2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)

Using the Semi-Stencil Algorithm to Accelerate High-Order Stencils on GPUs

Ryuichi Sai, John Mellor-Crummey, Xiaozhu Meng
Department of Computer Science, Rice University
Houston, TX, USA
{ryuichi, johnmc, xm13}@rice.edu

Mauricio Araya-Polo, Jie Meng
TotalEnergies EP Research & Technology USA, LLC.
Houston, TX, USA
{mauricio.araya, jie.meng}@totalenergies.com

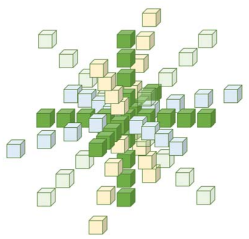
***Abstract*—Understanding how to develop efficient high-order stencils for Graphics Processing Units (GPUs) is a topic of great interest for many application domains. High-performance stencils on GPUs must be tailored for data parallel computation and to use the memory hierarchy efficiently. For data-intensive high-order stencils, the key to high performance on GPUs is reducing the shared memory footprint to enable a large thread block for hiding memory latency. In this paper, we use the semi-stencil algorithm to do so. On the NVIDIA A100, a CUDA implementation of the semi-stencil algorithm along with other optimizations achieves a $2.13\times$ speedup compared to an OpenACC reference implementation and 8.7% faster than the best conventional stencil computing out of shared memory. We evaluate the performance of our implementations on the latest NVIDIA GPUs.**

***Index Terms*—stencil computation, high-order, wave equation, HPC, GPU, semi-stencil**

I. Introduction

Accelerating high-order stencils on Graphics Processing Units (GPUs) is important for many application domains. However, implementing them efficiently is challenging because of the range of implementation choices, resource constraints, and varying characteristics of different GPUs.

Prior work on high-order GPU stencils by Sai et al. [1] studied 25-pt star stencils used to solve the acoustic wave equation on isotropic media (acoustic iso). In this paper, we study a compact-in-space [2] 73-pt stencil (shown below) used to solve the acoustic wave equation on tilted transversely isotropic media (acoustic TTI). Sai et al.'s acoustic iso implementation uses five full-size 3D data arrays; our acoustic TTI implementation needs 13. As shown later in Section V, applying the 73-pt stencil to values in GPU shared-memory is much faster than computing with values from global memory. However, for such data-intensive stencils, shared-memory usage limits the size of GPU thread blocks, which prevents from hiding data access latency.



In this work, we explore strategies for efficiently applying the 73-pt acoustic TTI stencil. The principal focus of this work is evaluating GPU implementations of the semi-stencil algorithm [3]. The semi-stencil algorithm reduces the shared memory footprint of a thread block to roughly half of that needed by conventional stencil implementations. The smaller shared memory footprint using the semi-stencil algorithm enables us to employ optimizations unavailable to conventional stencil implementations because of resource limitations. We compare our implementations of the acoustic TTI stencil using the semi-stencil algorithm with other approaches.

This paper makes the following contributions:

- it describes and evaluates strategies for high-performance implementation of the acoustic TTI stencil on GPUs;
- it explores the benefits of the semi-stencil algorithm for complex high-order stencils;
- it compares performance on NVIDIA A100 and V100 GPUs; and
- it quantitatively assesses our kernel implementations w.r.t. execution time, Roofline, memory footprint, code complexity, and performance portability.

The layout of the paper is as follows: Section II provides background about stencil computations, complex wave equation, GPU performance issues, 2.5D streaming, the semi-stencil approach, and other stencil optimizations. Section III describes our high-level approach and details semi-stencil implementation and its variants. Section IV describes optimizations in performance tuning. Section V describes our evaluation methodology, experimental results, and findings. Section VI summarizes our conclusions.

II. Background and Related Work

A. Stencil Computations

Stencil computations are used to update data elements, called points (usually a differential operator), on a data grid. A new data value for a point is computed as a weighted sum of products between values of a set of neighboring points and scaling coefficients. The set of points used in the calculation defines a stencil. The maximum distance from the center point along an axis to a neighboring point in the stencil defines the order R of the stencil. The halo size along each axis direction is defined by the maximum distance from the center to a neighbor along that axis direction.

B. Acoustic TTI Kernel

We investigate acoustic wave-propagation in Tilted Transversely Isotropic (TTI) media on a staggered Lebedev-grid, inspired by the formulation in [4], [5]. This approach is widely used in the energy industry when a complex medium (earth model) needs to be accurately reconstructed by simulation.

This method involves a coupled system of equations:

$$\frac{1}{V_p^2}\frac{\partial^2 p}{\partial t^2} = (1+2\delta)(\partial^2 - H)(p+q) + Hp \tag{1}$$

$$\frac{1}{V_p^2}\frac{\partial^2 q}{\partial t^2} = 2(\epsilon-\delta)(\partial^2 - H)(p+q) \tag{2}$$

where $H = (\sin\phi\partial + \cos\phi\partial)^2$, p represents the pressure field, q is an ancillary structure, δ and ϵ are anisotropic parameters as defined in [6], V_p is the medium property along the axis of symmetry and ϕ is the tilt from vertical. We solve these equations by applying a 73-pt $4th$ order stencil to the spatial discretization of p and q. To compute each grid point values from 13 different arrays are needed, where 4 arrays contain p and q, 4 arrays contain V_p, ϕ, ϵ and δ and the rest hold perfectly matched layer (see [7]) boundary condition values.

While this paper focuses on the acoustic TTI kernel, the techniques that we explore are useful for implementing other high-order stencils with boundary conditions on GPUs.

C. GPU Performance Issues

GPU's architectural characteristics impact performance.

Execution Model: GPU uses a Single-Instruction-Multiple-Thread (SIMT) execution model. A GPU bundles a group of SIMT threads known as a warp or a wavefront. All threads in a warp/wavefront execute the same instruction.

Memory Hierarchy: GPU memory hierarchies include multiple levels of cache as well as specialized resources such as constant memory, texture memory, and shared memory. Each of these resources has a usage quota at thread, warp, thread block, and device levels. Managing footprints in each of these memories is critical to achieve high performance.

Other GPU characteristics that an implementation must consider include branch divergence, work load balance, arithmetic intensity, occupancy; each of these can impact performance.

D. 2.5D Streaming

Nguyen et al. [8] describe 2.5D streaming as part of their 3.5D blocking algorithm. 2.5D streaming involves blocking in a 2D plane and streaming along a third dimension.

Micikevicius [9] describes a 2.5D streaming implementation on NVIDIA GPUs, which loads data points of the currently active plane in shared memory and employs registers to store data elements of a star stencil along a streaming dimension.

The AN5D framework [10] refines the previous work [9] with fixed register allocations, double buffering, and division of the streaming dimension.

Ernst et al. [11] describe a thread folding strategy. For 2.5D streaming, a thread folds the computation of multiple consecutive planes into one step to increase data reuse.

TABLE I: Implementation Strategies

Identifier	Description
gmem_3d_*	3D blocking using global memory only
smem_3d_*	3D blocking using shared memory for the data arrays
smem_25d_*	2.5D streaming with multiple planes in shared memory
semi_25d_*	2.5D streaming using semi-stencil

E. Semi-Stencil

The performance of stencil computations usually suffer from non-contiguous memory accesses, low computation/access, and low data reuse [3]. The semi-stencil algorithm significantly improved performance on CPUs by addressing these three issues by factoring a stencil computation into two halves–a forward update and a backward update. The semi-stencil algorithm loads only the halo on one side rather than both sides. The forward update stores a partial result for the stencil based on the left half of the points along one dimension; the backward update augments the partial result with the calculation using the right half of the points. The semi-stencil algorithm trades half of its loads for a store and reload of a partial result. This is beneficial for high-order stencils with a large halo size, as the reduction in loads becomes larger as stencil width increases.

F. Other Stencil Optimizations

Time skewing [12]–[15] increases data reuse and cache locality by skewing one or more data dimensions by the time dimension and computing several time steps while data are in cache. Overlapped tiling [16], [17] uses time skewing to trade redundant computation along the boundaries of overlapped tiles for a reduction in memory bandwidth. Split tiling [18] is an alternate to overlapped tiling that advancing points by multiple time steps with a two-phase computation—a hyper-trapezoidal tiling along the time dimension followed by a back-fill of the remaining points; Cache-oblivious algorithms [19]–[22] make optimal use of cache level by tiling the domain with a space cut or a time cut without the need to consider cache sizes as parameters.

Other work tackles stencil computations from different angles, including auto-tuning with dynamic resource allocations [23], DAG reordering [24], diamond tiling using a polyhedral model [25], [26], Domain-Specific Languages (DSLs) [11], [22], [27]–[30], functional programming [31], [32], and multi-layer intermediate representations [33]–[35].

III. Semi-Stencil on GPUs

To create a high-performance implementation of the 73-pt acoustic TTI stencil, we explored various issues, including domain decomposition, blocking, algorithmic approaches, and strategies for managing the memory hierarchy.

Prior work by Sai et al. [1] outlines various optimization strategies for high-order seismic stencils on GPUs. We found that their seven-region data domain decomposition works best for the acoustic TTI kernel with an inner region surrounded by a Perfectly Matched Layer (PML) region.

Table I outlines the principal implementation strategies we compared. In this section, we sketch how to use the semi-stencil strategy to implement high-order stencils on GPUs, highlight some important details, and describe two high-performance semi-stencil implementations on GPUs.

To facilitate discussion of our GPU kernels, let N_x, N_y, and N_z denote the extents of the input data region along the coordinate axes. Let D_x, D_y, and D_z be the thread block dimensions along the X, Y, and Z axes, respectively. For 2.5D streaming, z represents the induction variable while streaming along the Z dimension.

A. Semi-Stencil

In one step of a conventional 2.5D streaming stencil computation, values of points in $(2R+1)$ planes are loaded. Using the semi-stencil algorithm along the streaming dimension reduces the number of planes to $(R+1)$. On a GPU, we load these $(R+1)$ planes in shared memory. Each thread uses registers to hold partial results along the streaming dimension.

An implementation of the semi-stencil algorithm along the streaming dimension of a 2.5D algorithm on a GPU has three phases: (1) a prologue that loads several initial planes of data into shared memory and performs forward computations for a few planes to prepare for a streaming loop; (2) a streaming loop that performs both forward and backward computations on the planes in shared memory, writes one plane of values back to global memory, and loads the next input plane into shared memory; and (3) an epilogue that completes the computation for the last few planes using only backward computations. We describe these phases in more detail below.

Prologue: To prepare for streaming, data from planes of $z \leftarrow [-R..0)$ are loaded into shared memory. Next, we perform R streaming steps. Each step loads a *current plane* of data and uses data from R planes above the current plane to perform the forward computation. Each thread stores its partial result for each streaming step into a separate register.

Streaming Loop: Inside the streaming loop, for each $z \leftarrow [0..N_z)$, we first load the plane of data at $z+R$, i.e., R steps below the current plane at z. Then, each thread performs both the forward computation for its point at $z+R$ and the backward computation for its point at z. Each thread's forward computation computes a partial result for point $z+R$. Each thread's backward computation loads the partial result for point z, computed R steps earlier in the prologue or an earlier streaming step, and completes the stencil computation using values from shared memory in planes $z..(z+R)$. After completing both the forward and backward computations for a point, the kernel stores the final result back to global memory.

Epilogue: For each step $z \leftarrow [N_z..(N_z+R))$, we load data from plane $z+R$. We don't need any more forward computation in this phase; we only perform backward computation to complete the stencil computation for these planes.

The number of threads in a 2.5D streaming approach is determined by the size of the 2D plane. To maximize thread-level parallelism, the 2D plane size needs to be as large as possible. With $(R+1)$ planes, the required size of shared memory is $(D_x+2R) \times (D_y+2R) \times (R+1)$ per thread block. The data of a 2D plane is always loaded collaboratively with all threads in the 2D thread block.

B. Semi-Stencil with Buffering the Next Plane

This variant of the semi-stencil implementation simultaneously prefetches the plane at $(z+R+1)$ while performing forward and backward computations using planes $z..(z+R)$. This overlaps fetching the missing plane needed by the next iteration of the streaming computation while computing with the planes already in memory.

Overlapping the data fetching with computation hides the memory latency of the data fetch; but, it requires an additional plane in shared memory as the target for the asynchronous copy. Using this approach, the required size of shared memory increases to $(D_x+2R) \times (D_y+2R) \times (R+2)$ per thread block. If shared memory for the thread block is a limiting resource, the need for an extra plane might reduce size of the largest feasible 2D thread block. If so, that would reduce thread-level parallelism, which could hurt performance.

C. Semi-Stencil with 2z Thread Folding

Ernst et al. [11] describe benefits from folding the computation of f adjacent planes together to reuse data values in registers. We applied this idea to a 2.5D semi-stencil streaming approach by folding the computation of f planes into one streaming step. Instead of performing semi-stencil for one active plane, each step computes f planes. As each streaming step completes, z advances by f rather than by 1. With more planes being used in one step, this variant requires additional shared memory space to hold the points in the additional planes. In our case, due to resource limits on shared memory size, we can only reasonably do $f = 2$ planes at once, thus 2z folding. The shared memory required is $(D_x+2R) \times (D_y+2R) \times (R+3)$ per thread block.

IV. Optimizations

To accelerate kernel execution, we compose a variety of optimizations. We apply a set of optimizations described previously in the literature [1], which include using constant memory for stencil coefficients, allocating data in pinned memory, and padding to improve cache line alignment. This section describes optimizations that work well with the semi-stencil algorithm and speed up the acoustic TTI kernel.

A. Avoiding Unnecessary Data Movement

Data movement in GPUs requires multiple instruction cycles. Moving data needed by multiple threads can cause stalls from both the copy latency and thread synchronization. Our 2.5D streaming kernels avoid such stalls by maintaining a circular buffer of planes in shared memory. Each streaming iteration uses a different window of planes in the buffer. We use loop unrolling so that each iteration of the streaming algorithm accesses the proper window of planes. Several iterations after a plane is loaded, it will no longer be needed and will be replaced by another.

TABLE II: System Specifications

GPU	NVIDIA A100	NVIDIA V100
CPU	AMD EPYC 7402	IBM POWER9
CPU Cores	96	160
RAM	512 GB	256 GB
OS	RHEL v8.4	RHEL v7.7
Platform	CUDA 11.2	CUDA 11.0
GPU Driver	NV 470.57.02	NV 450.51.05

B. *Arithmetic Index*

High-order stencils load a large number of halo points. On a GPU, an inefficient strategy for loading halo points could cause two performance problems. First, there may be imbalanced work: while some threads are busy loading, others sit idle. Second, conditional checking on a GPU results in branch divergence among threads in a warp. Both have significant impact on GPU performance due to GPU's SIMT execution model. Our goals are to distribute work evenly and reduce conditionals, so that most or all threads contribute to the work evenly, shortening the overall execution time.

To achieve these goals, we choose *one* axis and partition the threads along that axis so that each thread fetches halo points to the left or the right along *each* axis. This approach uses a single conditional, fetches halos in parallel with multiple threads, and spreads the work evenly among the threads.

C. *Overlapping Computation with Data Fetching*

To hide memory access latency, our implementations overlap computation with data fetching. This approach improves performance by reducing stalls from thread barriers.

NVIDIA introduced hardware support for `memcpy_async` in their latest A100 GPU. With `memcpy_async`, which enables the data copies from global memory to shared memory without using registers; this leaves more registers available for use in a computation.

Some of our code versions overlap semi-stencil computation with data prefetching using `memcpy_async` when buffering for next plane. The hardware support for `memcpy_async` on the A100 boosts performance.

V. Evaluation

We evaluated our stencils on two NVIDIA GPU platforms. Table II describes hardware and software specifications for these systems. In the rest of the discussions, we refer to these systems by their GPU models.

We assess all implementations and their variants that differ in algorithmic choices, optimization strategies, and GPU thread block dimensions. For each machine, we run the kernels with a large grid size based on its device memory capacity. We run with grid size of 900^3 for the `A100` and 850^3 for the `V100`. We set compiler flags to `-O3 -arch=sm_80` for the `A100` and `-O3 -arch=sm_70` for the `V100`.

We first present a comparison of time measurements. Next, we present a table with Roofline measurements of our kernels on the A100, comparing achieved performance to empirical peak performance. Finally, we discuss our results and offer some of observations.

TABLE III: Time measurement in seconds on NVIDIA GPUs

Kernel Identifier	A100	V100
gmem_3d_8x8x8	129.85 ± 0.72	162.69 ± 0.04
gmem_3d_32x4x4	88.19 ± 0.16	120.72 ± 0.06
smem_3d_8x8x8	80.22 ± 0.27	112.96 ± 0.05
smem_3d_32x4x4	51.56 ± 0.09	76.54 ± 0.05
smem_25d_16x8	88.61 ± 0.10	126.94 ± 0.04
smem_25d_16x16	76.52 ± 0.16	122.98 ± 0.12
semi_25d_16x8	71.53 ± 0.08	155.04 ± 0.22
semi_25d_16x16	69.42 ± 0.09	145.07 ± 0.16
semi_25d_32x16	54.74 ± 0.16	97.84 ± 0.10
semi+b_25d_16x16	65.12 ± 0.09	130.42 ± 0.18
semi+b_25d_32x16	52.06 ± 0.22	84.47 ± 0.09
semi+b+a_25d_16x16	59.97 ± 0.10	130.41 ± 0.18
semi+b+a_25d_32x16	47.43 ± 0.01	84.44 ± 0.08
semi+2z_25d_16x8	66.75 ± 0.10	122.97 ± 0.15
semi+2z_25d_16x16	66.37 ± 0.04	117.63 ± 0.09

Kernels with `+b` buffers the next plane, `+a` uses `memcpy_async` for A100, and `+2z` uses 2z thread folding.

TABLE IV: DRAM Roofline for Top Performers on the A100

Kernel Identifier	FLOP ($\times10^{13}$)	AI*	Peak GF**	Achieved GF**	Achieved % peak
gmem_3d_32x4x4	18.496	3.73	4570	1999	43.74%
smem_3d_32x4x4	18.496	3.74	4576	3453	75.46%
smem_25d_16x16	17.818	4.19	5134	2221	43.26%
semi_25d_32x16	19.352	3.34	4084	3396	83.16%
semi+b_25d_32x16	19.352	3.37	4124	3585	86.94%
semi+b+a_25d_32x16	19.352	3.48	4263	3940	92.42%
semi+2z_25d_16x16	19.283	4.43	5424	2746	50.64%

*Arithmetic Intensity **GFLOPs

A. *Time measurements and comparison*

Table III presents time measurements for 1000 time steps for inner and PML regions. Prior to each simulation, we first execute a few warmup iterations. Then, we execute each simulation ten times. We report the average execution time of the simulations as well as their standard deviation. The block dimension variants can be inferred from the kernel identifiers. 3D blocking kernels and 2.5D blocking ones follow an `id_3d_Dx_Dy_Dz` and an `id_25d_Dx_Dy` pattern, respectively. Our simulations have halo size R of 4.

Our fastest implementation on the A100 is `semi+b+a_25d_32x16`. On the NVIDIA A100, it was 2.13× faster than an OpenACC reference implementation and 8.7% faster than the best conventional stencil computing out of shared memory.

B. *Roofline*

Table IV compares the performance of top-performing kernels with the bound imposed by the GPU DRAM bandwidth using the Roofline [36], [37] performance model.

Our best kernel `semi+b+a_25d_32x16` has a DRAM arithmetic intensity of 3.48 and achieved 3,940 GFLOPs, which is 92.42% of the DRAM performance bound.

C. *Discussion*

a) Semi-Stencil on GPUs: For high-order stencils, the semi-stencil algorithm reduces the shared memory footprint

TABLE V: Shared Memory Usage Table (in bytes) for Acoustic TTI Kernels

D_x	D_y	Number of Planes				
		5	6	7	8	9
8	8	10240	12288	14336	16384	18432
16	8	15360	18432	21504	24576	27648
16	16	23040	27648	32256	36864	41472
32	16	38400	46080	53760	61440	69120
32	32	64000	76800	89600	102400	115200

enabling high performance on GPUs. Because a GPU has limited on-device resources and each of thread block is bound by its own resource quotas (e.g. registers, shared memory), a reduction in resource use by a thread may permit launching a GPU block with more threads, yielding better thread-level parallelism. Semi-stencil is appealing for high-order stencils because the larger the halo size R, the higher shared memory savings one can achieve. For acoustic TTI kernels, without semi-stencil, the 2D block size is limited by shared memory to `16x16`; the block size doubles to `32x16` when using semi-stencil, which has better performance with twice the threads.

b) Buffering with Prefetching: Applying prefetching while holding the thread block size constant can improve performance. Without prefetching, one must synchronize before loading a new plane to ensure all threads are done with the plane being overwritten and then after loading a new plane to make sure that no thread accesses a value before it is present. With prefetching only one synchronization is needed per iteration because the prefetched plane is not used by the computation in the current iteration. Since stalls due to thread synchronization are expensive, removing one synchronization per iteration boosts performance.

In our experience, reducing thread block size to free up shared memory for prefetching isn't profitable.

c) Overlapping Prefetching with Computation: Because `memcpy_async` on the `A100` does not use registers, more registers are available to the computation. Overlapping the memory copy with the computation hides access latency and boosts performance.

d) Memory Footprint v.s. Block Dimensions: Achieving top performance on a GPU requires a delicate balance between competing concerns.

When using 2.5D blocking, using larger 2D plane tends to improve performance. We also observe performance improvements by applying optimization strategies, such as buffering next plane data using prefetching, especially when `memcpy_async` is available.

However, allocating larger 2D planes in shared memory in conjunction with optimizations stresses resource limits. For acoustic TTI kernels, Table V show the required shared memory size in bytes with respect to number of planes and 2D thread block dimensions (D_x and D_y). Because NVIDIA GPUs allow maximum 49152 bytes of static shared memory allocation per block, unachievable combinations are marked in red. One can use `cudaFuncSetAttribute` to increase the amount of shared memory at the expense of decreasing the L1 cache size. We leave this investigation as future work.

This shows that by balancing conflicting interests — maximizing thread-level parallelism with utilizing limited GPU hardware resources — semi-stencil with next plane buffering achieved excellent performance. Semi-stencil reduces the number of planes in shared memory from nine planes to five, enabling kernels to be launched with double the number of threads. Because of the dramatic reduction in shared-memory consumption with the semi-stencil algorithm, there is enough shared memory to support overlapping asynchronous prefetching of the next plane with the computation.

Our results show that although 2z thread folding increases performance with the same 2D plane size, it doesn't yield the best-performing acoustic TTI kernels because resource constraints require a smaller 2D plane, which lowers thread-level parallelism.

e) Code Portability: The `smem_3d_32x4x4` implementation delivers high performance across a range of GPUs. Its implementation has 211 lines of code while `semi+b+a_25d_32x16` has 513 lines, so it is easier to develop. But it is 8.7% slower than the fastest semi-stencil performance on A100 using asynchronized memory copy.

VI. Conclusions

This paper evaluates the performance of several implementations of a complex high-order stencil with boundary conditions. It is well known that GPU implementations of high-order stencils need careful design to best use limited hardware resources. In particular, memory hierarchy issues such as memory latency, data locality, data layout, and cache alignment have huge effects on GPU performance.

A new insight in this paper is that the semi-stencil algorithm is an essential component of the top performing GPU implementation of a complex high-order stencil used to perform a computation of industrial interest.

For large problems, using large blocks that occupy all GPU threads is an important ingredient for high performance implementations. For complex stencils, computing with values in GPU shared memory is typically faster than using global memory. An advantage of the semi-stencil algorithm is that it reduces the GPU shared memory footprint for a high-order stencil to roughly half of that used by the conventional approach. Beyond enabling a large block size that uses all GPU threads, the semi-stencil algorithm saves enough space in shared memory that we are able to allocate an extra plane of data that we can use to prefetch values for the next iteration. In our 2.5D streaming semi-stencil on the `A100`, `memcpy_async` overlaps the prefetch of a plane of data for the next iteration with the computation of a plane of outputs for the current iteration. This overlap enables an implementation that is 8.7% faster than the best conventional stencil computing out of GPU shared memory.

Acknowledgment

This work was supported in part by a contract from TotalEnergies EP Research & Technology USA, LLC.

References

[1] R. Sai, J. Mellor-Crummey, X. Meng, K. Zhou, M. Araya-Polo, and J. Meng, "Accelerating high-order stencils on gpus," *Concurrency and Computation: Practice and Experience*, vol. e6467.

[2] B. Hamilton, C. J. Webb, A. Gray, and S. Bilbao, "Large stencil operations for GPU-based 3-D acoustics simulations," in *Proceedings of the* 18^{th} *International Conference on Digital Audio Effects*. Norwegian University of Science and Technology, Nov. 2015, pp. 292–299, iSSN 2413-6689.

[3] R. de la Cruz and M. Araya-Polo, "Algorithm 942: Semi-Stencil," *ACM Transactions on Mathematical Software*, vol. 40, no. 3, pp. 23:1–23:39, Apr. 2014.

[4] K. P. Bube, T. Nemeth, J. P. Stefani, W. Liu, K. T. Nihei, R. Ergas, and L. Zhang, "First-order systems for elastic and acoustic variable-tilt ti media," *GEOPHYSICS*, vol. 77, no. 5, pp. T157–T170, 2012.

[5] K. Bube, T. Nemeth, P. Stefani, R. Ergas, W. Lui, T. Nihei, and L. Zhang, "On the instability in second-order systems for acoustic vti and tti media," *Geophysics*, vol. 77, pp. 171–186, 2012.

[6] L. Thomsen, "Weak elastic anisotropy," *GEOPHYSICS*, vol. 51, no. 10, pp. 1954–1966, 1986.

[7] R. Martin, D. Komatitsch, and S. D. Gedney, "A variational formulation of a stabilized unsplit convolutional perfectly matched layer for the isotropic or anisotropic seismic wave equation," *Comput. Model. Eng. Sci*, vol. 37, no. 3, pp. 274–304, 2008.

[8] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey, "3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs," in *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2010, pp. 1–13, iSSN: 2167-4337.

[9] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2. Washington, D.C., USA: Association for Computing Machinery, Mar. 2009, pp. 79–84.

[10] K. Matsumura, H. R. Zohouri, M. Wahib, T. Endo, and S. Matsuoka, "AN5D: automated stencil framework for high-degree temporal blocking on GPUs," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. San Diego, CA, USA: Association for Computing Machinery, Feb. 2020, pp. 199–211.

[11] D. Ernst, G. Hager, M. Holzer, M. Knorr, and G. Wellein, "Opening the Black Box: Performance Estimation during Code Generation for GPUs," 2021. [Online]. Available: https://arxiv.org/abs/2107.01143

[12] D. Wonnacott, "Achieving Scalable Locality with Time Skewing," *International Journal of Parallel Programming*, vol. 30, no. 3, pp. 181–221, Jun. 2002.

[13] G. Jin, J. Mellor-Crummey, and R. Fowler, "Increasing temporal locality with skewing and recursive blocking," in *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, ser. SC '01. Denver, Colorado: Association for Computing Machinery, Nov. 2001, p. 43.

[14] J. McCalpin and D. Wonnacott, "Time Skewing: A Value-Based Approach to Optimizing for Memory Locality," 1998.

[15] Y. Song and Z. Li, "New tiling techniques to improve cache temporal locality," in *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*. Atlanta, Georgia, USA: Association for Computing Machinery, May 1999, pp. 215–228.

[16] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Effective automatic parallelization of stencil computations," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 235–244, Jun. 2007.

[17] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on GPU architectures," in *Proceedings of the 26th ACM international conference on Supercomputing*, ser. ICS '12. San Servolo Island, Venice, Italy: Association for Computing Machinery, Jun. 2012, pp. 311–320.

[18] T. Grosser, A. Cohen, P. H. J. Kelly, J. Ramanujam, P. Sadayappan, and S. Verdoolaege, "Split tiling for GPUs: automatic parallelization using trapezoidal tiles," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. Houston, Texas, USA: Association for Computing Machinery, Mar. 2013, pp. 24–31.

[19] M. Frigo and V. Strumpen, "Cache oblivious stencil computations," in *Proceedings of the 19th annual international conference on Supercomputing*, ser. ICS '05. Cambridge, Massachusetts: Association for Computing Machinery, Jun. 2005, pp. 361–366.

[20] M. Frigo and V. Strumpen, "The cache complexity of multithreaded cache oblivious algorithms," in *Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, ser. SPAA '06. Cambridge, Massachusetts, USA: Association for Computing Machinery, Jul. 2006, pp. 271–280.

[21] R. Strzodka, M. Shaheen, D. Pajak, and H.-P. Seidel, "Cache oblivious parallelograms in iterative stencil computations," in *Proceedings of the 24th ACM International Conference on Supercomputing*. Tsukuba, Ibaraki, Japan: Association for Computing Machinery, Jun. 2010.

[22] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The Pochoir Stencil Compiler," in *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, ser. SPAA '11. San Jose, California, USA: Association for Computing Machinery, Jun. 2011, pp. 117–128.

[23] P. S. Rawat, M. Vaidya, A. Sukumaran-Rajam, A. Rountev, L.-N. Pouchet, and P. Sadayappan, "On Optimizing Complex Stencils on GPUs," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2019, pp. 641–652, iSSN: 1530-2075.

[24] P. S. Rawat, F. Rastello, A. Sukumaran-Rajam, L.-N. Pouchet, A. Rountev, and P. Sadayappan, "Register optimizations for stencils on GPUs," *ACM SIGPLAN Notices*, vol. 53, no. 1, pp. 168–182, Feb. 2018.

[25] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," *ACM SIGPLAN Notices*, vol. 43, no. 6, pp. 101–113, Jun. 2008.

[26] V. Bandishti, I. Pananilath, and U. Bondhugula, "Tiling stencil computations to maximize parallelism," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov. 2012, pp. 1–11, iSSN: 2167-4337.

[27] P. S. Rawat, M. Vaidya, A. Sukumaran-Rajam, M. Ravishankar, V. Grover, A. Rountev, L.-N. Pouchet, and P. Sadayappan, "Domain-Specific Optimization and Generation of High-Performance GPU Code for Stencil Computations," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1902–1920, Nov. 2018, conference Name: Proceedings of the IEEE.

[28] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, J. Absar, S. Van Haastregt, A. Kravets, A. Lokhmotov, R. David, and E. Hajiyev, "PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, Oct. 2015, pp. 138–149.

[29] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe, "Tiramisu: a polyhedral compiler for expressing fast and portable code," in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO 2019. Washington, DC, USA: IEEE Press, Feb. 2019, pp. 193–205.

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

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

[32] M. Lücke, M. Steuwer, and A. Smith, "A functional pattern-based language in mlir," p. 6, 2020.

[33] O. Fuhrer, C. Osuna, X. Lapillonne, T. Gysi, B. Cumming, M. Bianco, A. Arteaga, and T. Schulthess, "Towards a performance portable, architecture agnostic implementation strategy for weather and climate models," *Supercomputing Frontiers and Innovations: an International Journal*, vol. 1, no. 1, pp. 45–62, Apr. 2014.

[34] J.-M. Gorius and T. Grosser, "Modeling Stencils in a Multi-Level Intermediate Representation," p. 15, 2019.

[35] T. Gysi, C. Müller, O. Zinenko, S. Herhut, E. Davis, T. Wicky, O. Fuhrer, T. Hoefler, and T. Grosser, "Domain-Specific Multi-Level IR Rewriting for GPU," *arXiv:2005.13014 [cs]*, May 2020, arXiv: 2005.13014. [Online]. Available: http://arxiv.org/abs/2005.13014

[36] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009.

[37] C. Yang, "Empirical roofline toolkit," URL: https://bitbucket.org/berkeleylab/cs-roofline-toolkit, 2 2021.