Performance Portability Evaluation of Blocked Stencil Computations on GPUs

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

Tuowen Zhao
Samantha Hirsch
Priya Goyal
Mary Hall
mhall@cs.utah.edu
University of Utah
Salt Lake City, Utah, USA

ABSTRACT

In this new era where multiple GPU vendors are leading the supercomputing landscape, and multiple programming models are available to users, the drive to achieve performance portability across platforms faces new challenges. Consider stencil algorithms, where architecture-specific solutions are required to optimize for the parallelism hierarchy and memory hierarchy of emerging systems. In this work, we analyze performance portability of the BrickLib domain-specific library and vector code generator for stencils. BrickLib employs fine-grain data blocking to reduce the large amount of data movement associated with stencils. We compare different GPUs (NVIDIA, AMD and Intel) and their associated programming models (CUDA, HIP and SYCL). By testing a wide range of stencil configurations, we show that overall, BrickLib achieves good performance independent of machine or programming model. Moreover, we introduce correlation models as a new tool for comparing architectures and programming models from Roofline model data.

CCS CONCEPTS

• **Software and its engineering → Compilers**; **Software notations and tools**; • **Computer systems organization → Heterogeneous (hybrid) systems**.

KEYWORDS

Stencil computation, GPU, programming models, Roofline model, performance models, data blocking, vectorization

ACM Reference Format:
Oscar Antepara, Samuel Williams, Hans Johansen, Tuowen Zhao, Samantha Hirsch, Priya Goyal, and Mary Hall. 2023. Performance Portability Evaluation of Blocked Stencil Computations on GPUs. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3624062.3624177

SC-W 2023, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0785-8/23/11.
https://doi.org/10.1145/3624062.3624177

1 INTRODUCTION

Supercomputer architectures based on GPUs are prevalent and have dominated the Top500 list for the last four years [50]. Notably, one thing has changed starting in June 2022: the top two GPU architectures on the list incorporate AMD Instinct MI250X GPUs, well above the rankings of the platforms based on NVIDIA A100 and V100 GPUs. Soon Aurora [1] will introduce Intel GPUs to this list as well. With multiple GPU vendors leading the supercomputing landscape, we now have a variety of programming models and associated compilers available to target these architectures. However, programmers of these systems desire *performance portability*, i.e., achieving high performance across current and future GPU platforms, ideally using a single source code. Today's GPU systems support both native programming models (CUDA and HIP), and portable programming models such as SYCL that are intended to promote performance portability; therefore, another aspect of performance portability is whether a similar code written in either SYCL or a native programming model for the same platform both achieve high performance. At this inflection point in the supercomputing landscape, it is important to consider how much architecture, programming models, and maturity of compiler implementations impact performance portability for a given application class.

For such a performance-portability study, we consider the stencil computation, used to solve partial differential equations using the finite difference or finite volume methods, where the derivative at each point in space is calculated as a weighted sum of neighboring point values (a "stencil"). Stencils come in a variety of shapes corresponding to different discretizations and varying numbers of neighboring points. Low-order discretizations result in smaller stencils with limited data reuse, are typically bound by memory bandwidth, and thus underutilize the compute capability afforded by GPU architectures. High-order discretizations perform more floating point computations per point but can attain equal error with larger grid spacings (smaller array sizes); therefore, they exhibit higher arithmetic intensity and can derive solutions with less total data movement. However, exploiting data reuse in high-order stencils has been shown to exhibit increased data movement resulting from redundant loads and stores of temporary data. In practice, a high-performance stencil must incorporate architecture-specific optimizations to: (1) reduce data movement at multiple levels of the memory hierarchy (registers, caches, memory, TLBs) [4, 9, 10, 14, 16, 26, 27, 32, 34, 36, 45–48, 51–56, 58, 61, 66]; (2) exploit parallelism at multiple levels (across domains, nested

threading, and fine-grain SIMD parallelization) [17, 18, 34, 64, 65]; and (3) avoid redundant loads/stores and computation for stencils that exhibit high arithmetic intensity [5, 8, 10, 11, 13, 44, 48].

Given the need for architecture-specific optimization for stencils, a key question is how to achieve performance portability across different architectures and programming models. In this paper, we determine whether a domain-specific stencil library and code generator called *BrickLib* [6, 64, 65] delivers on its promises. When using BrickLib, stencil computations are expressed in a python-like DSL, and the system generates high-performance code for a variety of target programming models, including CUDA, HIP and SYCL. The defining feature of BrickLib is its use of fine-grain data blocking in the form of *bricks*, a data layout where mini subdomains are stored in contiguous memory so that accesses within a brick are part of a single address stream; neighboring subdomains are described with adjacency information, allowing flexibility in how bricks are organized in memory. In addition, BrickLib includes a vector code generator that exploits data reuse and eliminates redundant loads within a brick's computation. For benchmark comparisons, BrickLib also provides an option that defaults to a conventional array layout, while still exploiting the vector code generator.

In this paper, we study performance portability of stencil computations for different GPUs (NVIDIA A100, AMD M1250X and Intel PVC) and their associated programming models (CUDA, HIP and SYCL). We compare different data layouts (bricks vs. standard arrays), and isolate the contributions of bricks and the vector code generator by presenting results for arrays with vector code generation. We consider different stencil shapes and different problem sizes to observe how these variations strain memory bandwidth under different implementation scenarios. The paper makes the following contributions: (1) it is the first performance-portability study applied to stencils across recent NVIDIA, AMD and Intel GPUs; (2) it demonstrates differences in performance on the same platform using both vendor programming models and SYCL; (3) it introduces the *correlation model* and *potential speedup* as tools for comparing architectures and programming models using the Roofline model; and, (4) it demonstrates that optimizations used in BrickLib reduce data movement across all implementations and mitigate the differences between models on the same platform.

2 RELATED WORK

On structured grids, stencil computations are present at the core of several scientific and engineering applications. Historically, most optimization strategies for single nodes focus on spatial and temporal tiling to change memory access order and improve locality in cache [9, 10, 14, 16, 26, 27, 32, 34, 36, 45–47, 53–56, 58, 60, 61, 66]. As data movement increasingly dominates performance, some research focuses on instead on organizing data in a blocked physical representation to improve memory access order including BrickLib [64, 65], YASK [60] and RTM on the Cell processor [4]. All these fine-grained blocking techniques target large, compute-intensive stencils, and the small data blocks such as bricks do not have per-block ghost zones. TiDA [51, 52] uses coarse-grained data blocking, where the entire grid is tiled into sub-grids, each with its own ghost zone. High order stencils require additional optimizations to reduce redundant memory operations associated with temporary storage of reused data; techniques to reduce this data movement usually exploit *associative reordering* of computation, similar to identifying array common subexpressions and vector scatter used in the BrickLib vector code generator [5, 8, 10, 11, 13, 44, 48].

Research efforts on compilers and programming models are encouraged to achieve high performance on different heterogeneous systems: thus, achieving performance portability which enables scientific progress on the most powerful systems without code refactoring. A popular approach to compiler optimization of stencils is the use of domain-specific compilers for parallel code generation from a stylized stencil specification [7, 31, 49, 62, 63] or from a code excerpt [18]. Many of these target GPUs.

In this paper, we focus on the performance portability evaluation of fine-grain data blocking and vector code generation to leverage performance on modern supercomputers for stencil computations. We are looking to evaluate different techniques, such as tiling, data layouts, vector code generation, and warp-based computations on GPU based systems, such as NVIDIA, AMD and Intel, native programming models CUDA and HIP, and portability programming model SYCL. Since we are using an existing stencil framework BrickLib for this study, the primary contributions of this work is as a cross-platform performance-portability study of stencils on current-generation GPUs and available programming models.

We cite here prior cross-platform performance-portability studies of stencils and GPU programming models, and a recent experiment for tensor contraction on the same architectures. In the multi-core era, a study of stencil optimizations applied to the 3D Heat Equation (7pt stencil) on five multi-core architectures highlighted differences between these systems and pointed to the performance and energy benefits of many-core platforms and autotuning [10]. A previous study demonstrated performance portability of BrickLib across CPU (Intel KNL, Intel Skylake) and GPU (NVIDIA P100) [65]. Lee et al. examined performance portability of a stencil-based CFD application for thermodynamics simulation, performing an earlier generation cross-platform (NVIDIA and AMD GPUs, Intel Many Integrated Cores, and Altera FPGAs) and cross programming model (OpenMP+MPI, OpenACC+MPI, and CUDA+MPI) study [29].

Several recent performance portability studies have similarly evaluated the same or older GPU architectures on other applications. Mehta et al. compared OpenMP implementations of the SNAP application, a proxy for LAMMPS, on earlier GPUs, and evaluated the effectiveness of different layout optimizations in OpenMP using the Roofline model [33]. A comparison of Kokkos, SYCL and native programming models for the Milc-Dslash benchmark showed comparable performance, using speedup over a common baseline as the metric of interest [15]. Two articles present a comparison across mini-applications and the programming models used in their implementation applied the same performance portability metric as in this paper, along with metrics for evaluating consistency of performance [12, 28]. An examination of sparse block diagonal matrix times multiple vectors (SpMM) computation compared CUDA, HIP, OpenACC and Kokkos implementations; because the computation's performance varies with the input sparse matrix, the paper introduced a weighted aggregate performance portability metric to capture both variability and importance of individual measurements [19]. A recent performance portability experiment for tensor contraction compared native CUDA/HIP to performance portability

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

# Declare grid
input = Grid("in", 3)
output = Grid("out", 3)
a0 = ConstRef("MPI_B0")
a1 = ConstRef("MPI_B1")
a2 = ConstRef("MPI_B2")

# Express computation
calc=a0*input(i,j,k) + a1*input(i+1,j,k) + \
     a1*input(i-1,j,k) + a1*input(i,j+1,k) + \
     a1*input(i,j-1,k) + a1*input(i,j,k+1) + \
     a1*input(i,j,k-1) + a2*input(i+2,j,k) + \
     a2*input(i-2,j,k) + a2*input(i,j+2,k) + \
     a2*input(i,j-2,k) + a2*input(i,j,k+2) + \
     a2*input(i,j,k-2)
output(i, j, k).assign(calc)
```

BrickLib DSL Input

Figure 1: Python-syntax BrickLib DSL code to specify a star-shaped, radius 2 stencil in 3D.

frameworks Kokkos and SYCL on the same architectures as this study, showing a more significant performance loss when using Kokkos/SYCL as compared to CUDA on the NVIDIA GPU than the corresponding experiment on AMD [42].

As compared to prior performance portability studies applied to stencils, the work in this paper is distinguished by looking at recent GPUs and comparing native programming models to SYCL. We show BrickLib reduces data movement across the different implementations and mitigates the differences between native and SYCL implementations. Moreover, we introduce the use of correlation models as a tool for comparing architectures and programming models from Roofline model data.

3 BRICKLIB

For the sake of completeness, this section summarizes salient aspects of BrickLib, which are detailed in references upon which we build for this work [64, 65].

Brick data layout and fine-grained data blocking: The brick data layout embodies fine-grained data blocking (similar to references [4, 23, 60]) for stencil loops, without traditional "ghost cell" approaches and their associated memory overheads. For the purposes of this paper, bricks are 3D blocks stored in contiguous memory, specifically $4 \times 4 \times SIMD\_width$ for our experiments, where $SIMD\_width$ is architecture specific, as described in Section 4.4. 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, a $4 \times 4 \times SIMD\_width$ tile touches a large number of address streams, resulting in inefficient use of these hardware features and more data movement as compared to bricks.

BrickLib domain-specific library: BrickLib is a domain-specific library and code generator, in which the brick data layout underlies stencil grids. By applying code transformations for optimizations to a python-like stencil DSL, the final optimized kernel is able to target a specific architecture without low-level performance optimization. Stencils are defined in terms of constant weights for each offset, and can take any shape, from *star*-shaped grid-aligned, to *cube*-shaped full regions. Figure 1 shows the DSL input for a 3D, radius 2 star-shaped stencil over 13 points, which could be used to calculate a fourth-order accurate Laplacian stencil, for example. This format is fairly flexible and can be transformed to create compile-time layouts, improve register and intermediate reuse, and inject intrinsic instructions optimized for the stencil radius and brick dimensions.

Vector code generation: An important aspect of BrickLib is its domain-specific vector code generator. Since BrickLib is designed to target both CPUs and GPUs, it uses a common internal abstraction of vectors to develop the structure of the generated code, and subsequently map to architecture-specific instructions: SIMT code for GPUs, and wide SIMD instructions for CPUs. Figure 2 shows code snippets for GPU kernels using bricks with CUDA, HIP and SYCL for a star stencil.

There are three essential domain-specific optimizations in vector code generation. First, *vector folding* as described by Yount [59] is used to create long 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. [17]. BrickLib's vector code generator detects this reuse of *array common subexpressions* [5, 13], 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. [48].

The code resulting from vector code generation looks like a sequence of code blocks that compute portions of a brick's stencil grid. To achieve high performance on GPUs, data is moved through the register file of neighboring threads using *shuffle* primitives; the need for the movement is expressed with a macro that is replaced by architecture-specific implementations. For CUDA version 9 and later, $\_\_shfl\_down\_sync$ and $\_\_shfl\_up\_sync$, for earlier CUDA versions and HIP $\_\_shfl\_down$ and $\_\_shfl\_up$, and SYCL uses *sub_group_shfl_down* and *sub_group_shfl_up*. Although beyond the scope of this paper, architecture-specific implementations for CPUs include SIMD instructions in AVX2, AVX512, and SVE.

```
__global__ void d3star_brick
(unsigned (*grid)[STRIDEB][STRIDEB],
Brick <Dim<BDIM>, Dim<VFOLD>> bIn,
Brick <Dim<BDIM>, Dim<VFOLD>> bOut){
  long tk = GB + blockIdx.z;
  long tj = GB + blockIdx.y;
  long ti = GB + blockIdx.x;
  unsigned b = grid[tk][tj][ti];
  bOut[b][k][j][i] = (bIn[b][k+2][j][i] +
        bIn[b][k-2][j][i] + bIn[b][k][j+2][i] +
        bIn[b][k][j-2][i] + bIn[b][k][j][i+2] +
        bIn[b][k][j][i-2]) * MPI_B2 +
       (bIn[b][k+1][j][i] + bIn[b][k-1][j][i] +
        bIn[b][k][j+1][i] + bIn[b][k][j-1][i] +
        bIn[b][k][j][i+1] + bIn[b][k][j][i-1])
        * MPI_B1 + bIn[b][k][j][i] * MPI_B0;}
```

CUDA

```
__global__ void d3star_brick
(unsigned (*grid)[STRIDEB][STRIDEB],
Brick <Dim<BDIM>, Dim<VFOLD>> bIn,
Brick <Dim<BDIM>, Dim<VFOLD>> bOut){
  long tk = GB + hipBlockIdx_z;
  long tj = GB + hipBlockIdx_y;
  long ti = GB + hipBlockIdx_x;
  unsigned b = grid[tk][tj][ti];
  bOut[b][k][j][i] = (bIn[b][k+2][j][i] +
        bIn[b][k-2][j][i] + bIn[b][k][j+2][i] +
        bIn[b][k][j-2][i] + bIn[b][k][j][i+2] +
        bIn[b][k][j][i-2]) * MPI_B2 +
       (bIn[b][k+1][j][i] + bIn[b][k-1][j][i] +
        bIn[b][k][j+1][i] + bIn[b][k][j-1][i] +
        bIn[b][k][j][i+1] + bIn[b][k][j][i-1])
        * MPI_B1 + bIn[b][k][j][i] * MPI_B0; }
```

HIP

```
cgh.parallel_for<class d3star_brick>(nworkitem, [=](nd_item<3> WIid) {
   long bk = WIid.get_group(2); long k = WIid.get_local_id(2);
   long bj = WIid.get_group(1); long j = WIid.get_local_id(1);
   long bi = WIid.get_group(0); long i = WIid.get_local_id(0);
   bElem *bDat = (bElem *) bDat_s.get_pointer();
   auto bSize = cal_size<BDIM>::value;
   syclBrick<Dim<BDIM>, Dim<VFOLD>> bIn(bInfo_s.get_pointer(), bDat, bSize * 2, 0);
   syclBrick<Dim<BDIM>, Dim<VFOLD>> bOut(bInfo_s.get_pointer(), bDat, bSize * 2, bSize);
   unsigned b = bIdx_s[bi + (bj + bk * (STRIDEBY-2)) * (STRIDEBX-2)];
   bOut[b][k][j][i] = (bIn[b][k+2][j][i] + bIn[b][k-2][j][i] + bIn[b][k][j+2][i] +
     bIn[b][k][j-2][i] + bIn[b][k][j][i+2] + bIn[b][k][j][i-2]) * MPI_B2 +
    (bIn[b][k+1][j][i] + bIn[b][k-1][j][i] + bIn[b][k][j+1][i] + bIn[b][k][j-1][i] +
     bIn[b][k][j][i+1] + bIn[b][k][j][i-1]) * MPI_B1 + bIn[b][k][j][i] * MPI_B0;  } );
```

SYCL

Figure 2: Kernel example for a star stencil using bricks with different programming models (without vector code generation).

BrickLib performance portability. With the addition of auto-tuning for brick dimension, layout, and ordering, BrickLib demonstrates some level of *performance portability* [43], relative to the Roofline performance on a given architecture across a wide variety of stencils. Performance portability was demonstrated on Intel Xeon Phi, Intel Skylake, and NVIDIA P100 architectures in [65].

4 EXPERIMENTAL SETUP

In this section, we provide details of the GPU-based systems used for the performance and portability study. We also provide additional details about the programming models used in the evaluation.

4.1 GPU Architectures

Perlmutter [35] is the new 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's latest 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 x 16 link providing 32 GB/s. Nodes are connected with a Slingshot 11 interconnect system providing up to 12.5 GB/s bandwidth per NIC.

Crusher [40] is a testbed for the Frontier supercomputer at the Oak Ridge National Laboratory. Each node comprises one 64-core AMD EPYC 7A53 CPU and four AMD MI250X GPUs [2]. 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 but the NIC is attached directly to the GCDs. Thus, compared to Perlmutter's A100, each MI250X GCD provides more than twice peak FLOP rate for FP64, comparable bandwidth, and more overall network bandwidth.

Florentia [24] is a JLSE (Joint Laboratory for System Evaluation) testbed for application and software development prior to Aurora supercomputer at Argonne National Laboratory. Each Florentia node is a Quad GPU Early Silicon Ponte Vecchio (PVC) Node. Each node consists of four Intel Data Center GPUs, codenamed Ponte Vecchio, and two 4th Gen Intel Xeon Scalable processors [21]. Each PVC GPU is a two stack/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 Crusher's MI20X GCD, a PVC stack provides similar memory bandwidth, about 1.6x higher peak FLOP rate for FP64 than A100, and about 0.6x TFLOP/s less than AMD MI250X GCD.

In practice, programmers are advised to run one process per GCD on Crusher and one process per Stack on Florentia, which is what our tests use in this paper; our comparisons use either one A100 GPU on Perlmutter or one MI250X GCD on Crusher or one PVC GPU stack on Florentia.

4.2 Compilers and Profiling Tools

Table 1 lists the programming models, modules, and programming environments used on each of our target systems.

On Perlmutter, CUDA and HIP are available through NVIDIA CUDA compiler drivers and HIP installation [35]. The HIP module on Perlmutter functions as a wrapper that calls the NVIDIA compiler, while SYCL is available by using the Intel-llvm compiler installed on the system. We use NVIDIA Nsight Compute [38, 39] to profile and collect GPU performance metrics.

Crusher offers AMD compiler drivers [40] for HIP codes and an experimental module for DPC++ [41] to compile SYCL codes on AMD GPUs. We use AMD ROCm Profiler [3] and AMD Omniperf [30] to collect the GPU performance metrics on AMD MI250X.

The Florentia architecture provides Intel compiler drivers, libraries and tools for several languages. In this work we focus on a C++ SYCL code and Intel oneAPI products [22] are used to compile our tests on Intel PVC GPUs. We use Intel Advisor [20] to collect GPU performance metrics on Intel PVC.

4.3 Stencils

In this subsection, we describe the benchmark tests used to examine the performance of stencil computations. Broadly speaking, we define two classes of extensible stencils designed to proxy many common high-order finite difference stencils. *Star-shaped stencils* include points only along the axes with a distance (radius) of the central point. In 3D, this class nominally reproduces the classic 7-point stencil with $radius = 1$ and 13-point stencils with $radius = 2$. Our second class of stencils are classified as *cube-shaped* and include all points within a cubical bounding box of dimension $2 \times \text{radius} + 1$ centered on the central point. This class reproduces the common 27-point ($radius = 1$) and 125-point ($radius = 2$) stencils. In both cases, we use a minimal number (exploiting symmetry) of constant coefficients for each stencil (a 7-point stencil has two unique coefficients, while a 27-point stencil has four unique coefficients. Table 2 summarizes the stencils evaluated in our performance portability benchmarking effort. Experiments consist of computing stencils of different shapes and radii on a 3D grid with 512 double-precision elements in each dimension in an out-of-place manner.

Table 1: Programming models, modules and compiler versions used for CUDA, HIP and SYCL on Perlmutter, Crusher and Florentia.

HPC System	Progr. Model	Modules Load	Compiler version
Perlmutter NERSC	CUDA	cudatoolkit nvidia	NVHPC 22.7, CUDAToolkit 11.7, PrgEnv-nvidia/8.3.3, nvcc/11.7
	HIP	hip	hip/5.3.2, nvcc/11.7
	SYCL	intel-llvm	PrgEnv-llvm/0.1, intel-llvm/2023-WW13, clang++/17.0.0
Crusher OLCF	HIP	amd	PrgEnv-amd/8.3.3, amd/5.1.0, ROCm/5.2.0, AMD clang/14.0.0
	SYCL	dpcpp/22.09	dpcpp/22.09, ROCm/5.2.0, clang++/16.0.0
Florentia JLSE	SYCL	oneapi	oneapi/eng-compiler/ 2022.12.30.003, icpx/2023.1.0

Table 2: Stencils used for performance portability evaluation.

Stencil Shape	Radius	Points	Unique Coefficients
Star	1	7	2
	2	13	3
	3	19	4
	4	25	5
Cube	1	27	4
	2	125	10

4.4 Methodology

In this paper, we explore three combinations of data layout (array vs. bricks) and code generation (native compiler vs. vector codegen). Concurrently, we evaluate two metrics: performance (relative to Roofline), and performance portability. The evaluated ations are:

- **array:** Conventional array data layout is used, optimized with 3D tiling and a tile size of $4 \times 4 \times$ *SIMD_width* elements, which are mapped to the $\langle z, y, x \rangle$ thread dimensions. It uses a lexicographic representation of `i,j,k` for the input and output arrays.
- **array codegen:** Using a conventional array data layout, vector code generation available in the library computes stencils with a tiled lexicographic representation of the input and output arrays. When compared with **array**, this version permits isolating the benefits of the vector code generator.

- **bricks codegen:** Brick data layout is used, where each brick consists of $4 \times 4 \times SIMD\_width$ elements stored in contiguous memory, with adjacency information to preserve the logical ordering of the data. Vector code generation available in the library is applied. When compared with **array codegen**, this version isolates the benefits of the data layout in reducing data movement.

We use architecture-specific tile and brick sizes based on *SIMD_width* to show each system in its best light: 32 for NVIDIA A100; 64 for AMD MI250X; and for Intel PVC, where there is a choice between 16 or 32, we use 16 because it achieves better performance than 32. Note, for all kernels that use the vector code generator, the library applies different configurations for *vector_size* related to a specific GPU architecture. On NVIDIA A100, a *vector_size=32*, on AMD MI250X a *vector_size=64*, and on Intel PVC a *vector_size=16* are recommended in order to match warp/wave instructions to their respective architectures.

Performance: In this paper, we evaluate performance relative to the Roofline Model [57]. The Roofline model makes the simplifying assumption that computations are either memory- or compute-bound. This allows us to evaluate how close the various implementations come to the theoretical limits (imposed by bandwidth, FLOP/s, and theoretical stencil arithmetic intensity). We use the `mixbench` benchmark [25] to derive the double-precision Roofline plots for the NVIDIA A100 and AMD MI250X GPUs. In the case of Intel PVC, the Roofline plot was taken from Intel Advisor hl reports. For the Roofline model, GPU metrics – such as FLOP count, Bytes moved and kernel time – have been collected using NVIDIA Nsight Compute-CLI [38], AMD ROCm Profiler [3], AMD Omniperf [30], and Intel Advisor [20] on NVIDIA A100, AMD MI250X and Intel PVC GPUs, respectively. This performance study uses the same FLOP count for all kernels to avoid introducing FLOP count variations on the Roofline model due to different kernel implementations. The FLOP count for all the kernels has been selected as the minimum due to FLOP normalization in the Roofline model and to avoid inconsistent FLOP count given by the profilers.

Performance Portability: We define performance portability as attaining a similar fraction of Roofline across multiple architectures and/or moving the minimum amount of data from the GPU. The soft definition of performance portability allows the brick library to select the optimal programming model for the target architecture while a strict approach is evaluated for a given programming model.

5 RESULTS AND ANALYSIS

In this section, we analyze performance and data movement for each stencil shape and radius for NVIDIA, AMD and Intel GPUs using CUDA, HIP and SYCL. Next, we present new correlation and potential speed-up models that characterize stencil computations and give better insight into performance and data movement across programming models and GPU architectures by combining GPU metrics from different Roofline plots into one. We also present results for the established performance portability metric from Pennycook et al. [43].

5.1 Performance

Figure 3 shows the arithmetic intensity (x-axis) and performance in FLOP per second (y-axis) for the three kernel implementations, represented by colors with different stencil shapes and sizes. Cross symbols represent star-shaped stencils. Meanwhile, full square symbols represent cube-shaped stencils. Trend lines connecting the same color and symbols represent different radii for the same shaped stencil and are generally from lower to higher AI based on stencil radius. Note that we have a Roofline model per architecture and programming model. Overall results show that using **bricks** data layout gives a higher arithmetic intensity over tiled **array** data layout and that vector code generator implementations perform better than not using vector code generation. Observe that for all the plots considered here, we use the same number of FLOPs for the same stencil; thus, higher arithmetic intensity indicates reduced data movement resulting from improvements in data locality for the same stencil. As mentioned in Section 4.4, a minimum FLOP count among all kernels was used to calculate performance for the Roofline model.

On NVIDIA GPUs, CUDA brings the best overall performance among all programming models. Observe that vector code generation improves performance up to a factor of 1.3× for star stencils and up to 2× for cube stencils. Note that **bricks** brings the best performance for high order stencils, where a brick shape with the last dimension matching the warp size can achieve higher performance, especially for the compute-intensive 125-point stencil. Continuing the analysis on the NVIDIA A100, CUDA and HIP show the same performance and arithmetic intensity since the HIP interface is a wrapper for the NVIDIA compiler. Conversely, SYCL shows a noticeable difference in performance between kernels that use vector code generation – an improvement of up to 13× for star stencils and up to 26× for cube stencils compared with tiled **array** representations. Overall, **bricks codegen** achieves the highest performance and arithmetic intensity across all kernels and stencil shapes and sizes on the NVIDIA A100.

One GCD on an AMD MI250X GPU offers a similar HBM memory bandwidth as an NVIDIA A100 but with almost three times peak performance on double precision. A glance at figure 3 shows that all the kernels using HIP or SYCL on AMD GPU are in the same performance and arithmetic intensity range as CUDA or SYCL on NVIDIA GPU, and highlighting a performance and data movement consistency across different architectures. On AMD GPUs, and using HIP, **bricks codegen** achieves a performance improvement of up to 1.3× for star stencils and up to 3× for cube stencils compared to **array**. Moreover, **bricks codegen** shows better data locality, as the arithmetic intensity is higher than **array codegen**, improving performance by using a better data layout. A similar analysis can be derived for SYCL on AMD GPUs, where **arrays codegen** and **bricks codegen** take advantage of the vector code generator and improves the performance by up to 3× on star stencils and up to 9× on cube stencils. On the AMD MI250X, fine-grain data blocking plus vector code generator (**bricks codegen**) gives the best performance independent of the programming model being used.

One stack provides a similar HBM bandwidth on Intel PVC GPUs compared to an A100 NVIDIA or one GCD on AMD MI250X. On the other hand, the Peak FLOP rate for double precision is higher on

Intel GPU compared to NVIDIA but less high than AMD GPU peak FLOP rate. Kernels with vector code generation perform better than tiled **arrays**, as much as 3× on star stencils and up to 5× on cube stencils. The **array codegen** kernel shows a similar arithmetic intensity as **arrays**, where computing stencil outputs based on vector operations provided by the code generator makes the performance difference. Conversely, **bricks codegen** shows a better arithmetic intensity provided by the bricks data layout that incurs less data movement from the device memory and the vector code generator that improves GPU performance.

Additional performance details are in Figure 4 related to L1 data movement. We can observe that **bricks codegen** provides less variability on L1 data movement across all stencil shapes, programming models and architectures. The **array** implementation moves 10× or more L1 bytes compared to the vector code generator implementations. The main reason behind those differences is that **array codegen** or **bricks codegen** offers better register reuse by concentrating instructions in a warp/wave level on the vector code generation. Furthermore, **bricks codegen** takes advantage of the data layout and *shuffle* instructions available on CUDA, HIP and SYCL to further improve data reuse at the L1 memory hierarchy.

5.2 Performance Portability

In this section, we look closer at performance and data movement across programming models and architectures, using correlation plots to analyze performance portability. Moreover, we analyze different performance portability metrics as fraction of the Roofline and introduce fraction of theoretical arithmetic intensity, which could give us information about efficient hardware utilization and better data reuse, attaining minimum data movement.

5.2.1 Performance Portability Correlations. Figure 5 shows a scatter plot containing all the stencils and kernels (represented by symbols and colors respectively) to analyze different programming models on the NVIDIA A100 GPU.

Figure 5 (left) plots the performance as FLOPs for CUDA on the y-axis and FLOPs for SYCL on the x-axis on A100. A diagonal line is included to observe the pattern and relationship between programming models. Stencil symbols appearing above the diagonal indicate that CUDA outperforms SYCL on A100 GPUs. Results show that most of the stencils perform better using CUDA instead of SYCL. However, it is important to notice that most symbols pertaining to **bricks codegen** are closer to the diagonal, meaning that using fine grain data blocking with vector code generation reduces the performance gap between programming models.

Figure 5 (right) plots the measured bytes accessed using CUDA, represented on the y-axis, and SYCL, on the x-axis, on A100. Note that symbols falling below the diagonal indicate that SYCL has more byte accesses than CUDA. Dotted lines are included to denote an estimation of theoretical Bytes of data movement for this benchmark. Theoretical Bytes accessed have been measured as we have a 512^3 domain, one read and one write using double precision, giving us a total of 2.15 GBytes. The dotted lines represent a lower bound for Bytes accessed and estimate which programming model moves less data. We observe **array codegen** is moving closer to 4 Gbytes of data, indicating a lack of data reuse. Meanwhile, bricks kernels are significantly closer to the lower bound. Moreover, CUDA is moving 2× less data than SYCL, indicating a better locality.

Figure 6 (left) plots the performance in FLOPs for AMD MI250X GPU using HIP on the y-axis and SYCL on the x-axis. Similar to the analysis for the NVIDIA A100 GPU, symbols appearing above the diagonal indicate that HIP is performing better than SYCL on AMD GPUs. Results show a more balanced scenario where no clear programming model performs better than the other. **Array** without vector code generation performs better using HIP; **array codegen** and **bricks codegen** perform the same independently if HIP or SYCL is being used. Another observation, using the performance correlation plots, is that symbols going toward the upper right corner have the highest performance and are the fastest kernels. Overall, **bricks codegen** is faster and reduces the performance gap between programming models on AMD MI250X GPU.

Figure 6 (right) shows the Bytes accessed for the AMD GPU using HIP on the y-axis and SYCL on the x-axis. As a reminder, symbols falling below the diagonal indicate that SYCL has more byte accesses, and symbols above the diagonal involve more byte accesses with HIP. Kernels using HIP show Bytes accessed very close to the lower bound of 2.15 Gbytes, except for **array codegen**, which moves more than 10 Gbytes. SYCL on AMD attains better locality as fine-grain data blocking is used and reaches closer to the minimum for **bricks codegen**. Moreover, a similar effect happens here compared to the performance correlation where **bricks codegen** shows similar performance and data movement for HIP and SYCL on AMD MI250X GPU.

5.2.2 Performance Portability Metrics and Potential Speed-Up. In this paper, we use the performance portability metric described in [43]. Performance portability metric $\mathcal{P}$ is based on the harmonic mean of the application's performance efficiency across different platforms. Here, we want to explore using $\mathcal{P}$ across architectures and programming models to assess the performance portability for **bricks codegen**. We define performance portability $\mathcal{P}$, given a set of platforms and programming models H for an application a solving problem p is:

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

where $e_i(a,p)$ is the performance efficiency. In this paper, 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 $\mathcal{P}$: one based on fraction of roofline for empirical arithmetic intensity and another based on fraction of theoretical arithmetic intensity.

Table 3 presents performance portability using fraction of the Roofline with empirical AI as $e_i(a,p)$. Although depressed by the low fraction of Roofline for the 125pt stencil, **bricks codegen** still attains a $\mathcal{P}$ greater than 60% when averaged across all architectures and programming models.

Table 3 represents an assessment of our data structure and vector code generator's ability to saturate memory bandwidth and not an assessment on attaining ideal data locality. To that end, we also compare observed AI (basically inverse of data movement) to the theoretical bounds based on compulsory (cold) cache misses for

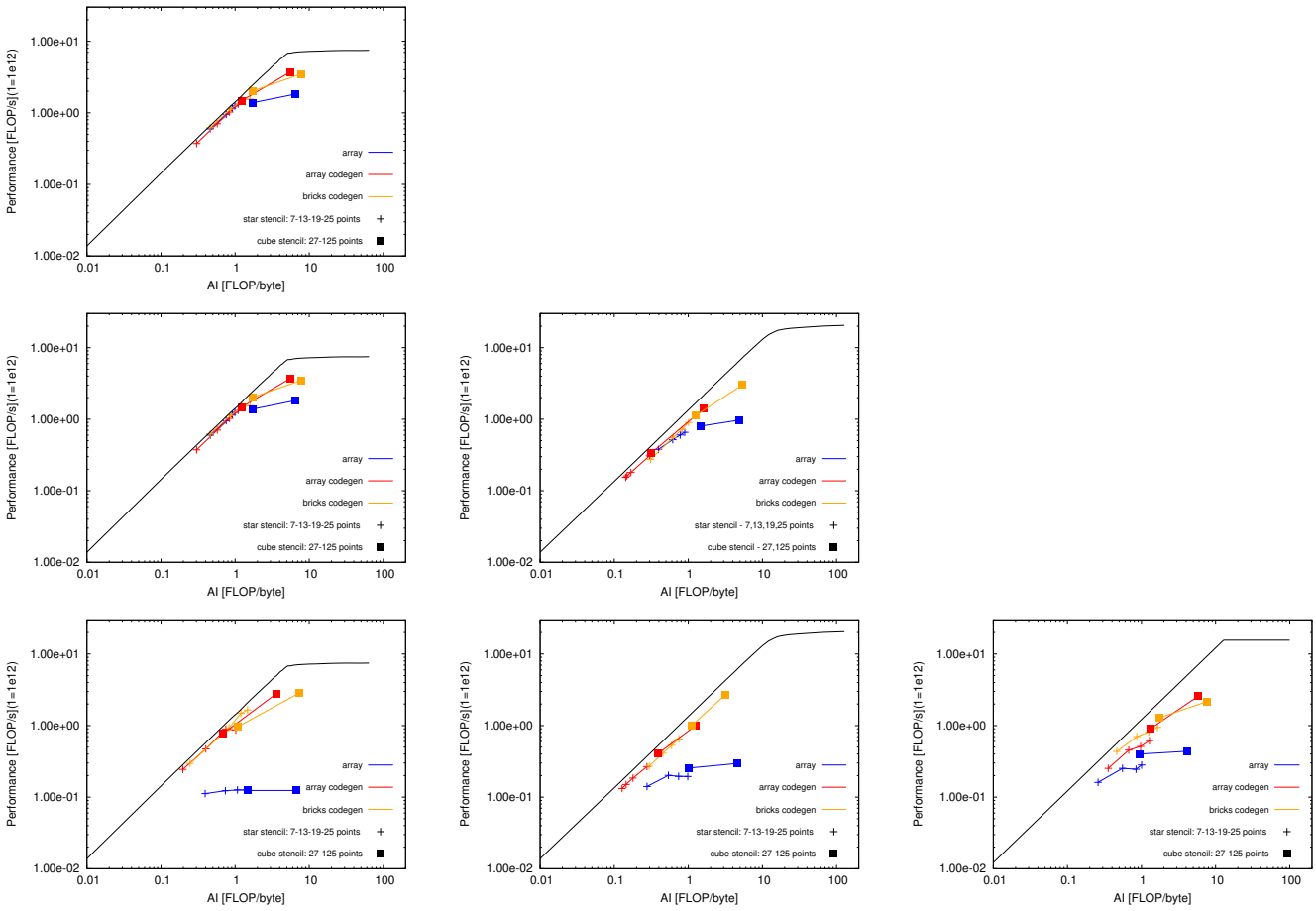

Figure 3: Roofline for stencil computations using CUDA (top row), HIP (middle row) and SYCL (bottom row) on NVIDIA A100 GPU (left column), single GCD on AMD MI250X GPU (middle column) and single stack on Intel PVC GPU (right column).

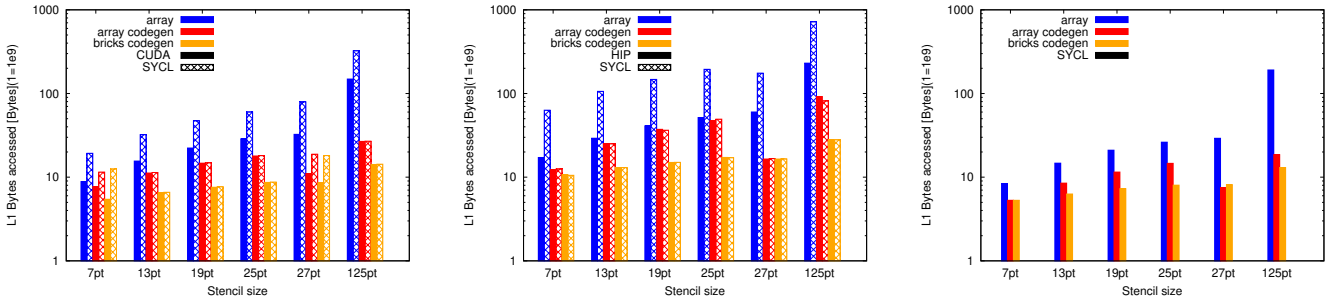

Figure 4: L1 data movement using CUDA or SYCL on NVIDIA A100 GPU (left), HIP or SYCL on a single GCD on AMD MI250X GPU (middle) and SYCL on a single stack on Intel PVC GPU (right). Note that bricks-codegen is the most efficient implementation related to L1 data movement by doing a better register reuse across programming models and GPU architectures (lower is better).

each stencil and GPU shown in Table 4. In essence, we assume a bound in 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 a new $\mathcal{P}$ based on fraction of theoretical arithmetic intensity and present the results for each stencil and GPU in Table 5. We observe that **bricks codegen** on finite GPU caches achieves nearly 70% portability when averaged over all architectures and

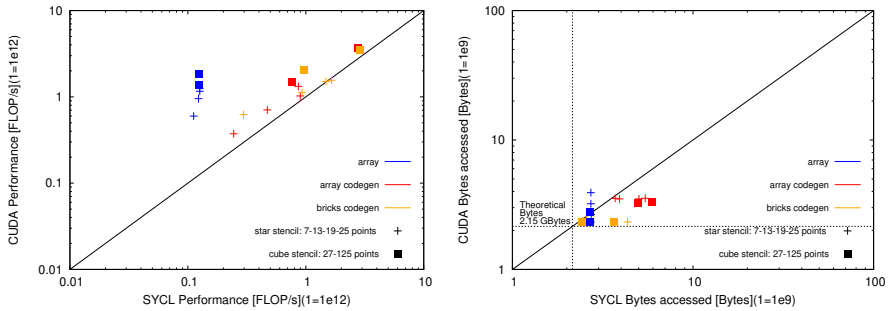

Figure 5: Performance (left) and Bytes accessed (right) correlation between CUDA and SYCL on NVIDIA A100 GPU. Observe, CUDA implementations consistently outperforms SYCL and usually moves near minimal data.

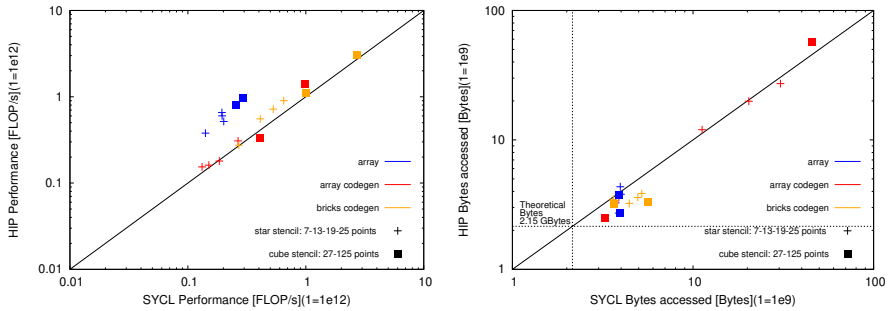

Figure 6: Performance (left) and Bytes accessed (right) correlation between HIP and SYCL on AMD MI250X GPU (single GCD). Dotted lines represent theoretical lower bounds. Observe, with the exception of array code generation, HIP implementations incur near minimal data movement and bricks codegen moved a minimum amount of data for both HIP and SYCL.

Table 3: Performance Portability metric $\mathcal{P}$ based on fraction of the Roofline for bricks codegen. bricks codegen achieves a $\mathcal{P}$ greater than 60% when averaged across all platforms and programming models.

Stencil	A100 CUDA	A100 SYCL	MI250X HIP	MI250X SYCL	PVC SYCL	$\mathcal{P}$
7pt	95%	84%	66%	68%	77%	77%
13pt	92%	79%	66%	67%	67%	73%
19pt	85%	87%	65%	66%	53%	69%
25pt	69%	79%	66%	64%	47%	63%
27pt	82%	60%	66%	67%	61%	66%
125pt	47%	39%	42%	63%	23%	38%
						61%

programming models, meaning that using an optimized data layout and vector operations ensures GPUs can keep data movement to within about 1.5× of what could be attained with infinite resources.

Figure 7 unifies these two performance portability metrics into a single figure where fraction of theoretical AI is the x-coordinate and fraction of the Roofline is the y-coordinate. 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.

Figure 7 shows with **bricks codegen**, NVIDIA and Intel show a higher fraction of AI, meaning they are closer to moving the

Table 4: Theoretical arithmetic intensity (FLOP:Byte) for all stencils shapes and sizes.

Stencil Shape	Number of points	Theoretical AI
Star	7	0.5
	13	0.9375
	19	1.375
	25	1.8125
Cube	27	1.875
	125	8.375

Table 5: Performance Portability metric $\mathcal{P}$ based on fraction of the theoretical arithmetic intensity for bricks codegen. Notice that bricks codegen a $\mathcal{P}$ of nearly 70% when averaged across all platforms and programming models.

Stencil	A100 CUDA	A100 SYCL	MI250X HIP	MI250X SYCL	PVC SYCL	$\mathcal{P}$
7pt	92%	49%	62%	59%	93%	67%
13pt	92%	88%	66%	48%	92%	72%
19pt	91%	87%	60%	43%	91%	68%
25pt	88%	81%	56%	41%	91%	65%
27pt	93%	59%	67%	59%	92%	71%
125pt	92%	89%	64%	38%	92%	67%
						68%

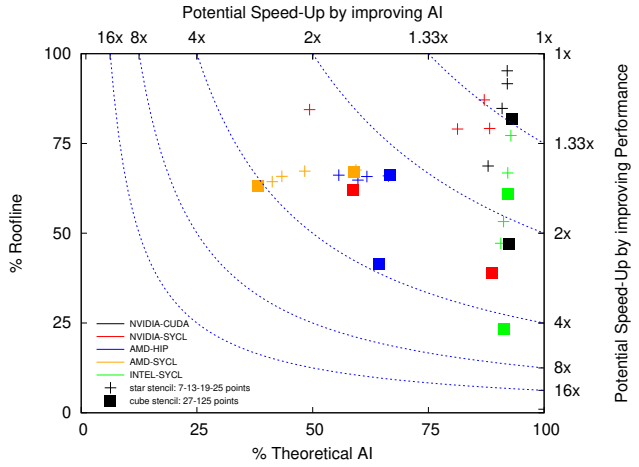

Figure 7: Potential Speed-Up plot for bricks-codegen implementation on NVIDA A100, AMD MI250X and Intel PVC using CUDA, HIP and SYCL. Bricks codegen attained over 50% of the Roofline and theoretical arithmetic intensity overall among all stencil configurations, programming models and GPU architectures.

minimum possible data from the device. On those architectures, there is still a potential speedup of up to 2× or 4× by improving kernel execution performance. Most of the stencils on AMD are at 50% on both metrics; there is still an overall potential speedup between 2× and 4× by improving data movement and/or code generation. In the context of stencil computation using bricks, one way to achieve this speedup is by changing the size of the brick which would expose more vector parallelism, amortize shuffling, and potentially improve data locality for a specific stencil on an architecture or programming model.

6 CONCLUSIONS

New emerging heterogeneous supercomputers increase HBM memory bandwidth and peak FLOPs, presenting opportunities to continue to scale application performance. However, as architectures advance, it is still challenging to develop portable solutions that can attain high performance independently of the system and without requiring total code rewriting.

This paper explores performance portability for stencil computations, which are widely used by the computational science and engineering community. Data reuse and vector operations (codegen) in the form of warp/wave level instructions are required to obtain high performance on stencil computations. These optimizations are provided by the BrickLib domain-specific library and code generator and reduce the reliance on such optimizations being available in performance-portable programming models. Results show fine-grain data blocking and vector code generation ensure performance and arithmetic intensity remain relatively similar between NVIDIA A100, AMD MI250X and Intel PVC GPUs and also among different programming models such as CUDA, HIP, and SYCL.

The performance gap between standard tiled arrays and fine-grain data blocking kernels is relatively small on CUDA or HIP, up to 2×, but it gets bigger with SYCL, between 4× to 10× for different stencil sizes. Still, BrickLib codegen improves the performance of SYCL substantially. Further compiler developments are required to ensure that a programming model designed for a diverse group of GPUs can be competitive against in-house compilers from a specific GPU vendor, primarily when domain-specific library optimizations are focused on vector shuffle and alignment operations.

Our work introduces correlation plots and performance metrics for performance portability analysis. By using Roofline metrics, we can make a direct comparison of programming models on the same GPU architecture or analyze metrics across architectures and programming models. Performance, measured in FLOPs, demonstrated that **bricks codegen** attained the highest performance overall on all three GPUs Additionally, bytes access demonstrated that bricks' data movement is comparable to the theoretical lower bound on CUDA and SYCL on NVIDIA A100, HIP and SYCL on AMD MI250X, and SYCL on Intel PVC. Among architectures, the fraction of Roofline can be analyzed as a metric for performance portability, where **bricks codegen** attained over 60% when averaged across all platforms and programming models. We also demonstrate the use of fraction of theoretical arithmetic intensity as a new performance portability metric, especially as stencil computations are mostly memory-bound operations, where **bricks codegen** also attained nearly 70% when averaged across all platforms and programming models. By merging performance metrics, such as fraction of the Roofline and fraction of theoretical arithmetic intensity, we could plot potential speed-up for an application by illustrating the same application on different architectures and programming models.

The results presented in this paper are encouraging, as fine-grain data blocking and vector code generators can improve the performance and data movement of stencil computations independently of the GPU architecture or the programming model. Moreover, correlation plots and performance portability metrics are fundamental tools to analyze and compare performance or data movement between GPUs or among programming models.

ACKNOWLEDGMENTS

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, and we gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory.

A ARTIFACT DESCRIPTION

A.1 Abstract

The paper focuses on the performance portability evaluation of blocked stencils on GPUs, as well as, different programming models such as CUDA, HIP and SYCL. Here, we describe the hardware and software used for stencil computations and the methodology

to extract the data needed for the Roofline and correlation plots included in the paper.

Machines: Results presented in this paper were obtained on a NVIDIA A100 GPU on Perlmutter at NERSC, AMD MI250X GPU on Crusher at OLCF and Intel PVC GPU on Florentia at JLSE. In all experiments, we use only a single process running on one GPU.

Application: We evaluated blocked stencil computations using BrickLib. The source code is available at https://github.com/OscarAntepara/bricklib/tree/artifact_sc23/ and can be cloned using:

```
git clone -b artifact_sc23
https://github.com/OscarAntepara/bricklib.git
```

B REPRODUCIBILITY OF EXPERIMENTS

We evaluated blocked stencil computations using BrickLib. The experiments source code is in the `examples/perf_port` directory. The `README` file in `examples/perf_port` contains all the instructions needed to compile and run the experiments on Perlmutter-NERSC, Crusher-OLCF and Florentia-JLSE. All the experiments are for star (radius 1, 2, 3, 4) and cube (radius 1, 2) stencils on a 512^3 domain. At configuration/compilation time, user must select radius and architecture via `-DSTENCIL_RADIUS` and `-DCODEGEN_ARCH`. Radius one and NVIDIA architecture are the default values, radius values are between 1 and 4, and architectures could be NVIDIA, AMD or INTEL. At run time, user can select to execute the experiment with star or cube stencil by adding "`<exe> -s star`" or "`<exe> -s cube`" as parameter. Each experiment has an execution time of a couple of seconds and the output will show the execution time in seconds for each stencil technology presented in the paper.

C GPU PROFILING AND DATA COLLECTION

In this section we describe the command lines given to the profiler to gather GPU metrics for the data on the Roofline and correlation figures described in the paper.

C.1 Perlmutter-NERSC

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

```
nv-nsight-cu-cli -k <kernel_name> --metrics
"sm__sass_thread_inst_executed_op_dfma_pred_on.sum,
sm__sass_thread_inst_executed_op_dadd_pred_on.sum,
sm__sass_thread_inst_executed_op_dmul_pred_on.sum,
sm__cycles_elapsed.avg,sm__cycles_elapsed.avg.per_second,
dram__bytes.sum" <exe> <params>
```

To compute FLOP count , the formula is:

```
FLOP=(sm__sass_thread_inst_executed_op_dadd_pred_on.sum +
sm__sass_thread_inst_executed_op_dmul_pred_on.sum + 2*
sm__sass_thread_inst_executed_op_dfma_pred_on.sum)
```

Time is calculated in nanoseconds as,

```
Time = (sm__cycles_elapsed.avg / sm__cycles_elapsed.avg.per_second)
```

Then, performance in *FLOPs* is computed as $FLOP/Time/1e9$ and arithmetic intensity $AI = FLOP/dram\_\_bytes.sum$. Another NVIDIA profiler to get the Roofline plots, and L1 data movement is NVIDIA Nsight Compute by using the next command line:

```
srun -n 1 ncu -o output_file --set full <exe> <params>
```

where results in the $output\_file$ can be visualized using the NVIDIA Nsight Compute API on your local machine. Examples of the command lines to collect GPU metrics for the experiments on Perlmutter-NERSC are in the `README` file in *examples/perf_port*.

C.2 Crusher-OLCF

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

```
rocprof -i input.txt --timestamp on -o out.csv <exe> <params>
```

AMD ROCm Profiler needs an input file with the kernel name and the metrics to be collected.

Inputs file for the experiments are in `examples/perf_port` directory, specifically, the files `prof_star.txt` and `prof_cube.txt` to gather GPU metrics for star and cube stencils, respectively. To compute Roofline metrics needed as FLOP count, time, FLOPs and Bytes moved, follow the instructions described here: https://docs.olcf.ornl.gov/systems/crusher_quick_start_guide.html#roofline-profiling-with-the-rocm-profiler. Another alternative to gather GPU metrics as FLOP count, arithmetic intensity or L1 data movement is by using omniperf. To gather GPU metrics and profile your application, the terminal command is:

```
omniperf profile -n output_name -- <exec> <params>
```

To visualize the results, as Roofline plots or memory hierarchy metrics per each kernel, the gui API was used as:

```
omniperf analyze -p workloads/output_name/mi200/ --gui
```

C.3 Florentia-JLSE

Intel provides several tools for GPU profiling. In this work, we have used Intel Advisor as the GPU metrics collector for Roofline data and L1 data movement. 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 stack. Intel Advisor will collect the information in a directory that will be needed to create a 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 Roofline plots for your kernels, as well as, general information of data movement across the memory hierarchy, FLOP count, instructions executed, etc.

REFERENCES

[1] ALCF. 2023. Aurora. https://www.alcf.anl.gov/aurora.

[2] AMD. 2022. AMD CDNA 2 ARCHITECTURE. https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf.

[3] AMD. 2023. AMD rocProf Documentation. https://docs.amd.com/projects/rocprofiler/en/docs-5.1.0/rocprof.html.

[4] Mauricio Araya-Polo, Félix Rubio, Raúl de la Cruz, Mauricio Hanzich, José María Cela, and Daniele Paolo Scarpazza. 2009. 3D Seismic Imaging Through Reverse-time Migration on Homogeneous and Heterogeneous Multi-core Processors. *Sci. Program.* 17, 1-2 (Jan. 2009), 185–198. https://doi.org/10.1155/2009/382638

[5] Protonu Basu, Mary Hall, Samuel Williams, Brian Van Straalen, Leonid Oliker, and Phillip Colella. 2015. Compiler-directed transformation for higher-order stencils. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 313–323.

[6] Brick Library 2021. BrickLib Documentation. https://bricks.run/.

[7] M. Christen, O. Schenk, and H. Burkhart. 2011. PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern

Microarchitectures. In *Parallel Distributed Processing Symposium (IPDPS)*. https://doi.org/10.1109/IPDPS.2011.70

[8] Kaushik Datta. 2009. *Auto-tuning Stencil Codes for Cache-Based Multicore Platforms*. Ph. D. Dissertation. EECS Department, University of California, Berkeley.

[9] Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. 2009. Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors. *SIAM Rev.* 51, 1 (2009), 129–159.

[10] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. 2008. Stencil Computation Optimization and Auto-Tuning on State-of-the-art Multicore Architectures. In *Supercomputing (SC)*.

[11] Raúl De La Cruz, Mauricio Araya-Polo, and José María Cela. 2010. Introducing the Semi-stencil Algorithm. In *International Conference on Parallel Processing and Applied Mathematics: Part I (PPAM)*. 11 pages.

[12] Tom Deakin, Simon McIntosh-Smith, James Price, Andrei Poenaru, Patrick Atkinson, Codrin Popa, and Justin Salmon. 2019. Performance Portability across Diverse Computer Architectures. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 1–13. https://doi.org/10.1109/P3HPC49587.2019.00006

[13] Steven J Deitz, Bradford L Chamberlain, and Lawrence Snyder. 2001. Eliminating redundancies in sum-of-product array computations. In *Proceedings of the 15th international conference on Supercomputing*. ACM, 65–77.

[14] Craig C. Douglas, Jonathan Hu, Markus Kowarschik, Ulrich Rüde, and Christian Weiss. 2000. Cache Optimization for Structured and Unstructured Grid Multigrid. *Elect. Trans. Numer. Anal* 10 (2000), 21–40.

[15] Amanda S. Dufek, Rahulkumar Gayatri, Neil Mehta, Douglas Doerfler, Brandon Cook, Yasaman Ghadar, and Carleton DeTar. 2021. Case Study of Using Kokkos and SYCL as Performance-Portable Frameworks for Milc-Dslash Benchmark on NVIDIA, AMD and Intel GPUs. In *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 57–67. https://doi.org/10.1109/P3HPC54578.2021.00009

[16] M. Frigo and V. Strumpen. 2005. Evaluation of cache-based superscalar and cacheless vector architectures for scientific computations. In *Proc. ACM International Conference on Supercomputing (ICS)*.

[17] Tom Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J Ramanujam, and P Sadayappan. 2011. Data layout transformation for stencil computations on short-vector simd architectures. In *Compiler Construction*. Springer, 225–245.

[18] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. 2012. High-performance code generation for stencil computations on GPU architectures. In *International Conference on Supercomputing (ICS)*.

[19] Khaled Z. Ibrahim, Chao Yang, and Pieter Maris. 2022. Performance Portability of Sparse Block Diagonal Matrix Multiple Vector Multiplications on GPUs. In *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 58–67. https://doi.org/10.1109/P3HPC56579.2022.00011

[20] INTEL. 2023. Intel Advisor tool on Florentia-JLSE. https://www.intel.com/content/www/us/en/developer/tools/oneapi/advisor.html.

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

[22] INTEL. 2023. Intel oneAPI on Florentia-JLSE. https://software.intel.com/ONEAPI.

[23] Jagan Jayaraj. 2013. *A strategy for high performance in computational fluid dynamics*. Ph. D. Dissertation. University of Minnesota.

[24] JLSE. 2023. JLSE: Florentia GPU Nodes. https://www.jlse.anl.gov/hardware-under-development

[25] Elias Konstantinidis and Yiannis Cotronis. 2017. A quantitative roofline model for GPU kernel performance estimation using micro-benchmarks and hardware metric profiling. *J. Parallel and Distrib. Comput.* 107 (2017), 37–56. https://doi.org/10.1016/j.jpdc.2017.04.002

[26] Markus Kowarschik and Christian Weiß. 2001. DiMEPACK - A Cache-Optimized Multigrid Library. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), volume I*.

[27] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. 2007. Effective automatic parallelization of stencil computations. In *Proc. ACM SIGPLAN conference on Programming language design and implementation (PLDI)*.

[28] JaeHyuk Kwack, John Tramm, Colleen Bertoni, Yasaman Ghadar, Brian Homerding, Esteban Rangel, Christopher Knight, and Scott Parker. 2021. Evaluation of Performance Portability of Applications and Mini-Apps across AMD, Intel and NVIDIA GPUs. In *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 45–56. https://doi.org/10.1109/P3HPC54578.2021.00008

[29] Seyong Lee, John Gounley, Amanda Randles, and Jeffrey S. Vetter. 2019. Performance portability study for massively parallel computational fluid dynamics application on scalable heterogeneous architectures. *J. Parallel and Distrib. Comput.* 129 (2019), 1–13. https://doi.org/10.1016/j.jpdc.2019.02.005

[30] Xiaomin Lu, Cole Ramos, Fei Zheng, Karl W. Schulz, Jose Santos, Keith Lowery, and Cristian Di Pietrantonio. 2023. *AMDResearch/omniperf: v1.0.8 (30 May 2023)*. https://doi.org/10.5281/zenodo.7314631

[31] Fabio Luporini, Mathias Louboutin, Michael Lange, Navjot Kukreja, Philipp Witte, Jan Hückelheim, Charles Yount, Paul H. J. Kelly, Felix J. Herrmann, and Gerard J. Gorman. 2020. Architecture and Performance of Devito, a System for Automated Stencil Computation. *ACM Trans. Math. Softw.* 46, 1, Article 6 (apr 2020), 28 pages. https://doi.org/10.1145/3374916

[32] J. McCalpin and D. Wonnacott. 1999. *Time skewing: A value-based approach to optimizing for memory locality*. Technical Report DCS-TR-379. Department of Computer Science, Rutgers University.

[33] Neil A. Mehta, Rahulkumar Gayatri, Yasaman Ghadar, Christopher Knight, and Jack Deslippe. 2021. Evaluating Performance Portability of OpenMP for SNAP on NVIDIA, Intel, and AMD GPUs Using the Roofline Methodology. In *Accelerator Programming Using Directives*, Sridutt Bhalachandra, Sandra Wienke, Sunita Chandrasekaran, and Guido Juckeland (Eds.). Springer International Publishing, Cham, 3–24.

[34] Paulius Micikevicius. 2009. 3D Finite Difference Computation on GPUs Using CUDA. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-2)*. 6 pages.

[35] NERSC. 2022. NERSC: Perlmutter GPU Nodes. https://docs.nersc.gov/systems/perlmutter/

[36] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 2010. 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)*.

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

[38] NVIDIA. 2023. NVIDIA Nsight Compute CLI Documentation. https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html.

[39] NVIDIA. 2023. NVIDIA Nsight Documentation. https://docs.nvidia.com/nsight-systems/UserGuide/index.html.

[40] OLCF. 2023. OLCF: Crusher GPU Nodes. https://docs.olcf.ornl.gov/systems/crusher_quick_start_guide.html

[41] oneAPI DPC++ Compiler 2022. DPC++ on Crusher-OLCF. https://github.com/intel/llvm/releases/tag/2022-09.

[42] M. Emin Ozturk, Omid Asudeh, Gerald Sabin, P. Sadayappan, and Aravind Sukumaran-Rajam. 2023. A Performance Portability Study Using Tensor Contraction Benchmarks. In *AsHES 2023: The Thirteenth International Workshop on Accelerators and Hybrid Exascale Systems (to appear)*. 37th IEEE International Parallel and Distributed Processing Symposium (IPDPS).

[43] S.J. Pennycook, J.D. Sewall, and V.W. Lee. 2019. Implications of a metric for performance portability. *Future Generation Computer Systems* 92 (2019), 947–958. https://doi.org/10.1016/j.future.2017.08.007

[44] Prashant Singh Rawat, Aravind Sukumaran-Rajam, Atanas Rountev, Fabrice Rastello, Louis-Noël Pouchet, and P. Sadayappan. 2018. Associative Instruction Reordering to Alleviate Register Pressure. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Dallas, Texas) *(SC '18)*. IEEE Press, Piscataway, NJ, USA, Article 46, 13 pages. http://dl.acm.org/citation.cfm?id=3291656.3291718

[45] G. Rivera and C. Tseng. 2000. Tiling Optimizations for 3D Scientific Computations. In *Supercomputing (SC)*.

[46] S. Sellappa and S. Chatterjee. 2004. Cache-Efficient Multigrid Algorithms. *International Journal of High Performance Computing Applications* 18, 1 (2004), 115–133.

[47] Y. Song and Z. Li. 1999. New tiling techniques to improve cache temporal locality. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[48] Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2014. A framework for enhancing data reuse via associative reordering. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 65–76.

[49] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. 2011. The pochoir stencil compiler. In *ACM symposium on Parallelism in algorithms and architectures*.

[50] TOP 500. 2023. TOP 500 website. https://www.top500.org/.

[51] Didem Unat, Anshu Dubey, Torsten Hoefler, John Shalf, Mark Abraham, Mauro Bianco, Bradford L. Chamberlain, Romain Cledat, H. Carter Edwards, Hal Finkel, Karl Fuerlinger, Frank Hannig, Emmanuel Jeannot, Amir Kamil, Jeff Keasler, Paul H J Kelly, Vitus Leung, Hatem Ltaief, Naoya Maruyama, Chris J. Newburn, and Miquel Pericás. 2017. Trends in Data Locality Abstractions for HPC Systems. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (2017), 3007–3020. https://doi.org/10.1109/TPDS.2017.2703149

[52] Didem Unat, Tan Nguyen, Weiqun Zhang, Muhammed Nufail Farooqi, Burak Bastem, George Michelogiannakis, Ann Almgren, and John Shalf. 2016. *TiDA: High-Level Programming Abstractions for Data Locality Management*. Springer International Publishing, Cham, 116–135.

[53] Gerhard Wellein, Georg Hager, Thomas Zeiser, Markus Wittmann, and Holger Fehske. 2009. Efficient Temporal Blocking for Stencil Computations by Multicore-Aware Wavefront Parallelization. In *International Computer Software*

and Applications Conference. https://doi.org/10.1109/COMPSAC.2009.82

[54] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick. 2008. Lattice Boltzmann Simulation Optimization on Leading Multicore Platforms. In *Interational Conference on Parallel and Distributed Computing Systems (IPDPS)*.

[55] Samuel Williams, Leonid Oliker, Jonathan Carter, and John Shalf. 2011. Extracting ultra-scale Lattice Boltzmann performance via hierarchical and distributed autotuning. In *Supercomputing (SC)*.

[56] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. 2006. The potential of the Cell processor for scientific computing. In *Proc. Conference on Computing Frontiers*.

[57] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (apr 2009), 65–76. https://doi.org/10.1145/1498765.1498785

[58] D. Wonnacott. 2000. Using Time Skewing to Eliminate Idle Time due to Memory Bandwidth and Network Limitations. In *Proc. Interational Conference on Parallel and Distributed Computing Systems*.

[59] C. Yount. 2015. 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*. 865–870.

[60] Charles Yount, Josh Tobin, Alexander Breuer, and Alejandro Duran. 2016. 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* (Salt Lake City, Utah) *(WOLFHPC '16)*. 10 pages.

[61] T. Zeiser, G. Wellein, A. Nitsure, K. Iglberger, U. Rude, and G. Hager. 2008. Introducing a parallel cache oblivious blocking approach for the lattice Boltzmann method. *Progress in Computational Fluid Dynamics* 8 (2008).

[62] N. Zhang, M. Driscoll, C. Markley, S. Williams, P. Basu, and A. Fox. 2017. Snowflake: A Lightweight Portable Stencil DSL. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 795–804.

[63] Yongpeng Zhang and Frank Mueller. 2012. Auto-generation and auto-tuning of 3D stencil codes on GPU clusters. In *International Symposium on Code Generation and Optimization (CGO)*.

[64] Tuowen Zhao, Protonu Basu, Samuel Williams, Mary Hall, and Hans Johansen. 2019. 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* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 52, 44 pages. https://doi.org/10.1145/3295500.3356210

[65] Tuowen Zhao, Samuel Williams, Mary Hall, and Hans Johansen. 2018. 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)*. 59–70. https://doi.org/10.1109/P3HPC.2018.00009

[66] Xing Zhou, Jean-Pierre Giacalone, María Jesús Garzarán, Robert H. Kuhn, Yang Ni, and David Padua. 2012. Hierarchical overlapped tiling. In *Proc. International Symposium on Code Generation and Optimization (CGO)*.