Performance Evaluation of Heterogeneous GPU Programming Frameworks for Hemodynamic Simulations

Aristotle Martin
Biomedical Engineering
Duke University
Durham, NC, USA
aristotle.martin@duke.edu

Geng Liu
Leadership Computing Facility
Argonne National Laboratory
Lemont, IL, USA
gliu@anl.gov

William Ladd
Biomedical Engineering
Duke University
Durham, NC, USA
william.ladd@duke.edu

Seyong Lee
Computer Science and Mathematics
Oak Ridge National Laboratory
Oak Ridge, TN, USA
lees2@ornl.gov

John Gounley
Computational Sciences and Engineering
Oak Ridge National Laboratory
Oak Ridge, TN, USA
gounleyjp@ornl.gov

Jeffrey Vetter
Computer Science and Mathematics
Oak Ridge National Laboratory
Oak Ridge, TN, USA
vetter@ornl.gov

Saumil Patel
Leadership Computing Facility
Argonne National Laboratory
Lemont, IL, USA
spatel@anl.gov

Silvio Rizzi
Leadership Computing Facility
Argonne National Laboratory
Lemont, IL, USA
srizzi@alcf.anl.gov

Victor Mateevitsi
Leadership Computing Facility
Argonne National Laboratory
Lemont, IL, USA
vmateevitsi@anl.gov

Joseph Insley
Leadership Computing Facility
Argonne National Laboratory
Lemont, IL, USA
insley@anl.gov

Amanda Randles
Biomedical Engineering
Duke University
Durham, NC, USA
amanda.randles@duke.edu

ABSTRACT

Preparing for the deployment of large scientific and engineering codes on upcoming exascale systems with GPU-dense nodes is made challenging by the unprecedented diversity of device architectures and heterogeneous programming models. In this work, we evaluate the process of porting a massively parallel, fluid dynamics code written in CUDA to SYCL, HIP, and Kokkos with a range of backends, using a combination of automated tools and manual tuning. We use a proxy application along with a custom performance model to inform the results and identify additional optimization strategies. At scale performance of the programming model implementations are evaluated on pre-production GPU node architectures for Frontier and Aurora, as well as on current NVIDIA device-based systems Summit and Polaris. Real-world workloads representing 3D blood flow calculations in complex vasculature are assessed. Our analysis highlights critical trade-offs between code performance, portability, and development time.

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

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

CCS CONCEPTS

• **Computing methodologies** → **Parallel programming languages**; *Massively parallel and high-performance simulations*; • **Hardware** → **Emerging architectures**.

KEYWORDS

Performance portability, Proxy applications, Computational fluid dynamics

ACM Reference Format:
Aristotle Martin, Geng Liu, William Ladd, Seyong Lee, John Gounley, Jeffrey Vetter, Saumil Patel, Silvio Rizzi, Victor Mateevitsi, Joseph Insley, and Amanda Randles. 2023. Performance Evaluation of Heterogeneous GPU Programming Frameworks for Hemodynamic Simulations. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (SC-W 2023).* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The rise of heterogeneous architectures in supercomputing platforms (e.g., Aurora, Summit, and Frontier) makes achieving performance portability across device types critical. The transition from predominantly NVIDIA GPU-based platforms of previous generations to new systems built around Intel (Aurora) and AMD (Frontier) GPUs means that HPC users will have to translate kernels written in CUDA to alternative programming languages supported on these devices. Vendors have developed software infrastructure

centered around different programming models and associated compilers tailored to their hardware. For instance, Intel has created the oneAPI software ecosystem, which includes the Data Parallel C++ (DPC++) compiler derived from the SYCL standard. Codes targeting Frontier GPUs will interact with AMD's ROCm programming environment through the Heterogeneous Interface for Portability (HIP). Users will also be able to choose from alternative heterogeneous programming frameworks such as Kokkos, which includes an ever-growing list of supported backends that includes CUDA, HIP, and SYCL, among others. Faced with the many choices of programming models and device architectures, developers of high performance computing (HPC) applications will have to decide how best to write code to maintain longevity, high performance, and portability. Selecting a programming model for a given application and hardware specification requires a detailed comparative analysis. The use of automated porting tools, proxy applications, and predictive performance models facilitates this iterative procedure (shown in Fig. 1) .

In light of the growing demand for GPU-accelerated supercomputing platforms, there is an urgent need for effective strategies for porting existing codebases to these platforms. In this paper, we address this challenge by systematically evaluating the SYCL, HIP, and Kokkos programming models on both current and upcoming GPU-centric supercomputing platforms. By analyzing the porting procedure and the resulting performance, we identify the trade-offs involved in porting a real-world, CUDA-based code for a production workflow. Our results demonstrate the strengths and weaknesses of each programming model and hardware configuration and provide insights into the most effective strategies for achieving performance portability in a range of computing environments. The primary contributions of this work are as follows.

(1) The development of a custom GPU performance model capable of predicting upper bounds on application performance for different problem sizes on any given node architecture
(2) Comparative analysis of scaling performance of SYCL, HIP, and Kokkos programming models on pre-production GPU node architectures
(3) Analyzing trade-offs between performance portability and porting times for the SYCL, HIP, and Kokkos programming models on different GPU architectures
(4) Insights into the applicability of automated tools for porting legacy CUDA codes to SYCL and HIP programming models
(5) Identification of the benefits and limitations of each programming model
(6) Evaluation of the impact of hardware architecture on the choice of programming model and code performance

In summary, this study makes important contributions to the field of GPU-accelerated computing by providing practical guidance for developers seeking to optimize code for a diverse range of GPU-centric supercomputing platforms. Ultimately, this study provides a roadmap for achieving strong performance portability across a range of GPU-accelerated computing environments, with important implications for the broader field of high-performance computing.

2 RELATED WORK

Recent studies have investigated the performance portability of various programming models on a variety of accelerator devices, including those from AMD, NVIDIA, and Intel [5, 10, 11, 14, 15]. Other experiences with using automated tools such as Intel's DPCT or AMD's HIPify to port legacy CUDA codes have been documented by multiple study authors [1, 3, 11]. However, there are some limitations to these previous works that are overcome in the present study. First, previous works have typically looked only at benchmark programs or mini-apps, which often do not exhibit the same behavior as a full-scale application. Second, while multiple studies have performed comparisons involving CUDA, HIP, SYCL, or Kokkos, to our knowledge, there has not been a study examining them all together on exascale hardware. Previous works that included Intel devices in their comparisons have used integrated graphics cards such as the Intel UHD Graphics P630, whereas in this study we show detailed performance results of a real-world application using Intel's forthcoming Ponte Vecchio (PVC) GPUs that will be used in the upcoming Aurora supercomputer.

3 APPLICATION OVERVIEW

This work evaluates performance portability using HARVEY [19], a massively parallel blood flow (hemodynamic) simulation software based on the lattice Boltzmann method (LBM) for fluid dynamics [20]. Advantages of LBM over other numerical solvers of the Navier-Stokes equations include its amenability to parallelization due to its underlying stencil structure and the local availability of physical quantities, eliminating the need for global communication among processors required of Poisson solvers [18].

The LBM models a fluid by tracking fictitious particles that represent probability distributions in the velocity space on a lattice grid structure. The algorithm consists of two main steps, *collision* and *streaming*. The collision step is a local operation, whereas streaming involves transferring particle populations between neighboring nodes in the lattice.

3.1 Simulation Inputs

To evaluate the performance of the different programming models, we performed simulations using HARVEY on two different input geometries. The first is a cylinder, which we use as a benchmark against our proxy application (described in the next section, see Fig. 2b for a description of the geometry). The second is a patient-derived aorta, which we selected to represent a real-world workload (as shown in Fig. 2a).

3.2 Proxy Application

Here we use an open source proxy application based on the LBM [17]. This code was developed to explore the performance-limiting aspects of HARVEY in a simplified environment that facilitates rapid prototyping on new systems and helps gauge the performance bounds of the full-scale application. We define performance as millions of fluid lattice updates per second (MFLUPS), which is a representative performance measure for LBM-based codes [19].

The proxy application solves a cylindrical channel flow problem depicted in Fig. 2b, characterized by an axial length of $84x$ and a radius of $8x$, where x is a scale factor specified by the user. The fluid

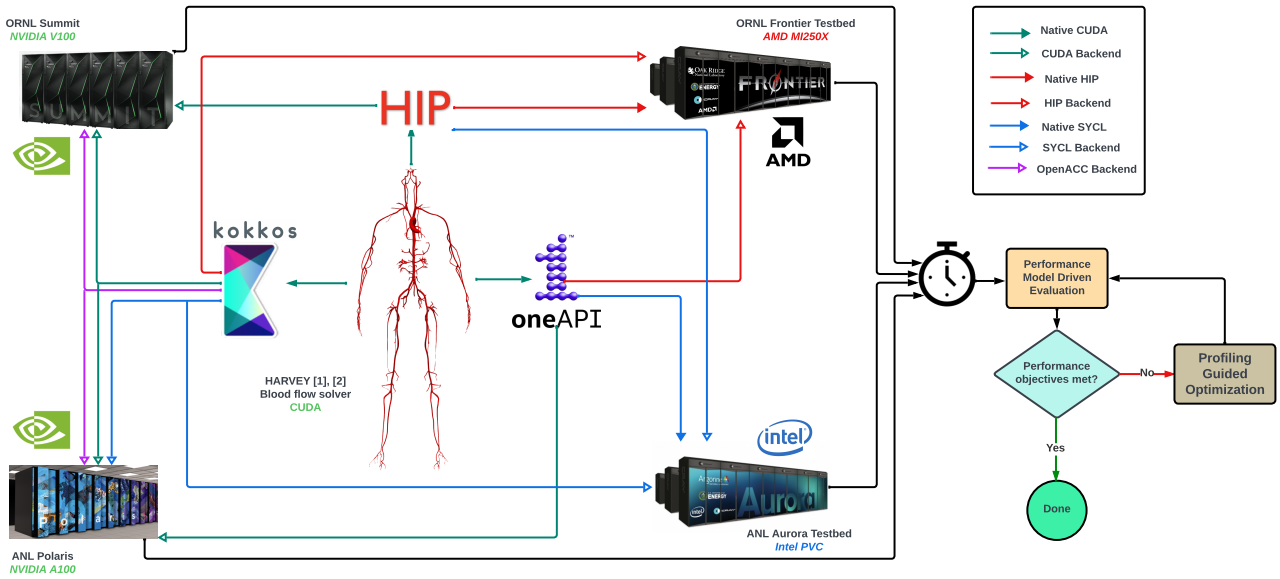

Figure 1: Overview of the performance portability study, which aims to evaluate the effectiveness of different programming models and hardware configurations for a real workload. This diagram illustrates the diverse range of programming models and hardware platforms investigated, as well as the iterative optimization process.

.

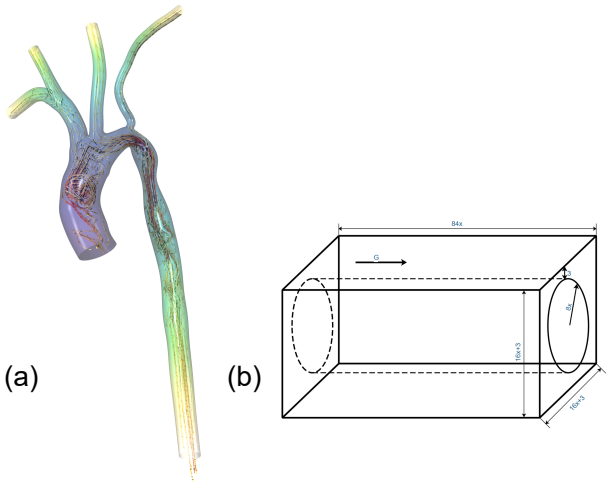

Figure 2: Geometries used for performance testing. (a) An image-derived geometry of a human aorta provides a realistic, pulsatile hemodynamic workflow for testing. Shown here are representative results with the domain shaded by pressure and streamlines indicating flow paths. (b) A cylindrical domain is used as an idealized test case for direct comparison between the LBM proxy app and production code.

inside the cylinder is computed and the nodal bounce is applied to the points of the channel wall [2].

4 HARDWARE OVERVIEW

The performance of the HARVEY and LBM proxy app codes is evaluated on several supercomputing systems, including Sunspot (Argonne National Laboratory), Crusher (Oak Ridge National Laboratory), Polaris (Argonne National Laboratory), and Summit (Oak Ridge National Laboratory). Sunspot is a Test and Development System for the upcoming Aurora system, with 128 nodes that each contain two 52-core Intel Xeon CPU Max Series (codename Sapphire Rapids) and six Intel Data Center GPU Max Series (codenamed Ponte Vecchio or PVC). Each PVC consists of two tiles that can be treated as subdevices and bound by individual MPI ranks, giving a total of 12 tiles (GPUs) per node. Crusher is an early-access testbed for the Frontier system, with 128 compute nodes that each have a single 64-core AMD EPYC 7A53 CPU and four AMD MI250X, each with two Graphics Compute Dies (GCDs) which act as separate GPUs, making for eight logical GPUs per node. Polaris is a system with 560 nodes based on the HPE Apollo 6500 Gen 10+ system. Each node has a single 2.8 GHz AMD EPYC Milan 7543P 32-core CPU with 512 GB of DDR4 RAM and four NVIDIA A100 GPUs connected via NVLink. Summit is an IBM system with 4,600 nodes, and each Summit node has two IBM POWER9 22-core CPUs and 6 NVIDIA V100 GPUs connected via NVLink. See Table 1 for a summary of all hardware specifications.

5 PROGRAMMING MODELS

Below we outline the programming models and frameworks employed in this study.

Table 1: System node characteristics. *Reported GPU memory bandwidth collected using BabelStream.

System	Sunspot	Crusher	Polaris	Summit
CPU	2x Xeon Max	1x EPYC 7A53	1x EPYC 7543P	2x POWER9
Cores/CPU	52	64	32	21
GPU	12x PVC Tiles (6 GPUs)	8x MI250X GCDs (4 GPUs)	4x A100 GPUs	6x V100 GPUs
GPU Memory	64 GB	64 GB	40 GB	16 GB
GPU Mem. Bandwidth*	0.997 TB/s	1.28 TB/s	1.30 TB/s	0.770 TB/s
GPU-CPU Interface	PCIe Gen5 (128 GB/s) [8]	Infinity Fabric CPU-GPU (72 GB/s)	NVLink (64 GB/s)	NVLink (50 GB/s)
Interconnect	Slingshot 11 (25 GB/s) [7]	4x HPE Slingshot (100 GB/s) [9]	Slingshot (25 GB/s) [6]	IB (25 GB/s)

5.1 CUDA

The CUDA programming model has almost become synonymous with GPU programming due to the ubiquity of NVIDIA devices on HPC platforms. With CUDA, users write data parallel kernels targeting NVIDIA GPUs, which are then launched on grids of thread blocks having user-defined dimensions. In traditional CUDA programming, device allocations referenced by C-style pointers are managed explicitly by the user and passed to device kernels. Since the introduction of CUDA 6.0, programmers can forgo explicit memory copies through the use of managed or unified memory allocations. The device code is compiled by a dedicated CUDA C compiler (NVCC). This allows for fine-grained control over the GPU's hardware resources, enabling developers to optimize their codes for the specific architecture of the NVIDIA GPU. In this study, our CUDA codes were executed on the Summit (NVIDIA V100) and Polaris (NVIDIA A100) systems.

5.2 SYCL

SYCL (from the Khronos SYCL standard) is a single source, heterogeneous programming model based on modern C++, designed for offload acceleration. In SYCL, kernels and data transfers are submitted to queues, which provide a concurrency mechanism similar to the function of CUDA streams. Kernels are defined using lambdas or functors and executed over workgroups that serve a purpose analogous to CUDA thread blocks. SYCL memory abstractions include unified shared memory (USM) and buffers. Buffers provide an abstract view of memory and are accessed indirectly through accessor objects in the host or device code. USM is pointer-based and is more familiar to users of CUDA C++. USM space arrays are allocated through SYCL functions and can be explicitly or implicitly managed by the user through host-device transfers or runtime management. The DPC++ compiler is Intel's implementation of SYCL, bundled with the Intel oneAPI toolkit, with support for CPUs, GPUs, and FPGAs, including a few extensions. Simulations using the SYCL implementations of HARVEY and the LBM proxy app were conducted on the Sunspot (Intel PVC), Polaris (NVIDIA A100), and Crusher (AMD MI250X) systems.

5.3 HIP

The HIP programming model is a C++ runtime API and programming language developed by AMD that shares a similar syntax to CUDA. Like CUDA, HIP launches kernels over grids of threads divided into thread blocks. Like SYCL, HIP is designed with portability in mind and is able to run on other architectures including NVIDIA platforms. Additionally, there are active projects currently underway to extend support for other backends, with a notable example being chipStar [23], an LLVM-based compiler with limited support for running HIP (and CUDA) on platforms that support SPIR-V as the device intermediate language, which includes Aurora hardware. In this work, performance runs of the HIP implementations were carried out on Crusher (AMD MI250X), Summit (NVIDIA V100), and Sunspot (Intel PVC).

5.4 Kokkos

Kokkos is a C++ library developed by Sandia National Laboratories to provide performance portability for scientific applications on heterogeneous HPC platforms. The library utilizes the Kokkos Views abstraction to automatically manage platform-dependent allocations and access patterns of device arrays. In the Kokkos programming model, a kernel is defined with a C++ functor that specifies a parallel pattern and execution policy, together with the body that contains the computational work to be carried out by parallel units.

Kokkos is attractive from a code portability perspective due to its higher-level abstractions, such as Kokkos Views, which make it easier to write portable code that can be executed on different architectures without significant modifications. Kokkos has built-in support for various backends, including CUDA, SYCL, HIP, and OpenMPTarget, allowing it to target other accelerators with the same application code. Each backend provides a unique set of features and capabilities, making it possible to select an optimal backend for a given platform. Ideally, users only need to maintain a single compilation base. They can pass in the appropriate switches to the build tool (e.g., CMake) to enable any given backend to build the program on a specific platform. Although sorting out the compiler details for multiple Kokkos backends is not trivial, it is more manageable than maintaining different code versions written with other offload acceleration languages. Within the context of this study, the Kokkos implementation was executed across all systems, among the CUDA (Summit, Polaris), SYCL (Sunspot), HIP (Crusher), and OpenACC (Summit, Polaris) backends deployed.

6 GPU PERFORMANCE MODEL

To facilitate evaluation of the performance of HARVEY and the LBM proxy application, we extend a performance model we previously developed in [16] to predict the optimal iteration time. Since LBM is memory-bandwidth-bound [19, 21, 22], we approximate the time required for a processor to process fluid points, streamcollide time

$t_{streamcollide}$, by dividing the number of bytes needed to process those fluid points by the memory bandwidth of the processor, equation 1. Unlike [16] which used individual CPU cores, for GPUs we model the processor as either the entire GPU or sub-device (logical GPU) of the discrete GPU, depending on the unit we measure memory bandwidth in. Then, we measure the memory bandwidth B_{mem} for the GPU or sub-device using the BabelSTREAM benchmark [4].

$$t_{streamcollide} = \frac{n_{bytes}}{B_{mem}} \tag{1}$$

When multiple processors are used to run an LBM simulation, halo exchanges between the processors, which adds to the runtime, are required. To account for this, we adapted the PingPong benchmark [13] to time communication between GPUs and memory transfers between the GPU and the CPU, considering all communication events for all message sizes so that we can sum all communication times and add them to the time of stream-collide as in Equation 2.

$$t = t_{streamcollide} + \sum_{j}^{n_{events}} t_{comm_j} \tag{2}$$

Predicting runtime with this model requires approximating the number of fluid points on each processor (and by extension the number of bytes n_{bytes}) and halo exchanges between each of the processors (byte size of communication event j leading to a communication time t_{comm_j}). We estimate the number of fluid points as the size of the problem domain, with increases during scaling as approximated by [16], and assume that each processor's subdomain is a perfect cube stacked together to form a box domain with minimal edge length. Using this same idealized cubic subdomain, we take the communication surface area as double the maximum area of the halo exchange boundary SA_{comm}, using the relation between volume (n_{bytes}) and the surface area given in equation 3 , as bytes are in an event sent away to another GPU and in another event received from another GPU.

$$SA_{comm} \approx w \times V^{\frac{2}{3}} \tag{3}$$

For low GPU counts, the cube with the maximum communication surface area will only have some, not all, of its faces used for communicating halo exchanges therefore we correct for this using equation 4.

$$w = 2 \times \min(\log_2(n_{GPUs}), 6) \tag{4}$$

To compare results independent of problem size and geometry, we use the MFLUPS units to quantify performance, as these are pure fluid simulations.

7 EVALUATION OF PORTING PROCESS

7.1 Porting to DPC++ with DPCT

To port the base HARVEY CUDA source code and LBM proxy app to DPC++, we used Intel's Data Parallel C++ Compatibility Tool (DPCT), part of the Intel oneAPI toolkit. Prior to porting, DPCT requires information about how source files are compiled. For simple test codes, this can be done directly through the command line utility. However, for larger Makefile based projects like HARVEY, a compilation database is needed. Intel provides an intercept-build script that automatically tracks and saves the compilation commands into a JSON file.

Table 2: DPCT Warning Breakdown

Category	Frequency(%)
Error handling	80.45
Unsupported feature	2.26
Functional equivalence	0.75
Kernel invocation	15.04
Performance improvement	1.50

The DPCT tool ported the proxy app without any intervention, but some manual tuning was required for HARVEY. DPCT processed 28 source code files, generating 133 warning messages, with the breakdown shown in Table 2. Most of the warnings were associated with error handling. Specifically, SYCL uses exceptions to report errors, whereas CUDA function calls use error codes. The warnings about unsupported features refer to CUDA API calls that do not have equivalent in DPC++.

In some cases, the DPC++ function that replaces a CUDA call may differ from an exact equivalent, as we encountered with a trigonometric function. In addition, the kernel invocation warnings inform the user that the auto-generated work group sizes may need to be adjusted to fit within the device. Lastly, performance improvement warnings are suggestions that may lead to faster code and can be generic or specific.

During compilation of the initial DPC++ HARVEY port, we encountered compiler errors in several places throughout the code. Many of these errors were associated with DPCT's handling of uninitialized dim3 objects, and were resolved by initializing the corresponding SYCL range objects with zeros. After addressing these issues, the DPC++ HARVEY port could successfully run on Intel PVC on Sunspot. To get a working port on Polaris and Crusher, some further changes were made, detailed below.

7.1.1 Polaris. In general, developing codes on moving systems is challenging due to lack of available modules that users of production systems have grown accustomed to. When we initially ported SYCL HARVEY code to Polaris, an earlier version of the open-source SYCL `clang`/`clang++`-based compiler was installed, with no support for the oneAPI libraries our DPCT-generated code relied on, specifically `dpct` and `oneDPL`. Our solution to this was to build the open-source versions of those libraries (e.g., SYCLomatic) from source and integrate them into our build chain. Now users can avoid these steps since oneAPI is installed with support for DPCT and oneDPL modules on Polaris.

7.1.2 Crusher. The open-source DPC++ compiler is installed on Crusher, but DPCT-generated code still requires external libraries, so we built those from source following the same steps outlined for Polaris. Additionally, we encountered compilation errors related to the use of unsupported atomic operations in experimental functions included in the latest build of SYCLomatic that are not supported on the MI250X at this time. As these routines were irrelevant to our HARVEY code, we could get around this by simply commenting out these sections of the `dpct` header. Beyond this, some additional tuning of the compilation chain was required.

7.2 Porting to HIP with HIPify

We used AMD's HIPify tool to port the HARVEY application code to HIP. There are two options for HIPify: HIPify-perl and HIPify-clang. The latter is similar to DPCT in that it uses a compilation database to transform the code with contextual information provided by the compiler. On the other hand, the former is a simple regex script that replaces instances of "cuda" with "hip" throughout the source code. This is made possible by mirroring the HIP API with the CUDA API (e.g., cudaMallocManaged versus hipMallocManaged). We chose HIPify-perl because it is straightforward and requires only a single command to complete the conversion. In our case, HIPify was able to perform the conversion without any errors. Specific details in porting HIP to different platforms are described below.

7.2.1 Crusher. Getting the HIP codes to run on Crusher was straightforward, requiring only the removal of a few CUDA header references without any other necessary changes for functional code.

7.2.2 Summit. While porting the HIP proxy app to Summit was trivial, porting HARVEY to HIP on Summit was slightly more involved compared to Crusher, for two main reasons. First, we encountered compilation errors arising from the use of `__constant__` arguments supplied to `hipMemcpyToSymbol` in certain areas of the code. The workaround was to change the constants to constant arrays of size 1. The second issue was encountered during linking and was due to the supply of files with different extensions to `hipcc`. We note that ROCM's `hipcc` on Crusher did not have this issue. The workaround we went with was to change the Makefile to pass object files to `hipcc` produced from `.cu` source files having `.o` endings instead of `.cu_o`, which we previously used to distinguish device and host source files that have the same name. It is also worth noting that GPU-aware MPI is not supported for HIP codes on Summit, so the code could run only by disabling this feature within HARVEY and the proxy app.

7.2.3 Sunspot. We employed the chipStar compiler (first discussed in Section 5.3) to get our HIP codes running on Intel PVC. We ran into issues with passing differing file extensions to the linker, which was resolved using the same strategy we applied on Summit. We also encountered compilation errors from the use of `hipMemPrefetchAsync`, which is not yet a supported feature of chipStar at this time. We commented out these lines of the code, accepting likely performance degradation as the trade-off for obtaining functional code. It is also worth noting that despite making it easier to get the HIP proxy app to compile on Sunspot, the chipStar compiler generated many warning messages pertaining to kernel arguments upon a successful build.

7.3 Fully Manual Porting to Kokkos

In contrast to the other programming models examined in this study, porting the code from CUDA to Kokkos must be done manually. We ported the proxy app to Kokkos as validation before working on HARVEY. Three significant changes were made during this process: (1) declaring and allocating device arrays are replaced by introducing Kokkos views, (2) data transfer between host and device is achieved by `Kokkos::deep_copy` function, and (3) kernels are launched by `Kokkos::parallel_for` with range policies that manage the execution space and kernel range. Namespace `Kokkos` is hereafter implied.

Kokkos usually automatically selects the device memory space if the environment is set correctly and the hardware is recognized. However, to ensure compatibility, memory spaces and their corresponding kernel range policies are defined as macros in the Kokkos header and are switched according to the user-controlled compiling flags. For example, the device memory space is `CudaSpace` with CUDA backend, and `Experimental::SYCLDeviceUSMSpace` with the SYCL backend. The Kokkos view elements are accessed with parentheses instead of brackets. Unlike porting the proxy app to Kokkos, converting the HARVEY code would be tedious if such minor changes appeared frequently. An alternative way to accessing Kokkos view elements in the device kernels through parentheses is to pass the data pointer to the launch interface as an input parameter. This pointer can be obtained by the `data()` function of Kokkos views (e.g., `distr.data()`). With this mechanism, many existing CUDA kernel bodies are inherited in the Kokkos functors.

In the HARVEY code, global constant device arrays, such as lattice velocities and weights for the distributions, are declared as constant device arrays. The declaration of these constant Kokkos device views is straightforward. However, initializing these views in Kokkos requires an additional step due to the inability to directly use `deep_copy` when the target view has constant elements. To work around this issue, the host view is first copied to an intermediate non-constant device view, and then the constant view is initialized with the non-constant view.

Although porting the proxy app was fairly straightforward, making the large code base of HARVEY work with multiple backends was more challenging. With the CUDA backend, global constant global device views can be accessed in the kernels without being passed through the launch interfaces. However, for other backends, such as SYCL, we had to pass the global views to the kernels. To take advantage of a single Kokkos codebase, passing global constant views explicitly was applied to all kernels since it works with multiple backends.

Ensuring the compatibility of the HARVEY Kokkos code across multiple backends can be a challenge, as more general expressions may not replace specific CUDA-specific keywords. For example, the use of `dim3` is prevalent in the HARVEY CUDA code, but to ensure cross-platform functionality, we have substituted that with a 3-element integer array in the Kokkos code. In addition, some variables in the native CUDA HARVEY code are defined as auto data types and receive their actual data types from initialization. However, this may not always apply in the Kokkos code because Kokkos is an abstraction of various backends. Hence, we have explicitly defined the data types of arrays when enabling different backends to share the same codebase is necessary.

HARVEY is ported to Kokkos with 452 modified code lines and another 1876 added (Table 3.) However, the difficulty in the porting process is not always code-wise. Setting up appropriate compiling environments on different platforms can also be complicated, especially when employing the unreleased OpenACC backend on the Summit or Polaris systems, which is different from other backends in many aspects. For example, the OpenACC backend also supports unified shared memory system as other backends such as CUDA and SYCL do; however, the current OpenACC specification does

Table 3: Number of Lines of Manual Code Needed for Ports

	DPCT	HIPify	Kokkos
Number of lines added	0	0	1876
Number of lines changed	27	0	452
Time scale	weeks	days	months

not provide any memory allocation API or directive to explicitly allocate host pinned memory or unified memory. Therefore, the Kokkos OpenACC backend does not provide Kokkos memory space variants for the unified memory as other backends do (e.g., CUDA backend provides CudaUVMSpace). Instead, the OpenACC back-end relies on the implicit conversion by the underlying OpenACC compiler (e.g., NVHPC OpenACC compiler automatically maps all dynamic data to the unified memory). However, the automatic, implicit conversion has some limits such as not fully supporting static data and may conflict with other parts of the program such as I/O operations, and thus we had to manually modify some I/O operations to avoid the conflicts, which would be unnecessary if explicit memory allocation methods were provided.

7.4 Quantifying Porting Time

We used lines of code to measure the relative efforts required to port HARVEY to each of SYCL, HIP, and Kokkos. Specifically, we monitored the number of lines of the application source code that were modified and added during the porting process, as shown in Table 3. It is important to note that we only tracked the lines of code related to completing the initial working port, rather than achieving optimal performance. Although we previously described minor code changes needed to get HIP running on NVIDIA GPUs on Summit and on Intel PVC on Sunspot (see Sections 7.2.2 and 7.2.3, respectively), by initial working port we are referring to the programming model of interest running on the native hardware in this case. The justification for this choice is that the point of tools like HIPify is to convert legacy CUDA code to the specific language native to the other vendor's hardware. Additionally, the last row of Table 3 provides the order of time taken by the authors to conduct each of the ports. As is clearly evident from Table 3, Kokkos required the most time as it involved not only rewriting all of the kernels, but also restructuring large portions of the code to be compatible with Kokkos constructs. Furthermore, as detailed in Section 7.3, ensuring portability of the Kokkos code across the different platforms involved a separate set of code changes, which in some cases were quite involved as with the case of the Kokkos-OpenACC backend. It is also worth noting that the number of lines of code does not necessarily correlate with the length of time, since some changes required more time to resolve than others.

8 PERFORMANCE EVALUATION

Below we detail the metrics used to evaluate the relative performance of programming models used in this work.

8.1 Quantifying Performance Efficiency

To assess the performance of each programming model, we evaluate two performance efficiency metrics while scaling over a range of GPUs for the device architectures listed in Table 1. These metrics are the 1) *architectural efficiency*, and 2) *application efficiency*. Within this study, we specify the architectural efficiency as the fraction of achieved performance, in MFLUPS, over the best case predicted, informed from our GPU performance model. We measure application efficiency using the fraction of performance achieved over the best observed performance at each GPU count among the implementations considered for a given system. To accurately scale and avoid problems from GPU workload saturation, we use the well-established approach of piece-wise scaling [12]. That is, we strong scale over a range of GPUs (equivalent to tiles on PVC or GCDs on MI250X) spanning four powers of 2, and then grow the problem size proportionately to the increase in GPU count. We again emphasize the one-to-one mapping between MPI ranks and sub-devices on the MI250X and PVC, versus single GPUs of A100 and V100 devices. This analysis is carried out in two parts: hardware and software back-end comparisons. In the former, we directly compare platforms, each represented by their native programming model (e.g., HIP for AMD MI250X, SYCL for Intel PVC, CUDA for NVIDIA A100 or V100). The second half of this analysis consists of evaluating each programming model that could be run for a given system, along with the native language. In all cases, we compare the performance of HARVEY with the LBM proxy app, as well as the performance predictions.

To further contextualize each programming model's performance to a real-world application, we conducted a similar study using a patient-derived aorta vascular geometry, which has nontrivial load balancing and sparser fluid points than the idealized cylinder. However, since the proxy app was not designed for this type of load balancing, we only consider GPU performance predictions and HARVEY performance in these cases.

9 RESULTS

9.1 Hardware Comparison

The results for the hardware comparison conducted through piece-wise strong scaling of the native programming models on each system are presented for the cylinder in Fig. 3 and the aorta in Fig. 4. In both data sets, we evaluate the raw MFLUPS attained by each native programming model over the span of GPU counts, against the predicted MFLUPS for the corresponding architecture. From both sets of data, it was observed that the HIP implementation of HARVEY performed worse than the other programming models for small numbers of GPUs (< 8 GPUs), but became competitive for multi-node runs, particular beginning at about 64 GPUs, at which point it generally outperforms the native HARVEY implementations on Summit and Sunspot. In the aorta case (Fig. 4), the HIP version of HARVEY running on Crusher's MI250X begins to outperform the A100 on Polaris starting at 512 GPUs. With respect to the HIP LBM proxy app, the performance is consistently better than the other native programming models except where the CUDA proxy app on A100 is concerned. However, the HIP proxy app appears to edge out the CUDA proxy app on A100 near the 1024 GPU count. In contrast to what we observe, our performance model suggests that native HIP on Crusher would perform at about the same or slightly better than CUDA on Polaris A100 for both geometries, over the full range of GCD counts. We also see in both

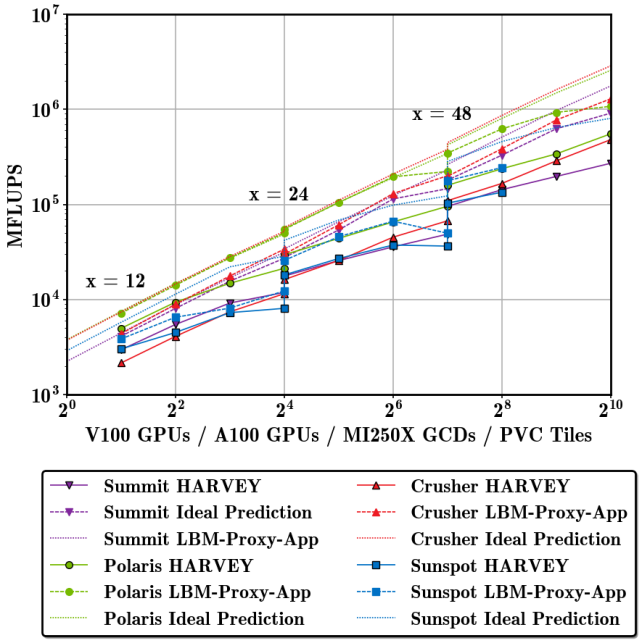

Figure 3: Comparison of relative HARVEY and LBM Proxy-App piecewise scaling performance using the idealized cylinder (with LBM Proxy-App simulation sizes of 12, 24, and 48 for GPU/GCD/Tile counts of 2-16, 16-128, and 128-1024 respectively) to ideal performance model prediction using the native backend to each system on Summit (CUDA), Crusher (HIP), Polaris (CUDA), and Sunspot (SYCL).

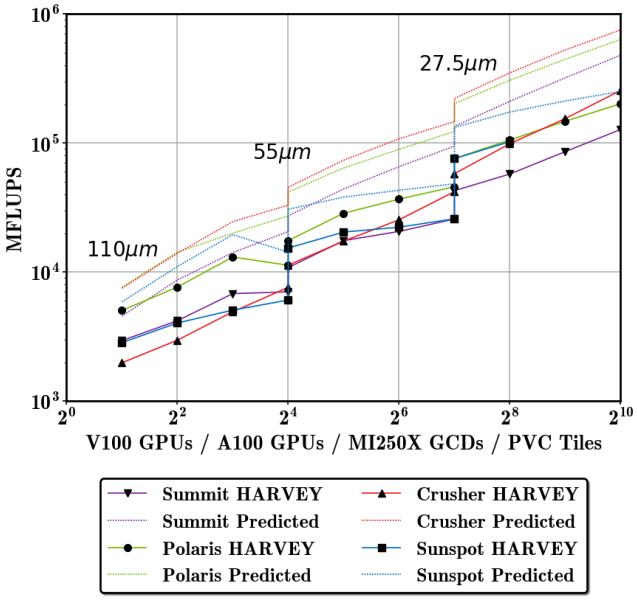

Figure 4: Aorta geometry hardware comparison of HARVEY piecewise scaling relative performance (with grid spacings of 110 microns, 55 microns, and 27.5 microns for GPU/GCD/Tile counts of 2-16, 16-128, and 128-1024 respectively) to ideal performance model prediction using the native backend to each system on Summit (CUDA), Crusher (HIP), Polaris (CUDA), and Sunspot (SYCL).

cases that the native SYCL implementation of HARVEY running on Sunspot PVC weak scales most efficiently, taken from the large jump discontinuities at each of the weak scaling points (i.e., at 16 and 128 GPU counts). This stepping behavior was also predicted by our performance model, and is more easily seen in the aorta case (Fig. 4). These results are not surprising since when compared to the NVIDIA GPUs (see Table 1), PVC tiles have more device memory (4x than Summit), so that the occupancy is consistently lower, especially at the end of each strong scaling section, where we lose the latency hiding benefit. With the MI250X on Crusher having the same device memory as PVC, it is possible that the AMD GPU is more efficient at handling the sparser fluid domains. Other important factors would include differences in interconnect speeds involving sub-devices on a single chip (PVC Xe Link versus MI250X Infinity Fabric) and between whole GPUs within a given node. In fact, device statistics alone are not sufficient to explain observed trends. For instance, with LBM being memory-bound, and from the reported bandwidths in Table 1, one might expect the PVC lines to generally be above the V100 curves. However, the trends (as well as predictions) show it is not as clear-cut. We suspect this is at least in part due to lower internodal interconnect latencies measured with our pingpong benchmark on Summit compared with Sunspot (not shown). Similarly, we measured lower internodal interconnect latencies on Crusher than on Sunspot, which are reflected in our performance model predictions. For native runs using the cylinder geometry input, the LBM proxy application consistently outperforms HARVEY, with a speedup of approximately 2 on average. In general, the gap between performance prediction and application runtime is narrower for the cylinder, which is not surprising given the simple load balancing in this case.

9.2 Software Backend Comparison

The results evaluating the software back-ends on each of the architectures are shown for the cylinder in Fig. 5 and the aorta in Fig. 6. Here we switch to our performance efficiency measures. In each figure, the first row of sub-plots displays the application efficiency, and the second row contains results for the architectural efficiencies. The application efficiencies for each system are calculated by normalizing the raw MFLUPS against the best observed case, which generally corresponds to the native programming model implementation with some exceptions, notably where we have Kokkos-SYCL being the best performing overall on Sunspot. Architectural efficiencies are calculated from normalization by the performance predictions for the system of interest, from our GPU performance model. Looking at the results for Summit (Fig. 5(a,e) and Fig. 6(a,e)), we see that the performance of the HIP proxy app with CUDA backend is on par with the native CUDA proxy app with respect to both performance measures, with the lines nearly completely overlapping. In contrast, HARVEY HIP generally lags behind native HARVEY CUDA, with a notable exception at the lowest task count, where by both performance measures and under both workloads,

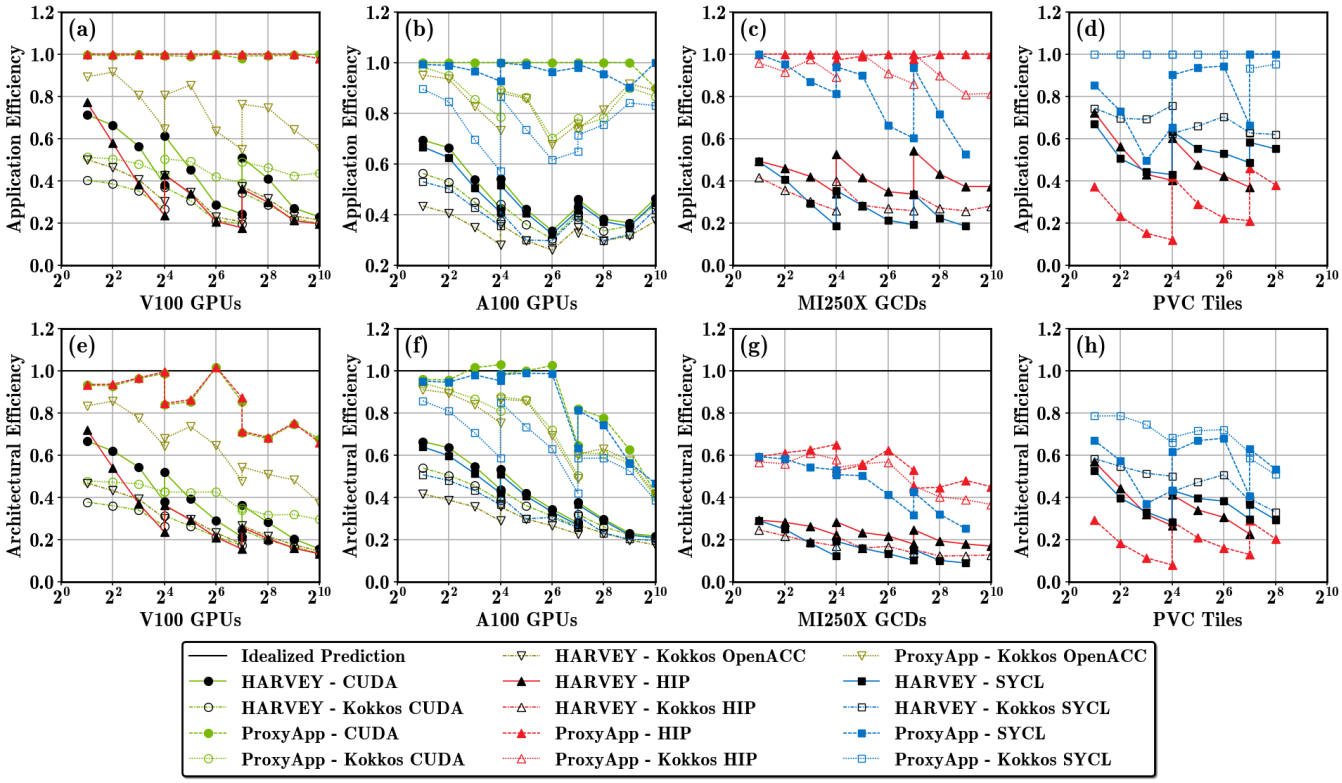

Figure 5: Piecewise scaling performance when using the cylinder input compared to the ideal performance model predictions using all of the backends available on 4 different systems: (a,e) Summit, (b,f) Polaris, (c,g) Crusher, (d,h) Sunspot. Problem sizes, in units of LBM Proxy-App simulation size, are 12, 24, and 48 for GPU/GCD/Tile counts of 2-16, 16-128, and 128-1024 respectively. Application efficiencies relative to best observed performing backend. Architectural efficiencies relative to idealized predictions from performance model.

the HIP HARVEY implementation outperforms the other HARVEY versions, followed by a steep drop in performance on the aorta. Among the non-native applications on Summit, it is interesting to see Kokkos-OpenACC consistently outperform Kokkos-CUDA irrespective of performance measure, which is especially evident for the proxy apps. Comparing with Summit results for the aorta, we observe the same general behavior between Kokkos-OpenACC and Kokkos-CUDA HARVEY implementations, but here the application efficiencies trend upward as we scale. Once again, HIP HARVEY tends to lag behind native CUDA, except for a notable exception at the first data point. It is also worth pointing out that in some instances, HARVEY in one programming model may outperform the proxy app in a different programming model, as seen for example with HARVEY CUDA compared with the Kokkos CUDA proxy app on Summit. As first mentioned in Section 7.2, the HIP code can only run on Summit with CPU-based message passing, and we acknowledge the performance degradation that can result. In both the cylinder and aorta datasets on Polaris (Fig. 5(b,f), and Fig. 6(b,f)), we observe that the SYCL implementations generally outperform the other non-native languages, and closely matches or even exceeds native CUDA performance (at the 1024 GPU count in Fig. 5(b)). The observed trends for the various Kokkos backends differ between proxy app and HARVEY. For instance, with respect to the LBM proxy app in the cylinder case in Fig. 5(b,f), the Kokkos-CUDA and Kokkos-OpenACC implementations are on par, with Kokkos-SYCL performing the worst. With HARVEY, however, we observe parity between Kokkos-CUDA and Kokkos-SYCL, while Kokkos-OpenACC performs the worst. The disparity between Kokkos-OpenACC and other programming models is most pronounced on the aorta geometry, up to 64 GPUs. One will note a few instances where the architectural efficiency of the CUDA proxy app on Polaris (in Fig. 5(b)) exceeds 1. This is possible due to caching effects not accounted for in the performance model. On Crusher (Fig. 5(c,g), and Fig. 6(c,g)), while native HIP generally outperforms the other programming models, there are some noteworthy differences in trends between the Kokkos-HIP and SYCL HARVEY implementations in the cylinder and aorta cases. With the cylinder, the Kokkos-HIP and SYCL HARVEY implementations are generally comparable, while the Kokkos-HIP proxy app generally performs better than the SYCL proxy app. The architectural efficiencies appear to be particularly low for both HARVEY and the proxy app on Crusher as seen in Fig. 5(c,g) and 6(c,g). Of note are the

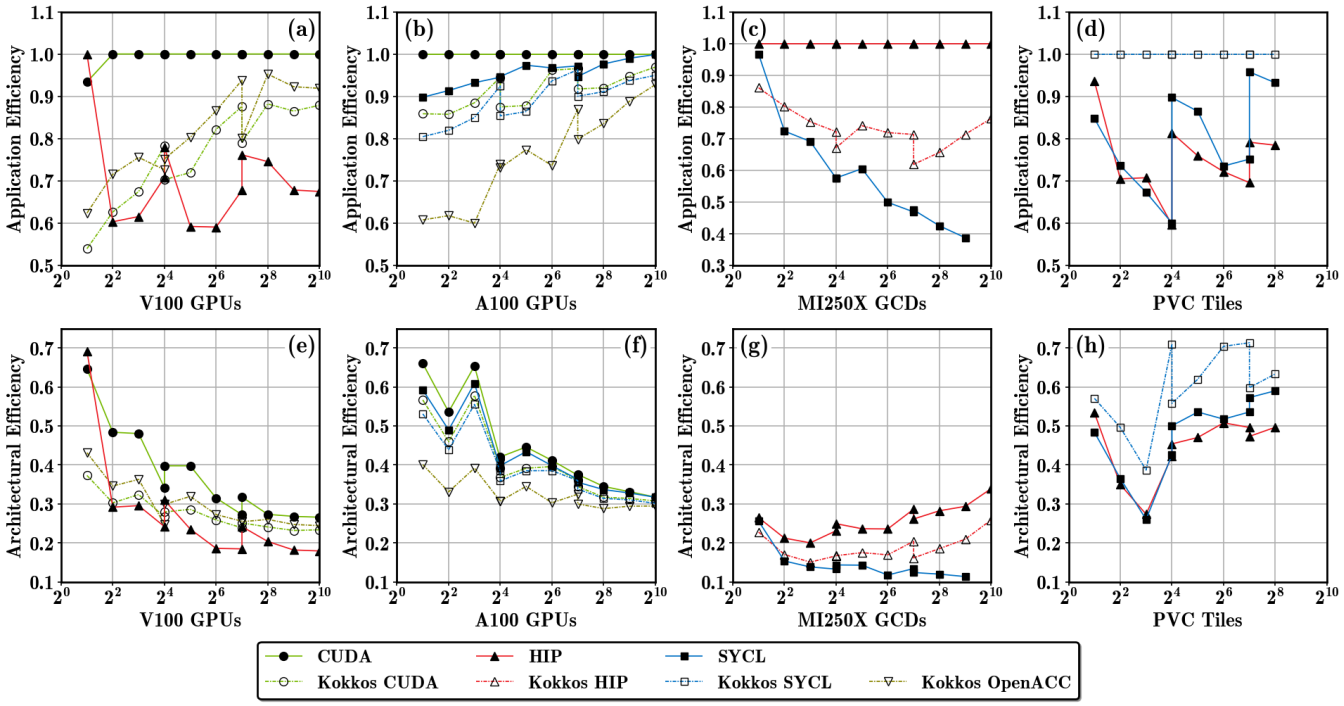

Figure 6: HARVEY piecewise scaling performance (with grid spacings of 110 microns, 55 microns, and 27.5 microns for GPU/GCD/Tile counts of 2-16, 16-128, and 128-1024 respectively) using the realistic workflow of the aorta dataset compared to the ideal performance model predictions using all of the backends available on 4 different systems: (a,e) Summit, (b,f) Polaris, (c,g) Crusher, (d,h) Sunspot. Application efficiencies relative to best observed performing backend. Architectural efficiencies relative to idealized predictions from performance model.

differing trend lines for the application efficiencies of SYCL HARVEY on Crusher for the cylinder versus the aorta cases (Fig. 5(c) and 6(c)). In the latter, the relative performance drops precipitously after the first data point, diverging from Kokkos-SYCL as the GPU count increases. Despite the downward trend on the aorta, even the lowest point has a higher efficiency value than the highest point on the cylinder, which in contrast appears to flat line in comparison. It is worth reminding the reader that the SYCL backend is still in an early development stage on the Crusher testbed. When looking at the Sunspot plots (Fig. 5(d,h), and Fig. 6(d,h)), one will notice that lines terminate at 256 GPUs, and this was due to the limited availability of hardware resources on the testbed system. One of the most striking features of these plots is that the Kokkos-SYCL implementations outperform the corresponding native SYCL codes nearly across the board. Generally, we observe HIP and SYCL HARVEY as being comparable, with the largest disparities observed with respect to the application efficiencies of the aorta (Fig. 6(d)). Interestingly, we can see from the cylinder data on Sunspot (Fig. 5(d,h)) that the HIP proxy app performs the worst among all programming models considered for the platform. Typically, proxy applications are expected to outperform their corresponding HARVEY applications since they represent an idealized version of HARVEY's core functionality. However, as mentioned in section 7.2.3, there were warning messages during the development of the chipStar miniapp, which likely impacted its performance. It's important to note that chipStar is currently in a heavy development phase, with functionality taking precedence over performance considerations.

9.3 Runtime Compositions

In evaluating hardware features for real applications, the composition of the runtime must be considered so that optimal hardware features can be chosen for a given application. HARVEY runtime composition on each vendor's hardware for an aorta geometry is shown in Figure 7. As expected, communication time increases with the number of GPUs used and as more internodal communication is required. When comparing the runtime compositions for each system (Fig. 7) against their hardware characteristics (Table 1), the increasing proportion of communication time from Crusher (MI250X GCDs) to Sunspot (PVC Tiles) to Polaris (A100 GPUs) is expected as the number of GPUs per node is the least on Polaris (four GPUs), causing high amounts of slow internodal communication on Polaris specifically, and the high-speed internodal interconnect on Crusher, having four times higher bandwidth, greatly diminishes the cost of internodal communication on Crusher.

10 LESSONS LEARNED

The insights gleaned from this study highlight several points of interest for researchers looking to move legacy codes from CUDA implementations to other programming models. Initially, we observed a significant trade-off between the level of effort required

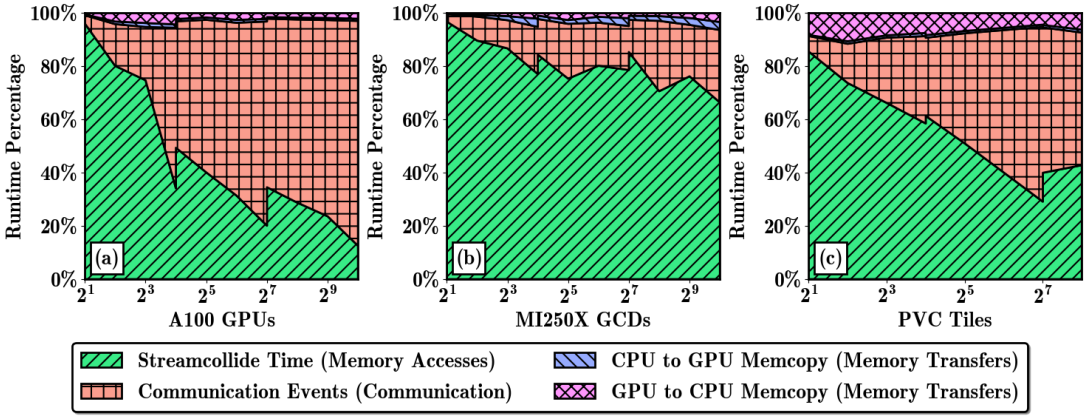

Figure 7: Composition of runtimes for the GPU with the greatest runtime running HARVEY Aorta piecewise strong scaling for each system: (a) Polaris - Nvidia A100 GPUs, (b) Crusher - AMD MI250X GCDs, (c) Sunspot - Intel PVC Tiles

for porting and the resulting portability achieved. The HIP and SYCL codes required relatively less effort than Kokkos with the use of automated porting assist tools, HIPify and DPCT, respectively. Between HIPify and DPCT, the former was more straightforward due to the similarity between the HIP and CUDA API (i.e., cudaMallocManaged versus hipMallocManaged), only requiring a regex script to perform the conversion. These tools reduced the barrier to obtaining ports that could each run on a subset of the platforms. On the other hand, each CUDA kernel had to be manually rewritten in Kokkos, with the resulting implementation being capable of being run on all of the systems. Therefore, the Kokkos implementation had the greatest portability, but not necessarily the best performance. This was most clearly demonstrated when comparing the application and architectural efficiencies between SYCL and Kokkos variants on NVIDIA based Polaris (Fig.5(b,f) and Fig.6(b,f)), with SYCL getting superior performance by both measures over the full range of GPU counts, under both types of workloads. A similar trend is observed between the SYCL and Kokkos implementations of the LBM proxy application. It follows from these findings that greater portability does not necessarily translate into greater portability of performance.

When considering how to navigate the exascale landscape, users of legacy HPC codes are faced with choosing between maintaining separate implementations for each programming model or opting for a one-size-fits-all approach with a portability framework such as Kokkos. Our results from Fig.5 and Fig.6 showed that the native programming model generally corresponded to the best observed performance for a given workload on a given system. It would be expected that the compilers for the native languages would be most optimized for the vendor's hardware. However, this trend did not hold for Sunspot, where we observed slightly better performance of Kokkos, mainly in the case of the HARVEY application runs. This is not completely surprising, given that the Kokkos implementation was manually tuned for Sunspot hardware. Nonetheless, the two implementations remained comparable. From these observations, one could argue that there remains a need to maintain multiple native implementations to maximize each system's compute capabilities. There are two counterpoints to this. First, again referring to Fig.5 and Fig.6, the native performance was not substantially higher than the other programming models for the system of interest. This is clear from the case of Kokkos, wherein we see performance comparable to that of the native language on all systems. Second, code maintainability becomes challenging when maintaining several GPU-supported implementations. There are two approaches to the maintainability issue. When a new feature is added, it can be either done directly in the original GPU implementation and then re-ported to each of the other programming models, or alternatively the user can manually translate the new feature into each implementation. Both options are error-prone. We elected to go with the latter strategy of implementing the new feature by hand in each programming model, to avoid potentially reintroducing any bugs from use of the port assist tools. This approach can be inefficient for large additional features or if new features are added frequently. In some cases, implementing a feature by hand using one programming model may require more code restructuring compared with another model, which generally occurs when there is not a direct correspondence between a CUDA construct and an equivalent in Kokkos or SYCL (this was generally not an issue with HIP). This was our experience with manually porting the CPU-based boundary conditions in HARVEY onto the GPU, which occurred after the initial porting process. Finally, maintaining multiple code bases can be challenging because obtaining expected performance on a given system may require specialized knowledge of the particular programming model API, while it is arguably favorable to acquire expert level knowledge of a single offload programming model. For these reasons, we appreciate the benefits of having a single-programming model implementation, such as Kokkos.

In this work, the process of porting and optimizing the main HARVEY application was enhanced by the use of proxy applications. In Figs. 3 and 5, we demonstrated that the proxy application can be used to inform the expected performance limits of each implementation of the main application programming model, on all systems of interest. Furthermore, the proxy application provided a useful testbed for experimenting with automated porting tools on a smaller codebase before moving to the main application, which enabled us to get working codes quickly on different hardware.

While both our performance model and the proxy application were useful in gauging expected performance, the LBM proxy application was unique in being able to identify areas of optimization due to its correspondence with HARVEY. That is, because both codes are essentially carrying out the same underlying algorithm at their core, it enables the user to identify bottlenecks in the main production code and draw from implementation differences in the proxy app to drive optimizations in the main application. However, it is worth cautioning that this approach can be limited by the fact that a proxy application may sometimes oversimplify certain aspects of the code for the sake of performance, but that there may not be a direct translation in the main application. A primary example of this lies in the domain decomposition schemes. HARVEY uses a sophisticated load bisection balancer algorithm designed to handle complex geometries, whereas the LBM proxy app uses a simplistic domain decomposition scheme that gives perfect load balancing in the cylindrical geometry it was programmed to solve. In summary, while proxy applications can be invaluable in optimization workflows, their limitations need also be considered.

11 CONCLUSION

Our study systematically evaluated SYCL, HIP, and Kokkos programming models for porting a CUDA-based code to various GPU-accelerated supercomputing platforms. We found that all models performed well in terms of the baseline code by applying manual optimizations to each port and benchmarking their performance with a GPU performance model and an LBM proxy app. However, we observed significant variation in the time required for each model to achieve a functional port, with HIP being the quickest and Kokkos requiring the most time. When tested with a real-world workload, the aorta, we found that Sunspot Nodes (Intel PVC) and Crusher Nodes (AMD MI250X) achieved better or comparable performance to Polaris and Summit Nodes (NVIDIA GPUs) for piecewise strong scaling. This study provides insights into the intricacies of transitioning CUDA-centric code to diverse programming models and hardware setups. Illustrated by the exemplary instance of HARVEY, these findings have the potential to guide the choices of HPC practitioners seeking to enhance their code for a variety of supercomputing platforms featuring GPU acceleration.

ACKNOWLEDGMENTS

Research reported in this work was supported by the National Institutes of Health under Award Numbers U01CA253511 and T32GM144291, and the ALCF Aurora Early Science Program. The content does not necessarily represent the official views of the NIH. This research used resources from the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported by the DE-AC02-06CH11357 Contract. An award of compute time was provided by the INCITE program. This research used resources from the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

REFERENCES

[1] Germán Castaño, Youssef Faqir-Rhazoui, Carlos García, and Manuel Prieto-Matías. 2022. Evaluation of Intel's DPC++ Compatibility Tool in heterogeneous computing. *J. Parallel and Distrib. Comput.* 165 (2022), 120–129.

[2] Cheng Chang, Chih-Hao Liu, and Chao-An Lin. 2009. Boundary conditions for lattice Boltzmann simulations with complex geometry flows. *Computers & Mathematics with Applications* 58, 5 (2009), 940–949. https://doi.org/10.1016/j.camwa.2009.02.016 Mesoscopic Methods in Engineering and Science.

[3] Steffen Christgau and Thomas Steinke. 2020. Porting a legacy cuda stencil code to oneapi. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 359–367.

[4] Tom Deakin, James Price, Matt Martineau, and Simon McIntosh-Smith. 2018. Evaluating attainable memory bandwidth of parallel programming models via BabelStream. *International Journal of Computational Science and Engineering* 17, 3 (2018), 247–262.

[5] 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)*. IEEE, 57–67.

[6] Argonne Leadership Computing Facility. 2021. Polaris. https://www.alcf.anl.gov/polaris.

[7] Argonne Leadership Computing Facility. 2022. Aurora/Sunspot Interconnect. https://www.alcf.anl.gov/support-center/aurora/interconnect.

[8] Argonne Leadership Computing Facility. 2022. Aurora/Sunspot Node Level Overview. https://www.alcf.anl.gov/support-center/aurora/node-level-overview.

[9] Oak Ridge Leadership Computing Facility. 2023. Crusher Quick-Start Guide. https://docs.olcf.ornl.gov/systems/crusher_quick_start_guide.html.

[10] William F Godoy, Pedro Valero-Lara, T Elise Dettling, Christian Trefftz, Ian Jorquera, Thomas Sheehy, Ross G Miller, Marc Gonzalez-Tallada, Jeffrey S Vetter, and Valentin Churavy. 2023. Evaluating performance and portability of high-level programming models: Julia, Python/Numba, and Kokkos on exascale nodes. *arXiv preprint arXiv:2303.06195* (2023).

[11] Muhammad Haseeb, Nan Ding, Jack Deslippe, and Muaaz Awan. 2021. Evaluating Performance and Portability of a core bioinformatics kernel on multiple vendor GPUs. In *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 68–78.

[12] Gregory Herschlag, Seyong Lee, Jeffrey S Vetter, and Amanda Randles. 2018. GPU data access on complex geometries for D3Q19 lattice Boltzmann method. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 825–834.

[13] Intel. 2017. Intel MPI Benchmarks Github. https://github.com/intel/mpi-benchmarks.

[14] Balint Joo, Thorsten Kurth, Michael A Clark, Jeongnim Kim, Christian Robert Trott, Dan Ibanez, Daniel Sunderland, and Jack Deslippe. 2019. Performance portability of a Wilson Dslash stencil operator mini-app using Kokkos and SYCL. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 14–25.

[15] 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)*. IEEE, 45–56.

[16] William Ladd, Christopher Jensen, Madhurima Vardhan, Jeff Ames, Jeff Hammond, Erik Draeger, and Amanda Randles. 2023. Optimizing Cloud Computing Resource Usage for Hemodynamic Simulation. In *IEEE 37th International Symposium on Parallel and Distributed Processing*. https://doi.org/10.1109/IPDPS54959.2023.00063

[17] Geng Liu and John Gounley. 2022. MINIAPP. https://github.com/lucaso19891019/MINIAPP.

[18] Ji Qiang and Robert D Ryne. 2001. Parallel 3D Poisson solver for a charged beam in a conducting pipe. *Computer physics communications* 138, 1 (2001), 18–28.

[19] Amanda Peters Randles, Vivek Kale, Jeff Hammond, William Gropp, and Efthimios Kaxiras. 2013. Performance Analysis of the Lattice Boltzmann Model Beyond Navier-Stokes. In *IEEE 27th International Symposium on Parallel and Distributed Processing*. 1063–1074. https://doi.org/10.1109/IPDPS.2013.109

[20] Sauro Succi. 2001. *The lattice Boltzmann equation: for fluid dynamics and beyond*. Oxford university press.

[21] Nhat Phuong Tran, Myungho Lee, and Dong Hoon Choi. 2016. Memory-Efficient Parallelization of 3D Lattice Boltzmann Flow Solver on a GPU. In *Proceedings - 22nd IEEE International Conference on High Performance Computing, HiPC 2015*. IEEE, 315–324. https://doi.org/10.1109/HiPC.2015.49

[22] G. Wellein, T. Zeiser, G. Hager, and S. Donath. 2006. On the single processor performance of simple lattice Boltzmann kernels. *Computers and Fluids* 35, 8-9 (2006), 910–919. https://doi.org/10.1016/j.compfluid.2005.02.008

[23] Jisheng Zhao, Colleen Bertoni, Jeffrey Young, Kevin Harms, Vivek Sarkar, and Brice Videau. 2022. HIPLZ: Enabling Performance Portability for Exascale Systems. In *European Conference on Parallel Processing*. Springer, 197–210.

Received 17 August 2023