Accelerating High-Order Stencils on GPUs

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

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

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

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

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

***Abstract*—While implementation strategies for low-order stencils on GPUs have been well-studied in the literature, not all of the techniques work well for high-order stencils, such as those used for seismic imaging. In this paper, we study practical seismic imaging computations on GPUs using high-order stencils on large domains with meaningful boundary conditions. We manually crafted a collection of implementations of a 25-point seismic modeling stencil in CUDA along with code to apply the boundary conditions. We evaluated our stencil code shapes, memory hierarchy usage, data-fetching patterns, and other performance attributes. We conducted an empirical evaluation of these stencils using several mature and emerging tools and discuss our quantitative findings. Some of our implementations achieved twice the performance of a proprietary code developed in C and mapped to GPUs using OpenACC. Additionally, several of our implementations have excellent performance portability.**

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

I. Introduction

In the oil and gas industry, seismic depth imaging based on high-order stencils is the main tool used to identify relevant subsurface structures. Today, HPC platforms often employ Graphics Processing Units (GPUs) to increase their computational power. For that reason, understanding how to develop efficient high-order stencils for GPUs is a topic of significant interest. However, the complexity of GPU architectures makes achieving top performance with high-order stencils surprisingly difficult. Without careful design, stencil computations on GPUs are likely to underperform. An efficient implementation of high-order stencils with boundary conditions on a GPU requires paying careful attention to data reuse, warp utilization, work balance, and arithmetic intensity among other issues.

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

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

- a careful comparison of existing approaches, including an assessment of their strengths and weaknesses when applied to high-order stencils with boundary conditions;
- the implementation and tuning of a collection of high-order stencil kernels with a selected set of algorithms and their variants using CUDA;
- a performance comparison of stencil implementations across multiple generations of NVIDIA GPUs (NVS510, P100, V100, and A100), along with a quantitative assessment of stencil performance using a Roofline performance model for GPUs; and
- an investigation of the characteristics of high-order stencil kernels that affect their performance.

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

II. Related Work

There are many papers that describe strategies for efficient stencil implementations on CPUs [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [14], [15], [11] and GPUs [12], [13], [16], [17], [18], [21], [22]. We discuss the most related efforts below.

Time skewing [6], [7] accelerates stencil computations by increasing data reuse and cache locality by skewing one or more data dimensions by the time dimension so that several time steps can be computed for a tile while values are in cache. It has been widely used on CPUs, e.g., [8], [9], and [10].

Overlapped tiling uses time skewing to increase the arithmetic intensity of parallel stencil computations by trading redundant computation along the boundaries of overlapped tiles for a reduction in memory bandwidth required [11], [12]. Overlapped tiling is effective on GPUs because loading data

978-0-7381-1048-6/20/$31.00 ©2020 IEEE
DOI 10.1109/PMBS51919.2020.00014

from a GPU's global memory is much more costly than data-parallel computation. Furthermore, redundant computation can be overlapped with data accesses to help hide memory latency. While overlapped tiling has been shown to improve the performance of low-order stencils on GPUs, for high-order stencils, redundant computation grows quickly when skewed across multiple time steps by the width of a high-order stencil.

Split tiling [13] is an alternate approach for accelerating computation with time skewing. Rather than using overlapped tiles, which can introduce large amounts of redundant computation, split tiling computes points in two phases. The first phase computes tiles in parallel as hypertrapezoids that taper along the time dimension. Once all tiles from the first phase have been computed, a second phase back-fills the missing points in the time dimension.

Nguyen et al. [16] introduce a 3.5D blocking algorithm as a mix of 2.5D spatial blocking with 1D temporal blocking. 2.5D spatial blocking involves blocking in a 2D plane and streaming along a third dimension. To increase data reuse, they store active 2D planes in GPU shared memory. In a 3.5D variant, they employ time skewing to advance the computation for multiple time steps before writing data back to the global memory. While the 3.5D algorithm works very well on CPUs, the 1D temporal blocking introduces two potential implementation challenges for high-order stencils with boundary conditions on GPUs: barrier synchronizations and limited parallelism. In this paper, we evaluate 2.5D spatial blocking of high-order stencils and plan to explore 3.5D blocking in future work.

Nguyen et al.'s approach [16] loads a central plane along with halo planes above and below into shared memory for faster data access while computing stencil operations for points in the central plane. While this strategy improves data reuse, the size of a data tile is limited by the GPU shared memory size. To reduce the shared memory pressure, we looked into the work that uses registers on GPUs. Micikevicius [17] also uses 2.5D blocking; however, his approach maintains data points along the third dimension in registers rather than in shared memory.

In the AN5D framework, Matsumura et al. [18] accelerate 2.5D and 3.5D stencils with three refinements: fixed register allocations, double buffering, and division of the streaming dimension. While these approaches work extremely well for simple single-statement kernels, neither boundary conditions nor multi-statement stencils are evaluated. In our work, we study a high-order stencil with boundary conditions, and part of our application has multiple statements, instead of simple single-statement stencil updates.

Other interesting approaches to tackle the stencil computations include auto-tuning with dynamic resource allocations [19], DAG reordering [20], diamond tiling using a polyhedral model [23], [24], functional programming [25], [26], and multi-layer intermediate representations [27], [28], [29].

From a software engineering perspective, there are two strategies for developing stencils for NVIDIA GPUs: hand-written kernels in CUDA and Domain-Specific Language (DSL)-based approaches [5], [21], [30], [31], [32], [33], [34]. DSL approaches can simplify the generation of code with complex logic. While we are interested in DSL-based approaches for the future, the focus of this paper is to understand in detail the strengths and weaknesses of various algorithmic strategies for achieving high performance and performance portability for high-order stencils. To avoid limitations as we explore this space, we chose to evaluate hand-written kernels.

III. Approach

We developed several implementations of the acoustic isotropic approximation of the wave equation [35] used for seismic imaging by the oil and gas industry. Solving this with finite differences involves using a high-order stencil-based solver with suitable boundary conditions. Oil and gas applications use such strategies on large grids to model subsurface and generate seismic data from source perturbations. In our work, we employ different code shapes that differ principally in how they organize the computation (e.g., 2D vs. 3D tiles) and how they manage the memory hierarchy. In the rest of this section, we briefly describe the acoustic isotropic approximation, explain the various data decomposition strategies we employ, describe blocking strategies, and discuss how we structure our implementations.

A. Seismic Modeling and Acoustic Isotropic Kernel

We study a stencil-based implementation of the acoustic isotropic wave equation approximation for seismic modeling. The details of the model are described in [35]. For our simulations, we employ a Perfectly-Matched Layer (PML) [36] boundary condition to the regions around the physical domain. The resulting extended domain consists of an "inner" region and a surrounding "PML" region.

To solve the acoustic isotropic approximation for the wave equation, in the inner region we apply a complex multi-statement stencil that is 8th-order in space and 2nd-order in time. This involves applying a star shaped 25-point stencil to data stored in the u-array. In the PML layer, we employ a 7-point star shaped stencil to compute boundary conditions. Data for this 7-point stencil is stored in the eta-array.

In production simulations, the grid, which represents the physical domain, is usually large with up to 4000 grid points in each dimension. To yield results of practical use, the stencil computations need to be applied iteratively for a large number of time steps.

While we specifically study the acoustic isotropic kernel as the seismic model wave approximation in this paper, we believe that our approach is general enough that it could be applied to other high-order stencils with boundary conditions.

B. Data Domain Decomposition

As described in the previous section, our data domain contains two regions, the inner region and the Perfectly Matched Layer (PML) boundaries. The inner region is a cubic grid sitting at the center of the data domain and the PML region represents the volume between the inner region and the data

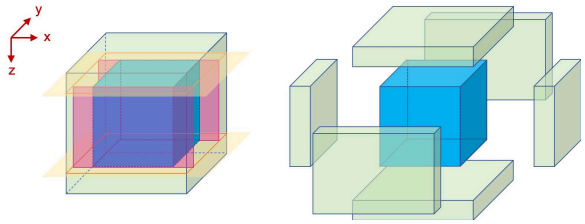


Fig. 1: Data domain decomposition.

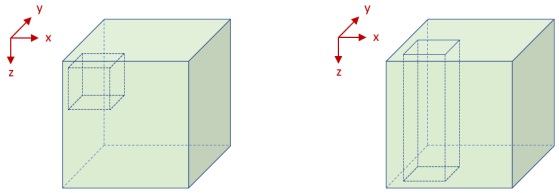


Fig. 2: Blocking strategies: (left) 3D blocking, (right) 2.5D blocking.

domain boundaries. The size of the inner region and the width of the PML region are defined as inputs to a simulation.

GPUs have different architectural characteristics than CPUs. As a result, GPU computations must be structured differently than CPU computations to achieve high performance. First, the Single-Instruction-Multiple-Thread (SIMT) execution model used by GPUs differs significantly from the execution model on CPUs. On NVIDIA GPUs, the execution model is realized by scheduling groups of 32 SIMT threads known as warps. To exploit thread-level parallelism in the SIMT model, GPU computations must utilize fine-grain data parallelism. A group of warps constitute a block which has its own quota for shared memory, registers, and other hardware resources; the number of blocks that can be active simultaneously is limited by the aggregated resource quota of active threads enforced by hardware limits. Second, since GPUs have a memory hierarchy distinct from CPUs, computations must be appropriately structured to exploit the GPU memory hierarchy. Finally, on NVIDIA GPUs, one could realize coarse-grain parallelism across Streaming Multiprocessors (SMs) by partitioning the computation into a sufficient number of blocks to keep the SMs busy.

We experimented with three decomposition strategies based on our data domain and boundary conditions.

First, we developed a single kernel that could be applied to any region of the data domain. The kernel contains conditionals that employ the PML calculations near any of the domain boundaries and compute the stencil for the acoustic isotropic wave function approximation in the inner region of the data domain. This strategy yields branch divergence for subregions that contain points in both the PML and inner regions which hurt performance.

Next, we developed separated kernels for the inner region and the PML region. These separate kernels can be launched concurrently. This strategy eliminates the need for checking whether the point is inside inner region or PML region in every kernel, thus, reducing the chance of branch divergence. Nevertheless, it leaves unbalanced work among threads, along the boundaries between the inner and PML regions when if the size of the GPU blocks doesn't evenly divide the extents of the PML and inner regions.

Lastly, we developed the strategy as shown in Figure 1. We separate the inner region from PML region, and further divide PML region into six subregions. We slice the domain along the top and bottom of the inner region, and this gives us a top block, a bottom block, and a border of four walls. We further slice along the front and back, and it becomes four separate walls. These four walls and the two subregions from the first cuts result in total of six subregions of the PML region, namely: top, bottom, front, back, left and right subregions. The symmetry of these subregions is a relevant characteristic that we discuss later along with our results. Next, we concurrently launch individual GPU kernels of stencil computations for each of the seven subregions: one for the inner region and six for the PML subregions. This approach does not have intrinsic branch divergence at the boundaries. While there are still work imbalances due to different grid sizes, they occur only for a few edge cases along the borders. We could further reduce unbalanced work by using automated code generation that tailors the number of threads to match the number of points at border locations.

C. Blocking Strategies

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

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

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

D. Kernels and their variants

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

IV. Implementations

In describing our implementations, we use R to denote the width of the halo, which is half the spatial order of the stencil. For the acoustic isotropic simulations in our experiments, R is 4. Let 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 denote the 3D coordinate for both a point location in a 3D block and the thread in a kernel thread block. Similarly, for 2D planes, (x, y) is used to locate a point in the 2D plane, as well as identifying the thread.

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

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

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

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

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

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

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

We refer to the implementation that uses 3D blocking and shared memory as $smem\_u$ in our experiments.

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

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

In terms of fetching eta into shared memory, we have two implementations that differ in the number of conditionals. In our experiments, we refer to the shared memory kernel implementation that uses three conditionals as $smem\_eta\_3$ and the implementation that uses one conditional as $smem\_eta\_1$. We let R_eta denote the width of halos for eta, and for the acoustic isotropic PML layer, R_eta is 1.

$smem\_eta\_3$ uses an approach similar to $smem\_u$, where the first $2R\_eta$ threads from each dimension fetch the halos. Since we have three dimensions, we need three conditionals, one for each dimension, respectively. Because R_eta is just 1, we only need two threads fetching halos along each thread dimension. This could introduce unbalanced fetch work because for a 3D block of `8x8x8`, if only two threads in each dimension perform halo fetching, that is one fourth of the threads. Because we have three dimensions, only $1/64$ of the threads perform fetching, while others are idle.

To address the work imbalance, we developed $smem\_eta\_1$ with only one condition, where we choose to use the first six threads from the `X` dimension to fetch halo points. Algorithm 1 describes how we tilt the six planes of threads to identify the halo point that the each thread is responsible for the fetching. For $8 \times 8 \times 8$ 3D blocks, because $6/8$ threads perform the fetching, in theory, we reduce thread idleness to just 20%. However, this algorithm has relatively complex arithmetic to tilt each thread to its proper halo position, so an experimental evaluation is needed to see whether the strategy is profitable.

Data: $\mathbf{xidz}, \mathbf{yidx}, \mathbf{zidx}$: thread index of x, y, and z dimension, respectively
Data: $\mathbf{nt}$: number of threads per block dimension
Result: $\mathbf{g}$: coordinate for global memory
Result: $\mathbf{s}$: 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
```

Algorithm 1: Shared memory halo fetching strategy for eta using only one conditional.

4) 2.5D Streaming with Multi-Plane using Shared Memory: Starting from this implementation, we use 2.5D blocking. As already described, the 2.5D algorithm mainly streams a 2D plane through the third dimension. In our implementations, we choose the 2D XY-subplane because X is the innermost dimension in our data layout. Let Dx and Dy denote the dimensions of the 2D tile along the X and Y axes, respectively. So we launch kernels using 2D thread blocks with the size of Dx by Dy and a total of $Dx \times Dy$ threads. The GPU grid size of each data region is $\lceil Nx/Dx \rceil \times \lceil Ny/Dy \rceil$.

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

Before we can start the streaming computation, points from the top halos are pre-loaded into buffer $B[0..R)$ and the first R XY-subplanes are pre-loaded into $B[R..2R)$. Then, in our streaming loop, for each $z \leftarrow [0..Nz)$, 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.

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

We refer to the family of kernel implementations of this strategy as $st\_smem\_\{Dx\}\_\{Dy\}$ in our experiments.

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

5) 2.5D Streaming using Register Shifting:

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

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

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

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

Although the notation we use above for registers might give the impression that we are using array indexing to access register values, in our implementation, registers are expressed explicitly as $2R+1$ scalar variables. Since our acoustic isotropic kernel has $R=4$, which is the same as the sample code by Micikevicius [17], our implementation uses the same variable names: `behind4`, `behind3`, `behind2`, `behind1`, `current`, `front1`, `front2`, `front3`, and `front4`.

6) 2.5D Streaming using Fixed Registers with Loop Unrolling:

	V100	P100	NVS510
CPU	IBM POWER9	IBM POWER8NVL	Intel Xeon E3-1245 v6
CPU Cores	160	160	8
RAM	256 GB	256 GB	16 GB
GPU	NVIDIA Tesla V100	NVIDIA Tesla P100	NVIDIA NVS 510
GRAM	32 GB	16 GB	2 GB
OS	RHEL v7.7	RHEL v7.4	Ubuntu 18.04 LTS
CUDA	10.2.89	10.1	10.2.89
NVIDIA Driver	440.33.01	418.39	440.33.01

TABLE I: Machine Specifications

Like the previous approach, this implementation uses shared memory for the current XY-subplane and registers for points along the z-axis – the streaming dimension. However, the values in the registers are fixed instead of being "shifted."

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

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

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

We refer to the family of kernel implementations using this strategy as $st\_reg\_fixed\_\{Dx\}\_\{Dy\}$ in our experiments.

V. Evaluation

A. Experimental Platforms

We evaluate all kernel implementations on several machines, each with a different generation of NVIDIA GPU. Table I lists the specifications for our primary experimental platforms. We refer to these machines by their GPU models.

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

We use the `-O3` optimization flag with `nvcc` when compiling kernels for each of the GPUs.

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

Additionally, NVIDIA also ran implementations of three of our fastest kernels on a system with both an A100 GPU and a V100 of their own.

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

We use HPCToolkit [40], [41] version 20200803, Empirical Roofline Toolkit [39], and NVIDIA Nsight Compute version 2019.5.0 during our evaluations.

B. Evaluation Methodologies

We evaluate all implementations and their variants. First, we conducte basic time measurements. Second, we use HPCToolkit's GPU support [41] to profile the kernel details with PC sampling. Third, we run Nsight Compute to measure device-specific kernel characteristics. Finally, we use the Empirical Roofline Toolkit to understand memory bandwidth limits on algorithm performance. We calculate the arithmetic intensity and the performance of every kernel, and compare them with the roofline chart. We describe each of our evaluation methods below.

1) Time Measurements: For each machine, based on its device memory size, we run the kernels with a large grid size supported by the device memory. For `V100` and `A100`, we use a grid size of 1000^3; for `P100`, we use a grid size of 893^3; for `NVS510`, we use a grid size of 300^3. As described in III-A, the stencil needs multiple iterations to converge. For benchmarking, we use 1000 iterations for all kernels on all machines. For each execution, we warm up the kernel by running the entire execution once, and then we repeat it five times, recording the average time for the five runs.

We provided three of our fastest implementations (`gmem_8x8x8`, `smem_u`, and `st_reg_fixed_32x32`) to NVIDIA, and NVIDIA performed time measurement on both the original (with different dimension sizes for the two 3D implementations, `gmem_32x4x4`, `smem_u_32x4x4`) and optimized versions on `A100` and `V100` systems of their own.

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

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

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

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

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

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

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

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

Kernel				Machine		
Kernel Identifier	Dx	Dy	Dz	V100	P100	NVS510
gmem_8x8x8	8	8	8	53.88	117.74	415.85
gmem_16x16x4	16	16	4	85.52	195.82	760.72
gmem_32x32x1	32	32	1	292.36	639.62	2507.22
smem_u	8	8	8	57.30	76.18	210.42
smem_eta_1	8	8	8	54.87	119.15	397.56
smem_eta_3	8	8	8	54.34	117.39	396.49
st_smem_8x16	8	16	-	113.46	105.41	439.47
st_smem_16x8	16	8	-	59.92	77.91	425.73
st_smem_16x16	16	16	-	55.87	72.73	349.45
st_reg_shft_16x16	16	16	-	65.79	80.23	182.52
st_reg_shft_32x16	32	16	-	60.83	70.63	171.30
st_reg_shft_32x32	32	32	-	93.92	76.27	167.29
st_reg_fixed_16x16	16	16	-	61.66	76.10	170.03
st_reg_fixed_32x16	32	16	-	62.45	66.60	162.05
st_reg_fixed_32x32	32	32	-	58.96	61.74	160.91

TABLE II: Time Measurement on V100, P100, and NVS510

Kernel Identifier	V100	A100	Speedup
gmem_32x4x4	54.33	34.53	1.57
smem_u_32x4x4	57.90	30.96	1.87
st_reg_fixed_32x32	57.10	28.89	1.98
gmem_32x4x4 (optimized)	48.00	24.14	1.99
smem_u_32x4x4 (optimized)	54.41	25.17	2.16
st_reg_fixed_32x32 (optimized)	45.93	25.79	1.78

TABLE III: Time Measurement on NVIDIA A100 and V100

C. Results

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

Table II presents time measurements for the kernels. For 3D blockings, the columns Dx, Dy, and Dz stand for the block dimensions along the x, y, and z axes, respectively. For 2.5D blockings, only columns for Dx and Dy are reported since the z-axis is unpartitioned.

Table III presents measurements conducted by NVIDIA with both derived and optimized versions based on three of our fastest kernels on their own `A100` and `V100` systems.

Table IV presents kernel characteristics for the 25-point stencil applied to the inner data region. Characteristics for low-order stencil used in the PML region can be found in a longer technical report [42].

Table V presents the performance characteristics of our implementations on the `V100`. Figures 3a and 3b visualize these performance characteristics using the roofline performance model, showing the rooflines for L2 and DRAM, respectively. The y-axes of these figures represent performance and x-axes show arithmetic intensity. An zoomed-in version of the roofline kernel characteristics can be found in a longer technical report [42].

From our results, we offer the following observations.

3D Blocking using Global Memory: The simplest implementation `gmem_8x8x8` using only the global memory yields the best performance on V100. With L1 data cache and shared memory combined into a single unified memory block on the V100 [43], we have a much larger data cache available on

Kernel Identifier	Block Size	Grid Size	Registers Per Thread	Achieved Active Warps	Achieved Occupancy	Theoretical Active Warps	Theoretical Occupancy
gmem_8x8x8	512	1,685,159	40	42.5	66.4	48.0	75.0
gmem_16x16x4	1,024	853,200	40	28.9	45.2	32.0	50.0
gmem_32x32x1	1,024	851,400	40	29.3	45.8	32.0	50.0
smem_u	512	1,685,159	38	44.6	69.7	48.0	75.0
smem_eta_1	512	1,685,159	40	42.4	66.3	48.0	75.0
smem_eta_3	512	1,685,159	40	42.4	66.2	48.0	75.0
st_smem_8x16	128	7,140	56	27.9	43.6	28.0	43.7
st_smem_16x8	128	7,140	56	27.9	43.5	28.0	43.7
st_smem_16x16	256	3,600	56	31.6	49.4	32.0	50.0
st_reg_shft_16x16	256	3,600	96	15.9	24.9	16.0	25.0
st_reg_shft_32x16	512	1,800	96	16.0	25.0	16.0	25.0
st_reg_shft_32x32	1,024	900	64	32.0	50.0	32.0	50.0
st_ref_fixed_16x16	256	3,600	78	23.9	37.4	24.0	37.5
st_ref_fixed_32x16	512	1,800	78	16.0	25.0	16.0	25.0
st_ref_fixed_32x32	1,024	900	64	32.0	50.0	32.0	50.0

TABLE IV: Characteristics of a 25-point stencil kernel for the interior region characteristics on V100.

Kernel Identifier	FLOP ($x10^{13}$)	Achieved Performance (GFLOPs)	L2 Transactions ($x10^{12}$)	L2 Arithmetic Intensity	L2 Machine Peak Performance (GFLOPs)	L2 Achieved Percentage	DRAM Transactions ($x10^{11}$)	DRAM Arithmetic Intensity	DRAM Machine Peak Performance (GFLOPs)	DRAM Achieved Percentage
gmem_8x8x8	4.453	770	1.79	0.78	2566	30.00%	7.26	1.92	1498	51.39%
gmem_16x16x4	4.453	485	2.45	0.57	1877	25.83%	6.67	2.08	1628	29.78%
gmem_32x32x1	4.453	142	13.90	0.10	330	42.95%	6.56	2.12	1656	8.57%
smem_u	4.453	724	1.82	0.77	2531	28.60%	7.37	1.89	1474	49.11%
smem_eta_1	4.453	756	1.82	0.76	2522	29.97%	7.31	1.90	1487	50.81%
smem_eta_3	4.453	763	1.81	0.77	2535	30.10%	7.31	1.90	1488	51.30%
st_smem_8x16	4.453	366	1.47	0.95	3130	11.68%	13.30	1.05	820	44.58%
st_smem_16x8	4.453	692	1.17	1.19	3933	17.59%	7.74	1.80	1404	49.27%
st_smem_16x16	4.453	742	1.04	1.34	4414	16.81%	6.97	2.00	1560	47.58%
st_reg_shft_16x16	4.453	630	1.20	1.16	3841	16.41%	7.22	1.93	1506	41.86%
st_reg_shft_32x16	4.453	682	0.94	1.47	4861	14.02%	6.94	2.00	1566	43.54%
st_reg_shft_32x32	4.453	442	1.67	0.83	2750	16.05%	15.50	0.90	701	62.95%
st_reg_fixed_16x16	4.453	673	1.18	1.18	3899	17.25%	7.71	1.80	1409	47.72%
st_reg_fixed_32x16	4.453	664	9.12	1.53	5043	13.17%	7.14	1.95	1522	43.62%
st_reg_fixed_32x32	4.453	703	1.09	1.27	4209	16.71%	9.08	1.53	1197	58.78%

TABLE V: Kernel Performance Characteristics on V100

the V100 than on previous generations of GPUs. Therefore, when retrieving data from global memory with a good access pattern, we can achieve very good performance.

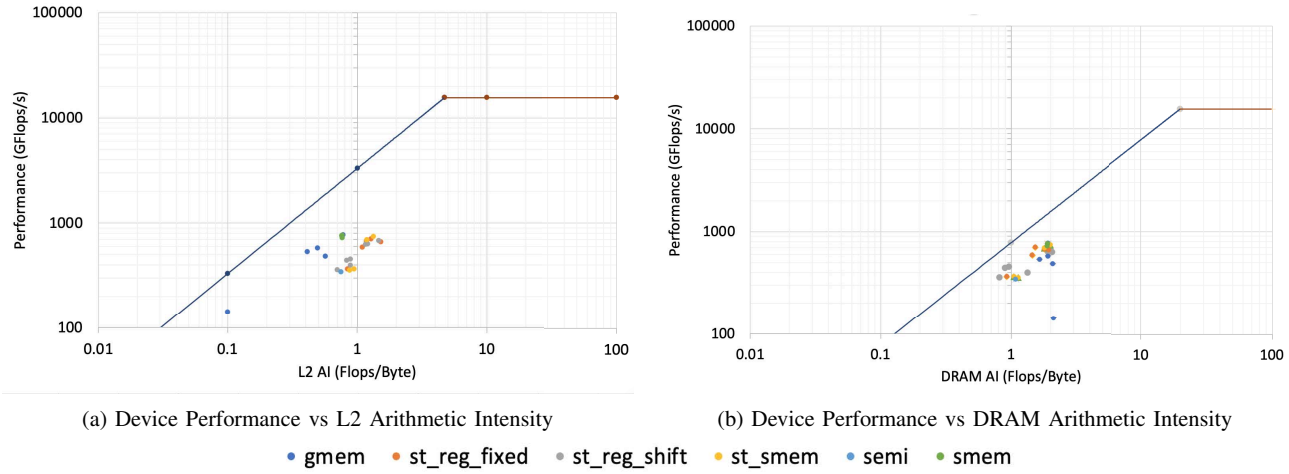
Comparing the performance of the 3D kernel across GPU generations, we notice its poor performance portability. It is one of the slowest implementations on P100 and the NVS510.

We tried several variants of the global memory implementation that differ each others in terms of block size, from smaller to larger, including `gmem_4x4x4`, `gmem_8x8x4`, `gmem_8x8x8`, `gmem_16x16x4`, and `gmem_32x32x1`. Our results show that, `gmem_8x8x8` is the best among them. We need to load all halos before performing stencil computations. For blocks smaller than `gmem_8x8x8`, such as `gmem_4x4x4` and `gmem_8x8x4`, the halo size for our 25-point stencil dominates the actual data points. Therefore, more time is spent on loading halos than points for the volume to be computed, which hurts performance. In addition, smaller block sizes also result in larger GPU grid size as we can see from Table IV, which means more kernel launches. For the smaller blocks, the additional overheads slow the overall execution. On the other hand, we also see performance degradation for larger blocks, `gmem_16x16x4` and `gmem_32x32x1`. Their larger block size results in a smaller grid size. However, both have low theoretical and achieved occupancy. Table V shows that the 3D kernels using larger blocks, especially `gmem_32x32x1`, incur more L1 cache misses, which increases the L2 transactions with a relative higher-latency loads.

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

Shared memory: Table II 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.

Recall that `smem_u` is a high-order stencil while



(a) Device Performance vs L2 Arithmetic Intensity

(b) Device Performance vs DRAM Arithmetic Intensity

Fig. 3: Roofline Performance on V100

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

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

Code Shape for 2.5D-Blockings: For implementations using 2.5D-blocking, we observe the larger the 2D plane, the better the performance. There are two main reasons for this. First, a larger 2D plane means a higher degree of concurrency. Second, with a larger plane, the percentage of halo points fetched into shared memory is smaller, which speeds up the overall performance.

In addition, our results show that `st_reg_shft_32x16` runs faster than `st_reg_shft_16x32`. From Table V, we see more L2 transactions with `st_reg_shft_16x32`, which in turn harms performance. Therefore, one should cut the plane so that the x-dimension of the GPU's thread block assigned to the innermost dimension has a relatively larger size.

Register Footprint in 2.5D-Blockings: When we evaluate `st_reg_shft_*` implementations, the variants with 2D plane size of 1024, namely `st_reg_shft_16x64`, `st_reg_shft_32x32`, and `st_reg_shft_64x16`, show poor performance on V100. The performance degradation is caused by register spilling. The maximum registers in a blockthread is $64 * 1024 = 65536$. Because we have 1024 threads for these implementations, we can only have maximum 64 registers for each thread. If we do not explicitly specify the register count to `nvcc`, it assigns 80 and 96 registers to the PML and inner kernels, respectively. Running the generated binaries for these register footprints yields incorrect results. To avoid this problem, we use compiler flag `-maxrregcount=64` to limit the maximum register usage per thread. Unfortunately 64 registers are not enough to hold all of the variables at the same time, causing register spilling. The register shifting approach exacerbates register spilling due to its high frequency of register access.

However, although register spilling occurs in the register shifting kernels, for the `st_fixed_reg_32x32` kernel, we don't see a performance degradation because the code uses fixed registers with loop unrolling. Because the registers are fixed, the frequency of register data movement is smaller than for kernels using the register shifting approach. This allows the performance impact of register spilling to be hidden by other thread activities.

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

Performance Portability: The best performing implementations on P100 and NVS510 come from 2.5D approaches. Although they are not the best kernels on V100, they are still in the fastest tier. Thus, if performance portability is a concern, implementations using 2.5D blocking, such as `st_reg_fixed_32x32`, are attractive.

Gaps to the Roofline Ceilings: Our interpretation of the performance gaps are twofold:

First, although our current implementations realize a good performance for high-order stencils with boundary conditions, we see room for further performance tuning. We could improve the arithmetic intensities by designing new GPU code shapes, e.g., by employing the semi-stencil algorithm [14], [15], which reduces data movement for high-order stencils, or employing time-skewing to increase data reuse. While all of our implementations were manually written in CUDA, we could develop a new DSL approach or build a framework that enables us to explore more sophisticated approaches.

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

NVIDIA Optimization Insights: NVIDIA ran our kernels on internal systems with either a V100 or A100 GPU and tuned our kernels to improve memory hierarchy performance. Table III shows measurements of our kernels before and after their optimizations on both V100 and A100 GPUs. Their memory hierarchy optimizations yielded speedups ranging from and 6.4% to 24.3% on a V100 and 12% to 43% on an A100. Their optimization insights were the following:

First, `cudaMallocHost` allocates page-locked memory on the host, and accelerates data copying between host and device. Our original kernel used `malloc` to support several programming models, including `OpenMP` and `OpenACC`, but it was easily adjusted to employ this platform-specific allocation strategy.

Second, NVIDIA relocated our stencil coefficients to constant memory. While accessing the coefficients in constant memory itself may not improve performance, using constant memory for stencil coefficients reduces kernel register footprints, which in some cases enables larger number of threads in a thread block, which improves warp utilization.

Third, 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 achieve this, padding can be added to an array so that the length of each row is a multiple of 128 bytes—the cache line size on NVIDIA GPUs. For our stencil, due to boundary conditions, the kernels working on the PML and inner regions can have conflicting alignment requirements. The best approach seems to be adding lead padding to the array so that the memory accessed by each of the inner tiles is aligned to cache line boundaries. This is best since most of the time and memory accesses are spent applying the high-order stencil to the the inner region.

VI. Conclusions and Future Work

This paper evaluated the performance of high-order stencils with boundary conditions on four generations of 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 largest tile size possible without exceeding GPU resource bounds. In addition, kernel implementations that compute stencils directly from global-memory, despite being the simplest, deliver the best performance on NVIDIA's V100 and A100 GPUs. We observed that 2.5D streaming algorithms deliver high performance and have the best performance portability across NVIDIA GPU generations. Also, careful alignment of tiles with cache lines improves performance.

We began our evaluation by computing 25-point stencil algorithms over the entire data domain in a single kernel launch. Inefficiency caused by branch divergence led us to apply domain decomposition and compute the boundaries separately from the center region. While this improved efficiency, having the regions separate impedes our ability to apply time skewing along the streaming dimension for the 2.5D algorithm. We plan to reintegrate boundary computations with the inner region to evaluate 3.5D algorithms on high-order stencils that apply time skewing along the streaming dimension. In addition, we plan to explore whether applying the semi-stencil algorithm [14] along the streaming dimension improves performance by reducing the memory hierarchy footprint of a block's stencil calculations and increasing the arithmetic intensity of high-order kernels.

In the near future, we will incorporate NVIDIA's optimization insights for improving cache utilization into all of our existing implementations and then compare and analyze their performance impacts on V100 and P100 machines. Once we have direct access to an NVIDIA A100, we plan to experiment with the range of kernels to assess performance portability of kernels to A100 GPUs. We will also explore the additional benefits provided by using the `float4` type.

Furthermore, we plan to expand the scope of our evaluation to explore the performance of various high-order stencil implementations on leading-edge GPUs from other vendors, as soon as we can gain access to them and results on them are not embargoed.

Acknowledgment

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 for performing the experiments on A100 and providing optimization insights. We thank Keren Zhou from Rice University for reviewing the drafts of this paper and helping us use his emerging GPU Performance Advisor tool, which offered insights for tuning some of the kernels we studied.

References

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

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

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

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

[5] 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, San Jose, California, USA, Jun. 2011, pp. 117-128.

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

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

[8] G Jin, J. Mellor-Crummey, and R. Fowler, "Increasing Temporal Locality with Skewing and Recursive Blocking," in SC01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing, Nov. 2001, pp. 57.

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

[10] Y. Song and Z. Li, "New tiling techniques to improve cache temporal locality," SIGPLAN Not., vol. 34, no. 5, pp. 215-228, May 1999.

[11] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Effective automatic parallelization of stencil computations," SIGPLAN Not., vol. 42, no. 6, pp. 235-244, Jun. 2007.

[12] 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, San Servolo Island, Venice, Italy, Jun. 2012, pp. 311-320.

[13] 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, Mar. 2013, pp. 24-31.

[14] R. de la Cruz, M. Araya-Polo, and J. M. Cela, "Introducing the Semi-stencil Algorithm," in Parallel Processing and Applied Mathematics, Berlin, Heidelberg, 2010, pp. 496-506.

[15] R. de la Cruz and M. Araya-Polo, "Algorithm 942: Semi-Stencil," ACM Trans. Math. Softw., vol. 40, no. 3, p. 23:1-23:39, Apr. 2014.

[16] 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.

[17] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, Washington, D.C., USA, Mar. 2009, pp. 79-84.

[18] 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, Feb. 2020, pp. 199-211.

[19] 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.

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

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

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

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

[24] 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.

[25] 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.

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

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

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

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

[30] R. Baghdadi et al., "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.

[31] R. Baghdadi et al., "Tiramisu: a polyhedral compiler for expressing fast and portable code," in Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization, Washington, DC, USA, Feb. 2019, pp. 193-205, Accessed: Jul. 20, 2020. [Online].

[32] 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.

[33] M. Louboutin et al., "Devito (v3.1.0): an embedded domain-specific language for finite differences and geophysical exploration," Geoscientific Model Development, vol. 12, no. 3, pp. 1165-1187, Mar. 2019.

[34] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, Seattle, Washington, USA, Jun. 2013, pp. 519-530.

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

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

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

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

[39] C. Yang, "Hierarchical Roofline Analysis on GPUs," ECP Annual Meeting, February 2020,

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

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

[42] R. Sai, J. Mellor-Crummey, X. Meng, M. Araya-Polo, and J. Meng, "Accelerating High-Order Stencils on GPUs," Sept. 2020, Accessed: Sept. 11, 2020. [Online]. Available: https://arxiv.org/abs/2009.04619

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

APPENDIX: ARTIFACT DESCRIPTION/ARTIFACT EVALUATION

SUMMARY OF THE EXPERIMENTS REPORTED

We manually wrote a collection of implementations of a 25-points star shaped stencil in CUDA along with code to apply the boundary conditions. We evaluated our implementations on several systems with generations of NVIDIA GPUs. We use a variety of tools, such as HPCToolkit, Nsight Compute, and Empirical Roofline Toolkit to evaluate our performance.

ARTIFACT AVAILABILITY

Software Artifact Availability: Some author-created software artifacts are NOT maintained in a public repository or are NOT available under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: There are no author-created data artifacts.

Proprietary Artifacts: There are associated proprietary artifacts that are not created by the authors. Some author-created artifacts are proprietary.

URL/DOI List: https://github.com/rsrice/PMBS2020-Artifact

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: IBM POWER9 + NVIDIA TESLA V100

Operating systems and versions: Red Hat Enterprise Linux Server 7.7

Compilers and versions: NVCC 10.2.89

```
CPLUS_INCLUDE_PATH=/home/USER/release-planning/openmp/openmp-install/include:/
    home/USER/staging/llvm/llvm-10.0.0-rc3/include:/usr/local/cuda-11.0/extras/
    CUPTI/include:/usr/local/cuda-11.0/include
MANPATH=/home/USER/opt/pgi/linuxpower/2019/mpi/openmpi-3.1.3/share/man:/home/USER
    /opt/pgi/linuxpower/19.10/man:/projects/spack/pkgs/linux-rhel7-ppc64le/gcc
    -4.8.5/zlib-1.2.11-4v3ticyykh2xcgw5fzltjrfuxkx7ipqh/share/man:/projects/spack/
    pkgs/linux-rhel7-ppc64le/gcc-4.8.5/gcc-7.3.0-ploqhwmybyd3wojblviucugocg3w7rdo/
    share/man:/opt/puppetlabs/puppet/share/man
XDG_SESSION_ID=16727
HOSTNAME=HOSTNAME.cs.rice.edu
SPACK_ROOT=/home/USER/spack
SELINUX_ROLE_REQUESTED=
SHELL=/bin/bash
TERM=xterm
HISTSIZE=1000
SSH_CLIENT=168.7.22.23 36669 22
LIBRARY_PATH=/home/USER/release-planning/openmp/openmp-install/lib:/home/USER/
    staging/llvm/llvm-10.0.0-rc3/libexec:/home/USER/staging/llvm/llvm-10.0.0-rc3/
    lib64:/home/USER/staging/llvm/llvm-10.0.0-rc3/lib:/usr/local/cuda-11.0/lib64:/
    projects/spack/pkgs/linux-rhel7-ppc64le/gcc-4.8.5/zlib-1.2.11-4
    v3ticyykh2xcgw5fzltjrfuxkx7ipqh/lib:/projects/spack/pkgs/linux-rhel7-ppc64le/
    gcc-4.8.5/gcc-7.3.0-ploqhwmybyd3wojblviucugocg3w7rdo/lib64:/projects/spack/
    pkgs/linux-rhel7-ppc64le/gcc-4.8.5/gcc-7.3.0-ploqhwmybyd3wojblviucugocg3w7rdo/
    lib
SELINUX_USE_CURRENT_RANGE=
QTDIR=/usr/lib64/qt-3.3
QTINC=/usr/lib64/qt-3.3/include
SSH_TTY=/dev/pts/18
QT_GRAPHICSSYSTEM_CHECKED=1
USER=USER
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd
    =40;33;01:or=40;31;01:mi=01;05;37;41:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow
    =34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arc=01;31:*.arj=01;31:*.taz
    =01;31:*.lha=01;31:*.lz4=01;31:*.lzh=01;31:*.lzma=01;31:*.tlz=01;31:*.txz
    =01;31:*.tzo=01;31:*.t7z=01;31:*.zip=01;31:*.z=01;31:*.Z=01;31:*.dz=01;31:*.gz
    =01;31:*.lrz=01;31:*.lz=01;31:*.lzo=01;31:*.xz=01;31:*.bz2=01;31:*.bz=01;31:*.
```

```
tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01;31:*.jar=01;31:*.war
=01;31:*.ear=01;31:*.sar=01;31:*.rar=01;31:*.alz=01;31:*.ace=01;31:*.zoo
=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.cab=01;31:*.jpg=01;35:*.jpeg
=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tga
=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg
=01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01;35:*.mov=01;35:*.mpg=01;35:*.mpeg
=01;35:*.m2v=01;35:*.mkv=01;35:*.webm=01;35:*.ogm=01;35:*.mp4=01;35:*.m4v
=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;35:*.wmv=01;35:*.asf
=01;35:*.rm=01;35:*.rmvb=01;35:*.flc=01;35:*.avi=01;35:*.fli=01;35:*.flv
=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xwd=01;35:*.yuv=01;35:*.cgm
=01;35:*.emf=01;35:*.axv=01;35:*.anx=01;35:*.ogv=01;35:*.ogx=01;35:*.aac
=01;36:*.au=01;36:*.flac=01;36:*.mid=01;36:*.midi=01;36:*.mka=01;36:*.mp3
=01;36:*.mpc=01;36:*.ogg=01;36:*.ra=01;36:*.wav=01;36:*.axa=01;36:*.oga
=01;36:*.spx=01;36:*.xspf=01;36:
LD_LIBRARY_PATH=/home/USER/release-planning/openmp/openmp-install/lib:/home/USER/
release-planning/hpctoolkit/hpctoolkit-install/lib/hpctoolkit/ext-libs:/home/
USER/release-planning/hpctoolkit/hpctoolkit-install/lib/hpctoolkit:/home/USER/
staging/llvm/llvm-10.0.0-rc3/libexec:/home/USER/staging/llvm/llvm-10.0.0-rc3/
lib64:/home/USER/staging/llvm/llvm-10.0.0-rc3/lib:/home/USER/opt/pgi/
linuxpower/2019/mpi/openmpi-3.1.3/lib:/home/USER/opt/pgi/linuxpower/19.10/lib
:/usr/local/cuda-11.0/lib64:/projects/spack/pkgs/linux-rhel7-ppc64le/gcc
-4.8.5/zlib-1.2.11-4v3ticyykh2xcgw5fzltjrfuxkx7ipqh/lib:/projects/spack/pkgs/
linux-rhel7-ppc64le/gcc-4.8.5/gcc-7.3.0-ploqhwmybyd3wojblviucugocg3w7rdo/lib64
:/projects/spack/pkgs/linux-rhel7-ppc64le/gcc-4.8.5/gcc-7.3.0-
ploqhwmybyd3wojblviucugocg3w7rdo/lib
CPATH=/projects/spack/pkgs/linux-rhel7-ppc64le/gcc-4.8.5/zlib-1.2.11-4
v3ticyykh2xcgw5fzltjrfuxkx7ipqh/include:/projects/spack/pkgs/linux-rhel7-
ppc64le/gcc-4.8.5/gcc-7.3.0-ploqhwmybyd3wojblviucugocg3w7rdo/include
CPP=/bin/cpp
PGI=/home/USER/opt/pgi
PATH=/home/USER/release-planning/hpctoolkit/hpctoolkit-install/libexec/hpctoolkit
:/home/USER/release-planning/hpctoolkit/hpctoolkit-install/bin:/home/USER/
staging/llvm/llvm-10.0.0-rc3/bin:/home/USER/opt/llvm/bin:/home/USER/opt/go/bin
:/home/USER/usr/local/bin:/home/USER/cs-roofline-toolkit/
Empirical_Roofline_Tool-1.1.0:/home/USER/spack/bin:/home/USER/opt/pgi/
linuxpower/2019/mpi/openmpi-3.1.3/bin:/home/USER/opt/pgi/linuxpower/19.10/bin
:/usr/local/cuda-11.0/bin:/projects/spack/pkgs/linux-rhel7-ppc64le/gcc-4.8.5/
cmake-3.12.4-o5rgr6d6nl7hs746vk5ljmmr4wffmgcw/bin:/projects/spack/pkgs/linux-
rhel7-ppc64le/gcc-4.8.5/gcc-7.3.0-ploqhwmybyd3wojblviucugocg3w7rdo/bin:/usr/
lib64/qt-3.3/bin:/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/opt/
puppetlabs/bin:/home/USER/.local/bin:/home/USER/bin
MAIL=/var/spool/mail/USER
_=/usr/bin/env
C_INCLUDE_PATH=/home/USER/release-planning/openmp/openmp-install/include:/home/
USER/staging/llvm/llvm-10.0.0-rc3/include:
PWD=/home/USER/Author-Kit
F90=/home/USER/opt/pgi/linuxpower/19.10/bin/pgfortran
_LMFILES_=/projects/spack/modules/linux-rhel7-ppc64le/gcc-7.3.0-gcc-4.8.5-ploqhwm
:/projects/spack/modules/linux-rhel7-ppc64le/cmake-3.12.4-gcc-4.8.5-o5rgr6d:/
projects/spack/modules/linux-rhel7-ppc64le/zlib-1.2.11-gcc-4.8.5-4v3ticy:/usr/
local/modules//cuda/11.0:/home/USER/opt/pgi/modulefiles/pgi/19.10:/home/USER/
opt/pgi/modulefiles/openmpi/3.1.3/2019:/home/USER/modules/hpctoolkit:/home/
USER/modules/openmp
CUDA_VISIBLE_DEVICES=3
LANG=en_US.UTF-8
```

```
MODULEPATH=/home/USER/modules:/home/USER/spack/share/spack/modules/linux-rhel7-
    power9le:/home/USER/opt/pgi/modulefiles:/usr/local/modules:/projects/spack/
    modules/linux-rhel7-ppc64le:/usr/share/Modules/modulefiles:/etc/modulefiles
LOADEDMODULES=gcc-7.3.0-gcc-4.8.5-ploqhwm:cmake-3.12.4-gcc-4.8.5-o5rgr6d:zlib
    -1.2.11-gcc-4.8.5-4v3ticy:cuda/11.0:pgi/19.10:openmpi/3.1.3/2019:hpctoolkit:
    openmp
SELINUX_LEVEL_REQUESTED=
F77=/home/USER/opt/pgi/linuxpower/19.10/bin/pgfortran
HISTCONTROL=ignoredups
CXX=/home/USER/opt/pgi/linuxpower/19.10/bin/pgc++
TOT_C=/home/USER/HPC_ProgrammingModels/code/Minimig_C
HOME=/home/USER
SHLVL=2
FC=/home/USER/opt/pgi/linuxpower/19.10/bin/pgfortran
LOGNAME=USER
QTLIB=/usr/lib64/qt-3.3/lib
SSH_CONNECTION=168.7.22.23 36669 128.42.128.30 22
XDG_DATA_DIRS=/home/USER/.local/share/flatpak/exports/share:/var/lib/flatpak/
    exports/share:/usr/local/share:/usr/share
MODULESHOME=/usr/share/Modules
LESSOPEN=||/usr/bin/lesspipe.sh %s
PKG_CONFIG_PATH=/projects/spack/pkgs/linux-rhel7-ppc64le/gcc-4.8.5/zlib-1.2.11-4
    v3ticyykh2xcgw5fzltjrfuxkx7ipqh/lib/pkgconfig
ACLOCAL_PATH=/projects/spack/pkgs/linux-rhel7-ppc64le/gcc-4.8.5/cmake-3.12.4-
    o5rgr6d6nl7hs746vk5ljmmr4wffmgcw/share/aclocal
XDG_RUNTIME_DIR=/run/user/14742
CC=/home/USER/opt/pgi/linuxpower/19.10/bin/pgcc
CMAKE_PREFIX_PATH=/projects/spack/pkgs/linux-rhel7-ppc64le/gcc-4.8.5/zlib
    -1.2.11-4v3ticyykh2xcgw5fzltjrfuxkx7ipqh:/projects/spack/pkgs/linux-rhel7-
    ppc64le/gcc-4.8.5/cmake-3.12.4-o5rgr6d6nl7hs746vk5ljmmr4wffmgcw:/projects/
    spack/pkgs/linux-rhel7-ppc64le/gcc-4.8.5/gcc-7.3.0-
    ploqhwmybyd3wojblviucugocg3w7rdo/
BASH_FUNC_module()=() {  eval `/usr/bin/modulecmd bash $*`
}
BASH_FUNC_spack()=() {  if [ -n "${LD_LIBRARY_PATH-}" ]; then
 export SPACK_LD_LIBRARY_PATH=$LD_LIBRARY_PATH;
 fi;
 if [ -n "${DYLD_LIBRARY_PATH-}" ]; then
 export SPACK_DYLD_LIBRARY_PATH=$DYLD_LIBRARY_PATH;
 fi;
 if [ -n "${ZSH_VERSION:-}" ]; then
 emulate -L sh;
 fi;
 _sp_flags="";
 while [ ! -z ${1+x} ] && [ "${1#-}" != "${1}" ]; do
 _sp_flags="$_sp_flags $1";
 shift;
 done;
 if [ -n "$_sp_flags" ] && [ "${_sp_flags#*h}" != "${_sp_flags}" ] || [ "${
     _sp_flags#*V}" != "${_sp_flags}" ]; then
 command spack $_sp_flags "$@";
 return;
 fi;
 _sp_subcommand="";
 if [ ! -z ${1+x} ]; then
 _sp_subcommand="$1";
```

```
shift;
fi;
case $_sp_subcommand in
"cd")
_sp_arg="";
if [ -n "$1" ]; then
_sp_arg="$1";
shift;
fi;
if [ "$_sp_arg" = "-h" ] || [ "$_sp_arg" = "--help" ]; then
command spack cd -h;
else
LOC="$(spack location $_sp_arg "$@")";
if [ -d "$LOC" ]; then
cd "$LOC";
else
return 1;
fi;
fi;
return
;;
"env")
_sp_arg="";
if [ -n "$1" ]; then
_sp_arg="$1";
shift;
fi;
if [ "$_sp_arg" = "-h" ] || [ "$_sp_arg" = "--help" ]; then
command spack env -h;
else
case $_sp_arg in
activate)
_a="$@";
if [ -z ${1+x} ] || [ "${_a#*--sh}" != "$_a" ] || [ "${_a#*--csh}" != "$_a" ] ||
    [ "${_a#*-h}" != "$_a" ]; then
command spack env activate "$@";
else
eval $(command spack $_sp_flags env activate --sh "$@");
fi
;;
deactivate)
_a="$@";
if [ "${_a#*--sh}" != "$_a" ] || [ "${_a#*--csh}" != "$_a" ]; then
command spack env deactivate "$@";
else
if [ -n "$*" ]; then
command spack env deactivate -h;
else
eval $(command spack $_sp_flags env deactivate --sh);
fi;
fi
;;
*)
command spack env $_sp_arg "$@"
;;
esac;
```

```
 fi;
 return
 ;;
 "load" | "unload")
 _a=" $@";
 if [ "${_a#* --sh}" != "$_a" ] || [ "${_a#* --csh}" != "$_a" ] || [ "${_a#* -h}"
      != "$_a" ] || [ "${_a#* --help}" != "$_a" ]; then
 command spack $_sp_flags $_sp_subcommand "$@";
 else
 eval $(command spack $_sp_flags $_sp_subcommand --sh "$@" ||
                         echo "return 1");
 fi
 ;;
 *)
 command spack $_sp_flags $_sp_subcommand "$@"
 ;;
 esac
}
+ lsb_release -a
LSB Version:        :core-4.1-noarch:core-4.1-ppc64le:cxx-4.1-noarch:cxx-4.1-
    ppc64le:desktop-4.1-noarch:desktop-4.1-ppc64le:languages-4.1-noarch:languages
    -4.1-ppc64le:printing-4.1-noarch:printing-4.1-ppc64le
Distributor ID:     RedHatEnterpriseServer
Description:        Red Hat Enterprise Linux Server release 7.7 (Maipo)
Release:    7.7
Codename:   Maipo
+ uname -a
Linux HOSTNAME.cs.rice.edu 4.14.0-115.16.1.el7a.ppc64le #1 SMP Wed Nov 27
    18:01:42 UTC 2019 ppc64le ppc64le ppc64le GNU/Linux
+ lscpu
Architecture:          ppc64le
Byte Order:            Little Endian
CPU(s):                160
On-line CPU(s) list:   0-159
Thread(s) per core:    4
Core(s) per socket:    20
Socket(s):             2
NUMA node(s):          6
Model:                 2.2 (pvr 004e 1202)
Model name:            POWER9, altivec supported
CPU max MHz:           3800.0000
CPU min MHz:           2300.0000
L1d cache:             32K
L1i cache:             32K
L2 cache:              512K
L3 cache:              10240K
NUMA node0 CPU(s):     0-79
NUMA node8 CPU(s):     80-159
NUMA node252 CPU(s):
NUMA node253 CPU(s):
NUMA node254 CPU(s):
NUMA node255 CPU(s):
+ cat /proc/meminfo
MemTotal:       261819328 kB
MemFree:        173085248 kB
MemAvailable:   254032704 kB
```

```
Buffers:         2361536 kB
Cached:         75190400 kB
SwapCached:        24192 kB
Active:         49405056 kB
Inactive:       29230144 kB
Active(anon):     388480 kB
Inactive(anon):  1437952 kB
Active(file):   49016576 kB
Inactive(file): 27792192 kB
Unevictable:           0 kB
Mlocked:               0 kB
SwapTotal:       1540032 kB
SwapFree:        1395008 kB
Dirty:                 0 kB
Writeback:             0 kB
AnonPages:       1061376 kB
Mapped:           484672 kB
Shmem:            743104 kB
Slab:            8112832 kB
SReclaimable:    5321472 kB
SUnreclaim:      2791360 kB
KernelStack:       28208 kB
PageTables:        21568 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:    132449664 kB
Committed_AS:     3442432 kB
VmallocTotal:   549755813888 kB
VmallocUsed:           0 kB
VmallocChunk:          0 kB
HardwareCorrupted:     0 kB
AnonHugePages:         0 kB
ShmemHugePages:        0 kB
ShmemPmdMapped:        0 kB
CmaTotal:        6717440 kB
CmaFree:         5500096 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
+ inxi -F -c0
./collect_environment.sh: line 14: inxi: command not found
+ lsblk -a
NAME                    MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sdb                       8:16   1  1.8T  0 disk
sda                       8:0    1  1.8T  0 disk
|-sda2                    8:2    1  500M  0 part /boot
|-sda3                    8:3    1  1.8T  0 part
| |-VolGroup00-lv_swap 253:1    0  1.5G  0 lvm  [SWAP]
| |-VolGroup00-lv_root 253:0    0  1.8T  0 lvm  /
|-sda1                    8:1    1    4M  0 part
+ lsscsi -s
[0:0:0:0]    disk    ATA       ST2000NX0253     BE35  /dev/sda    2.00TB
[1:0:0:0]    disk    ATA       ST2000NX0253     BE35  /dev/sdb    2.00TB
```

```
+ module list
++ /usr/bin/modulecmd bash list
Currently Loaded Modulefiles:
  1) gcc-7.3.0-gcc-4.8.5-ploqhwm       5) pgi/19.10
  2) cmake-3.12.4-gcc-4.8.5-o5rgr6d    6) openmpi/3.1.3/2019
  3) zlib-1.2.11-gcc-4.8.5-4v3ticy     7) hpctoolkit
  4) /cuda/11.0                        8) openmp
+ eval
+ nvidia-smi
Mon Sep 14 21:32:22 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 450.51.05    Driver Version: 450.51.05    CUDA Version: 11.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla V100-SXM2...  On   | 00000004:04:00.0 Off |                    0 |
| N/A   34C    P0    54W / 300W |    628MiB / 32510MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  Tesla V100-SXM2...  On   | 00000004:05:00.0 Off |                    0 |
| N/A   36C    P0    41W / 300W |      2MiB / 32510MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   2  Tesla V100-SXM2...  On   | 00000035:03:00.0 Off |                    0 |
| N/A   31C    P0    40W / 300W |      2MiB / 32510MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   3  Tesla V100-SXM2...  On   | 00000035:04:00.0 Off |                    0 |
| N/A   37C    P0    40W / 300W |      2MiB / 32510MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A     38347      C   ./main                           313MiB |
|    0   N/A  N/A    161396      C   ./main                           313MiB |
+-----------------------------------------------------------------------------+
+ cat
+ lshw -short -quiet -sanitize
WARNING: you should run this program as super-user.
H/W path        Device      Class      Description
============================================
                            system     8335-GTH (ibm,witherspoon)
/0                          bus        Motherboard
/0/1                        processor  02CY210
/0/1/0                      memory     32KiB L1 Cache (instruction)
/0/1/1                      memory     32KiB L1 Cache (data)
/0/1/2                      memory     512KiB L2 Cache (unified)
/0/1/3                      memory     10MiB L3 Cache (unified)
/0/16                       processor  02CY210
/0/16/0                     memory     32KiB L1 Cache (instruction)
```

```
/0/16/1                 memory      32KiB L1 Cache (data)
/0/16/2                 memory      512KiB L2 Cache (unified)
/0/16/3                 memory      10MiB L3 Cache (unified)
/0/20                   processor   02CY210
/0/20/0                 memory      32KiB L1 Cache (instruction)
/0/20/1                 memory      32KiB L1 Cache (data)
/0/20/2                 memory      512KiB L2 Cache (unified)
/0/20/3                 memory      10MiB L3 Cache (unified)
/0/24                   processor   02CY210
/0/24/0                 memory      32KiB L1 Cache (instruction)
/0/24/1                 memory      32KiB L1 Cache (data)
/0/24/2                 memory      512KiB L2 Cache (unified)
/0/24/3                 memory      10MiB L3 Cache (unified)
/0/28                   processor   02CY210
/0/28/0                 memory      32KiB L1 Cache (instruction)
/0/28/1                 memory      32KiB L1 Cache (data)
/0/28/2                 memory      512KiB L2 Cache (unified)
/0/28/3                 memory      10MiB L3 Cache (unified)
/0/40                   processor   02CY210
/0/40/0                 memory      32KiB L1 Cache (instruction)
/0/40/1                 memory      32KiB L1 Cache (data)
/0/40/2                 memory      512KiB L2 Cache (unified)
/0/40/3                 memory      10MiB L3 Cache (unified)
/0/44                   processor   02CY210
/0/44/0                 memory      32KiB L1 Cache (instruction)
/0/44/1                 memory      32KiB L1 Cache (data)
/0/44/2                 memory      512KiB L2 Cache (unified)
/0/44/3                 memory      10MiB L3 Cache (unified)
/0/48                   processor   02CY210
/0/48/0                 memory      32KiB L1 Cache (instruction)
/0/48/1                 memory      32KiB L1 Cache (data)
/0/48/2                 memory      512KiB L2 Cache (unified)
/0/48/3                 memory      10MiB L3 Cache (unified)
/0/52                   processor   02CY210
/0/52/0                 memory      32KiB L1 Cache (instruction)
/0/52/1                 memory      32KiB L1 Cache (data)
/0/52/2                 memory      512KiB L2 Cache (unified)
/0/52/3                 memory      10MiB L3 Cache (unified)
/0/56                   processor   02CY210
/0/56/0                 memory      32KiB L1 Cache (instruction)
/0/56/1                 memory      32KiB L1 Cache (data)
/0/56/2                 memory      512KiB L2 Cache (unified)
/0/56/3                 memory      10MiB L3 Cache (unified)
/0/60                   processor   02CY210
/0/60/0                 memory      32KiB L1 Cache (instruction)
/0/60/1                 memory      32KiB L1 Cache (data)
/0/60/2                 memory      512KiB L2 Cache (unified)
/0/60/3                 memory      10MiB L3 Cache (unified)
/0/4                    processor   02CY210
/0/4/0                  memory      32KiB L1 Cache (instruction)
/0/4/1                  memory      32KiB L1 Cache (data)
/0/4/2                  memory      512KiB L2 Cache (unified)
/0/4/3                  memory      10MiB L3 Cache (unified)
/0/64                   processor   02CY210
/0/64/0                 memory      32KiB L1 Cache (instruction)
/0/64/1                 memory      32KiB L1 Cache (data)
```

```
/0/64/2                 memory      512KiB L2 Cache (unified)
/0/64/3                 memory      10MiB L3 Cache (unified)
/0/68                   processor   02CY210
/0/68/0                 memory      32KiB L1 Cache (instruction)
/0/68/1                 memory      32KiB L1 Cache (data)
/0/68/2                 memory      512KiB L2 Cache (unified)
/0/68/3                 memory      10MiB L3 Cache (unified)
/0/72                   processor   02CY210
/0/72/0                 memory      32KiB L1 Cache (instruction)
/0/72/1                 memory      32KiB L1 Cache (data)
/0/72/2                 memory      512KiB L2 Cache (unified)
/0/72/3                 memory      10MiB L3 Cache (unified)
/0/76                   processor   02CY210
/0/76/0                 memory      32KiB L1 Cache (instruction)
/0/76/1                 memory      32KiB L1 Cache (data)
/0/76/2                 memory      512KiB L2 Cache (unified)
/0/76/3                 memory      10MiB L3 Cache (unified)
/0/80                   processor   02CY210
/0/80/0                 memory      32KiB L1 Cache (instruction)
/0/80/1                 memory      32KiB L1 Cache (data)
/0/80/2                 memory      512KiB L2 Cache (unified)
/0/80/3                 memory      10MiB L3 Cache (unified)
/0/84                   processor   02CY210
/0/84/0                 memory      32KiB L1 Cache (instruction)
/0/84/1                 memory      32KiB L1 Cache (data)
/0/84/2                 memory      512KiB L2 Cache (unified)
/0/84/3                 memory      10MiB L3 Cache (unified)
/0/88                   processor   02CY210
/0/88/0                 memory      32KiB L1 Cache (instruction)
/0/88/1                 memory      32KiB L1 Cache (data)
/0/88/2                 memory      512KiB L2 Cache (unified)
/0/88/3                 memory      10MiB L3 Cache (unified)
/0/92                   processor   02CY210
/0/92/0                 memory      32KiB L1 Cache (instruction)
/0/92/1                 memory      32KiB L1 Cache (data)
/0/92/2                 memory      512KiB L2 Cache (unified)
/0/92/3                 memory      10MiB L3 Cache (unified)
/0/2048                 processor   02CY210
/0/2048/0               memory      32KiB L1 Cache (instruction)
/0/2048/1               memory      32KiB L1 Cache (data)
/0/2048/2               memory      512KiB L2 Cache (unified)
/0/2048/3               memory      10MiB L3 Cache (unified)
/0/2052                 processor   02CY210
/0/2052/0               memory      32KiB L1 Cache (instruction)
/0/2052/1               memory      32KiB L1 Cache (data)
/0/2052/2               memory      512KiB L2 Cache (unified)
/0/2052/3               memory      10MiB L3 Cache (unified)
/0/2056                 processor   02CY210
/0/2056/0               memory      32KiB L1 Cache (instruction)
/0/2056/1               memory      32KiB L1 Cache (data)
/0/2056/2               memory      512KiB L2 Cache (unified)
/0/2056/3               memory      10MiB L3 Cache (unified)
/0/2060                 processor   02CY210
/0/2060/0               memory      32KiB L1 Cache (instruction)
/0/2060/1               memory      32KiB L1 Cache (data)
/0/2060/2               memory      512KiB L2 Cache (unified)
```

```
/0/2060/3               memory      10MiB L3 Cache (unified)
/0/2064                 processor   02CY210
/0/2064/0               memory      32KiB L1 Cache (instruction)
/0/2064/1               memory      32KiB L1 Cache (data)
/0/2064/2               memory      512KiB L2 Cache (unified)
/0/2064/3               memory      10MiB L3 Cache (unified)
/0/2068                 processor   02CY210
/0/2068/0               memory      32KiB L1 Cache (instruction)
/0/2068/1               memory      32KiB L1 Cache (data)
/0/2068/2               memory      512KiB L2 Cache (unified)
/0/2068/3               memory      10MiB L3 Cache (unified)
/0/2072                 processor   02CY210
/0/2072/0               memory      32KiB L1 Cache (instruction)
/0/2072/1               memory      32KiB L1 Cache (data)
/0/2072/2               memory      512KiB L2 Cache (unified)
/0/2072/3               memory      10MiB L3 Cache (unified)
/0/2076                 processor   02CY210
/0/2076/0               memory      32KiB L1 Cache (instruction)
/0/2076/1               memory      32KiB L1 Cache (data)
/0/2076/2               memory      512KiB L2 Cache (unified)
/0/2076/3               memory      10MiB L3 Cache (unified)
/0/2096                 processor   02CY210
/0/2096/0               memory      32KiB L1 Cache (instruction)
/0/2096/1               memory      32KiB L1 Cache (data)
/0/2096/2               memory      512KiB L2 Cache (unified)
/0/2096/3               memory      10MiB L3 Cache (unified)
/0/2100                 processor   02CY210
/0/2100/0               memory      32KiB L1 Cache (instruction)
/0/2100/1               memory      32KiB L1 Cache (data)
/0/2100/2               memory      512KiB L2 Cache (unified)
/0/2100/3               memory      10MiB L3 Cache (unified)
/0/2104                 processor   02CY210
/0/2104/0               memory      32KiB L1 Cache (instruction)
/0/2104/1               memory      32KiB L1 Cache (data)
/0/2104/2               memory      512KiB L2 Cache (unified)
/0/2104/3               memory      10MiB L3 Cache (unified)
/0/2108                 processor   02CY210
/0/2108/0               memory      32KiB L1 Cache (instruction)
/0/2108/1               memory      32KiB L1 Cache (data)
/0/2108/2               memory      512KiB L2 Cache (unified)
/0/2108/3               memory      10MiB L3 Cache (unified)
/0/2112                 processor   02CY210
/0/2112/0               memory      32KiB L1 Cache (instruction)
/0/2112/1               memory      32KiB L1 Cache (data)
/0/2112/2               memory      512KiB L2 Cache (unified)
/0/2112/3               memory      10MiB L3 Cache (unified)
/0/2116                 processor   02CY210
/0/2116/0               memory      32KiB L1 Cache (instruction)
/0/2116/1               memory      32KiB L1 Cache (data)
/0/2116/2               memory      512KiB L2 Cache (unified)
/0/2116/3               memory      10MiB L3 Cache (unified)
/0/2120                 processor   02CY210
/0/2120/0               memory      32KiB L1 Cache (instruction)
/0/2120/1               memory      32KiB L1 Cache (data)
/0/2120/2               memory      512KiB L2 Cache (unified)
/0/2120/3               memory      10MiB L3 Cache (unified)
```

```
/0/2124                 processor   02CY210
/0/2124/0               memory      32KiB L1 Cache (instruction)
/0/2124/1               memory      32KiB L1 Cache (data)
/0/2124/2               memory      512KiB L2 Cache (unified)
/0/2124/3               memory      10MiB L3 Cache (unified)
/0/2128                 processor   02CY210
/0/2128/0               memory      32KiB L1 Cache (instruction)
/0/2128/1               memory      32KiB L1 Cache (data)
/0/2128/2               memory      512KiB L2 Cache (unified)
/0/2128/3               memory      10MiB L3 Cache (unified)
/0/2132                 processor   02CY210
/0/2132/0               memory      32KiB L1 Cache (instruction)
/0/2132/1               memory      32KiB L1 Cache (data)
/0/2132/2               memory      512KiB L2 Cache (unified)
/0/2132/3               memory      10MiB L3 Cache (unified)
/0/2136                 processor   02CY210
/0/2136/0               memory      32KiB L1 Cache (instruction)
/0/2136/1               memory      32KiB L1 Cache (data)
/0/2136/2               memory      512KiB L2 Cache (unified)
/0/2136/3               memory      10MiB L3 Cache (unified)
/0/2140                 processor   02CY210
/0/2140/0               memory      32KiB L1 Cache (instruction)
/0/2140/1               memory      32KiB L1 Cache (data)
/0/2140/2               memory      512KiB L2 Cache (unified)
/0/2140/3               memory      10MiB L3 Cache (unified)
/0/2                    memory      254GiB System memory
/0/2/0                  memory      8GiB RDIMM DDR4 2666 MHz (0.4ns)
/0/2/1                  memory      8GiB RDIMM DDR4 2666 MHz (0.4ns)
/0/2/2                  memory      8GiB RDIMM DDR4 2666 MHz (0.4ns)
/0/2/3                  memory      8GiB RDIMM DDR4 2666 MHz (0.4ns)
/0/2/4                  memory      8GiB RDIMM DDR4 2666 MHz (0.4ns)
/0/2/5                  memory      8GiB RDIMM DDR4 2666 MHz (0.4ns)
/0/2/6                  memory      8GiB RDIMM DDR4 2666 MHz (0.4ns)
/0/2/7                  memory      8GiB RDIMM DDR4 2666 MHz (0.4ns)
/0/2/8                  memory      8GiB RDIMM DDR4 2666 MHz (0.4ns)
/0/2/9                  memory      8GiB RDIMM DDR4 2666 MHz (0.4ns)
/0/2/a                  memory      8GiB RDIMM DDR4 2666 MHz (0.4ns)
/0/2/b                  memory      8GiB RDIMM DDR4 2666 MHz (0.4ns)
/0/2/c                  memory      8GiB RDIMM DDR4 2666 MHz (0.4ns)
/0/2/d                  memory      8GiB RDIMM DDR4 2666 MHz (0.4ns)
/0/2/e                  memory      8GiB RDIMM DDR4 2666 MHz (0.4ns)
/0/2/f                  memory      8GiB RDIMM DDR4 2666 MHz (0.4ns)
/0/3                    generic     bmc-firmware-version
/0/5                    generic     buildroot
/0/6                    generic     capp-ucode
/0/7                    generic     hcode
/0/8                    generic     hostboot
/0/9                    generic     hostboot-binaries
/0/a                    generic     linux
/0/b                    generic     machine-xml
/0/c                    generic     occ
/0/d                    generic     petitboot
/0/e                    generic     sbe
/0/f                    generic     skiboot
/0/10                   generic     version
/0/100                  bridge      POWER9 Host Bridge (PHB4)
```

```
/0/101                  bridge     POWER9 Host Bridge (PHB4)
/0/101/0                bus        TUSB73x0 SuperSpeed USB 3.0 xHCI Host
    Controller
/0/102                  bridge     POWER9 Host Bridge (PHB4)
/0/102/0                bridge     AST1150 PCI-to-PCI Bridge
/0/102/0/0              display    ASPEED Graphics Family
/0/103                  bridge     POWER9 Host Bridge (PHB4)
/0/104                  bridge     POWER9 Host Bridge (PHB4)
/0/104/0                bridge     PLX Technology, Inc.
/0/104/0/2              bridge     PLX Technology, Inc.
/0/104/0/2/0            storage    88SE9235 PCIe 2.0 x2 4-port SATA 6 Gb/s
    Controller
/0/104/0/a              bridge     PLX Technology, Inc.
/0/104/0/a/0            display    GV100GL [Tesla V100 SXM2 32GB]
/0/104/0/b              bridge     PLX Technology, Inc.
/0/104/0/b/0            display    GV100GL [Tesla V100 SXM2 32GB]
/0/104/0/c              bridge     PLX Technology, Inc.
/0/104/0.1              generic    PLX Technology, Inc.
/0/104/0.2              generic    PLX Technology, Inc.
/0/104/0.3              generic    PLX Technology, Inc.
/0/104/0.4              generic    PLX Technology, Inc.
/0/105                  bridge     POWER9 Host Bridge (PHB4)
/0/105/0      enP5p1s0f0  network    NetXtreme BCM5719 Gigabit Ethernet PCIe
/0/105/0.1    enP5p1s0f1  network    NetXtreme BCM5719 Gigabit Ethernet PCIe
/0/106                  bridge     IBM
/0/107                  bridge     IBM
/0/108                  bridge     IBM
/0/109                  bridge     IBM
/0/10a                  bridge     IBM
/0/10b                  bridge     IBM
/0/10c                  bridge     IBM
/0/10d                  bridge     IBM
/0/10e                  bridge     IBM
/0/10f                  bridge     IBM
/0/110                  bridge     IBM
/0/111                  bridge     IBM
/0/112                  bridge     POWER9 Host Bridge (PHB4)
/0/113                  bridge     POWER9 Host Bridge (PHB4)
/0/114                  bridge     POWER9 Host Bridge (PHB4)
/0/0                    bridge     POWER9 Host Bridge (PHB4)
/0/0/0                  bridge     PLX Technology, Inc.
/0/0/0/4                bridge     PLX Technology, Inc.
/0/0/0/4/0              display    GV100GL [Tesla V100 SXM2 32GB]
/0/0/0/5                bridge     PLX Technology, Inc.
/0/0/0/5/0              display    GV100GL [Tesla V100 SXM2 32GB]
/0/0/0/d                bridge     PLX Technology, Inc.
/1            bond0     network    Ethernet interface
/2            br0       network    Ethernet interface
WARNING: output may be incomplete or inaccurate, you should run this program as
    super-user.
```