High-Performance, Scalable Geometric Multigrid via Fine-Grain Data Blocking for GPUs

Oscar Antepara, Samuel Williams, Hans Johansen
Lawrence Berkeley National Laboratory
Berkeley, California, USA
{oantepara, swwilliams, hjohansen}@lbl.gov

Mary Hall
University of Utah
Salt Lake City, Utah, USA
mhall@cs.utah.edu

***Abstract*—We present a performance study of geometric multigrid (GMG) on NVIDIA, AMD, and Intel GPU-accelerated supercomputers. The approach employs fine-grain data blocking in BrickLib, which reduces data movement in the GMG V-cycle by optimizing storage order for stencil access and communication. Our GMG attains 73% in a peak performance portability metric, and 87% parallel efficiency when weak scaling to 512 GPUs on all three GPU-accelerated supercomputers. Analysis shows stencil performance and MPI communication is well-correlated with a traditional linear model from which we can extract empirical latency, overhead, bandwidth, and throughput for comparison to theoretical GPU and network limits. Observations show NVIDIA GPUs provide the lowest overhead and highest throughput per process with AMD and Intel GPUs delivering comparable performance. Conversely, despite all three platforms employing the same Slingshot network, sustained bandwidth and latency vary widely when each GPU is dedicated one NIC.**

***Index Terms*—GPU, performance portability, multigrid, data blocking, latency, network**

I. Introduction

Linear solvers are widely used in scientific computing and engineering applications as part of the numerical solution for partial differential equations (PDEs) in the form $A\,x = b$. Geometric multigrid (GMG) [1] is a matrix-free approach, with the main advantage that simple stencil calculations for the linear operator have the potential to be computationally fast and efficient, with near-linear scaling.

With ongoing trends towards heterogeneous architectures, multigrid implementations on CPUs/GPUs are relevant due to their wide use in scientific computing. Multigrid uses a nested hierarchy of meshes in a *V-cycle*, where error on finer meshes is reduced using approximate solutions on coarser meshes, through a series of *restriction, smoothing, and interpolation* operations recursively applied on all meshes. As a result, porting multigrid to new emerging supercomputer architectures and maximizing performance continue to present unique challenges, with more GPUs per node, and different ratios of FLOPs, memory and network bandwidth. The hierarchical stencil operations lead to complex, memory- and communication-bound performance, especially deep in the multigrid cycle where the problem size becomes smaller and performance is limited by data movement, communication, and any associated latency or overheads.

In this paper, we optimize memory and on-node communication-related data movement using BrickLib [2], to provide fine-grain data blocking [3], [4]. In Bricklib, lexicographical `ijk` data layouts are comprised of smaller *bricks* (e.g. 8^3) of contiguous data. When coupled with a vector code generator, BrickLib has demonstrated significant reduction in data movement and increased arithmetic intensity for stencil application as compared to tiled implementations [5]. Moreover, bricks can be stored in an optimized physical ordering to avoid on-node packing for communication [6].

Prior work in BrickLib focused on optimizing just stencil application [7]; in this paper we use bricks for all operations in the multigrid V-cycle, including inter-layer interpolation and restriction. These optimizations enable bricks to achieve data reuse across neighboring bricks in 3 dimensions of a grid, as well as across levels in multigrid. We additionally reduce communication costs by exploiting message aggregation across multiple smoothing operations, using GPU-Aware MPI and optimized CPU-GPU-NIC bindings, and improving NIC communication performance for small messages.

The performance and performance-portability of the GMG implementation is assessed on NVIDIA, AMD and Intel GPUs on Perlmutter-NERSC, Frontier-OLCF, and Sunspot-ALCF. We introduce performance models to evaluate latency, throughput, and bandwidth for computation and communication throughout the multigrid V-cycle and across architectures. Our performance portability evaluation is rigorously based on fraction of a *roofline model* [8] theoretical arithmetic intensity, to compare computation kernels across different GPUs.

This paper makes the following unique contributions:

- To the best of our knowledge, it is the first GMG performance analysis on the three most recent DOE GPU-accelerated supercomputers.
- Our performance portability assessment includes a GMG implementation in SYCL for INTEL GPUs, and compares it to both NVIDIA and AMD GPUs.
- We present performance results and models for GMG computation and communication on all multigrid levels.
- GMG attains better than 73% of the theoretical on-node peak performance portability metric [9], and 87% parallel efficiency when weak scaling to 512 GPUs on all three GPU-accelerated supercomputers.

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00159

II. Related Work

For structured grid applications, stencil computations are the core of the geometric multigrid algorithm. The literature for stencil optimizations is extensive (an incomplete list might include [4], [10]–[13]), with block and tiling optimizations often used to improve cache reuse on both CPUs and GPUs. However, as new supercomputer architectures evolve with higher peak multi-node performance, the growing gap between DRAM bandwidth and floating point operations (FLOPs) has focused research on increasing performance through data locality, particularly for bandwidth-bound computations.

Other efforts have introduced MG frameworks that could efficiently exploit resources on GPU architectures. Examples of recent multiphysics applications, where multigrid solvers are key for elliptic equations, on structured or unstructured grids include [14]–[17]. Many of these efforts have been focused on GPU acceleration and, to some extent performance portability, since the focus is on different supercomputers with different GPU vendors. However, because of the complexity of multigrid solvers, it is an ongoing challenge to do detailed performance analysis and explore optimization techniques that exploit increasing GPU parallelism, GPU memory, and network bandwidth.

Performance analysis and optimization for geometric multigrid on many core and GPU architectures include [18]–[23]; in most cases, platform-specific approaches were used for GPUs, such as blocking optimizations, auto-tuned approaches, and methods to reduce communication costs. Clearly, techniques tailored to certain programming models or GPUs are limited, indicating a need for more performance portability efforts as supercomputers continue to diversify system architectures.

Additional work has explored MG optimizations for smoother operations [24]–[26] or communication optimizations [27], by comparing performance improvements against GPU bandwidth or enabling performance on a specific architecture or programming model. In [28], the authors examined speedups for mixed precision implementations for AMG on the most recent GPU devices, such as NVIDIA H100, AMD MI250X, and INTEL PVC. However, their focus was on the speedups enabled by mixed precision in an iterative refinement strategy, compared to double precision. Our work expands the efforts by evaluating portability and scalability for Geometric Multigrid solvers by using fine-grain data blocking, including a new set of optimization strategies and performance evaluation models for GPU-accelerated architectures.

III. Bricks - Optimized layout for block structured grids

For completeness, this section summarizes salient aspects of BrickLib from previous work [2], [5]. Additionally, in [29], there is a description of several applications where BrickLib provided a boost in performance. The section ends with a brief discussion of new operators added for this paper to support GMG.

Brick data layout and fine-grained data blocking: The brick data layout uses fine-grained data blocking (similar to references [3], [4], [30]) for stencil loops, without traditional "ghost cell" approaches and their associated memory overheads. In this paper, bricks are 3D blocks stored contiguously in memory, specifically 8^3 or 4^3 for our experiments, as described in Section V. These fine-grained data blocks take advantage of hardware features that optimize data movement of contiguous addresses, such as multi-word cache lines, prefetch engines, and TLBs. In contrast, when using a conventional array data layout for 3D stencils, an 8^3 tile touches a large number of separate address streams, resulting in more streams and cache misses, resulting in more data movement.

BrickLib DSL Input

```
# Declare indices
i = Index(0)
j = Index(1)
k = Index(2)

# Declare grid
input = Grid("x", 3)
output = Grid("Ax", 3)
alpha = ConstRef("MPI_ALPHA")
beta = ConstRef("MPI_BETA")

# Express computation
# output[i, j, k] is assumed
calc = alpha * input(i, j, k) + \
       beta * input(i + 1, j, k) + \
       beta * input(i - 1, j, k) + \
       beta * input(i, j + 1, k) + \
       beta * input(i, j - 1, k) + \
       beta * input(i, j, k + 1) + \
       beta * input(i, j, k - 1)
output(i, j, k).assign(calc)
```

Fig. 1. Python-syntax BrickLib DSL code to specify a 7-pt stencil for `applyOp()` function in the V-cycle.

BrickLib domain-specific library: BrickLib is a domain-specific library and code generator, in which the brick data layout provides indirection from `ijk` stencil grids. Using a simple python-like stencil DSL, code transformations and optimizations can be applied and the final optimized kernel is able to target a specific architecture without low-level performance optimization in end-user code. Figure 1 shows the DSL input for a 3D, radius 1 star-shaped stencil over 7 points, which is used in the V-cycle smooth operation, for example. This format is fairly flexible, including larger stencils, non-constant coefficients, conditionals, and different coarse/fine index spaces. It is then transformed to create compile-time layouts, improve register and intermediate reuse, and inject GPU (or CPU) intrinsic instructions optimized for the stencil radius and brick dimensions.

Vector code generation: BrickLib uses a domain-specific vector code generator [5] to target both CPUs and GPUs. It uses a common internal abstraction of vectors to structure the generated code, and subsequently map to architecture-specific instructions, including SIMT code for GPUs, and wide SIMD instructions for CPUs.

There are three essential domain-specific optimizations in vector code generation. First, *vector folding* as described by

Yount [31] creates longer vectors by collapsing brick dimensions. Second, in a stencil pattern, some input data is reused from computing neighboring output points, but shifted in the 3D domain requiring data reorganization in the vectors as observed by Henretty et al. [32]. BrickLib's vector code generator detects this reuse of *array common subexpressions* [33], [34], exploiting reuse in buffers and shifting iteration spaces rather than data. Third, for high-order stencils, it is often profitable to eliminate redundant loads by *scattering* an input to all the outputs that use it to avoid data movement associated with the large amount of temporary data when *gathering*; *vector scatter* is used when profitable in conjunction with the reuse of buffers described above in a vector version of the *associative reordering via statement splitting* approach described by Stock et al. [35].

New operators in BrickLib for multigrid. In this paper, we extend BrickLib for multigrid computations. Because GMG has different coarse and fine grids that interact, additional stencils are needed for *restriction* and *interpolation*. In restriction, fine cells are coarsened by volume-averaging to the coarse grid cells, brick by brick, which requires no communication between neighbors, just between multigrid levels. Similarly, interpolation copies values from the coarse grid to correct fine grid data. This is a piecewise-constant interpolation that also does not require communication.

IV. Experimental Setup

A. GPU Architectures

Perlmutter [36] is the HPE Cray EX supercomputer at the National Energy Research Scientific Computing Center (NERSC), Lawrence Berkeley National Laboratory. Each Perlmutter GPU node contains one AMD EPYC 7763 CPU and four of NVIDIA Ampere A100 GPUs [37]. Each GPU includes 108 streaming multiprocessors (SM) each with four warp schedulers of 16 integer units and 8 double-precision floating point units. The GPU provides a peak performance of about 9.77 TFLOP/s in double-precision. The SMs each include a 192KB shared memory/data cache and share a 40 MB L2 cache and 40 GB of HBM accessible at 1.5TB/s. The GPUs are individually connected to the CPU with a PCIe 4.0 x16 link providing 32 GB/s. Nodes are connected with a Slingshot 11 interconnect system providing up to 25 GB/s bandwidth per NIC.

Frontier [38] is the most recent supercomputer at the Oak Ridge National Laboratory. Each node comprises one 64-core AMD EPYC 7A53 CPU and four AMD MI250X GPUs [39]. Each MI250X instantiates two Graphical Compute Dies (GCDs) each with 110 compute units (CU). Each CU includes four 16-wide 64b SIMD units to execute either integer or floating-point instructions and a small L1 cache. Each GCD also includes an 8MB L2 cache, provides a peak FP64 performance of about 24 TFLOP/s, and is connected to 4 HBM stacks of 64 GB providing 1.6 TB/s. Network connection between nodes uses Slingshot 11 system with the NIC attached directly to the GCDs. Thus, compared to Perlmutter's A100, each MI250X GCD provides more than twice the peak FLOP rate for FP64, comparable HBM bandwidth and network bandwidth with the difference that the NICs are attached directly to the GCDs.

Sunspot [40] is a testbed for application and software development prior to Aurora supercomputer at Argonne National Laboratory. Each Sunspot node consists of two Intel Xeon CPU Max Series (codename Sapphire Rapids or SPR) and six Intel Data Center GPU Max Series (codename Ponte Vecchio or PVC) [41]. Each PVC GPU is a two tile architecture interconnected by Xe links with the other PVC GPUs. Each stack has 64-GB of HBM, 208MB L3 per stack and 448KB L1 with two 80KB lcache per Xe-core. The entire GPU supports a total of 1024 execution units, each with a SIMD width of 512b. Eight EUs are grouped together into an Xe-core with a shared cache. Sixteen Xe-core form a slice, and four slices form a stack providing a peak FP64 performance of about 16TFLOP/s and 1.64TB/s of memory bandwidth per stack. Compared to Perlmutter's A100 and Frontier's MI20X GCD, a PVC stack provides similar memory bandwidth, about 1.6× higher peak FLOP rate for FP64 than A100, and about 0.6× TFLOP/s less than AMD MI250X GCD. Interconnect is provided by Slingshot 11 with eight NICs per node giving the same NIC bandwidth as Perlmutter and Frontier but a higher overall network bandwidth compared to Perlmutter and Frontier.

B. Compilers and Profiling Tools

Our work evaluates a geometric multigrid solver on three different GPU supercomputers. Table I shows the corresponding GPU programming model, modules, compilers, and environment variables for Perlmutter, Frontier, and Sunspot. Notice that on Perlmutter and Frontier, high performance is attainable using GPU-Aware MPI. Conversely, on Sunspot, we observed that not using the GPU-Aware MPI feature delivered better performance. It is important to point out that Sunspot is a test and development System with early versions of the Aurora software development kit in an early deployment state with non-final system configurations and could get hardware and software instabilities.

Perlmutter and Frontier additional environment variables are dedicated to managing message protocols that go through the fabric to decide between eager or rendezvous protocols according to message sizes. In addition, using `FI_CXI_RX_MATCH_MODE=hardware` enables hardware support for message matching by the Cassini NIC — an execution mode that could provide performance improvements for specific MPI ranks, as mentioned in [42].

Profiling timings are completed by recording events or wall-clock time to capture kernel and MPI operations timings using CUDA, HIP, or SYCL. For performance portability analysis, which consists of metrics from the roofline model [8], we gather the data with the profilers NVIDIA Nsight Compute [43], RocProf [44], and Intel Advisor [45] for NVIDIA-, AMD-, and Intel GPUs, respectively.

TABLE I
COMPILER VERSIONS, FLAGS AND ENVIRONMENT VARIABLES USED FOR CUDA, HIP AND SYCL ON PERLMUTTER-NERSC, FRONTIER-OLCF AND SUNSPOT-ALCF.

HPC System	Progr. Model	Modules	Compiler version	Env. variables
Perlmutter-NERSC	CUDA	cudatoolkit, nvidia	NVHPC 23.9, CUDAToolkit 12.2, PrgEnv-nvidia/8.5.0, nvcc/12.2, cray-mpich/8.1.28, libfabric/1.15.2.0	MPICH_GPU_SUPPORT_ENABLED=1 MPICH_OFI_NIC_POLICY=GPU FI_CXI_RDZV_EAGER_SIZE=0 FI_CXI_RDZV_THRESHOLD=0 FI_CXI_RDZV_GET_MIN=0
Frontier-OLCF	HIP	PgrEnv-amd, craype-accel-amd-gfx90a	PrgEnv-amd/8.3.3, amd/5.3.0, ROCm 5.3.0, AMD clang/15.0.0, cray-mpich/8.1.28, libfabric/1.15.2.0	MPICH_GPU_SUPPORT_ENABLED=1 MPICH_OFI_NIC_POLICY=GPU FI_CXI_RDZV_EAGER_SIZE=0 FI_CXI_RDZV_THRESHOLD=0 FI_CXI_RDZV_GET_MIN=0 FI_CXI_RX_MATCH_MODE=hardware
Sunspot-ALCF	SYCL	oneapi	oneapi/eng-compiler/ 2023.12.15.001, icpx/2024.0.0, mpich/52.2-1024/icc-all-pmix-gpu, libfabric/1.15.2.0	

C. Geometric Multigrid Solver

Geometric multigrid is a matrix-free iterative solver which accelerates convergence by creating a hierarchy of grid levels by recursively updating an initial guess on the finest grid with corrections produced on coarsest grids to improve overall convergence rate. NOTE: there are many variations of multigrid; the point of this paper is not to optimize numerical convergence, but to demonstrate performance portability using bricks for the typical elements of GMG.

In *finite volume* geometric multigrid, the V-cycle updates an initial guess "down" through all grid levels to compute a correction to the solution. With h defined as the grid size on the finest grid, on each coarser level the grid size is multiplied by two and represented by the superscript $2h, 4h, \ldots$, etc. Figure 2 shows a schematic of the hierarchy of grids and the main operations:

- Apply the linear operator to the approximate solution. We use a simple 7 point stencil defined at the finest resolution, x^h.
- Do a point Jacobi smoothing operation to x^h (alternative smoothers could include successive over-relaxation or Gauss-Seidel with similar performance characteristics).
- Compute the residual at the finest level $r^h = b^h - Ax^h$.
- Apply a restriction operator to the residual. In finite volume context, a coarse cell contains 8 fine cells, and its average is $b^{2h} = R^h(r^h)$.
- Solve $A\,x^{2h} = b^{2h}$ by applying the linear operator and smoothing the correction x^{2h}.
- Recursively coarsen until the coarse problem is sufficiently small to solve directly using point Jacobi (again, other solvers might be more effective).
- Apply an interpolation operator to approximate x^h. This is done with piece-wise constant interpolation from the coarse grid to the fine grid and updating the solution by $x^h = x^h + I^{2h}(x^{2h})$

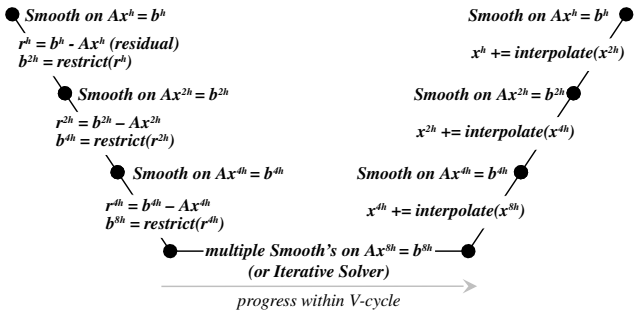

Fig. 2. Geometric multigrid V-cycle schematic that represents the process to solve a linear problem $Ax = b$ on a set of hierarchy grids where V-cycle operations as smoothing, restriction and interpolation+increment are depicted. At the coarsest level, a direct or an iterative solver is used to eventually correct the solution at the finest grid.

- Apply the linear operator and smooth the updated solution, x^h.

The amount of coarse grid data is 1/8 that of the finer grid in 3D; we therefore expect that the computation time scales eight times between levels. For communication operations related to the grid surface, this is a 2D set on a 3D region, so we expect the data volume to scale by four times between levels (for large grids - we will see that other factors dominate on smaller grids).

Algorithms for this GMG V-cycle are listed in Algorithm 1 and 2. We define `applyOp` as applying the 7-point stencil to the solution. This requires ghost cells from `exchange()` of surface data from one subdomain to 26 neighboring subdomains using `MPI_ISend, MPI_IRecv, MPI_WaitAll` functions. Smoothing is performed in `smooth+residual`, which simultaneously calculates the residual. The residual is averaged from eight fine cells into one coarse cell in `restriction`, which is written into the right-hand side for the next coarsest level. `interpolation+increment`

uses a piece-wise constant interpolation from a coarse cell to increment eight elements in the next finest grid.

Algorithm 1 Geometric Multigrid

```
1: while maxNormRes > 1e − 10 do
2:     V-cycle()
3:     Compute maxNormRes <<<>>>
4: end while
```

Algorithm 2 V-cycle

```
1: for level = 0, 1, ..., numLevels do
2:     for iteration = 0, 1, ..., max_smooths do
3:         exchange()
4:         applyOp <<<>>>
5:         smooth + residual <<<>>>
6:     end for
7:     restriction <<<>>>
8:     initZero <<<>>>
9: end for
10: \\Bottom Solver at level = numLevels
11: for iteration = 0, 1, ..., max_smooths do
12:     exchange()
13:     applyOp <<<>>>
14:     smooth <<<>>>
15: end for
16: for level = numLevels, numLevels − 1, ..., 0 do
17:     interpolation + increment <<<>>>
18:     for iteration = 0, 1, ..., max_smooths do
19:         exchange()
20:         applyOp <<<>>>
21:         smooth + residual <<<>>>
22:     end for
23: end for
```

To test our multigrid implementation using fine-grain data blocking, we use a simple stencil for 3D Poisson's equation in a cubic domain with periodic boundary conditions. Again, we choose this multigrid model problem for easy performance comparison and load balancing, although BrickLib can also generate code for more complicated stencils, data distribution, and domain boundary conditions. The right hand side of the equation is initialized to $b = sin(2\pi x)sin(2\pi y)sin(2\pi z)$, where x, y and z are spatial locations. The matrix operator A is the standard 7 point stencil with center coefficient $\alpha = -6/h^2$ and neighboring cells with $\beta = 1/h^2$ and h is the grid spacing on a given level. The residual is computed as $r = b - A\,x$ and the smoothing function is a point Jacobi operation defined as $x := x + \gamma(A\,x - b)$, with $\gamma = h^2/12$. The convergence criterion is when the maximum residual in the entire domain is less than 1e-10.

V. Implementation details and Optimizations

Geometric multigrid solvers are a combination of computation and communication operations. Our work examines several optimizations, settings, and tools to achieve higher performance on modern supercomputers' GPUs and networks; this requires several key optimizations:

Fine-grain data blocking: BrickLib provides fine-grain data blocking that uses hardware features to optimize data movement of contiguous addresses and reduce memory overheads. BrickLib provides a code generator that applies code transformations and optimizations for stencil computations using a Python-like stencil domain-specific library (DSL in Figure 1). The 7-point stencil in `applyOp()` is repeatedly applied to the entire domain, and is one of the most time consuming operations on a V-cycle, especially on the coarsest level where it is applied until the coarse solution is obtained. For the remaining point-wise operations, such as `smooth`, `residual` and the inter-layer operations, `restriction` and `interpolation+increment` using fine-grain data blocking data structures enhance data locality and thus improve performance on GPU-like architectures. Therefore, we have set optimal brick sizes according to our observations, as $8 \times 8 \times 8$ elements on Perlmutter and Frontier and a brick size of $4 \times 4 \times 4$ on Sunspot to exploit their corresponding GPU architectures to reduce data movement and improve on-node performance. For the `applyOp()` operation, which uses the vector code generator to compute the 7-point stencil, we set the number of threads per block equal to $SIMD\_width$ for each architecture, where for NVIDIA is equal to 32, for AMD is 64 and for INTEL we have found that 16 is the most optimal selection. With the vector code generator, BrickLib moves data through the register file for neighboring threads by using shuffle primitives to achieve high-performance for stencils on GPUs.

Communication-Avoiding: In the V-cycle, the `exchange()` operation extracts surface data from each subdomain and exchanges ghost zones between subdomains. Applying fine-grain data blocking expands our ghost zone to the size of the bricks, and as a result, the smoothing operation can be applied multiple times without further communication, redundantly computing results with other processes. This reduces the MPI communication frequency by a factor of the ghost zone size, but as a consequence, communication is done with all 26 neighbors (faces, edges, and corners). We use several BrickLib MPI optimizations described in [6] to improve communication performance: using an optimal brick layout to reduce data movement, consolidating to minimize the number of messages, and reducing latency with packing- and unpacking-free communication buffers.

Optimal Mapping between CPU-GPU-NIC: Selecting the correct mapping between CPU, GPU, and NIC is crucial to avoid unnecessary data movement and increase performance on applications that communicate data among several ranks on intra- or inter-node systems. On Perlmutter and Sunspot, the NICs are connected to the CPUs; meanwhile, on Frontier, the NICs are connected directly to the GPUs. Consequently, users should know and test their mappings to ensure that MPI ranks are bound to their closest GPU-CPU-NIC or CPU-GPU-NIC. On Perlmutter and Frontier, which use HPE Cray MPI, the `MPICH_OFI_NIC_POLICY=GPU` variable can be used to

select the NIC that is closest to the CPU or GPU being used for each process. On Sunspot, affinity and rank placements are done through the job script by manually assigning the cores, GPUs, and NICs to interact over the shortest path possible.

GPU-Aware MPI Communication: MPI libraries on all the GPU systems in this paper offer GPU-Aware MPI support to perform MPI operations with GPU buffers. This feature provides support for MPI operations for inter-node and intra-node MPI transfers and also leverages the hardware to handle data transfers efficiently on modern high-performance systems. For this implementation, during the `exchange()` operation use GPU-Aware MPI features that give high performance on NVIDIA and AMD GPUs. On Sunspot, we observed better results copying the data from the GPU to the CPU and then performing MPI operations, so GPU-Aware MPI is not used in those results.

Small Message Communication Protocols: As we descend in the V-cycle, each level has an $8\times$ smaller problem size. Therefore, small message transfers that go through the NICs for inter-node applications could encounter different implementations and protocols to do the communication between processes. Consequently, faster communication as we descend in the V-cycle is challenging since the default mechanisms for small message transfers may not be the most adequate for our application. The CXI provider allows setting different message protocols by environment variables (see Table I) that may improve MPI communication performance by using the rendezvous protocol for small message sizes and NIC hardware support, as we have observed on Frontier.

VI. Performance of the V-Cycle

This section's experiments consist of 8 nodes and one MPI rank per node, where we want to analyze the performance of the computation kernels on a single GPU/GCD/tile and communication operations using a single NIC. Each MPI rank binds to one CPU and one NVIDIA A100, GCD AMD MI250X, or Intel PVC tile. The domain consists of 512^3 elements per node, and we measure the total time per level across six multigrid levels. All timings are for the best mapping found, with GPU-Aware MPI on Perlmutter and Frontier, and using host pointers for MPI communication on Sunspot provided the best performance.

In Figure 3, the total execution time for all operations on each multigrid level is shown, for all three platforms, using communication-avoiding (CA) and all the optimizations available on each machine. The problem is considered converged in 12 V-cycles, using 12 smooth steps on each level, and at the coarsest level 100 smooth steps completely solve the coarse grid problem (but leading to a significant increase in wall clock time). Notet that this has been designed to proxy GMG solvers and to evaluate performance portability of the main GMG components individually.

Observe that our implementation presents good scaling between levels, closer to $4\times$, which is the ratio of the surface size between levels as we decrease the problem size since communication dominates over computation for this

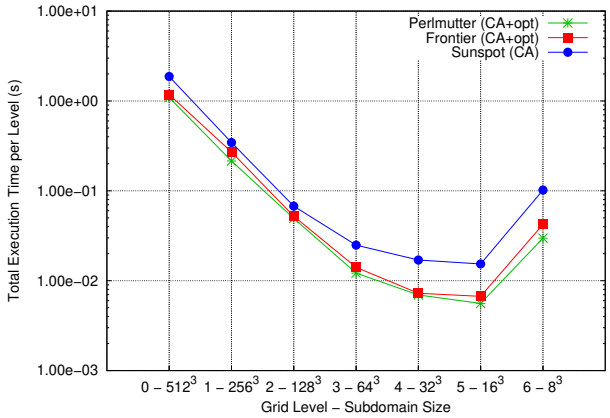

Fig. 3. Total execution time per level in seconds to solve a constant coefficient Poisson equation. Geometric multigrid uses a V-cycle with six levels deep, with 100 smoothing iterations on the coarsest level. Total domain size consists of 1024^3 elements distributed on eight nodes, where each node uses one CPU and a single A100, MI250X GCD, or PVC tile.

test problem. It is also clear that GPU-Aware MPI has a significant performance impact on Perlmutter and Frontier compared to Sunspot. We note that Sunspot is still in an early stage of maturity compared to Perlmutter and Frontier, which have spent significantly more time making improvements for production runs.

Another feature that we can observe in Figure 3 is the timing differences between Perlmutter, Frontier and Sunspot at coarsest levels, where the domain sizes are smaller and therefore, small messages are getting communicated through the NIC between subdomains. Perlmutter and Frontier systems can get faster at the coarsest levels compared to Sunspot, and this is a result achieved in a combination of using CA techniques plus the environment variables from Table I, indicating that using CXI settings and using rendezvous protocols could improve the communication operations for the smaller problem sizes, where GPU kernel computations are negligible compared to communication.

In Figure 4, we present an additional comparison between our GMG performance results and HPGMG, the CUDA version of the open-source benchmark designed as a proxy for finite volume based geometric multigrid linear solvers [46]. To make the right comparison, we have compiled and tested HPGMG-cuda with their second order finite volume implementation for a 3D periodic 1024^3 domain distributed across 8 nodes and each node using one CPU and one A100 GPU. We have used their optimized implementation for the GMG operators, selected the Jacobi smoother and a sequence of point-relaxation steps for the bottom solvers, as in our implementation. Relative performance against HPGMG is evaluated using the time per V-cycle, and we can observe that our implementation is $1.58\times$ and $1.46\times$ faster on Perlmutter and Frontier, respectively. Similar performance is shown between HPGMG and our Sunspot result. However, is important to note that our GMG implementation is portable and provides

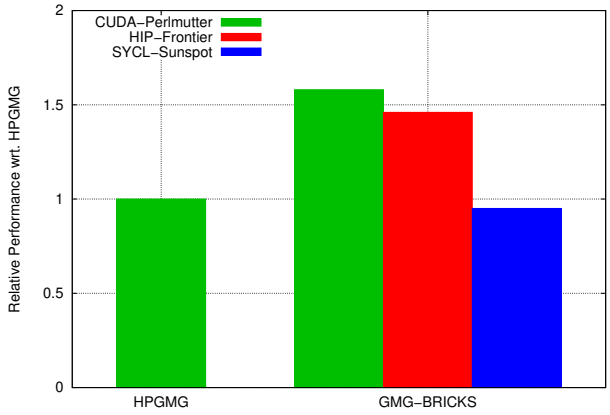

Fig. 4. Relative performance of our GMG implementation using Bricks, based on the time spent per Vcycle, with respect to HPGMG. Note that HPGMG is a CUDA-oriented implementation and it does not have a HIP or SYCL version.

TABLE II
PERCENTAGE OF THE TOTAL TIME AT THE FINEST GRID (LEVEL 0) FOR EACH OPERATION IN THE V-CYCLE USING COMMUNICATION AVOIDING AND THE PERTINENT OPTIMIZATIONS ON ALL PLATFORMS AND PROGRAMMING MODELS .

Operation	A100 GPU CUDA	MI250X GCD HIP	PVC Tile SYCL
applyOp	25.0%	30.7%	22.5%
smooth+residual	54.5%	50.0%	53.1%
restriction	1.0%	1.1%	1.5%
interpolation+increment	1.9%	5.4%	2.5%
exchange	17.5%	12.8%	20.4%

relative better or similar performance, on all three GPUs, compared to a architecture-driven optimized GMG version.

From the highest point of view, most of our solver's time is spent on the finest level. Table II shows the percentage of the total time at the finest grid (512^3 problem size) for each kernel computation and the communication operation on all platforms. The vast majority of time is spent on `applyOp`, `smooth+residual`, and `exchange`. Similar percentages for the computation kernels are achieved across all GPUs, likely because fine-grain data blocking provides similar performance across programming models and GPUs. However, communication operations differ between Sunspot and the other two systems, again likely due to GPU-Aware MPI on Perlmutter and Frontier.

A. Latency, overheads, throughput and bandwidth anaylisis

To compare stencil performance and MPI communication to theoretical GPU and network limits, we employ a traditional linear model to extract empirical latency, overhead, bandwidth, and throughput:

$$f(x) = \frac{x}{\alpha + x/\beta}$$

For computation kernels, x is the problem size, $f(x)$ is GStencil/s, α is latency in nanoseconds and β is attainable

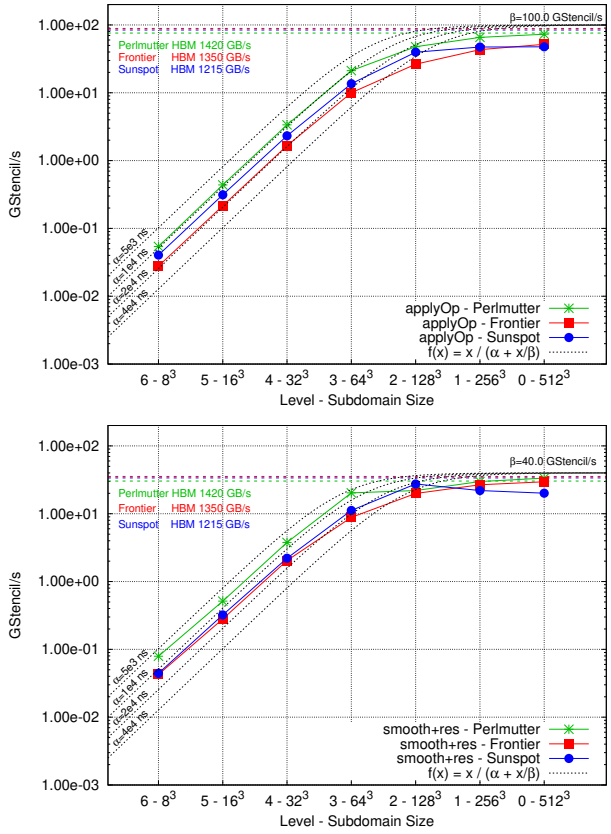

Fig. 5. GStencils/s, using time per invocation, for the operation `applyOp` (top) and `smooth+residual` (bottom) on Perlmutter, Frontier and Sunspot at all the levels of the V-cycle. Dashed lines, in colors, show the theoretical peak for this operation that consists of one read and one write using double precision. Performance model $f(x)$ describes the relation between latency and performance and gives a clear insight about the latency differences on all the machines and at which level computation becomes predominant in the V-cycle.

GStencil/s. GStencil/s is the performance metric that evaluates the speed of calculating one billion (10^9) stencil points per second. For a given computation kernel, such as `applyOp` or `smooth+residual`, this model indicates when a specific problem size is mostly memory bandwidth-bound, or mostly latency-bound.

Figure 5 shows the two most expensive computational kernels in the V-cycle on a GStencil/s, computed with time per invocation, and subdomain size plot. Dashed black lines are our model with different empirical latency values, in nanoseconds, and with an empirical flat part being set at 100 GStencil/s for `applyOp` and 40 GStencil/s for `smooth+residual`. Colored dashed lines represent the theoretical GPU limits for each architecture based on the kernels' number of reads and writes in double precision. For example, `applyOp` requires one read and one write in double precision, so for Perlmutter's measured HBM with 1420 GB/s, we obtain a theoretical ceiling for GStencil/s by dividing the GPU HBM bandwidth by the number of reads plus writes multiplied by eight bytes (double

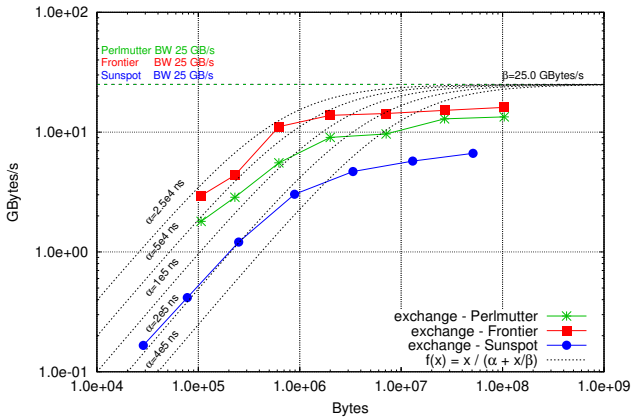

Fig. 6. GB/s, using time per invocation, for the `exchange` operation on Perlmutter, Frontier and Sunspot at all the levels of the V-cycle. The theoretical peak for communication in this experiment is limited by the peak NIC bandwidth at 25 GB/s. The performance model $f(x)$ depicts the relation between latency and bandwidth for MPI communications, where latency dominates the V-cycle for total message size smaller than one MegaByte.

precision); the result is 88.75 GStencil/s for Perlmutter.

Figure 5 presents the GStencil/s attained by `applyOp` and `smooth+residual` showing that our implementation that uses fine-grain data blocking and vector operations for stencil computations achieves near ideal performance throughput for the finest grids, independent of the programming model or the architecture. As we descend in the V-cycle, the problem size shrinks and at level two or three, depending on the kernel, the performance starts to drop linearly. Here, the kernel performance correlates to empirical latency values between $5\mu s$ and $20\mu s$. NVIDIA GPUs provide the lowest overhead and highest throughput per process for both kernels, while AMD and Intel GPUs provide comparable results.

We can derive a similar model to evaluate network throughput and empirical latency and overhead. In this case, we can use x as the total message size in bytes, $f(x)$ is GB/s, α is latency in nanoseconds, and β is attainable GB/s that we can set as the NIC bandwidth. Since all three systems work with Slingshot 11 as their network system, we can set the theoretical attainable limit at 25 GB/s.

Figure 6 shows the GB/s attained at different levels by doing the `exchange` operation. Note that this test using a single node with one A100, MI250X GCD, or PVC tile per node, which provides insight into the slowest part of the network by exercising a single NIC directly. We can observe that Frontier provides the highest bandwidth at 16 GB/s, followed closely by Perlmutter. Sunspot falls behind since it does not use the GPU-Aware MPI feature, which is still being developed and optimized on that platform. Regardless, the simple linear model correlates well with the empirical network data, showing the Frontier system with the lowest overhead and highest communication bandwidth, with Perlmutter and Sunspot falling behind with peak bandwidths between 7 and 14 GB/s and latency values between 25 μs and 200 μs.

TABLE III
PERFORMANCE PORTABILITY METRIC Φ BASED ON FRACTION OF THE ROOFLINE FOR THE OPERATIONS IN THE V-CYCLE AT THE FINEST LEVEL. USING BRICKLIB ACHIEVES A Φ EQUAL TO 73% WHEN HARMONIC AVERAGED ACROSS ALL PLATFORMS AND PROGRAMMING MODELS.

Operation	A100 CUDA	GCD MI250X HIP	tile PVC SYCL	Φ
applyOp	90%	77%	66%	76%
smooth	98%	87%	64%	80%
smooth+residual	94%	87%	71%	83%
restriction	95%	79%	62%	76%
interpolation+increment	88%	42%	52%	55%
				73%

Please note that these experiments included typical shared network variability, as opposed to the best possible performance that might be obtained from dedicated, isolated compute nodes.

VII. PERFORMANCE PORTABILITY ON A SINGLE GPU

Next we explore the performance portability of the computational kernels on the finest grid in the V-cycle by comparing performance on a single NVIDIA A100, AMD MI250X GCD, or Intel PVC tile. We adopt the definition of performance portability Φ in [9], so that given a set of platforms and programming models H for an application a solving problem p is:

$$\Phi(a,p,H) = \begin{cases} \frac{|H|}{\sum_{i \in H} \frac{1}{e_i(a,p)}}, & \text{if } i \text{ is supported } \forall i \in H \\ 0, & \text{otherwise} \end{cases}$$

As described in [7], in order to orthogonalize code generation efficiency from innate cache subsystem performance, we examine two efficiencies $e_i(a,p)$ and two performance portability metrics Φ: one based on fraction of Roofline for empirical arithmetic intensity and another based on fraction of theoretical arithmetic intensity. Empirical roofline limits were extracted from the `mixbench` benchmark [47] for the NVIDIA A100 and AMD MI250X GPUs. In the case of Intel PVC, the Roofline limits were found using Intel Advisor.

Table III presents performance portability based on the fraction of peak Roofline performance, using the empirical AI as $e_i(a,p)$. We compute metrics for all kernels in the V-cycle at the finest grid. Our implementation attains a Φ greater than 73% when considering all architectures and programming models.

TABLE IV
THEORETICAL ARITHMETIC INTENSITY (FLOP:BYTE) FOR THE V-CYCLE OPERATIONS AT THE FINEST LEVEL.

Operation	Theoretical AI (FLOP/B)
applyOp	0.50
smooth	0.125
smooth+residual	0.15
restriction	0.11
interpolation+increment	0.06

TABLE V
PERFORMANCE PORTABILITY METRIC $\mathcal{P}$ BASED ON FRACTION OF THE THEORETICAL ARITHMETIC INTENSITY FOR THE OPERATIONS IN THE V-CYCLE AT THE FINEST LEVEL. USING BRICKLIB ACHIEVES A $\mathcal{P}$ EQUAL TO 92% WHEN HARMONIC AVERAGED ACROSS ALL PLATFORMS AND PROGRAMMING MODELS.

Operation	A100 CUDA	GCD MI250X HIP	tile PVC SYCL	$\mathcal{P}$
applyOp	98%	88%	86%	90%
smooth	96%	100%	94%	97%
smooth+residual	100%	100%	71%	88%
restriction	99%	99%	86%	94%
interpolation+increment	100%	74%	100%	90%
				92%

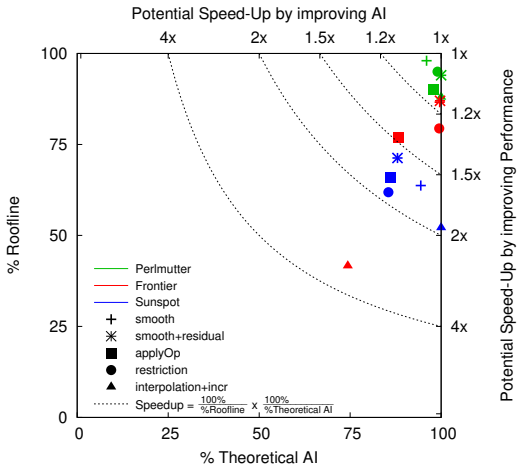

Fig. 7. Potential Speed-Up plot for all the operations in the V-cycle at the finest grid on one NVIDIA A100, one AMD MI250X GCD and one Intel PVC tile using CUDA, HIP and SYCL. Our implementation attained over 70% of the Roofline and theoretical arithmetic intensity overall among all operations, programming models and GPU architectures.

Table III demonstrates the data structure and vector code generator's ability to achieve peak memory bandwidth; it is not an assessment of cache misses or data locality. To that end, Table IV compares observed AI (approximately the inverse of data movement) to the theoretical bounds based on compulsory (cold) cache misses for each V-cycle operation and GPU architecture. In essence, this assumes the ideal bound for which each GPU has an infinite capacity, fully associative cache. Proximity to this highly idealized bound would represent near perfect cache performance using finite hardware. Thus, we define an additional $\mathcal{P}$ based on fraction of theoretical arithmetic intensity and present the results for each V-cycle operation and GPU in Table V. We observe that our implementation on actual GPU caches achieves nearly 92% portability when averaged over all architectures and programming models, meaning that using an optimized data layout and vector operations ensures GPUs keep data movement very close to what could be attained with infinite resources.

Figure 7 unifies these two performance portability efficiencies into a single plot where fraction of theoretical AI is the x-coordinate and fraction of the Roofline is the y-coordinate for memory bound operations. One can define a set of iso-curves of constant potential speedup (any mix of improved code generation/bandwidth with improved data locality) in order to quantify overall implementation performance in this equation:

$$Speedup = \frac{100\%}{\%Roofline} \times \frac{100\%}{\%TheoreticalAI}.$$

Although GMG does not have any compute-bound kernels, an equivalent expression could be used in that case by assuming 100% as theoretical arithmetic intensity.

In Figure 7, we measured fraction of theoretical AI and fraction of the Roofline for the V-cycle operations on all three GPU architectures. We can notice NVIDIA GPUs achieved the highest performance efficiencies for all operations with a potential speed-up of at most 1.2×. One GCD AMD MI250X GPU presented high efficiencies for almost all operations with a potential speed-up range between 1.2× and 1.5×, with an outlier close to 4× for interpolation+increment, since that kernel appears to unnecessarily move additional data, which we are investigating. In addition, one Intel PVC tile presented similar performance efficiencies with a slightly higher potential speed-up ranging between 1.5× and 2×.

VIII. SCALABILITY WITH FULL NODES

This section presents the scalability tests of our implementation up to 128 nodes on all three architectures, for a total range of 8 to 512 NVIDIA A100 GPUs, 8 to 512 AMD MI20X GPUs and 12 to 96 INTEL PVC GPUs, where one AMD MI20X GPU contains two GCDs and one INTEL PVC GPU has two tiles. Note that we show results for Perlmutter and Frontier up to 128 nodes but up to 16 nodes on Sunspot, since Sunspot is a testbed comprising 128 nodes and access to the full system was not possible for this paper. For weak and strong scalability tests, we map one MPI rank to one A100 NVIDIA GPU, one GCD AMD MI250X GPU, and one tile INTEL PVC GPU. With this configuration, an entire Perlmutter node has four MPI ranks, a Frontier node has eight MPI ranks, and a Sunspot node has twelve MPI ranks.

Figure 8 shows the performance throughput of our implementation in GStencil/s by using the total time to converge to the solution of a linear system $Ax = b$ with a subdomain size of 512^3 per rank on 2-16-128 nodes. Results show a good scaling trend on all three architectures, with the difference that each architecture node handles a different number of ranks per node and each rank solves a 512^3 problem size. As a result, Frontier presents almost double GStencil/s performance compared to Perlmutter, showcasing the ability of AMD GPUs to pack more performance throughput in a single node. Sunspot performance falls behind with performance in GStencil/s closer to the Perlmutter numbers, even though it had more GPUs available per node compared to the other two systems. However, the main reason for this lack of

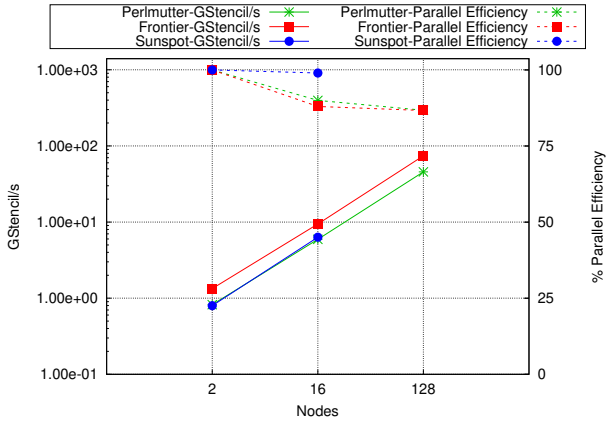

Fig. 8. GStencil/s, using total time, and parallel efficiency for weak scalability test solving a linear system $Ax = b$ with a subdomain size of 512^3 points per rank. Perlmutter is using four ranks per node, where each rank binds to one CPU/GPU. Frontier uses eight ranks per node, also binding each rank to one CPU/GCD. Sunspot uses 12 ranks per node with each rank binding to one CPU/tile. Our implementation achieves over 87% parallel efficiency with Frontier getting close to 100 GStencil/s as each node can solve twice the problem size compared to Perlmutter.

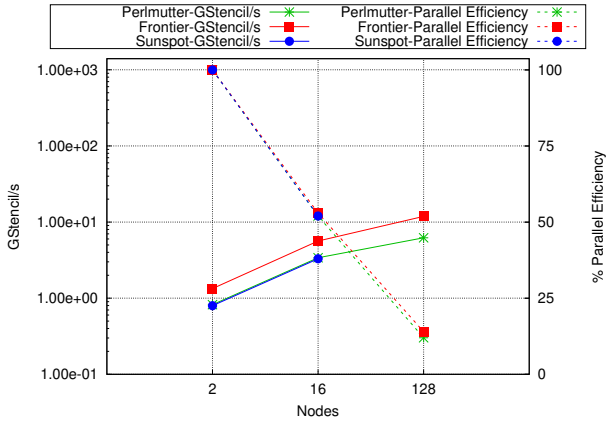

Fig. 9. GStencil/s, using total time, and parallel efficiency for strong scalability test solving a linear system $Ax = b$ with a total domain size of 1024^3 on Perlmutter, 2×1024^3 on Frontier and 3×1024^3 on Sunspot. Perlmutter is using four ranks per node, where each rank is binded to one CPU/GPU. Frontier uses eight ranks per node with each rank binded to one CPU/GCD. Sunspot uses 12 ranks per node with each rank binded to one CPU/tile. Our implementation achieves close to 12 GStencil/s on Frontier but performance becomes hinder by increasing concurrency on a fixed domain since decreasing the problem size per rank makes the V-cycle to be latency bound making parallel efficiency to nose dive.

performance is the network drawbacks, especially for this application where MPI communication constitutes closer to 20% of the time at the finest grid in the V-cycle.

In the same figure 8, we show the parallel efficiencies for weak scaling, where our test achieves more than 87%. Note that the parallel efficiencies on Perlmutter and Frontier follow a similar trend, indicating that both systems are scaling at similar rates, demonstrating the performance portability of our implementation given that both systems have the same network interconnect and one A100 NVIDIA GPU has comparable GPU HBM to one GCD AMD MI250X.

Figure 9 shows the performance throughput in GStencil/s for a strong scaling test where the total domain size is fixed and consists of 1024^3 cells in Perlmuter, 2×1024^3 on Frontier and 3×1024^3 on Sunspot. We use the same configuration of ranks-per-node in each architecture by using four ranks per node on Perlmutter, where each rank is binded to one GPU, eight ranks per node on Frontier with each rank binded to one GCD and twelve ranks on Sunspot with each rank binded to one tile. Thus, we keep the same fixed problem size to be solved by each rank on all architectures, but we start doubling the number of ranks in each dimension for a total of 128 nodes (512 GPUs) on Perlmutter and Frontier and 16 nodes (96 GPUs) on Sunspot.

Observe that the performance throughput on Frontier is close to double that of Perlmutter, which is what we could expect as each Frontier node can handle twice the problem size compared to Perlmutter. However, as we increase the number of ranks with a fixed problem size, we are increasing concurrency and solving a smaller problem on each rank, pushing us into the latency limits as described in Section VI-A. As computation and communication timings plateau at latency/overhead limits, performance efficiencies start to drop as we increase concurrency and total time becomes constant. On the other hand, Sunspot shows a similar performance compared to Perlmutter, which is related to the lack of performance of the MPI communications in the system. Finally, we can consider that weak and strong tests show similar performance efficiencies on the systems studied here, showing the performance portability capabilities of our implementation on systems with similar GPU and network capacities.

IX. Discussion and Future Work

Linear solvers, such as geometric multigrid, present some challenges in achieving high performance on modern supercomputers since multigrid solvers rely on computation and communication operations on a hierarchy of grids with a dynamic range of orders of difference between the finest grid and the coarsest grid. It becomes imperative to restructure those operations to minimize data transfers at the GPUs and the network, thus minimizing time-to-solution and improving scientific applications that heavily use linear solvers in their computational frameworks.

In this paper, we explore and shed some light on achieving high performance on GPUs for GMG operations, such as stencil computations and inter-grid operations, using fine-grain data blocking and communication optimizations. Our implementation can attain over 73% of the theoretical on-node performance on all three GPU-accelerated systems and presented a speedup of $1.6\times$ per V-cycle compared against a similar setup of HPGMG-CUDA. Effectively using fine-grain data blocking can reduce unnecessary data movement in the GPU memory hierarchy, and by optimizing stencil computations using warp-based operations, we can reuse data

more effectively at the register level, improving data locality and performance on all GPUs and programming models.

Our work also shows that to attain high performance for communication operations, the right CPU-GPU-NIC mapping, GPU-Aware MPI feature, setting the most convenient messaging protocols, and the use of an optimal data layout for communications to reduce the number of messages and avoid packing can be beneficial for GMG. In addition, deepening the ghost zone size to avoid communicating data more frequently can increase performance, especially at small problem sizes, due to communication overheads being close to ten times larger than kernel launching overheads.

In the context of performance analysis on a dynamic range of problem sizes in the V-cycle, we found that traditional linear models are well-correlated with computation and communication operations in our implementation, from which we can extract empirical latency, overhead, bandwidth, and throughput to compare to theoretical limits. As a result, NVIDIA GPUs provide the lowest overhead and highest throughput for most of the computation operations, while AMD and INTEL GPUs provide comparable performance. Meanwhile, the Frontier system showed better network performance in the latency and bandwidth regions, with Perlmutter bringing a comparable performance and Sunspot falling back since it is still a testbed prior to Aurora and is a less mature system.

Parallel efficiencies reached 87% when weak scaling up to 128 nodes (512 GPUs) and analyzing performance in GStencil/s gives us an insight into the ability of the architecture nodes to deliver performance by packing more GPUs per node. Strong scaling tests highlight the latency/overhead drawbacks on a dynamic range of problem sizes in a V-cycle. In order to improve strong scaling parallel efficiencies for this application, we should focus our attention on improving the latency/overhead of kernel launching on the GPUs and of the MPI communications at the network or restructure the algorithm to scale on latency bound applications by exploring the ability to pack more computation from several ranks into fewer ones to avoid network contention or solving small size problems on the CPU where latency/overhead timings could be significantly less than the GPU ones if the copy operations between CPUs and GPUs are also being mitigated.

Future work will complete our studies on performance portable techniques to improve our multigrid implementation when facing small-size problems and enhancing strong scaling. We will also extend our work to explore adaptive mesh refinement, where specific grid regions are subjected to refinement and load balancing becomes critical. Finally, we will also explore other smoothers, operators and bottom solvers that could improve time-to-solution for multigrid solvers and evaluate their performance and scalability on modern supercomputers.

ACKNOWLEDGMENT

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This work was done on a pre-production supercomputer with early versions of the Aurora software development kit, and we gratefully acknowledge the computing resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

REFERENCES

[1] P. Wesseling and C. Oosterlee, "Geometric multigrid with applications to computational fluid dynamics," *Journal of Computational and Applied Mathematics*, vol. 128, no. 1, pp. 311–334, 2001, numerical Analysis 2000. Vol. VII: Partial Differential Equations. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0377042700005173

[2] T. Zhao, S. Williams, M. Hall, and H. Johansen, "Delivering performance-portable stencil computations on cpus and gpus using bricks," in *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2018, pp. 59–70.

[3] M. Araya-Polo, F. Rubio, R. de la Cruz, M. Hanzich, J. M. Cela, and D. P. Scarpazza, "3d seismic imaging through reverse-time migration on homogeneous and heterogeneous multi-core processors," *Sci. Program.*, vol. 17, no. 1-2, pp. 185–198, Jan. 2009. [Online]. Available: http://dx.doi.org/10.1155/2009/382638

[4] C. Yount, J. Tobin, A. Breuer, and A. Duran, "Yask-yet another stencil kernel: A framework for hpc stencil code-generation and tuning," in *Proceedings of the Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for HPC*, ser. WOLFHPC '16, 2016.

[5] T. Zhao, P. Basu, S. Williams, M. Hall, and H. Johansen, "Exploiting reuse and vectorization in blocked stencil computations on cpus and gpus," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356210

[6] T. Zhao, M. Hall, H. Johansen, and S. Williams, "Improving communication by optimizing on-node data movement with data layout," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 304–317. [Online]. Available: https://doi.org/10.1145/3437801.3441598

[7] O. Antepara, S. Williams, H. Johansen, T. Zhao, S. Hirsch, P. Goyal, and M. Hall, "Performance portability evaluation of blocked stencil computations on gpus," in *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1007–1018. [Online]. Available: https://doi.org/10.1145/3624062.3624177

[8] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, p. 65–76, apr 2009. [Online]. Available: https://doi.org/10.1145/1498765.1498785

[9] S. Pennycook, J. Sewall, and V. Lee, "Implications of a metric for performance portability," *Future Generation Computer Systems*, vol. 92, pp. 947–958, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X17300559

[10] 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 *Proc. ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[11] X. Zhou, J.-P. Giacalone, M. J. Garzarán, R. H. Kuhn, Y. Ni, and D. Padua, "Hierarchical overlapped tiling," in *Proc. International Symposium on Code Generation and Optimization (CGO)*, 2012.

[12] D. Unat, T. Nguyen, W. Zhang, M. N. Farooqi, B. Bastem, G. Michelogiannakis, A. Almgren, and J. Shalf, *TiDA: High-Level Programming Abstractions for Data Locality Management*. Cham: Springer International Publishing, 2016, pp. 116–135.

[13] F. Luporini, M. Louboutin, M. Lange, N. Kukreja, P. Witte, J. Hückelheim, C. Yount, P. H. J. Kelly, F. J. Herrmann, and G. J. Gorman, "Architecture and performance of devito, a system for automated stencil computation," *ACM Trans. Math. Softw.*, vol. 46, no. 1, apr 2020. [Online]. Available: https://doi.org/10.1145/3374916

[14] R. Anderson, J. Andrej, A. Barker, J. Bramwell, J.-S. Camier, J. Cerveny, V. Dobrev, Y. Dudouit, A. Fisher, T. Kolev, W. Pazner, M. Stowell, V. Tomov, I. Akkerman, J. Dahm, D. Medina, and S. Zampini, "Mfem: A modular finite element methods library," *Computers & Mathematics with Applications*, vol. 81, pp. 42–74, 2021, development and Application of Open-source Software for Problems with Numerical PDEs. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0898122120302583

[15] W. Zhang, A. Myers, K. Gott, A. Almgren, and J. Bell, "Amrex: Block-structured adaptive mesh refinement for multiphysics applications," *The International Journal of High Performance Computing Applications*, vol. 35, no. 6, pp. 508–526, 2021. [Online]. Available: https://doi.org/10.1177/10943420211022811

[16] P. Grete, J. C. Dolence, J. M. Miller, J. Brown, B. Ryan, A. Gaspar, F. Glines, S. Swaminarayan, J. Lippuner, C. J. Solomon, G. Shipman, C. Junghans, D. Holladay, J. M. Stone, and L. F. Roberts, "Parthenon—a performance portable block-structured adaptive mesh refinement framework," *The International Journal of High Performance Computing Applications*, vol. 37, no. 5, pp. 465–486, 2023. [Online]. Available: https://doi.org/10.1177/10943420221143775

[17] J. Watkins, M. Carlson, K. Shan, I. Tezaur, M. Perego, L. Bertagna, C. Kao, M. J. Hoffman, and S. F. Price, "Performance portable ice-sheet modeling with mali," *The International Journal of High Performance Computing Applications*, vol. 37, no. 5, pp. 600–625, 2023. [Online]. Available: https://doi.org/10.1177/10943420231183688

[18] S. Williams, D. D. Kalamkar, A. Singh, A. M. Deshpande, B. Van Straalen, M. Smelyanskiy, A. Almgren, P. Dubey, J. Shalf, and L. Oliker, "Optimization of geometric multigrid for emerging multi- and manycore processors," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–11.

[19] J. Sebastian, N. Sivadasan, and R. Banerjee, "Gpu accelerated three dimensional unstructured geometric multigrid solver," in *2014 International Conference on High Performance Computing & Simulation (HPCS)*, 2014, pp. 9–16.

[20] D. Göddeke and R. Strzodka, "Cyclic reduction tridiagonal solvers on gpus applied to mixed-precision multigrid," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 22–32, 2011.

[21] I. Stroia, L. Itu, C. Niţă, L. Lazăr, and C. Suciu, "Gpu accelerated geometric multigrid method: Performance comparison on recent nvidia architectures," in *2015 19th International Conference on System Theory, Control and Computing (ICSTCC)*, 2015, pp. 175–179.

[22] W. Ma, Y. Ao, and S. Williams, "Solving a trillion unknowns per second with hpgmg on sunway taihulight," *Cluster Computing*, vol. 23, pp. 493–507, 2020.

[23] N. Onodera, Y. Idomura, Y. Hasegawa, S. Yamashita, T. Shimokawabe, and T. Aoki, "Gpu acceleration of multigrid preconditioned conjugate gradient solver on block-structured cartesian grid," in *The International Conference on High Performance Computing in Asia-Pacific Region*, ser. HPCAsia '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 120–128. [Online]. Available: https://doi.org/10.1145/3432261.3432273

[24] M. Geveler, D. Ribbrock, D. Göddeke, P. Zajac, and S. Turek, "Towards a complete fem-based simulation toolkit on gpus: Unstructured grid finite element geometric multigrid solvers with strong smoothers based on sparse approximate inverses," *Computers & Fluids*, vol. 80, pp. 327–332, 2013, selected contributions of the 23rd International Conference on Parallel Fluid Dynamics ParCFD2011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0045793012000345

[25] N. Zhang, M. Driscoll, C. Markley, S. Williams, P. Basu, and A. Fox, "Snowflake: A lightweight portable stencil dsl," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2017, pp. 795–804.

[26] Q. Zhu, H. Luo, C. Yang, M. Ding, W. Yin, and X. Yuan, "Enabling and scaling the hpcg benchmark on the newest generation sunway supercomputer with 42 million heterogeneous cores," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476158

[27] A. Bienz, W. D. Gropp, and L. N. Olson, "Reducing communication in algebraic multigrid with multi-step node aware communication," *The International Journal of High Performance Computing Applications*, vol. 34, no. 5, pp. 547–561, 2020. [Online]. Available: https://doi.org/10.1177/1094342020925535

[28] Y.-H. M. Tsai, N. Beams, and H. Anzt, "Three-precision algebraic multigrid on gpus," *Future Generation Computer Systems*, vol. 149, pp. 280–293, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X23002741

[29] M. Lakshminarasimhan, O. Antepara, T. Zhao, B. Sepanski, P. Basu, H. Johansen, M. Hall, and S. Williams, "Bricks: A high-performance portability layer for computations on block-structured grids," *The International Journal of High Performance Computing Applications*, vol. 0, no. 0, p. 10943420241268288, 0. [Online]. Available: https://doi.org/10.1177/10943420241268288

[30] J. Jayaraj, "A strategy for high performance in computational fluid dynamics," Ph.D. dissertation, University of Minnesota, 2013.

[31] C. Yount, "Vector folding: Improving stencil performance via multi-dimensional simd-vector representation," in *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, Aug 2015, pp. 865–870.

[32] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan, "Data layout transformation for stencil computations on short-vector simd architectures," in *Compiler Construction*. Springer, 2011, pp. 225–245.

[33] S. J. Deitz, B. L. Chamberlain, and L. Snyder, "Eliminating redundancies in sum-of-product array computations," in *Proceedings of the 15th international conference on Supercomputing*. ACM, 2001, pp. 65–77.

[34] P. Basu, M. Hall, S. Williams, B. Van Straalen, L. Oliker, and P. Colella, "Compiler-directed transformation for higher-order stencils," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 2015, pp. 313–323.

[35] K. Stock, M. Kong, T. Grosser, L.-N. Pouchet, F. Rastello, J. Ramanujam, and P. Sadayappan, "A framework for enhancing data reuse via associative reordering," in *ACM SIGPLAN Notices*, vol. 49, no. 6. ACM, 2014, pp. 65–76.

[36] NERSC, "NERSC: Perlmutter gpu nodes," 2024. [Online]. Available: https://docs.nersc.gov/systems/perlmutter/architecture/

[37] NVIDIA, "NVIDIA A100 GPU architecture," 2020. [Online]. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[38] OLCF, "OLCF: Frontier gpu nodes," 2024. [Online]. Available: https://docs.olcf.ornl.gov/systems/frontier_user_guide.html

[39] AMD, "AMD CDNA 2 architecture," 2022. [Online]. Available: https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf

[40] ALCF, "ALCF: Sunspot gpu nodes," 2024. [Online]. Available: https://www.alcf.anl.gov/support-center/aurorasunspot/getting-started-sunspot

[41] INTEL, "INTEL IRIS XE GPU architecture," 2023. [Online]. Available: https://www.intel.com/content/www/us/en/docs/oneapi/optimization-guide-gpu/2023-0/intel-iris-xe-gpu-architecture.html

[42] K. Kandalla, K. McMahon, N. Ravi, T. White, L. Kaplan, and M. Pagel, "Designing the hpe cray message passing toolkit software stack for hpe cray ex supercomputers," ser. Cray User Group Proceedings, 2023.

[43] NVIDIA, "NVIDIA Nsight Compute CLI documentation," 2024. [Online]. Available: https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html

[44] AMD, "AMD rocProf documentation," 2024. [Online]. Available: https://rocm.docs.amd.com/projects/rocprofiler/en/docs-5.3.0/rocprof.html

[45] INTEL, "Intel Advisor documentation," 2024. [Online]. Available: https://www.intel.com/content/www/us/en/docs/advisor/user-guide/2024-2/overview.html

[46] N. Sakharnykh, "Heterogeneous HPGMG-FV implementation using CUDA with Unified Memory," https://bitbucket.org/nsakharnykh/hpgmg-cuda/src/master/, 2021.

[47] E. Konstantinidis and Y. Cotronis, "A quantitative roofline model for gpu kernel performance estimation using micro-benchmarks and hardware metric profiling," *Journal of Parallel and Distributed Computing*, vol. 107, pp. 37–56, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731517301247

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

I. Overview of Contributions and Artifacts

A. Paper's Main Contributions

The paper focuses on the performance of geometric multigrid (GMG) on NVIDIA A100, AMD MI250X and INTEL PVC GPU-accelerated supercomputers. The main contributions are listed as follows.

- C_1 Geometric Multigrid performance analysis on the three most recent DOE GPU-accelerated supercomputers.
- C_2 Performance portability assessment that includes a GMG implementation in SYCL for INTEL GPUs and it is compared with NVIDIA and AMD GPUs.
- C_3 GMG attains better than 73% of the theoretical on-node peak performance portability metric and 87% parallel efficiency when weak scaling.

B. Computational Artifacts

We evaluated our geometric multigrid using BrickLib. The source code is in GitHub and is available at

- A_1 https://github.com/OscarAntepara/bricklib/tree/gmg_brick/

The next table shows the relation between the computational artifact and its relations to the contributions. The artifact execution will provide the necessary results and data to generate the most relevant figures in the paper.

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1	Figure 3,4,5,6
	C_2	Table 3,5
	C_3	Figure 7,8

II. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

The computational artifact provides the source code for the implemented geometric multigrid using fine-grain data blocking for NVIDIA A100, AMD MI250X, and INTEL PVC GPU-accelerated supercomputers, such as Perlmutter, Frontier, and Sunspot.

Expected Results

The experiment's source code is in the `examples/gmg` directory. Here, there are three directories, `nvidia_cuda`, `amd_hip`, `intel_sycl`, where each one contains a `README` file with the instructions needed to compile and run the experiments on Perlmutter-NERSC, Forntier-OLCF and Sunspot-ALCF. Each test is independent where `nvidia_cuda` timings should be closer to `amd_hip` with `intel_sycl` being behind.

Expected Reproduction Time (in Minutes)

The expected computational time of this artifact on Perlmutter and Frontier for the 8-node test is about 1 minute. On Sunspot, it is about 2 minutes. The weak and strong scaling tests take about 15 to 20 minutes on all machines.

Artifact Setup (incl. Inputs)

Hardware: Results presented in this paper were obtained on an NVIDIA A100 GPU on Perlmutter at NERSC, AMD MI250X GPU on Frontier at OLCF and Intel PVC GPU on Sunspot at ALCF. In all experiments, we used only a single process running on one A100 GPU, one GDC on MI250X, and one tile on Intel PVC GPU.

Software: We evaluated our geometric multigrid implementation using BrickLib. The source code is available at `https://github.com/OscarAntepara/bricklib/tree/gmg_brick/` and can be cloned using: `git clone -b gmg_brick https://github.com/OscarAntepara/bricklib.git`.

On Perlmutter the modules are:

- NVHPC 23.9
- CUDAToolkit 12.2
- PrgEnv-nvidia/8.5.0
- nvcc/12.2
- cray-mpich/8.1.28
- libfabric/1.15.2.0

And the environmental variables are:

- MPICH_GPU_SUPPORT_ENABLED=1
- MPICH_OFI_NIC_POLICY=GPU
- FI_CXI_RDZV_EAGER_SIZE=0
- FI_CXI_RDZV_THRESHOLD=0
- FI_CXI_RDZV_GET_MIN=0

On Frontier the modules are:

- PrgEnv-amd/8.3.3
- amd/5.3.0
- craype-accel-amd-gfx
- ROCm 5.3.0
- AMD clang/15.0.0
- cray-mpich/8.1.28
- libfabric/1.15.2.0

And the environmental variables are:

- MPICH_GPU_SUPPORT_ENABLED=1
- MPICH_OFI_NIC_POLICY=GPU
- FI_CXI_RDZV_EAGER_SIZE=0
- FI_CXI_RDZV_THRESHOLD=0
- FI_CXI_RDZV_GET_MIN=0
- FI_CXI_RX_MATCH_MODE=hardware

On Sunspot the modules are:

- oneapi/eng-compiler/2023.12.15.001
- icpx/2024.0.0
- mpich/52.2-1024/icc-all-pmix-gpu
- libfabric/1.15.2.0

Datasets / Inputs: No Datasets or/and Inputs required.

Installation and Deployment: The README file in BrickLib explains all the dependencies required. The requirements are:

- C++14* compatible compiler
- OpenMP
- MPI library
- CMake
- Optional backends: CUDA, HIP and SYCL.

We uses CMake to find libraries. If some library fails to load, be sure to check the find module for it in CMake and set the corresponding paths.

Artifact Execution

After the artifact is compiled and built for each GPU-accelerated supercomputer, there are job scripts for the 8-node test, `script_perlmutter.sh`, in each directory in `examples/gmg`.

At configuration/compilation time, users can do `cmake -DBRICK_GPU_AWARE=ON ..` for Perlmutter, `cmake -DBRICK_USE_HIP=ON -DBRICK_GPU_AWARE=ON ..` for Frontier and `cmake -DHAS_SYCL=ON ..` for Sunspot in a build directory created to test this artifact.

At run time, each test is done with `"<exe> -s 512,512,512 -I 10 -l 6 -n 20"` as parameters. Where $-s$ is the subdomain size, $-I$ is the number of iterations to gather timing statistics, $-l$ is the number of levels in the V-cycle and $-n$ is the maximum number of iterations allowed for the solver. Each experiment solves a linear system by means of a geometric multigrid, where we solve the problem ten times for warm-up and solve it another ten times to gather timings statistics.

Artifact Analysis (incl. Outputs)

The output will include data used to generate the figures in the paper. Data includes the average timing for each V-cycle operation and per level, time per invocation for each V-cycle operation, total time per level, total time to solve the problem and performance in GStencil/s. In the output, an example of the timing per each V-cycle operation and per level is depicted as: `level 0 applyOp [0.265012, 0.265184, 0.265346] (σ: 9.20184e-05)`, where we have [min, avg, max] and (stdev) time across all ranks.

For the performance portability results that require GPU roofline metrics, we present the command lines given to the profiler for the performance portability analysis based on the fraction of the roofline and a fraction of the arithmetic intensity described in the paper.

On Perlmutter-NERSC, NVIDIA Nsight Compute command line to gather GPU metrics for double precision on a specific kernel is depicted below:

```
ncu --set full -s 3 -c 2 \\
--kernel-name-base=demangled \\
-k regex:"applyOp_kernel" \\
-o reportApplyOp.%q{SLURM_PROCID} \\
.\exec
```

where roofline metrics in the report can be visualized using the NVIDIA Nsight Compute API on your local machine.

On Frontier-OLCF, Amd-ROCProf is the profiler available on Frontier-OLCF to collect GPU metrics. The command line used is:

```
rocprof -i  input_file.txt --timestamp on
-o my_output.csv <exe> <params>
```

AMD ROCm Profiler needs an input file with the kernel name and the metrics to be collected. An example of the input file is shown below:

```
kernel: <kernel_name>
pmc : SQ_INSTS_VALU_ADD_F16
SQ_INSTS_VALU_MUL_F16
SQ_INSTS_VALU_FMA_F16
SQ_INSTS_VALU_TRANS_F16
pmc : SQ_INSTS_VALU_ADD_F32
SQ_INSTS_VALU_MUL_F32
SQ_INSTS_VALU_FMA_F32
SQ_INSTS_VALU_TRANS_F32
pmc : SQ_INSTS_VALU_ADD_F64
SQ_INSTS_VALU_MUL_F64
SQ_INSTS_VALU_FMA_F64
SQ_INSTS_VALU_TRANS_F64
pmc : SQ_INSTS_VALU_MFMA_MOPS_F16
SQ_INSTS_VALU_MFMA_MOPS_BF16
SQ_INSTS_VALU_MFMA_MOPS_F32
SQ_INSTS_VALU_MFMA_MOPS_F64
pmc : TCC_EA_RDREQ_32B_sum TCC_EA_RDREQ_sum
TCC_EA_WRREQ_sum TCC_EA_WRREQ_64B_sum
gpu: 0
```

To compute roofline metrics as performance in GFLOPS/s and arithmetic intensity can be found here: https://docs.olcf.ornl.gov/systems/frontier_user_guide.html#getting-started-with-the-rocm-profiler.

On Sunspot-ALCF, Intel provides several tools for GPU profiling. In this work, we have used Intel Advisor as the GPU metrics collector for roofline metrics. To profile our application with Intel Advisor, we used the following command line:

```
ZE_AFFINITY_MASK=0.0 advisor
--collect=roofline
--profile-gpu -- <exec> <params>
```

where $ZE\_AFFINITY\_MASK = 0.0$, is at the beginning of the line to run our executable with one tile. Intel Advisor will collect the information in a directory that will be needed to create an html report with the next line:

```
advisor --report=all --project-dir=.
--report-output=roofline.html
```

The *html* file can be opened with a web browser and it contains general information such as data movement across the memory hierarchy, FLOP count, instructions executed, etc.