shared across blocks, the data dependency for time step T'' is only available in the shared memory. To make stencil computations correct, when each point reads its neighbor points, because these points are now in the shared memory, so we must treat points along the border of each 2D tile as halo points for the second time step computation. So in time step T'', only the $(Dx - 2 \times R) \times (Dy - 2 \times R)$ threads at the center of the 2D tile compute using the T' results from shared memory. After performing two time steps of the stencil computation, the results are stored back to global memory.

There are three issues that require some additional attention for the acoustic isotropic kernel. First, as we stream a 2D plane through a 3D data domain, to compute T'' time step for z index, T' results of $[(z + 1) \dots (z + R)]$ from the streaming dimension are needed. In our implementations, after computing T' time step for a z index, we store the results into shared memory, and hold the T'' computation for z index until the T' results of $[(z + 1) \dots (z + R)]$ become available in a later streaming iteration. Second, since computation of the acoustic isotropic kernel is 2nd-order in time, results for both T' and T'' are stored back to the global memory as both are needed in the computation of the next iteration. The need to save results for both time steps offsets the data movement savings obtained using time-skewing. Third, the acoustic isotropic kernel requires additional data arrays for its computation. While they can be loaded from global memory directly in previous approaches, they have to be preserved in shared memory for the computation in the second time step. As a result, additional shared memory space is needed for these values. Since they have no impact on the algorithm other the data footprint in shared memory, we do not describe their use in detail. These 3.5D blocking implementations that fetch stencil points directly from global memory for each pair of time steps are referred as *ol_gmem_*{*Dx*}_{*Dy*} in our experiments.

5.9 | 3.5D streaming using register shifting

This 3.5D blocking approach is implemented on top of 3.5D strategy described in the previous section. Instead of fetching data directly from global memory in time step T', similar to its 2.5D blocking implementation, points of an active plane are loaded into shared memory first and points along the z-axis are stored in registers.

This implementation allocates a shared memory tile of size $(Dx + 2R) \times (Dy + 2R)$ to hold the active plane and holds $2R + 1$ halo points in registers—the current point and its neighbors along the z-axis. For thread (x, y), let $S(x, y)$ be the point in shared memory and $Reg(x, y)[i]$ denote the `i`th register.

As before, preparations are needed before the streaming iteration, with each thread (x, y) fetching data values from (x, y, z) for $z \leftarrow [-R \dots R)$ and storing them into registers $Reg(x, y)(0 \dots 2R]$, respectively. For each index $z \leftarrow [0 \dots Nz)$ in the streaming iteration, time step T' shifts the register indices and loads the leading point $(x, y, z + R)$ into register $Reg(x, y)[2R]$. Next, it fetches data (x, y, z) from global memory into $S(x, y)$. It uses data from shared memory for the active plane and data from registers for halos along the streaming dimension. It computes results for time step T' and stores them into shared memory. The computation at time step T'' remains the same as in the previous section: all time step T' results are available in shared memory, and we can compute just by fetching them from shared memory. We still compute only the central piece of the 2D tile in T'' time step. After computing two time steps, the results are stored to global memory.

The difficulties outlined in the previous section that affect the profitability of applying time-skewing to the acoustic isotropic kernel remain in this 3.5D streaming implementation. We refer to the family of kernel implementations using this strategy as *ol_ref_shft_*{*Dx*}_{*Dy*} in our experiments.

6 | OPTIMIZATIONS

To improve the performance, we augmented our implementations with a variety of optimizations. In this section, we describe optimizations that provided significant performance benefits.

6.1 | Pinned memory

Our original implementation allocated data for our simulation domain using `malloc` to support several programming models, including `OpenMP` and `OpenACC`. However, any data transferred between a host and a GPU must pass through page-locked memory on the host, commonly known as pinned memory. One can avoid the need for the host to copy data in or out of pinned memory by allocating data for our simulation domain directly in pinned memory. It is easy to adjust to this platform-specific allocation strategy. To allocate data directly in pinned memory, we use `cudaMallocHost` on NVIDIA GPUs, and we use `hipHostMalloc` on AMD GPUs.

6.2 | Constant memory

Our initial implementations stored coefficients used in the stencil computations in registers. We later relocated them to constant memory. While accessing the coefficients in constant memory itself may not improve performance, using constant memory for stencil coefficients reduces a kernel's register footprint. In some cases, the smaller register footprint enables an increase in the number of threads in a thread block, which improves occupancy.

6.3 | Cache line alignment

To minimize the number of bytes being transferred to and from memory, it is best to align a tile's memory accesses on cache line boundaries to avoid kernels from unnecessarily straddling cache lines. To achieve this, padding can be added to the data array so that the length of each row is a multiple of the cache line size on GPUs.

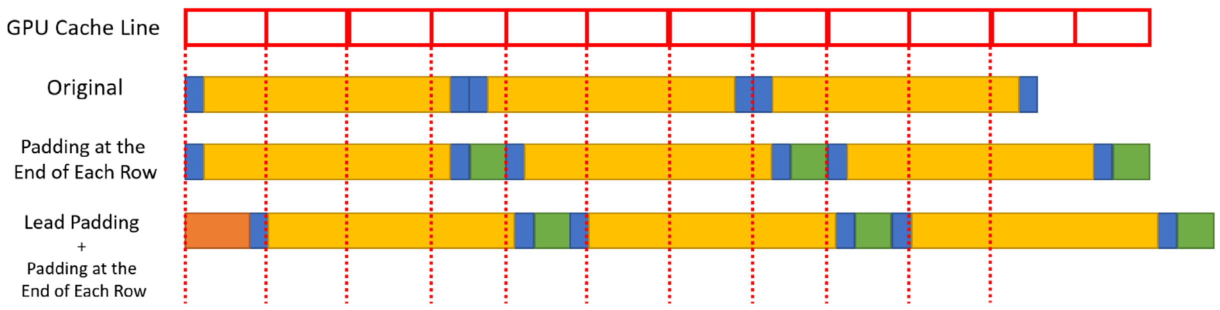
GPU Cache Line
Original
Padding at the End of Each Row
Lead Padding + Padding at the End of Each Row

FIGURE 4 Padding strategies for cache line alignment

As shown in Figure 4, we first add padding at the end of each row as necessary, which ensures that the volume of each row is a multiple of the cache line size.

For our stencil, thread blocks working on the PML and inner regions can have conflicting alignment preferences. We found that adding lead padding to the array and cache-aligning the leading edge of the inner region in each row improves performance because most of the time and memory accesses are spent applying the high-order stencil to the inner region.

6.4 | Function template

To perform stencil computations with a halo size of R, threads with an index less than R away from a boundary are responsible for reading halo points. We added conditionals to check the thread index, and threads fetch halo data when the conditionals evaluate to true. However, on GPUs, these conditionals add stalls that hurt performance.

Our acoustic isotropic kernel has a halo size of 4. For a grid size such as `32x4x4`, the conditionals are always true for the Y and Z dimensions, and threads should always perform a halo fetch. We use a function template to eliminate conditionals for short dimensions in such cases. We provide constants for the halo size and tile dimensions as template parameters. At compile time, an optimizing compiler can dead code eliminate tautological conditions based on knowledge of these constants. This avoids unnecessary runtime checks and the associated stalls, improving performance for these kernel variants. This optimization is suggested by our GPA[37] tool, as are the next two optimizations.

6.5 | Code reordering

We reordered our code to read subscripted values from global memory well ahead of their use in the stencil computation. This adjustment can overlap memory fetches with a long computation. Hiding memory latency improves performance.

6.6 | Double buffering

To further hide memory latency by overlapping memory accesses with computation, our 2.5D implementations employs double buffering. Instead of keeping just a single plane in shared memory, we allocate space for two planes in shared memory. While reading values from one plane and performing the stencil computation at one value of z, we simultaneously load the values of the other plane with values needed for the next value for z. In each iteration, we alternate the role of the planes, reading from one while filling the other.

Double buffering also significantly reduces synchronization stalls. Without double buffering, two barrier synchronizations are needed per iteration to prevent data races: one after filling the plane and another after reads of the plane are complete. With double buffering, only one barrier is required per iteration to signal that both a fill of one plane and read operations on the other are complete before swapping roles of the planes.

NVIDIA introduced hardware support for `memcpy_async` in their latest A100 GPUs. When copying data from global memory to shared memory with `memcpy_async` on an A100, data need not pass through the registers as it must when using older NVIDIA GPUs. This frees registers from the task of moving data so that they can be used in the computation. To further improve performance on NVIDIA A100 GPUs, we implemented data copies for double buffering using `memcpy_async`.

7 | EVALUATION

In this section, we describe how we evaluate our implementations on different GPUs, present their timing results, and discuss performance characteristics.

7.1 | Experimental platforms

We evaluated all of our kernel implementations across multiple generations of AMD and NVIDIA GPUs. For NVIDIA GPUs, we evaluated on NVS510, P100, V100, and the newest A100. We also evaluated our kernels on AMD MI50 and MI100 GPUs. In this article, we only report detailed results for the best performing GPUs, namely the NVIDIA V100, NVIDIA A100, and AMD MI100. Where appropriate, we comment about performance portability of kernels to earlier GPU versions, such as NVIDIA's NVS510 and P100 as well as AMD's MI50. Table 2 lists the specifications for our primary experimental platforms and their respective software stacks. We refer to these systems by their GPU models.

We use `__launch_bounds__` to specify the maximum threads per block, letting compilers limit register usage as needed to guarantee thread block sizes up to the size specified by the launch bounds. However, we pay very close attention to the resulting register usage as it is a critical determinant of performance.

7.2 | Tools

This section describes several tools we use to evaluate the performance of stencil implementations: HPCToolkit, GPA, and the Empirical Roofline Toolkit (ERT).

7.2.1 | HPCToolkit

HPCToolkit is a full-featured suite of tools for performance measurement and analysis.[38] Recently, it has been extended to support analysis of GPU-accelerated applications.[39] In this article, we use the August 2020 release of HPCToolkit to measure kernel performance using PC sampling and associate exposed latencies and stall reasons with program source. In addition, we use HPCToolkit to collect GPU kernel metrics, such as register usage, block size, and grid size.

HPCToolkit's `hpcviewer` is a graphical user interface for analysis of program performance. Its code-centric view enables us to easily spot source lines that have the most significant performance issues, and its trace view enables us to inspect the entire program execution over time and identify idleness and its associated calling contexts. We describe some of our findings using HPCToolkit in our discussion of evaluation results.

TABLE 2 System specifications

	V100	A100	MI100
CPU	IBM POWER9	AMD EPYC 7402	AMD EPYC 7252
CPU cores	160	96	16
RAM	256 GB	512 GB	256 GB
GPU	NVIDIA Tesla V100	NVIDIA A100	AMD MI100
CUDA cores/stream processors	5120	6912	7680
GRAM	32 GB	40 GB	32 GB
OS	RHEL v7.7	RHEL v8.3	CentOS v8
Platform	CUDA 11.0	CUDA 11.2	ROCm 4.1
GPU driver	NV 450.51.05	NV 460.27.04	ROCm 4.1
Compiler flags	`-O3 -arch=sm_70`	`-O3 -arch=sm_80`	`-O3 -amdgpu-target=gfx908`

7.2.2 | GPA

GPA[37] is a performance tool that employs a combination of static and dynamic analysis, to provide optimization suggestions for programs executed on NVIDIA GPUs. We use GPA to obtain optimization insights into the performance of our stencil implementations. We applied top-ranked advice from GPA to tune several our stencils, and measured its performance impact.

7.2.3 | NVIDIA tools

We use two profiling tools from NVIDIA, `nvprof`[40] and Nsight Compute.[41]

We run `nvprof` to collect a set of metrics on a GPU, including kernel execution, data transfers, cache information, and other events for a CUDA kernel. On `V100` and older NVIDIA GPUs, we use `nvprof` to capture performance measurements needed for computing kernel arithmetic intensities and other data points needed for Roofline plots described in the next section.

Nsight Compute (`ncu`)[41] provides insights when performance differences are driven by the kernel characteristics. We run Nsight Compute to examine such characteristics, including theoretical and achieved occupancy. When low occupancy happens, Nsight Compute reports whether or not the problem seems to be associated with the register footprint, the shared memory footprint, or the number of threads. On the A100, we also use Nsight Compute for performance metrics collection for Roofline analysis.

Profiling using Nsight Compute has a huge measurement overhead, as it replays every kernel execution multiple times to collect a complete set of measurements. To profile our high-order stencils on a large data volume with 1000 iterations, it takes an unreasonable amount of time to finish. So, when we use Nsight Compute, we use it to measure only five iterations. We run 1000 iterations when profiling with `nvprof` as its overhead is acceptable.

7.2.4 | AMD rocprofiler

We use `rocprofiler`[42] from AMD to collect hardware performance counters for kernels running on AMD GPUs.

7.2.5 | Roofline performance model

The Roofline[43] performance model visually shows code performance with relative to a machine's practical peak performance. It combines a code's arithmetic intensity, memory bandwidth, and performance into a single chart. It also can provide some optimization insights by comparing a code's performance against a platform's performance ceiling.

Recently, the Roofline model has been extended for GPUs. To capture the performance characterizations for our experimental platforms, we use the ERT,[44] which empirically measures the performance of a GPU using benchmarks. It provides us with a GPU's achievable performance bound for computations with various arithmetic intensities. For memory-bound kernels such as high-order stencils, the achievable peak, as limited by memory bandwidth, is substantially lower than the theoretical peak claimed by the manufacturers.

We characterize kernels using NVIDIA's `nvprof` on `V100` and `ncu` on `A100` by measuring several kernel performance metrics, including FLOPs, L2 read and write transactions, as well as DRAM read and write transactions. We wrote a Python script that takes the output from either `nvprof` or `ncu` and calculates both the performance and the arithmetic intensity for each kernel. Performance is calculated by dividing the measured FLOPS by the recorded execution time. Arithmetic intensities are calculated by dividing the measured FLOPS by the measured bytes accessed on DRAM and L2 cache, respectively.

We identify the performance gap between the kernel performance and the peak performance possible for a kernel on a particular GPU given the kernel's arithmetic intensity. We then consider how to increase the arithmetic intensity of a kernel to increase the maximum achievable performance. We also compare kernels by their arithmetic intensities and their relative performance.

7.3 | Evaluation methodology

We evaluate all implementations and their variants. Our kernels are written in CUDA, and they can be compiled and directly on NVIDIA platforms. For AMD GPUs, we use AMD's `hipify` tool to transform CUDA source code to HIP code. Depending on the platform, we compile kernels with NVIDIA's `nvcc` or AMD's `hipcc`.

For each machine, we run the kernels with a large grid size based on its device memory size. We use the same grid size of 1000^3 for `V100`, `A100`, and `MI100`. For benchmarking, we measure 1000 iterations of all kernels on all machines.

First, we collect time measurements for each kernel. We warm up the kernel by running the entire execution once, then we repeat it 10 times and record the average time and the standard deviation of the 10 runs.

Second, we use HPCToolkit's GPU support to profile the kernel details with PC sampling. Third, we run GPA and look for optimization opportunities. Finally, we measure device-specific kernel characteristics and metrics using Nsight Compute. We calculate the arithmetic intensity and performance metrics of every kernel. We used the ERT to measure our experimental platforms and compare our kernel performance with respect to the empirical machine performance roofline. We describe each of our evaluation methods below.

7.4 | 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 3 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 and 3.5D blockings, only columns for *Dx* and *Dy* are reported since the *z*-axis is unpartitioned. For each system, we show each kernel's average execution time and standard deviation. Table 3 has three sections. The first section contains the time measurement for kernel implementations of our own. The second section presents the measurements of a code version, that NVIDIA optimized from our global memory implementation. While this code version is tuned for NVIDIA GPUs, we also `hipify` this implementation, ran it on the `MI100` GPU, and recorded its time measurements. The third section has the measurements for a code version provided by AMD, this code version is fine tuned for AMD MI100 system, and we also ran it on NVIDIA systems. Table 4 presents the kernel characteristics of each 25-point stencil applied to the inner data region.

Table 5 presents the performance characteristics of our implementations on the `A100` GPU. We combine the metrics from both inner region and PML region in the same table, so that we can discuss the entire execution. Figure 5 compares the performance of select top-performing kernels with the performance bound imposed by GPU DRAM bandwidth using the roofline performance model. The *y*-axes of these figures represent the performance in GFLOPs/second and *x*-axes show arithmetic intensity in FLOPs/byte.

In the rest of the section, we discuss our findings.

7.4.1 | 3D blocking using global memory

As shown in Table 3, simple kernel implementations using only GPU global memory yield great performance on the latest GPUs, NVIDIA V100 and A100. Since Tesla V100, L1 data cache and shared memory are combined into a single unified memory block, providing a large data cache size. When accessing data from global memory with a good access pattern that exploits global memory coalescing, we can achieve very good performance.

While the 3D kernel had great performance on NVIDIA's latest GPUs, it is one of the slowest implementations on older NVIDIA GPUs, such as P100 and NVS510. So its performance portability is poor.

For the 3D global memory kernel, we evaluated several thread block size variants of the global memory implementation are evaluated, from small to large, including `gmem_4x4x4`, `gmem_8x8x4`, `gmem_8x8x8`, `gmem_16x16x4`, `gmem_32x4x4`, `gmem_32x8x4`, `gmem_32x32x1`. We observe the following phenomena: First, when loading the points before performing stencil computations, for kernels with smaller block sizes, the halo size for our 25-point stencil dominates the actual data points. Therefore, more time is spent on loading halos than the points for the volume to be computed, which hurts performance. In addition, smaller block size also leads to larger GPU grid size, which results in more kernel launches. These additional overheads from kernel launches slow the overall execution. Second, with a good cache line alignment, the 3D blocking kernels with a larger innermost dimension tend to perform better.

In summary, despite that 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, one can achieve amazingly good performance with little effort on the late-model GPU architectures. The simplicity and the low optimization effort make the global memory implementations a very attractive option from a software engineering perspective, as they are easy to understand and have a low maintenance cost.

7.4.2 | Shared memory

Table 3 shows that using shared memory can boost performance. The performance gain is more significant on older generation GPUs, such as P100 and NVS510, which is consistent with results in previous research. We attribute this to the architectural changes in V100, which combines the L1

TABLE 3 Time measurement on V100, A100, and MI100

Kernel				System					
				V100		A100		MI100	
Kernel identifier	Dx	Dy	Dz	Avg.	SD	Avg	SD	Avg.	SD
gmem_8x8x8	8	8	8	53.65	0.02	31.82	0.51	612.08	0.03
gmem_32x8x4	32	8	4	49.81	0.02	27.21	0.03	56.90	0.05
gmem_32x4x4	32	4	4	47.62	0.03	25.66	0.04	48.50	0.03
smem_u_8x8x8	8	8	8	54.96	0.01	33.00	0.16	310.19	0.02
smem_u_32x8x4	32	8	4	49.87	0.01	29.54	0.06	45.71	0.09
smem_u_32x4x4	32	4	4	46.62	0.02	25.01	0.04	44.95	0.02
smem_eta_1_8x8x8	8	8	8	54.36	0.01	34.66	0.22	597.76	0.02
smem_eta_3_8x8x8	8	8	8	55.27	0.02	34.93	0.18	592.76	0.06
st_smem_16x8	16	8	–	55.97	0.03	35.37	0.09	100.73	1.06
st_smem_16x16	16	16	–	51.77	0.01	35.78	0.10	65.95	0.69
st_reg_shft_16x16	16	16	–	59.84	0.04	33.50	0.07	69.35	1.53
st_reg_shft_32x16	32	16	–	48.89	0.01	30.86	0.05	49.16	0.08
st_reg_shft_32x32	32	32	–	50.50	0.01	33.31	0.05	47.02	0.20
st_reg_shft_16x16 (db)	16	16	–	67.92	0.03	31.63	0.04	75.25	0.62
st_reg_shft_32x16 (db)	32	16	–	53.99	0.01	23.23	0.05	47.80	0.10
st_reg_shft_32x32 (db)	32	32	–	55.06	0.01	23.76	0.04	45.87	0.16
st_reg_fixed_16x16	16	16	–	59.51	0.01	31.87	0.05	69.47	0.86
st_reg_fixed_32x16	32	16	–	46.28	0.01	25.99	0.04	51.04	0.15
st_reg_fixed_32x32	32	32	–	47.19	0.00	27.34	0.03	48.94	0.16
st_reg_fixed_16x16 (db)	16	16	–	60.04	0.02	29.63	0.07	86.02	0.85
st_reg_fixed_32x16 (db)	32	16	–	44.15	0.02	22.29	0.03	56.37	0.54
st_reg_fixed_32x32 (db)	32	32	–	43.93	0.00	22.84	0.04	54.40	0.42
st_semi_16x16	16	16	–	59.68	0.03	31.66	0.04	68.73	1.72
st_semi_32x16	32	16	–	46.43	0.02	25.92	0.05	48.69	0.16
st_semi_32x32	32	32	–	47.06	0.00	27.54	0.03	45.38	0.29
st_semi_16x16 (db)	16	16	–	67.64	0.03	29.26	0.03	68.50	0.69
st_semi_32x16 (db)	32	16	–	49.75	0.01	22.14	0.04	47.36	0.07
st_semi_32x32 (db)	32	32	–	49.46	0.01	23.37	0.04	45.76	0.06
ol_gmem_16x16	16	16	–	337.21	0.68	101.67	0.43	346.20	0.09
ol_gmem_24x24	24	24	–	128.31	0.12	64.05	0.28	144.82	0.05
ol_reg_shft_16x16	16	16	–	168.26	0.04	86.94	0.34	344.26	0.16
ol_reg_shft_24x24	24	24	–	158.83	0.04	80.38	0.23	145.32	0.03
nv_gmem_f4	32	4	4	43.55	0.03	21.84	0.02	83.09	0.03
amd_reg_shft	32	16	–	46.12	0.25	24.12	0.04	38.11	0.96

Note: Kernels with `(db)` use double buffering.

TABLE 4 Characteristics of a 25-point stencil kernel for the interior region characteristics on the A100

Kernel identifier	Block size	Grid size	Registers per thread	Theoretical active warps	Theoretical occupancy	Achieved occupancy	Achieved active warps
gmem_8x8x8	512	1, 685, 159	32	64	100	86.08	55.09
gmem_32x8x4	1, 024	846, 090	32	64	100	79.67	50.99
gmem_32x4x4	512	1, 685, 070	32	64	100	87.01	55.69
smem_u_8x8x8	512	1, 685, 159	28	64	100	94.27	60.33
smem_u_32x8x4	1, 024	846, 090	28	64	100	92.98	59.51
smem_u_32x4x4	512	1, 685, 070	26	64	100	93.33	59.73
smem_eta_1_8x8x8	512	1, 685, 159	32	64	100	86.08	55.09
smem_eta_3_8x8x8	512	1, 685, 159	32	64	100	86.09	55.10
st_smem_16x8	128	7, 140	48	40	62.50	59.68	38.19
st_smem_16x16	256	3, 600	48	40	62.50	60.08	38.45
st_reg_shft_16x16	256	3, 600	48	40	62.50	59.98	38.39
st_reg_shft_32x16	512	1, 800	48	32	50	48.64	31.13
st_reg_shft_32x32	1, 024	900	48	32	50	50.00	32.00
st_reg_shft_16x16 (db)	256	3, 600	32	64	100	93.86	60.07
st_reg_shft_32x16 (db)	512	1, 800	32	64	100	93.29	59.71
st_reg_shft_32x32 (db)	1, 024	900	32	64	100	95.15	60.89
st_reg_fixed_16x16	256	3, 600	54	32	50	48.35	30.94
st_reg_fixed_32x16	512	1, 800	54	32	50	48.66	31.14
st_reg_fixed_32x32	1, 024	900	54	32	50	50.00	32.00
st_reg_fixed_16x16 (db)	256	3, 600	64	32	50	48.78	31.22
st_reg_fixed_32x16 (db)	512	1, 800	64	32	50	48.83	31.25
st_reg_fixed_32x32 (db)	1, 024	900	64	32	50	49.99	31.99
st_semi_16x16	256	3, 600	54	32	50	48.41	30.98
st_semi_32x16	512	1, 800	55	32	50	48.62	31.11
st_semi_32x32	1, 024	900	55	32	50	50.00	32.00
st_semi_16x16 (db)	256	3, 600	62	32	50	48.67	31.15
st_semi_32x16 (db)	512	1, 800	62	32	50	48.80	31.23
st_semi_32x32 (db)	1, 024	900	62	32	50	50.00	32.00

Note: Kernels with `(db)` use double buffering.

data cache with shared memory. As discussed previously, on the V100, with good access patterns for global memory, one can achieve great performance with little effort. The overhead of using shared memory on the V100 in 3D blocking introduced overheads that slow all but the `32x4x4` kernel compared to the corresponding global memory variants. In contrast, older generation GPUs do not have this new feature, so shared memory provides performance benefits that outweigh its overhead.

It is not hard to fill the shared memory provided by GPU hardware when using high-order stencils, which have a large halo size. Limited shared memory capacity precludes using large block sizes for high-order stencils while staging all data in shared memory.

7.4.3 | Mediocre performance of several kernels on the MI100

Despite that the majority of the MI100 results are on-par with those on the V100, our results in Table 3 show unexpected (not seen in MI50) performance on the MI100 for all the 3D blocking kernels with shape of `8x8x8`. We collected performance counters for `gmem_8x8x8` and `gmem_32x4x4`

TABLE 5 Kernel performance characteristics on the A100

Kernel identifier	FLOP ($\times 10^{13}$)	Achieved performance (GFLOPs)	L2 transactions ($\times 10^{12}$)	L2 arithmetic intensity	L2 achieved percentage	DRAM transactions ($\times 10^{11}$)	DRAM arithmetic intensity	DRAM achieved percentage
gmem_8x8x8	4.453	1302	1.88	0.74	68.52%	8.31	1.67	63.55%
gmem_32x8x4	4.453	1523	1.69	0.82	72.07%	9.45	1.47	84.49%
gmem_32x4x4	4.453	1616	1.83	0.76	82.75%	9.45	1.47	89.61%
smem_u_8x8x8	4.453	1256	1.92	0.72	67.47%	8.36	1.67	61.63%
smem_u_32x8x4	4.453	1403	1.72	0.81	67.40%	9.51	1.46	78.33%
smem_u_32x4x4	4.453	1657	1.86	0.75	85.96%	9.48	1.47	92.25%
smem_eta_1_8x8x8	4.453	1196	1.90	0.73	63.50%	8.35	1.67	58.66%
smem_eta_3_8x8x8	4.453	1187	1.89	0.73	62.79%	8.35	1.67	58.21%
st_smem_16x8	4.453	1172	1.47	0.94	48.27%	8.17	1.70	56.23%
st_smem_16x16	4.453	1159	1.37	1.02	44.32%	7.80	1.78	53.04%
st_reg_shft_16x16	4.453	1237	1.49	0.93	51.47%	7.90	1.76	57.39%
st_reg_shft_32x16	4.453	1343	1.33	1.05	49.96%	7.57	1.84	59.75%
st_reg_shft_32x32	4.453	1244	1.29	1.08	44.71%	7.44	1.87	54.36%
st_reg_shft_16x16 (db)	4.453	1310	1.52	0.92	55.50%	8.43	1.65	64.88%
st_reg_shft_32x16 (db)	4.453	1784	1.39	1.00	69.25%	7.68	1.81	80.42%
st_reg_shft_32x32 (db)	4.453	1745	1.34	1.04	65.07%	7.58	1.84	77.66%
st_reg_fixed_16x16	4.453	1301	1.43	0.97	51.91%	7.84	1.78	59.87%
st_reg_fixed_32x16	4.453	1595	1.30	1.07	58.05%	7.67	1.82	71.79%
st_reg_fixed_32x32	4.453	1516	1.26	1.11	53.19%	7.43	1.87	66.19%
st_reg_fixed_16x16 (db)	4.453	1399	1.53	0.91	59.60%	8.02	1.74	65.87%
st_reg_fixed_32x16 (db)	4.453	1859	1.39	1.00	72.36%	7.68	1.81	83.85%
st_reg_fixed_32x32 (db)	4.453	1815	1.31	1.06	66.41%	7.44	1.87	79.27%
st_semi_16x16	4.889	1438	1.46	1.05	53.33%	7.85	1.95	60.33%
st_semi_32x16	4.889	1756	1.32	1.16	58.93%	7.59	2.01	71.32%
st_semi_32x32	4.890	1653	1.27	1.20	53.36%	7.43	2.06	65.70%
st_semi_16x16 (db)	4.890	1556	1.49	1.02	59.04%	7.98	1.91	66.38%
st_semi_32x16 (db)	4.890	2056	1.36	1.12	71.26%	7.59	2.01	83.43%
st_semi_32x32 (db)	4.891	1948	1.29	1.19	63.67%	7.38	2.07	76.88%
ol_gmem_16x16	11.030	1010	5.95	0.58	67.81%	16.81	2.05	40.26%
ol_gmem_24x24	7.198	1046	3.02	0.75	54.56%	11.00	2.04	41.81%
ol_reg_shft_16x16	11.030	1181	3.61	0.96	48.03%	15.36	2.24	43.00%
ol_reg_shft_24x24	7.198	833	1.72	1.31	24.77%	7.26	3.10	22.00%

Note: Kernels with `(db)` use double buffering.

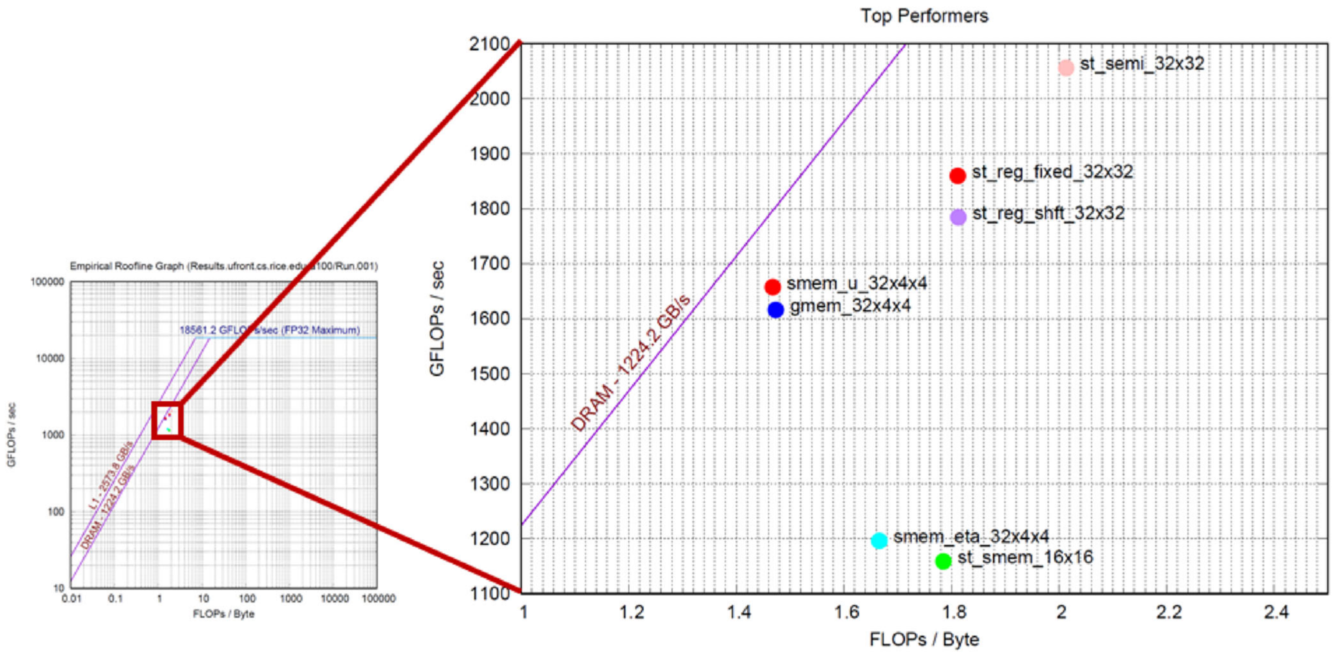

FIGURE 5 DRAM roofline chart for top performers on the A100

and found that `gmem_8x8x8` incurred 4 times L2 cache hits compared to `gmem_32x4x4` while these two kernels have similar numbers of L2 cache misses and similar numbers of dynamic scalar and vector instruction counts. AMD GPUs currently do not provide performance counters for L1 cache, but our results strongly indicates that the abysmal performance for `gmem_8x8x8` is caused by excessive L1 cache conflict misses.

7.4.4 | Code shape for 2.5D blockings

For implementations using 2.5D-blocking, we observe that a larger 2D plane tends to have better performance. There are two main reasons for this. First, a larger 2D plane brings a higher degree of concurrency. Second, the halo-to-point ratio is smaller for a larger 2D plane, which boosts overall performance.

A larger 2D plane means a bigger thread block, which demands more hardware resources, which puts pressure on GPU resources such as registers and shared memory. Fully utilizing a GPU's hardware resources to achieve top performance requires a delicate balance between completing concerns, for example, register pressure and thread block size. Our results show that, 2.5D kernels with a block size of `32x16` perform better than `32x32`, because with the same hardware quota per thread block, `32x16` tiles have more resources per thread, which leads to better performance.

7.4.5 | Register footprint in 2.5D blockings

With the maximum 2D plane size allowed by GPUs, high-order stencils with blocks of 1024 threads struggle with register pressure. We evaluated the variants of the `st_reg_shft_*` implementations with 2D plane size of 1024, namely `st_reg_shft_16x64`, `st_reg_shft_32x32`, and `st_reg_shft_64x16`, and they show lower performance results. The performance degradation is caused by register spilling. The maximum number of registers in a threadblock is $64 * 1024 = 65{,}536$. 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 nor provide hints to compilers, `nvcc` assigns 80 and 96 registers to the PML and inner kernels, respectively. Running the generated binaries for these register footprints yields the CUDA error of `too many resources requested for launch`. We use `__launch_bounds__` to instruct the compilers to limit the maximum number of registers per thread to support parallelism up to the specified level. (For NVIDIA GPUs, one can also use compiler flag `-maxrregcount=64` to achieve the same goal.) However, 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.

Although register spilling occurs in both the register shifting and register fixed kernels, for the register fixed kernels, we do not see a performance degradation because the code uses fixed registers with loop unrolling. We illustrate the differences in Figure 3. Because most of the values in registers are kept stationary, the frequency of register data movement is smaller than for kernels that use the register shifting approach. This enables the performance impact of register spilling to be amortized by other thread activities.

7.4.6 | Semi-stencil

We believe that we are the first to implement the semi-stencil algorithm on GPUs, and it shows excellent performance on all GPUs. Table 3 shows that `st_semi_32x16` kernel with double buffering using A100's `memcpy_async` is the fastest kernel on the A100 and among all of our experiments.

By increasing arithmetic intensity and changing the ratio between loads and stores, the semi-stencil algorithm can improve the performance of high-order stencils not only on CPUs but also on GPUs. For a 3D stencil with halo size of R, with typical approach, it requires to load $6 * R + 1$ 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 for the u-array is $(6 * R + 1) : 1$. Our semi-stencil strategy, on the other hand, only reads the center point and one side of the halo with $R + 1$ loads on the streaming dimension. Instead of one single store at the end of stencil computation, semi-stencil requires one additional store to save the partial results from the forward phase, resulting in a total of two stores. The load-store-ratio for our semi-stencil implementation on the Z-dimension is $(R + 1) : 2$, and remains the same for the other two dimensions. Because our semi-stencil approach trades R loads for one load and one store, even on just one dimension, this is very appealing for high-order stencils because the larger the halo size R, the higher potential benefits one could achieve.

7.4.7 | Impact of memory-related optimizations

We discussed a few memory-related optimizations such as using pinned memory, constant memory, and cache line alignment, in Section 6. Each of them improves performance. We use the `gmem_32x4x4` kernel to illustrate their performance impact. Allocating data directly in the pinned memory avoids extra copy in and out of GPU from host memory, and it yields a 1.02× speedup. While storing coefficients used in the stencil computations into constant memory provides us speed up that is barely measurable for this kernel, it lowers register pressure, which is important for other kernels. Cache line alignment significantly improves performance. To align memory access with cache line boundaries, we add padding to the end of each row, as well as the leading edge of the inner region. The padding between rows yields an 1.03× speedup, and adding the leadpad yields a 1.09× speedup, and two paddings together yield a total of 1.12× speedup.

7.4.8 | Function template

We used function templates to eliminate stalls caused by the branch conditions. For kernels with grid size of $32 \times 4 \times 4$, because the number of threads on the Y and Z axis are all equal to the halo size, threads always perform a halo fetch. Using function templates to provide the grid block size as a constant in the kernel enables the compiler to remove the conditional checking at compilation time. Our experiments show that, for `smem_u_32x4x4`, this optimization provides a 1.07× speedup.

7.4.9 | Double buffering

On the A100, with the use of `memcpy_async`, double buffering always improves performance, which illustrates the utility of this new feature. However, employing double buffering on other GPUs may degrade performance due to increased register pressure because copies to shared memory go through the registers. Table 3 shows a performance degradation for 2.5D `reg_shft_*` kernels on the V100, as well as 2.5D `ref_fixed_*` kernels on the MI100.

7.4.10 | 3.5D blocking

Overlapped tiling introduces redundant computation, which is costly for high-order stencils, as the overlap region is large for high-order stencils. The large overlap region not only increases the level of redundant computation, but also the data volume needed for the computation. Hardware resource requirements also restrict our 2D tile size, which limits the number of time steps we can do in a single pass.

For first-order algorithms, overlapped tiling reduces data movement because it can advance multiple time steps and write out only the result of the final time step. However, the acoustic isotropic kernel, which is second-order in time, requires data values for both the current time step and the previous time step. A second-order kernel computing two time steps in a single pass using a 3.5D approach must write both the final time step and the previous time step as input for the next iteration. This doubles the number of writes compared to the 2.5D algorithm, which only needs to write the value computed in the current time step and retain the value from the previous time step. The cost of the extra array write for the second-order 3.5D algorithm, the redundant computation for the 3.5D algorithm, and the additional shared memory footprint to accommodate a double-width halo outweigh the benefits of computing two time steps in a single pass over the data domain.

Furthermore, using a unified domain, which is necessary for time skewing, adds overhead due to branch divergence from conditionals.

In our experience, manually writing 3.5D implementations is a challenging task, which makes it difficult to combine with other algorithmic approaches.

7.4.11 | GPU occupancy

We observe from Table 4 that, while 3D blocking implementations have higher theoretical occupancies, 2.5D blocking realizes a higher achieved occupancy.

7.4.12 | Performance portability

Table 3 clearly shows that implementations using 2.5D blocking have a better performance portability. The good performance portability can be seen not only on generations of GPUs from the same vendor, but also GPUs from different vendors.

7.4.13 | Gaps to the roofline ceilings

We offer two interpretations for the performance gaps observed in our implementations:

While our current implementations already achieve good performance for high-order stencils with boundary conditions, we see tuning opportunities for additional performance. We can design new GPU code shapes to improve arithmetic intensities, and we can apply other algorithmic approaches to increase performance. We manually wrote our implementations, in the future, we can develop a new DSL approach or build a framework that enables us to explore more sophisticated approaches.

In contrast to the ERT, which uses simple micro-benchmarks for profiling the machines, acoustic isotropic kernels are complex. Their high-order with boundary conditions produces high memory pressure on GPUs. Their complex multiple statements also have challenges to efficiently utilize all GPU resources, making it difficult to hit the roofline ceiling.

7.4.14 | NVIDIA code version

Given a copy of our codes, NVIDIA produced `nv_gmem_f4`, a variant of our `gmem_32x4x4`implementation. Instead of using the typical `float` data type, it employs the CUDA built-in vector type `float4` for the innermost dimension. `nvcc` generates the `LD.128` instruction for `float4` to allow for coalesced access to a vector of four `float` elements. On the `A100`, for `gmem_32x4x4`, this yields a 1.17× speedup. We also applied the `float4` to our other global memory implementations, and we can observe similar performance improvements on NVIDIA systems.

The `hipified` version of this implementation has a degraded performance on the `MI100`GPU.

7.4.15 | AMD code version

Given a copy of our codes, AMD produced `amd_reg_shft`, a kernel similar to our register “shifting” implementation. It improves performance by further dividing the inner region into smaller blocks on the *z*-dimension. Cutting the inner region along the *z*-dimension reduces load imbalance and keeps the GPU busy. This strategy improves performance on the `MI100`GPU.

This code version is also competitive on the `V100`, but slightly shy of our register “shifting” performance on the `A100` GPU.

8 | CONCLUSIONS AND FUTURE WORK

This article evaluates the performance of high-order stencils with boundary conditions on several generations of AMD and NVIDIA GPUs. Our experiments show that the key to performance for a high-order stencil is to maximize utilization of GPU threads by using the tile size as large as possible without exceeding GPU resource bounds. The GPU algorithms also need to be carefully designed to avoid over-consumption of limited resources. Also, data layout and padding have a surprisingly large impact on performance. Careful alignment of tiles with cache lines significantly improves performance. We noticed that kernel implementations that compute stencils directly from global-memory, despite being the simplest, deliver reasonably good performance on NVIDIA's V100 and A100 GPUs. We also observed that 2.5D streaming algorithms deliver excellent performance and have the best performance portability across generations of GPUs from different vendors. In addition, semi-stencil algorithm also show great applications on GPUs for high-order stencils.

Our evaluations on 3.5D algorithms are based on manually written kernels. To ease the implementation, we chose the unified domain data decomposition while knowing its inefficiency caused by branch divergence from our early experiments. Our results once again show this branch divergence on GPUs is costly. We plan to develop a DSL or build a framework that automates code generation, which will facilitate composing tuning strategies and applying them to other high-order stencils. Automated code generation will also help us to migrate our kernels on emerging accelerators to evaluate performance portability on a broad range of platforms. In the immediate future, we plan to evaluate our approaches on stencils for other commonly used seismic imaging approximations to the wave equation.

ACKNOWLEDGMENTS

This work was supported in part by a contract from Total E&P Research & Technology USA, LLC. We thank Guillaume Thomas-Collignon and Ken Hester from NVIDIA and Yaoming Mu from AMD for providing the `nv_gmem_f4` and `amd_reg_shft` kernel implementations and optimization insights.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available at https://github.com/rsrice/CPE21-Artifact.

ORCID

Ryuichi Sai https://orcid.org/0000-0001-8372-401X
John Mellor-Crummey https://orcid.org/0000-0002-9026-5453
Xiaozhu Meng https://orcid.org/0000-0003-3716-9072
Keren Zhou https://orcid.org/0000-0002-7977-3182

REFERENCES

1. Meng J, Atle A, Calandra H, Araya-Polo M. Minimod: a finite difference solver for seismic modeling; 2020. arXiv:2007.06048v1.
2. Komatitsch D, Tromp J. A perfectly matched layer absorbing boundary condition for the second-order seismic wave equation. *Geophys J Int*. 2003;154(1):146-153. https://doi.org/10.1046/j.1365-246X.2003.01950.x
3. Frigo M, Leiserson CE, Prokop H, Ramachandran S. Cache-oblivious algorithms. Paper presented at: Proceedings of the 40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039). New York City, CY; October 17, 1999:285-297; IEEE.
4. Frigo M, Strumpen V. Cache oblivious stencil computations. Paper presented at: Proceedings of the 19th Annual International Conference on Supercomputing ICS '05; 2005:361-366; Association for Computing Machinery, Cambridge, MA.
5. Frigo M, Strumpen V. The cache complexity of multithreaded cache oblivious algorithms. *SPAA '06*. Cambridge, MA: Association for Computing Machinery; 2006:271-280.
6. Strzodka R, Shaheen M, Pajak D, Seidel HP. Cache oblivious parallelograms in iterative stencil computations. Paper presented at: Proceedings of the 24th ACM International Conference on Supercomputing ICS '10; 2010:49-59; Association for Computing Machinery, Tsukuba, Ibaraki, Japan.
7. Tang Y, Chowdhury RA, Kuszmaul BC, Luk CK, Leiserson CE. The pochoir stencil compiler. Paper presented at: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures SPAA '11; 2011:117-128; Association for Computing Machinery, San Jose, CA.
8. Wonnacott D. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. Paper presented at: Proceedings of the 14th International Parallel and Distributed Processing Symposium IPDPS 2000. Cancun, Mexico; 2000:171-180.
9. Wonnacott D. Achieving scalable locality with time skewing. *Int J Parallel Program*. 2002;30(3):181-221. https://doi.org/10.1023/A:1015460304860
10. Jin G, Mellor-Crummey J, Fowler R. Increasing temporal locality with skewing and recursive blocking. Proceedings of the 2001 ACM/IEEE Conference on Supercomputing SC '01; 2001:43; Association for Computing Machinery, Denver, Colorado.
11. McCalpin J, Wonnacott D. Time skewing: a value-based approach to optimizing for memory locality; 1998. https://doi.org/10.7282/T3ZG6WVV
12. Song Y, Li Z. New tiling techniques to improve cache temporal locality. *PLDI '99*. Atlanta, Georgia: Association for Computing Machinery; 1999:215-228.
13. de la Cruz R, Araya-Polo M, Cela JM. Introducing the semi-stencil algorithm. In: Wyrzykowski R, Dongarra J, Karczewski K, Wasniewski J, eds. *Parallel Processing and Applied Mathematics Lecture Notes in Computer Science*. Berlin/Heidelberg, Germany: Springer; 2010:496-506.
14. de la Cruz R, Araya-Polo M. Algorithm 942: semi-stencil. *ACM Trans Math Softw*. 2014;40(3):23:1-23:39. https://doi.org/10.1145/2591006
15. Krishnamoorthy S, Baskaran M, Bondhugula U, Ramanujam J, Rountev A, Sadayappan P. Effective automatic parallelization of stencil computations. *ACM SIGPLAN Not*. 2007;42(6):235-244. https://doi.org/10.1145/1273442.1250761
16. Holewinski J, Pouchet LN, Sadayappan P. High-performance code generation for stencil computations on GPU architectures. *ICS'12*. San Servolo Island, Venice, Italy: Association for Computing Machinery; 2012:311-320.
17. Grosser T, Cohen A, Kelly PHJ, Ramanujam J, Sadayappan P, Verdoolaege S. Split tiling for GPUs: automatic parallelization using trapezoidal tiles. Paper presented at: Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units; GPGPU-6; 2013:24-31; Association for Computing Machinery, Houston, TX.

18. Nguyen A, Satish N, Chhugani J, Kim C, Dubey P. 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. Paper presented at: SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. New Orleans, LA; 2010:1-13.
19. Micikevicius P. 3D finite difference computation on GPUs using CUDA. Paper presented at: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units GPGPU-2; 2009:79-84; Association for Computing Machinery, Washington, DC.
20. Matsumura K, Zohouri HR, Wahib M, Endo T, Matsuoka S. AN5D: automated stencil framework for high-degree temporal blocking on GPUs. Paper presented at: Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization CGO 2020; 2020:199-211; Association for Computing Machinery, San Diego, CA.
21. Rawat P, Kong M, Henretty T, et al. SDSLc: a multi-target domain-specific compiler for stencil computations. Paper presented at: Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing WOLFHPC '15. Association for Computing Machinery; 2015:1-10; Austin, TX.
22. Rawat PS, Vaidya M, Sukumaran-Rajam A, et al. Domain-specific optimization and generation of high-performance GPU code for stencil computations. *Proc IEEE*. 2018;106(11):1902-1920. https://doi.org/10.1109/JPROC.2018.2862896
23. Rawat PS, Vaidya M, Sukumaran-Rajam A, Rountev A, Pouchet LN, Sadayappan P. On optimizing complex stencils on GPUs. Paper presented at: Proceedings of the 2019 IEEE International Parallel and Distributed Processing Symposium. Rio de Janeiro, Brazil; 2019:641-652.
24. Rawat PS, Rastello F, Sukumaran-Rajam A, Pouchet LN, Rountev A, Sadayappan P. Register optimizations for stencils on GPUs. *ACM SIGPLAN Not*. 2018;53(1):168-182. https://doi.org/10.1145/3200691.3178500
25. Bondhugula U, Hartono A, Ramanujam J, Sadayappan P. A practical automatic polyhedral parallelizer and locality optimizer. *ACM SIGPLAN Not*. 2008;43(6):101-113. https://doi.org/10.1145/1379022.1375595
26. Bandishti V, Pananilath I, Bondhugula U. Tiling stencil computations to maximize parallelism. Paper presented at: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis SC'12. Salt Lake City, UT; 2012:1-11.
27. Steuwer M, Remmelg T, Dubach C. LIFT: a functional data-parallel IR for high-performance GPU code generation. Paper presented at: Proceedings of the 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2017). Austin, TX; 2017:74-85.
28. Lücke M, Steuwer M, Smith A. A functional pattern-based language in mlir. Paper presented at: Proceedings of the 2nd Workshop on Accelerated Machine Learning@ ISCA 2020. Valencia, Spain; 2020:6.
29. Fuhrer O, Osuna C, Lapillonne X, et al. Towards a performance portable, architecture agnostic implementation strategy for weather and climate models. *Supercomput Front Innovat Int J*. 2014;1(1):45-62. https://doi.org/10.14529/jsfi140103
30. Gorius JM, Grosser T. Modeling stencils in a multi-level intermediate representation; 2019:15.
31. Gysi T, Müller C, Zinenko O, et al. Domain-Specific Multi-Level IR Rewriting for GPU; 2020. arXiv:2005.13014[cs].
32. Baghdadi R, Beaugnon U, Cohen A, et al. PENCIL: a platform-neutral compute intermediate language for accelerator programming. Paper presented at: Proceedings of the 2015 International Conference on Parallel Architecture and Compilation. San Francisco, CA; 2015:138-149.
33. Baghdadi R, Ray J, Romdhane MB, et al. Tiramisu: a polyhedral compiler for expressing fast and portable code. *CGO '2019*. Washington, DC: IEEE Press; 2019:193-205.
34. Christen M, Schenk O, Burkhart H. PATUS: a code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. Paper presented at: Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium. Anchorage, AK; 2011:676-687.
35. Louboutin M, Lange M, Luporini F, et al. Devito (v3.1.0): an embedded domain-specific language for finite differences and geophysical exploration. *Geosci Model Develop*. 2019;12(3):1165-1187. Publisher: Copernicus GmbH. https://doi.org/10.5194/gmd-12-1165-2019
36. Ragan-Kelley J, Barnes C, Adams A, Paris S, Durand F, Amarasinghe S. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. Paper presented at: Proceedings of the PLDI '13; 2013:519-530; Association for Computing Machinery, Seattle, Washington, DC.
37. Zhou K, Meng X, Sai R, Mellor-Crummey J. GPA: a GPU performance advisor based on instruction sampling. In: *CGO'21*; 2021.
38. Adhianto L, Banerjee S, Fagan M, Krentel M, Mellor-Crummey J, Tallent N. HPCToolkit: tools for performance analysis of optimized parallel programs. *Concurr Comput Pract Exper*. 2010;22(6):685-701. https://doi.org/10.1002/cpe.1553.10.1002/cpe.1553
39. Zhou K, Krentel MW, Mellor-Crummey J. Tools for top-down performance analysis of GPU-accelerated applications. *ICS'20*. New York, NY: Association for Computing Machinery; 2020.
40. NVIDIA. Profiler-CUDA toolkit documentation; 2021. https://docs.nvidia.com/cuda/profiler-users-guide/index.h%tml#nvprof-overview.
41. NVIDIA. Nsight compute roofline analysis; 2021. https://docs.nvidia.com/nsight-compute/ProfilingGuide/ind%ex.html#roofline.
42. AMD. ROC-profiler; 2021. https://github.com/ROCm-Developer-Tools/rocprofiler.
43. Williams S, Waterman A, Patterson D. Roofline: an insightful visual performance model for multicore architectures. *Commun ACM*. 2009;52(4):65-76. https://doi.org/10.1145/1498765.1498785
44. Yang C. Empirical roofline toolkit; 2021. https://bitbucket.org/berkeleylab/cs-roofline-toolkit.

How to cite this article: Sai R, Mellor-Crummey J, Meng X, Zhou K, Araya-Polo M, Meng J. Accelerating high-order stencils on GPUs. *Concurrency Computat Pract Exper*. 2021;e6467. https://doi.org/10.1002/cpe.6467