Performance Portability Evaluation of Fluid-Structure Interaction Simulations on Heterogeneous Platforms

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

Ayman Yousef
Biomedical Engineering
Duke University
Durham, USA
ayman.yousef@duke.edu

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

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

Antigoni Georgiadou
National Center for Computational Sciences
Oak Ridge National Laboratory
Oak Ridge, USA
georgiadoua@ornl.gov

Jorik Stoop
Biomedical Engineering
Duke University
Durham, USA
jorik.stoop@duke.edu

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

***Abstract*—The rapid proliferation of heterogeneous programming languages and multi-vendor hardware has underscored the critical need to evaluate the performance portability of scientific applications. In this work, we present the systematic porting and optimization of a massively parallel fluid-structure interaction code across multiple heterogeneous programming frameworks for deployment on leadership-class supercomputers from major vendors. Our analysis focuses on at-scale performance for simulations involving hundreds of millions of deformable cells, executed on a combination of CPUs and GPUs spanning thousands of nodes on exascale machines. We benchmark the performance of each implementation, highlighting the trade-offs inherent in adopting diverse programming models. Key insights regarding the portability of CUDA on multi-vendor platforms, the superior multi-core CPU performance from SYCL, and architectural considerations on performance optimization are distilled from our experience, offering guidance to other users of high performance computing based on our findings.**

***Index Terms*—Computational fluid dynamics, fluid structure interaction, GPU computing, high performance computing, performance portability**

I. Introduction

With the recent proliferation of programming models targeting heterogeneous architectures, there is a need to devise and evaluate strategies to achieve performance portability for science applications at scale. Legacy CUDA-based applications, widely adopted for graphics processing unit (GPU) acceleration, have required significant porting efforts to adapt to alternative programming models introduced by diverse hardware vendors. However, recent advances in compiler technologies have enabled CUDA to function as a portable language across hardware from the major device vendors, including NVIDIA, AMD, and Intel, raising critical questions about the necessity and trade-offs of porting.

Simultaneously, the increasing popularity of GPUs has spurred a strong focus on offload acceleration. However, many HPC applications have underutilized the parallelism available in high core counts of modern CPUs present on leadership-scale system nodes. This imbalance underscores the need for a holistic evaluation of programming models across heterogeneous architectures to inform developers on optimizing performance while minimizing development overhead.

In this paper, we systematically evaluate the performance portability of HARVEY, a multiphysics HPC application [Randles et al.(2013)], [Gounley et al.(2019)], across heterogeneous nodes from leadership class systems, including Aurora (ALCF), Frontier (ORNL), and Polaris (ALCF) (Fig. 1). The primary contributions of this work are as follows.

1) **A comparative analysis of scaling performance** of CUDA, SYCL, HIP, Kokkos and OpenMP programming models on leadership-class heterogeneous architectures.
2) **An evaluation of trade-offs** between performance portability and the development effort required for each programming model.
3) **An analysis of GPU progamming optimizations**, focusing on the impact of hardware architecture on atomic operations and GPU-centric communication.
4) **Actionable insights** into the benefits and drawbacks of each programming model within the context of a real-world scientific application

By addressing these points, this study makes important contributions to accelerator programming by providing practical guidance to help HPC users navigate the challenges and opportunities of exascale computing effectively.

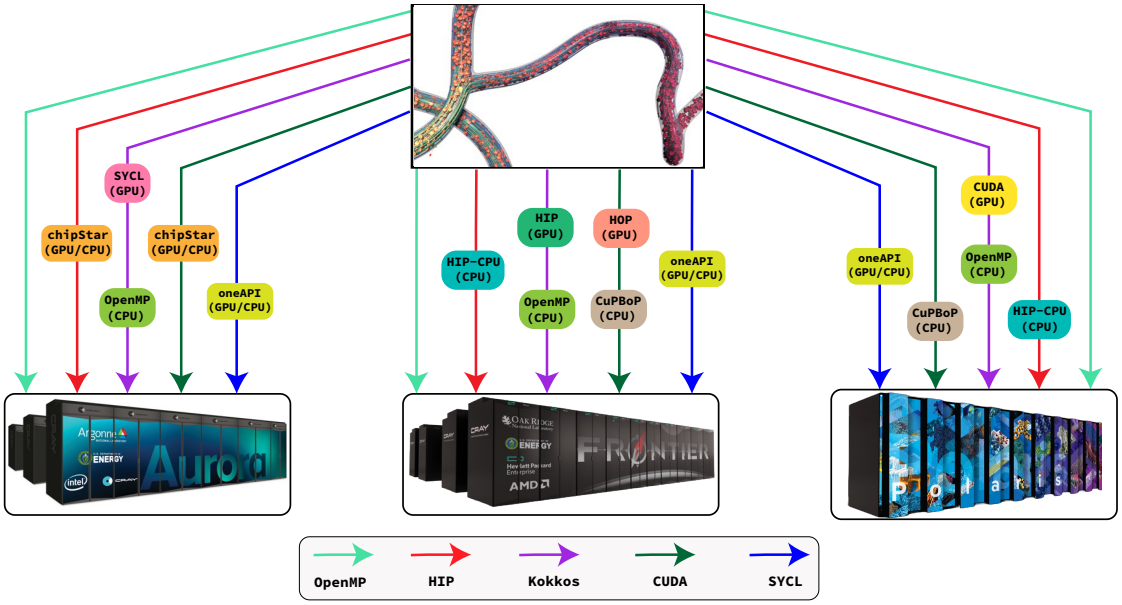

Fig. 1. Overview of the performance portability study, where programming models are assessed in terms of a real application, HARVEY, both on CPUs and GPUs supported by a variety of compiler toolchains across leadership class supercomputers. The programming models evaluated in this study are OpenMP (light green), HIP (red), Kokkos (purple), CUDA (dark green), and SYCL/oneAPI (blue). Each programming model was evaluated both on GPUs and CPUs on each of three platforms, Aurora, Frontier, and Polaris. Arrays denoting the programming models emanate from the HARVEY application icon to the platforms they were evaluated on. Text bubbles overlaid onto arrows indicate specific compiler tools used to support GPU and CPU backends for the given programming model. Missing labels on arrows denote native backend support requiring no additional toolchain steps.

II. Related Work

GPU offloading has become the standard approach to improving application performance on modern GPU-dense nodes. Given the diversity of available software-hardware combinations, performance portability has become a central concern [Pennycook et al.(2016)]. The performance of SYCL across various solvers and hardware vendors has been extensively analyzed by [Reguly(2023)], [Reguly(2019)], [Rangel et al.(2023)]. Broad evaluations of the well-known heterogeneous programming models targeting GPUs have been conducted [Davis et al.(2024b)], [Ruzicka et al.(2024)]. Numerous works have detailed the porting of benchmark applications and real-world HPC codes to heterogeneous programming frameworks [Howard et al.(2017)], [Joo et al.(2019)], [Gschwandtner et al.(2021)]. Complementing these efforts, the development of APIs that enable system-level portability at compilation time has gained traction. These APIs mitigate the need for extensive source code modifications while targeting native system architectures. For example, [Chen et al.(2023)] and [Louhivuori(2023)] introduce solutions that allow CUDA code to run seamlessly on HIP-native systems, highlighting the potential for cross-platform execution without significant code refactoring.

III. Application Overview

This work employs HARVEY [Randles et al.(2013)], a massively parallel, multiphysics code. HARVEY uses the lattice Boltzmann method (LBM) [Succi(2001)], [Krüger et al.(2016)] to solve the underlying fluid flow. The LBM is a mesoscopic method that tracks fictitious particle packets that represent probability densities in a velocity space on a Cartesian grid. The Immersed Boundary method (IBM) [Peskin(1977)] is used to couple the LBM fluid solver with the finite element (FEM) module [Krüger et al.(2011)] for computing the deformation response of biological cells. HARVEY is an ideal candidate for performance portability evaluation since it is a production HPC code with characteristics representative of a broader set of scientific codes, including a mixture of memory bandwidth-bound and compute-bound kernels, and stencil-based communication patterns.

Due to an efficient parallelization scheme, HARVEY is capable of simulating millions of red blood cells in complex arterial vessel geometries. Fig. 2 shows a representative HARVEY simulation with many red blood cells flowing through a complex vascular anatomy.

An inherent advantage of HARVEY that facilitates portability evaluation across a diverse set of platforms is its minimal dependence on external libraries.

IV. Platforms

This work exercises HARVEY on leadership class systems containing devices from the major GPU vendors. These platforms include the exascale machines Frontier (Oak Ridge National Laboratory, AMD) and Aurora (Argonne National Laboratory, Intel), as well as Polaris (Argonne National Laboratory,

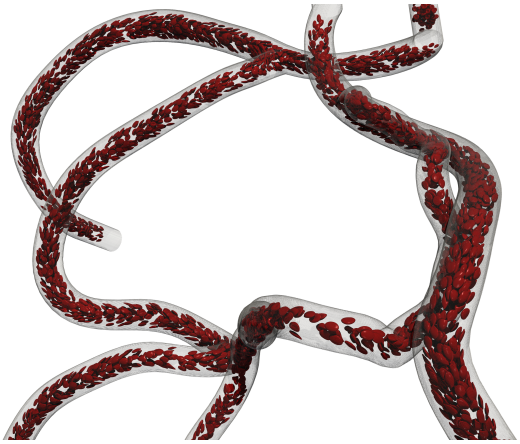

Fig. 2. Representative HARVEY simulation of red blood cells flowing through a complex vascular network.

NVIDIA). The hardware characteristics of these machines are summarized in Table I.

V. Programming Models

In this paper, we evaluate performance portability of the HARVEY application using five well-known heterogeneous programming frameworks: CUDA, HIP, SYCL, Kokkos, and OpenMP. These programming models were chosen since they are the major supported languages on the current-generation machines.

VI. Porting Methodology

Porting of the HARVEY application to heterogeneous programming models was performed using a combination of semi-automated porting tools and manual code transformations. Execution of GPU programming models on nonnative hardware was enabled by leveraging numerous compiler technologies (Table II), which in some cases required further changes to the source code.

A. Enabling Portability of CUDA

HARVEY GPU kernels were originally written in CUDA. To run CUDA on the AMD MI250X GPUs of Frontier, we used the Header Only Porting (HOP) library [Louhivuori(2023)], which performs compilation-time redefinition of CUDA identifiers with HIP equivalents. Usage of HOP only required minor edits to the HARVEY build files. Generation of binaries compatible with the Intel GPUs of Aurora was facilitated by the LLVM-based chipStar compiler, which provides support for CUDA on platforms with SPIR-V as the device intermediate language. At the time of publication, chipStar does not support commonly used CUDA libraries such as Thrust, which was a HARVEY dependency. Our workaround involved re-writing subroutines that originally made use of `thrust::reduce_by_key` with handwritten segmented scan kernels [Sengupta et al.(2011)]. Toward this aim, the restructuring of HARVEY CUDA for deployment on Aurora with chipStar involved adding several new functions for custom shared memory algorithms of segmented scans which amounted to a few hundred lines of new code.

The deployment of HARVEY CUDA on CPUs relied on usage of CUDA for Parallelized and Broad-range Processors (CuPBoP) [Han et al.(2022)], [Han et al.(2024a)], [Han et al.(2024b)], a framework supporting execution of CUDA source code on non-NVIDIA devices, including several CPU backends. CuPBoP works by converting CUDA source code to LLVM bitcode, and performing compiler optimizations on the intermediate representation (IR) files. Therefore, application of CuPBoP to HARVEY CUDA required modifications to the HARVEY build files. While no source file changes were required, editing build files to generate the intermediates needed for CuPBoP from HARVEY source files was not a straightforward process. CuPBoP proved to be highly portable, enabling HARVEY CUDA to execute on the CPUs across all systems in this study.

B. Porting to HIP with HIPify

Porting the native CUDA source code of HARVEY to HIP involved straightforward application of the perl-based HIPify tool, which acts essentially as a simple regex script. With a single command, HIPify automatically converted the vast majority of the HARVEY source code, with minor exceptions to header file include statements. HIP natively supports AMD and NVIDIA devices, and therefore required no source code modifications to execute HARVEY HIP on Frontier or Polaris. Similarly to CUDA, chipStar also supports HIP applications, so once the code changes were applied for running HARVEY CUDA on Intel GPUs on Aurora, HIP could readily be run with the HIPify tool.

Deploying HARVEY HIP on CPUs relied on a combination of compiler frameworks. Since chipStar comes with an OpenCL backend for targeting Intel CPUs, simply passing in the OpenCL backend to chipStar during the build procedure was needed to execute HARVEY HIP on the Intel Xeon CPU Max on Aurora. No additional code changes were necessary. Execution of HARVEY HIP on the AMD EPYC CPUs on Frontier and Polaris was facilitated by usage of the HIP CPU Runtime, a header-only library originally developed by AMD based on the Parallel Algorithms component of the C++ library, allowing the execution of unmodified HIP code on CPUs. Usage of the HIP CPU Runtime requires providing paths to the compiler identifying the HIP CPU Runtime library, and the Intel oneAPI Thread Building Blocks (oneTBB) library. Otherwise, no additional source file changes were necessary.

C. Incremental Porting to SYCL

Porting of the HARVEY CUDA code to SYCL was performed incrementally. An initial port was facilitated by usage of the Intel Data Parallel C++ Compatibility Tool (DPCT). Once the compilation database is provided to DPCT, the tool automatically converts the majority of CUDA syntax to SYCL-like data-parallel C++ (DPC++), Intel's implementation of SYCL. DPC++ headers include wrappers serving as

TABLE I
SYSTEM NODE CHARACTERISTICS.

System	Aurora	Frontier	Polaris
CPU	2x Intel Xeon CPU Max 9470C	1x AMD EPYC 7A53	1x AMD EPYC 7543P
Cores/CPU	52	64	32
GPU	6x Intel Data Center GPU Max 1550 (12 GPUs)	4x AMD MI250X GCDs (8 GPUs)	4x NVIDIA A100 GPUs
GPU Memory	64 GB	64 GB	40 GB
GPU Mem. Bandwidth	1.23 TB/s	1.28 TB/s	1.30 TB/s
GPU-CPU Interface	PCIe Gen5 (128 GB/s)	AMD Infinity Fabric CPU-GPU (72 GB/s)	PCIe Gen4 (64 GB/s)
GPU-GPU Intranode	Intel X^e Link (23 GB/s)	AMD Infinity Frabric GPU-GPU (100 GB/s)	NVLink (600 GB/s)
Interconnect	Slingshot 11 (25 GB/s)	4x HPE Slingshot (100 GB/s)	Slingshot (25 GB/s)

abstractions for CUDA constructs such as constant and shared device memories that simplify the porting procedure. Still, DPCT emitted a number of error messages requiring manual intervention. Once a working DPCT port of HARVEY was verified, the next step involved phasing out code referencing the DPCT namespace, and instead replacing these pieces of code with standard SYCL. While DPCT allows C pointer-style arithmetic on device allocations, conversion to SYCL required mixing the pointer-style conventions of CUDA with SYCL buffers and accessor semantics that were necessary for replacing `dpct::constant_memory`. In all, the porting procedure to SYCL was more time-consuming than HIP, requiring significantly more user intervention that amounted to several hundred lines of code. Once HARVEY was ported to SYCL, no further code changes were required to run on the GPU or CPU backends of any systems examined here, given out-of-the-box support for heterogeneous offload from oneAPI.

D. Fully Manual Porting to Kokkos

The procedure for porting HARVEY to Kokkos was performed completely manually. CUDA kernel launch syntax was replaced with `Kokkos::parallel_for` blocks. Allocations in CUDA shared memory syntax were replaced with Kokkos team scratch memory. CUDA constant device allocations were replaced with Kokkos unmanaged Views. Parallel scan primitives were readily available in the Kokkos library via `kokkos::parallel_scan`. In all, a couple thousand lines of code were added or changed during the port to Kokkos. Execution of HARVEY Kokkos binaries on each platform was straightforward, only requiring specification of the correct backend for each platform to the CMake system. To run Kokkos on the multi-core CPUs, the OpenMP backend was specified.

E. Manual Porting to Directive-based OpenMP

Like Kokkos, the conversion of CUDA source code to OpenMP 5.0 was performed completely manually. While OpenMP allows for implicit data management between host and device with the `map` clause, for performance reasons, we opted for an explicit memory management approach using OpenMP analogs of C-style memory allocation routines like `malloc`, those being `omp_target_alloc` and `omp_target_free`. Simple CUDA kernels were straightforward to port with `omp target teams` directives. CUDA constant device memory declarations were wrapped in `omp declare target` directives. CUDA local shared memory variable allocations were replaced with OpenMP 5.0 allocator semantics, such as `omp_pteam_mem_alloc`. While OpenMP 5.0 introduced support for inclusive and exclusive scans with the `scan` directive, manual implementation of segmented scans was still required. In total, the OpenMP port involved a comparable amount of code changes as Kokkos. Minimal code changes were necessary to running HARVEY OpenMP on each platform. Executing OpenMP on the CPUs was straightforward, only requiring adjustments to environment variables.

VII. PERFORMANCE EVALUATION

To measure the performance portability of programming models, we followed the definition proposed by Pennycook *et. al.* [Pennycook et al.(2016)], repeated in Equation 1 below.

$$P(a,p,H) = \left\{ \begin{array}{ll} \frac{|H|}{\sum i \in H \frac{1}{e_i(a,p)}}, & \text{if } i \text{ is supported } \forall i \in H \\ 0, & \text{otherwise} \end{array} \right\} \quad (1)$$

Equation 1 states that the performance portability P of an application a solving problem p over a set of platforms H is the harmonic mean of the performance efficiencies $e_i(a,p)$ of the application a computing the problem p on each platform i. Specifically, we computed the performance portability P for each HARVEY programming model implementation over the three hardware sets, H_{CPU}, H_{GPU}, and $H_{GPU} \cup H_{CPU}$. Architectural differences between CPUs and GPUs are accounted for by representing P for each platform separately in addition to their combined evaluation. Furthermore, following the nomenclature of [Pennycook et al.(2016)], we calculated P based on the application efficiency, defined as the fraction of performance achieved over the best observed performance on a given platform.

VIII. RESULTS

A. Optimizations

Several code optimizations were implemented to better utilize the hardware resources on each platform. The impact of each of these code optimizations is reflected in Fig. 3 for each programming model running on a single node of each platform. One key optimization was to replace atomic

TABLE II
COMPILER TOOLCHAINS.

Programming Model	Aurora	Frontier	Polaris
SYCL	Aurora MPICH(mpicxx)	oneAPI(icpx)	oneAPI(icpx)
Kokkos	Aurora MPICH(mpicxx)	ROCm(hipcc)/Cray MPICH(mpicxx)	NVHPC(nvc++)
OpenMP	Aurora MPICH(mpicxx)	HIP-Clang(amdclang++)	clang++
CUDA	chipStar-Level Zero(cucc)/chipStar-OpenCL(cucc)	HOP(amdclang++)/CuPBoP(clang++)	NVHPC(nvcc)/CuPBoP(clang++)
HIP	chipStar-Level Zero(hipcc)/chipStar-OpenCL(hipcc)	HIP-Clang(amdclang++)/HIP-CPU(icpx)	ROCm(hipcc)/HIP-CPU(icpx)

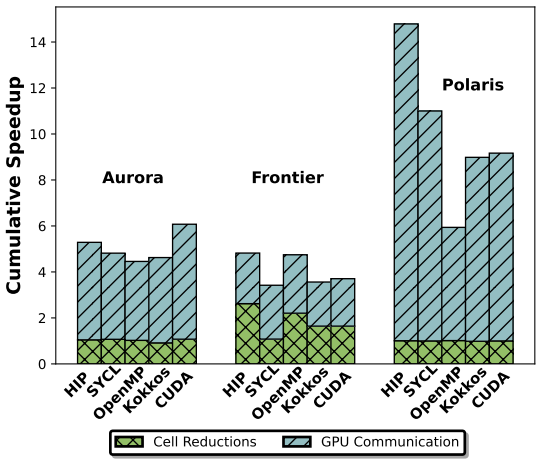

Fig. 3. Cumulative speedups resulting from code optimizations applied uniformly to each programming model, evaluated on Aurora (leftmost cluster), Frontier (middle cluster), and Polaris (rightmost cluster).

additions with shared memory reductions (indicated as "Cell Reductions") in device kernels involved in calculating cell centroids and volumes. Another significant optimization involved restructuring host-centric IBM cell communication tasks to the GPUs (marked as "GPU Communication"), as detailed in [Martin et al.(2024)].

The results reveal that the "Cell Reductions" optimization had a substantial impact only on Frontier, with the exception of SYCL. This behavior is likely due to the higher cost of atomic operations on AMD GPUs, which appears to be better mitigated by the oneAPI compiler. The most significant performance improvements were achieved through the second set of optimizations ("GPU Communication"), as it reduced the data movement between host and device and eliminated the bottleneck caused by CPU-mediated bookkeeping. The fact that HIP achieved the greatest speedup among all programming models on Polaris can be explained by the lower baseline performance of HIP on the NVIDIA A100 (not shown).

B. Kernel-Level Analysis

A kernel-level analysis of HARVEY was conducted for each of the programming models being evaluated on a single GPU of each system using roofline plots. The analysis focused on several kernels representative of the fluid-structure interaction aspects of the program for the analysis, which were the FEM (FEM), LBM fluid update (LBM), IBM force spreading (Spreading), and IBM interpolation (Interpolation). The roofline plots for Aurora, Frontier, and Polaris are shown in Fig. 4. Roofline plots were generated from Intel Advisor, NVIDIA Nsight, and AMD Omniperf. While most kernels were memory-bandwidth bound, the FEM kernel (circle) was compute-bound with the exception of Frontier. Generally, among the memory-bandwidth-bound kernels, the LBM fluid update (triangle) achieved the highest FLOP/s. On Aurora (Fig. 4(a)), the memory-bandwidth bound kernels were clustered close together, indicating similar performance among the programming models. In contrast, for the compute-bound FEM, Kokkos and OpenMP achieved the highest FLOP/s, and HIP and CUDA achieved the fewest FLOP/s. HIP and CUDA were the most similar in performance, which was expected as both were built with chipStar. There was generally more spread in kernel performance among programming models on the other platforms (Fig. 4(b), 4(c)). Kernel performance was the most consistent for the LBM fluid update (triangle) and FEM (circle), with the exception of OpenMP. In general, kernel performance was the most variable for OpenMP.

C. Weak Scaling

Weak scaling studies were conducted to evaluate the ability of each system to efficiently handle increasing problem sizes. To ensure consistency, the problem size was adjusted so that the number of simulated red blood cells per process remained constant. Furthermore, to account for differences in the number of GPUs per node between systems, the problem sizes were scaled to ensure that the number of cells per GPU was the same for any node count between systems. We acknowledge that ensuring equal problem size per GPU translates to differences in GPU memory utilization, and in turn impact performance trends. Performance was measured as time per iteration, calculated by dividing the maximum wall-time recorded among MPI ranks by the total number of iterations.

1) GPU weak scaling: The weak scaling performance of each programming model on the GPUs is shown in Fig. 5, with one MPI rank assigned to each logical GPU (sub-device). Overall, the programming models exhibited similar performance, with a few notable exceptions. On Aurora (Fig. 5(a)), a significant runtime spike was observed for Kokkos beginning at about 512 nodes, and a sudden drop in SYCL runtime seen at 128 nodes. These performance anomalies observed on Aurora can likely be attributed to the fact that Aurora was still in a pre-production stage at the time of this study. Additionally, a systematic runtime discrepancy was noted for OpenMP on

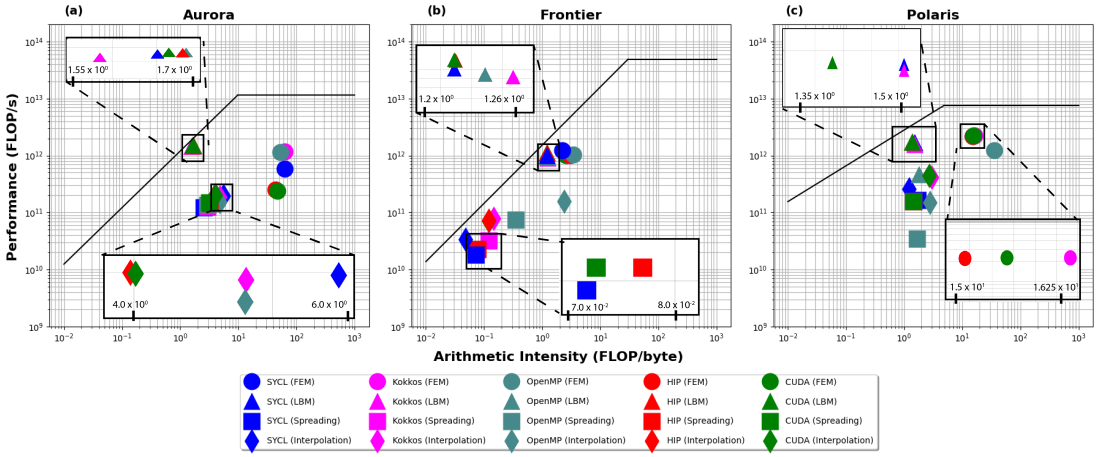

Fig. 4. Roofline plots for HARVEY running on a single GPU on (a) Aurora (Intel Advisor), (b) Frontier (AMD Omniperf), and (c) Polaris (NVIDIA Nsight). Each kernel is denoted by a different symbol, and the programming model indicated by a unique color.

Frontier compared to all other programming models, which was traced to the under-performance of the interpolation kernel (Fig. 4(b)). For OpenMP, the interpolation kernel exhibited significantly higher arithmetic intensity at comparable FLOP/s relative to other programming models, which increased runtime. Since the IBM cell interpolation algorithm was identical across programming models, this difference likely resulted from how HIP-Clang interacted with the relevant OpenMP Target constructs. Among the platforms, Polaris exhibited the shortest runtimes overall (Fig. 5(c)). On Frontier and Polaris, native programming models consistently delivered the best performance (Fig. 5(b,c)), which was in contrast to Aurora data (Fig. 5(a)) indicating parity among programming models.

2) CPU weak scaling: Weak scaling of each programming model on the CPUs is shown in Fig. 6. As was done with GPU scaling, the number of MPI ranks on a node was matched to the number of GPUs, and the number of hardware threads was evenly split among the MPI ranks. Among the CPU backends, we observed that SYCL compiled with Intel's oneAPI compiler consistently outperformed the other programming models, even on AMD CPUs (Fig. 6(b,c)). To better understand these results, the disassembled code was analyzed using the `objdump` program. A notable trend was observed: the disassembled SYCL kernels exhibited up to 60% fewer instructions compared to other CPU-backends, such as Kokkos-OpenMP and HIP-CPU. This reduction in instruction count suggests that the faster performance of SYCL on CPUs may be partially attributed to its lower instruction overhead.

D. Evaluating the Performance Portability Metric

We employed Equation 1 to evaluate the portability of application performance among three sets of platforms: H_{GPU}, H_{CPU}, and $H_{GPU} \cup H_{CPU}$, with H_{GPU} denoting the set of logical GPUs, H_{CPU} comprising all available CPU hardware threads, and $H_{GPU} \cup H_{CPU}$ representing the union of the two sets. prior. The resulting performance portabilities are plotted in Fig. 7. In general, the HARVEY CUDA implementation achieved the greatest performance portability over H_{GPU} (Fig. 7(a)), whereas SYCL achieved the highest performance portability over H_{CPU} (Fig. 7(b)). When both CPUs and GPUs were taken into account as represented by $H_{GPU} \cup H_{CPU}$ (Fig. 7(c)), SYCL consistently outperformed the other programming models in this category.

E. Runtime composition

In order to gain insight into differences between the CPU and GPU backends of programming models, as well as the impact of architectural differences between platforms, we analyzed runtime compositions shown in Fig. 8. These plots were derived from internal profiling statistics of the weak scaling runs presented in Fig. 5 and 6. Each platform (represented as a column in Fig. 8) corresponds to its native programming model, so that we have the leftmost column as SYCL for Aurora, the middle column as HIP on Frontier, and the rightmost column being CUDA on Polaris. Native programming model performance on the GPUs is represented in the top row, while native programming model performance with the CPU backends indicated in Fig. 1 is shown in the bottom row. MPI communication time in each subplot is subdivided into the three bottom layers of different shades of blue, representing in order of lightest to darkest shades the IBM Cell Update Communication, the IBM Cell Velocity Communication, and the LBM Fluid Communication. The IBM Cell Update Communication concerns MPI-related bookkeeping associated with tracking changes to cell vertex ownership as cells travel between tasks. The IBM Cell Velocity Communication component denotes the MPI exchange of IBM cell vertex velocities. The LBM Fluid Communication subroutine involves exchange of LBM fluid distribution data at the task boundaries. With the exception of Frontier, the GPU runtimes (top row) were dominated by MPI communication overhead stemming primarily from IBM Cell Velocity Communication, which tended to increase at higher node counts, and was followed by the FEM kernel (light purple). On

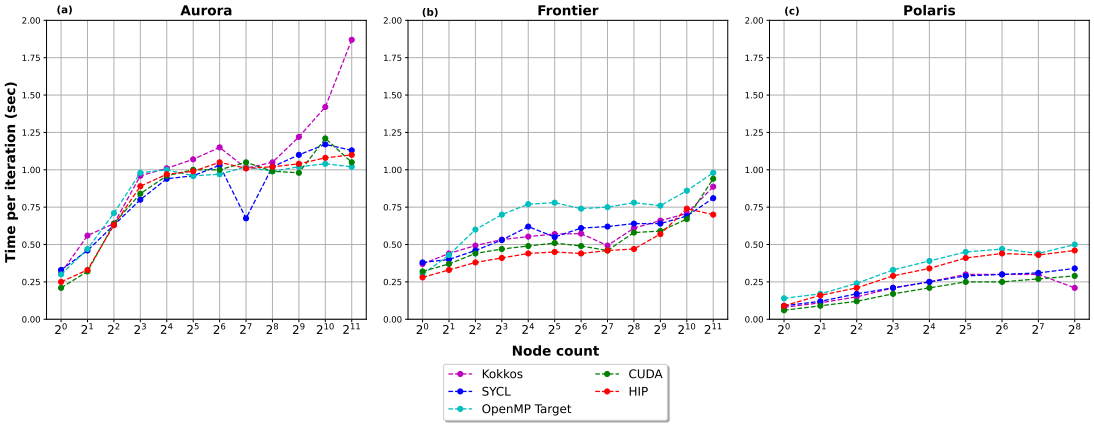

Fig. 5. Weak scaling comparison of programming models on GPUs from (a) Aurora, (b) Frontier, and (c) Polaris.

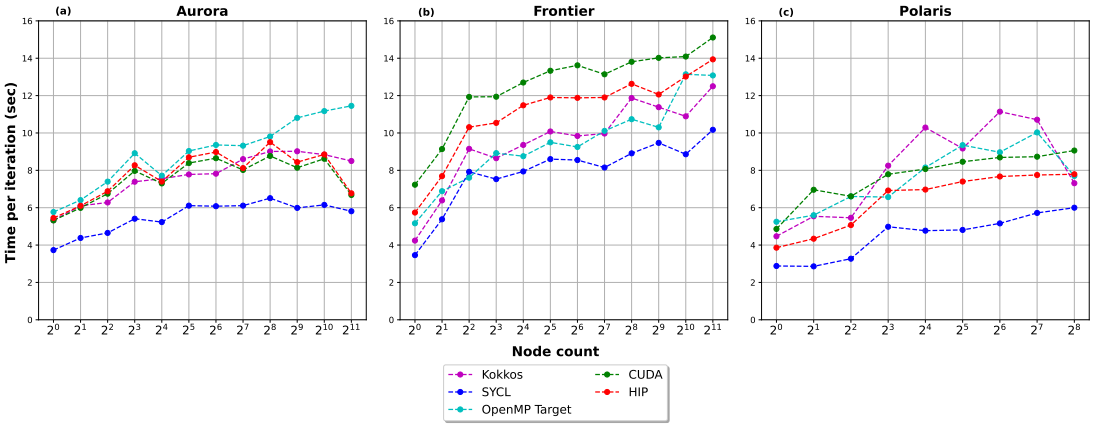

Fig. 6. Weak scaling comparison of CPU-backends of programming models on CPUs from (a) Aurora, (b) Frontier, and (c) Polaris.

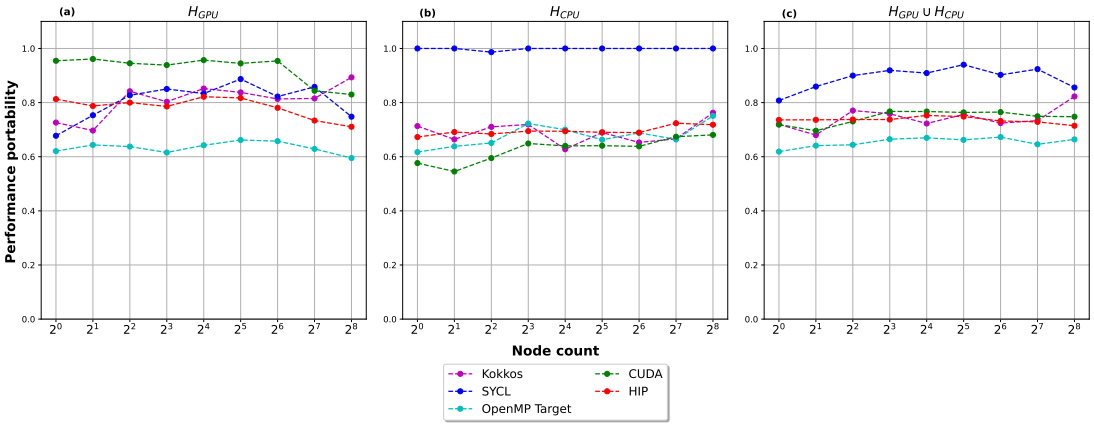

Fig. 7. Performance portability metric computed based on application efficiencies among (a) H_{GPU}, (b) H_{CPU}, and (c) $H_{GPU} \cup H_{CPU}$. H_{GPU} consists of all available logical GPUs, H_{CPU} comprises the available hardware threads on each node, and $H_{GPU} \cup H_{CPU}$ is the union of both the GPUs and CPUs over the full scaling range of each platform.

Frontier, however, MPI-related overhead was overshadowed by the IBM spreading kernel (brown). We suspected that these trends were attributable to differences in how atomic addition operations in the IBM spreading routine were handled between programming models or by the underlying hardware, and predict that replacing these atomics with a more optimized parallel reduction would lead to a runtime composition comparable to what was observed for Aurora and Polaris. In general, the relative MPI overhead among the CPU backends (bottom row of Fig. 8) was less than on the GPUs, being replaced by the more computationally intensive IBM kernels like FEM. In contrast to the GPU backends, the MPI communication is dominated by IBM Cell Update Communication. Interestingly for Frontier (middle column), the runtime became dominated not by IBM spreading but instead by the IBM cell property update. The IBM cell property update includes calculating the updated cell centroids, min and max positional coordinates, and cell volumes. Notably, the cell property update relies on shared memory algorithms for segmented scan operations that make extensive use of barrier semantics, which according to the HIP CPU Runtime documentation can incur significant slowdown. Interestingly, SYCL (left column) exhibited the highest MPI overhead among CPU backends, suggesting that compute-intensive kernels were better parallelized by the oneAPI compiler. This is evident in the relatively efficient handling of the cell property update (green) compared to other CPU backends.

F. Comparing CPU versus GPU performance

We show the relative speedup of each GPU backend over the corresponding CPU backend for each programming model in Fig. 9. The GPU backends consistently obtained about an order-of-magnitude speedup over their CPU counterparts, with the largest speedups observed on Polaris. These results were unsurprising, given the superior memory bandwidths of GPUs relative to CPUs, together with the fact that the majority of kernels sampled for HARVEY were memory bandwidth bound (Fig. 4).

IX. Discussion

This study provides critical insights into the performance portability of GPU programming models, offering practical takeaways for developers navigating heterogeneous environments. We expect the findings presented here to generalize to a broad set of HPC applications, given the fact that HARVEY exhibits characteristics of both memory bandwidth-bound and compute-bound codes (Fig. 4), employs common memory access and communication patterns (e.g., stencil-based, hybrid MPI+X), and contains kernels commonly found in scientific applications (e.g., finite element solver). One of the key findings is the comparable performance across most GPU programming models on all tested platforms, with the exception of OpenMP, as indicated in Fig. 5. On the NVIDIA-based Polaris system, OpenMP was consistently the worst performing implementation, which was similarly observed by [Davis et al.(2024a)]. On Frontier, OpenMP underperformed compared to other programming models on multiple nodes, with a consistent $\sim$30% slowdown. However, when Frontier is excluded, OpenMP's competitiveness with other models improves, emphasizing the need to account for platform-specific factors when evaluating programming models. In contrast to the GPU backends, greater performance variability was observed with respect to the CPU backends. This result was unsurprising given the relative immaturity of compilers (e.g., CuPBoP) supporting CPU offload of GPU-centric programming languages.

Among GPUs, CUDA demonstrated the highest performance portability, which is unexpected given its reputation as a non-portable language compared to alternatives like Kokkos. This result highlights a reassuring point for users of legacy HPC codes based on CUDA C++: they may not have to invest heavily in porting their code to other programming models. Instead, they can focus on optimizing their existing CUDA-based implementations, as CUDA continues to deliver strong performance across GPU platforms. That said, CUDA's portability on CPUs remains a challenge, ranking lowest among models tested. This limitation underscores the importance of ongoing developments such as CuPBoP, which may enhance support for CPU backends in the future. In the meantime, CUDA's presence on CPUs does offer significant debugging advantages, even if performance lags.

SYCL emerged as a standout in this study, demonstrating unmatched performance on CPUs and the highest overall performance portability across the combined set of GPU and CPU platforms. SYCL has previously been shown to exhibit superior performance on multi-core CPUs [Breyer et al.(2022)]. Furthermore, SYCL was the only programming model that could be compiled on all platforms using the same compiler (oneAPI/DPC++). This versatility makes SYCL a compelling option for developers targeting heterogeneous systems. For teams working with a mixture of CPUs and GPUs, SYCL presents a strong case as both a starting point for new projects and a migration target for legacy codes. While SYCL offers excellent performance portability across both CPUs and GPUs, our findings highlight several important nuances. Specifically, CUDA's superior performance portability on GPUs (Fig. 7(A)), coupled with its strong native performance on NVIDIA devices (Fig. 5(C)), may justify the continued maintenance of a legacy CUDA codebase alongside SYCL. This finding is particularly relevant given the widespread use of NVIDIA GPUs and the increasing availability of projects like chipStar and CuPBoP that aim to improve CUDA portability.

The sensitivity of program performance to code optimizations across different programming models and node architectures underscores the importance of platform-specific considerations. One striking example is the application of cell reduction optimizations that removed certain atomic operations. As shown in Fig. 3, these changes yielded substantial speedup on Frontier, with notable exception of SYCL, likely due to differences in compiler optimizations and GPU hardware intrinsics. Similarly, offloading CPU communication tables

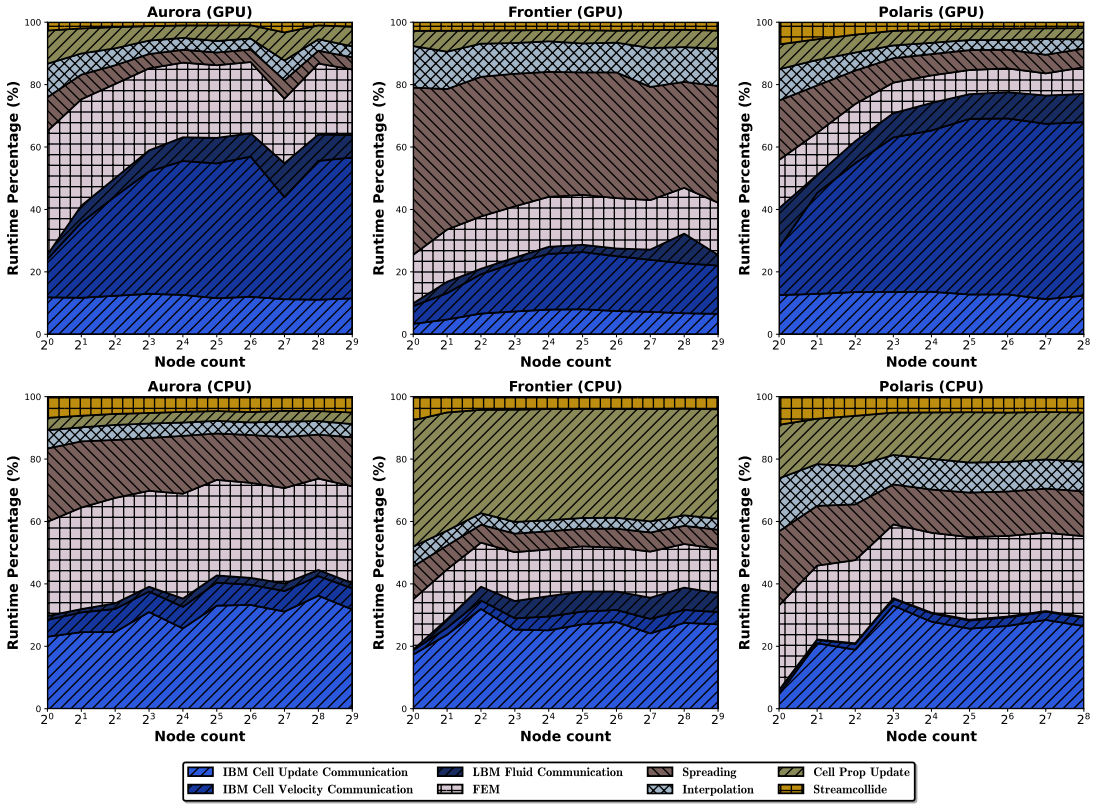

Fig. 8. Runtime compositions among GPU and CPU backends for the native programming models to each system. From left to right by column: SYCL (Aurora), HIP (Frontier), and CUDA (Polaris). From top to bottom by row: GPU backend, CPU backend.

onto GPUs resulted in significant performance gains across all platforms (about an order-of-magnitude in many cases on a single node), especially on the NVIDIA GPUs. These results highlight that while general optimizations are valuable, their effectiveness can vary greatly depending on the hardware and programming mdoel specifics.

Assessing the effort required to port applications to different programming models remains a complex task, often relying on subjective measures. While lines of code (LOC) changed are commonly cited, they may not fully capture the nuances of porting large-scale applications [Harrell et al.(2018)]. For instance, porting CUDA to HIP involves minimal effort due to the close similarity between the two, often achievable through automated scripts or simple manual regular expressions. In contrast, porting CUDA applications to SYCL requires a deeper understanding of the SYCL programming model, especially its buffer-based memory management, which differs significantly from CUDA's constant memory paradigm. Although tools like Intel's Data Parallel C++ Compatibility Tool (DPCT) can expedite the process, the resulting code may include wrappers that are less portable and harder to debug. We found that rewriting these wrappers in pure SYCL not only enhanced portability but also provided an opportunity to better understand and leverage the SYCL programming model.

Directive-based programming approaches, such as OpenMP, present their own challenges when applied to GPUs. While OpenMP offers a relatively simple mechanism for parallelizing multicore CPU applications, its GPU implementations require careful tuning to achieve comparable performance. Limitations in the OpenMP standard, including inconsistent support for advanced features like memory allocators across compilers, can hinder adoption. By comparison, high-level abstractions provided by programming models like Kokkos or oneAPI/DPC++ offer more robust support for parallel algorithms, albeit with a steeper learning curve for users accustomed to lower-level models like CUDA.

Porting from CUDA applications to Kokkos can involve substantial code changes due to differences in memory models and abstractions. However, this investment pays dividends in terms of growing community support, portability, and a rich set of features including support for distributed shared memory with Kokkos Remote Spaces. Ultimately, the choice of programming model often depends on user preferences and the specific requirements of the target platform, as our results showed comparable performance among GPU programming models (Fig. 5).

As detailed in Section III, HARVEY's independence from external libraries provided significant flexibility during this study, allowing us to replace Thrust-based reduction operations with custom kernels to support chipStar. However, this level of independence may not be feasible for many HPC applications, where leveraging highly optimized, well-established libraries

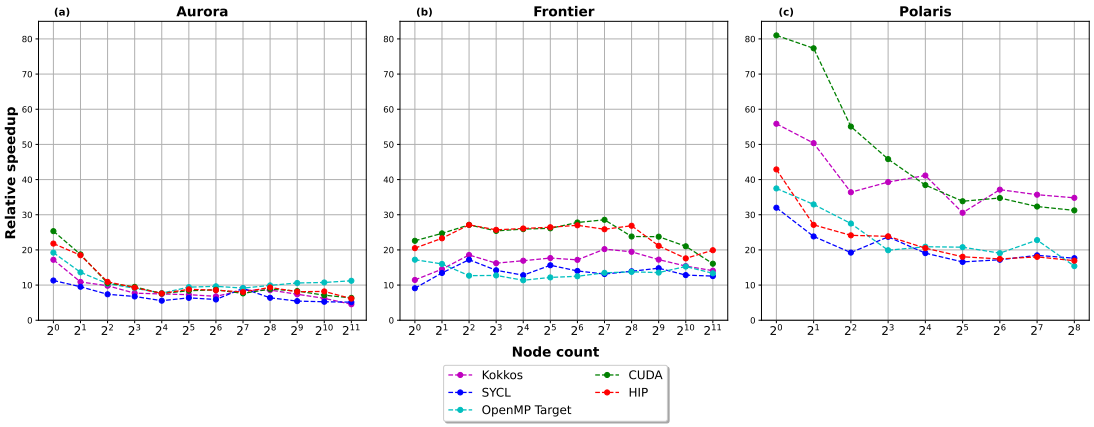

Fig. 9. Relative speedups between the GPU and CPU implementations for (a) Aurora, (b) Frontier, and (c) Polaris. Relative speedup is calculated as the total simulation loop time taken by the GPUs divided by the corresponding time taken by the CPU-backend.

is critical. We caution against a blanket approach of replacing libraries for the sake of portability, as doing so can introduce unnecessary complexity and performance trade-offs.

While offload acceleration is conventionally thought of in terms of GPU programming, our experiences from this study have taught us that many GPU programming methodologies find good application in multi-core CPU programming. In several cases, kernels originally written for GPUs outperformed traditional MPI + OpenMP implementations on CPUs. However, results from Fig. 8 showing high overhead from shared memory algorithms with the HIP CPU runtime underscore the importance of considering hardware differences between multi-core CPUs and GPUs when writing SIMT-style kernels. Overall, our finding highlights the broad applicability of accelerator-style programming, which fosters parallel programming practices that are increasingly valuable in today's heterogeneous computing environments.

X. Conclusion

This work has highlighted actionable strategies for achieving performance portability in large-scale scientific codes on leadership-class supercomputers with heterogeneous architectures. By running HARVEY, a single-source, large-scale scientific code, across CPUs and GPUs from all major vendors, we gained critical insights into the strengths and trade-offs of various programming models, providing a roadmap for developers navigating the complex landscape of HPC.

A key takeaway is CUDA's surprising viability as a performance-portable programming language for GPU-based workloads, achieving order-of-magnitude speedups over its CPU counterparts. This result, combined with tools like CuPBoP, suggests that developers of legacy HPC codes based on CUDA C++ can extend their applications to exascale machines with minimal rework, enabling them to fully exploit modern heterogeneous systems while focusing on optimization rather than large-scale rewrites.

SYCL emerged as the most performance-portable programming model across heterogeneous nodes, consistently delivering strong performance on both the CPUs and GPUs, even as scaling demands increased. Its cross-platform flexibility positions SYCL as a compelling choice for applications targeting diverse hardware ecosystems. While CUDA may still outperform SYCL on GPUs, SYCL's versatility offers a distinct advantage for future-proofing codes, particularly for those seeking a unified approach across mixed architectures.

This study highlights the versatility of accelerator-style programming, demonstrating that GPU-designed kernels can outperform traditional MPI + OpenMP approaches on multi-core CPUs, emphasizing the need for parallel programming practices that work across diverse hardware. Platform-specific optimizations, such as offloading CPU communication tables to GPUs, further illustrate the benefits of tailoring code to hardware-specific characteristics.

Our results also show that developers are no longer limited by vendor-specific solutions. Advances in compiler ecosystems and heterogeneous offloading now make scalable performance achievable across architectures, reducing barriers to adopting exascale systems. By aligning the programming models with the strengths of the platform, HPC developers can maximize efficiency and portability.

XI. Acknowledgments

The authors thank Guinevere Ferreira and Timothy King for fruitful discussions. Compute time at ORNL and ANL was provided by the U.S. Department of Energy (DOE) INCITE Program. This research used resources of the Argonne Leadership Computing Facility, a U.S. DOE Office of Science user facility at Argonne National Laboratory and is based on research supported by the U.S. DOE Office of Science-Advanced Scientific Computing Research Program, under Contract No. DE-AC02-06CH11357. ORNL is supported by the U.S. DOE Office of Science under Contract DE-AC05-00OR22725. Computing support for this work came from the Argonne National Laboratory (ANL) Aurora Early Science program.

REFERENCES

[Breyer et al.(2022)] Marcel Breyer, Alexander Van Craen, and Dirk Pflüger. 2022. A comparison of sycl, opencl, cuda, and openmp for massively parallel support vector machine classification on multi-vendor hardware. In *Proceedings of the 10th International Workshop on OpenCL*. 1–12.

[Chen et al.(2023)] Jun Chen, Xule Zhou, and Hyesoon Kim. 2023. CuPBoP-AMD: Extending CUDA to AMD Platforms. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. 1093–1104.

[Davis et al.(2024a)] Joshua Hoke Davis, Pranav Sivaraman, Isaac Minn, Konstantinos Parasyris, Harshitha Menon, Giorgis Georgakoudis, and Abhinav Bhatele. 2024a. *An Evaluative Comparison of Performance Portability across GPU Programming Models*. Technical Report. Lawrence Livermore National Laboratory (LLNL), Livermore, CA (United States).

[Davis et al.(2024b)] Joshua H Davis, Pranav Sivaraman, I Minn, Konstantinos Parasyris, Harshitha Menon, G Georgakoudis, and A Bhatele. 2024b. Taking GPU Programming Models to Task for Performance Portability. *arXiv preprint arXiv:2402.08950* (2024).

[Gounley et al.(2019)] John Gounley, Erik W Draeger, and Amanda Randles. 2019. Immersed boundary method halo exchange in a hemodynamics application. In *Computational Science–ICCS 2019: 19th International Conference, Faro, Portugal, June 12–14, 2019, Proceedings, Part I 19*. Springer, 441–455.

[Gschwandtner et al.(2021)] Philipp Gschwandtner, Ralf Kissmann, David Huber, Philip Salzmann, Fabian Knorr, Peter Thoman, and Thomas Fahringer. 2021. Porting Real-World Applications to GPU Clusters: A Celerity and Cronos Case Study. In *2021 IEEE 17th International Conference on eScience (eScience)*. IEEE, 90–98.

[Han et al.(2024a)] Ruobing Han, Jun Chen, Bhanu Garg, Xule Zhou, John Lu, Jeffrey Young, Jaewoong Sim, and Hyesoon Kim. 2024a. CuPBoP: Making CUDA a Portable Language. *ACM Transactions on Design Automation of Electronic Systems* 29, 4 (2024), 1–25.

[Han et al.(2022)] Ruobing Han, Jaewon Lee, Jaewoong Sim, and Hyesoon Kim. 2022. COX: Exposing CUDA warp-level functions to CPUs. *ACM Transactions on Architecture and Code Optimization (TACO)* 19, 4 (2022), 1–25.

[Han et al.(2024b)] Ruobing Han, Jisheng Zhao, and Hyesoon Kim. 2024b. Unleashing CPU Potential for Executing GPU Programs through Compiler/Runtime Optimizations. In *2024 57th IEEE/ACM International Symposium on Microarchitecture*. IEEE.

[Harrell et al.(2018)] Stephen Lien Harrell, Joy Kitson, Robert Bird, Simon John Pennycook, Jason Sewall, Douglas Jacobsen, David Neill Asanza, Abaigail Hsu, Hector Carrillo Carrillo, Hessoo Kim, et al. 2018. Effective performance portability. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 24–36.

[Howard et al.(2017)] Micah Howard, Andrew Bradley, Steven W Bova, James Overfelt, Ross Wagnild, Derek Dinzl, Mark Hoemmen, and Alicia Klinvex. 2017. Towards performance portability in a compressible cfd code. In *23rd AIAA Computational Fluid Dynamics Conference*. 4407.

[Joo et al.(2019)] 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.

[Krüger et al.(2011)] Timm Krüger, Fathollah Varnik, and Dierk Raabe. 2011. Efficient and accurate simulations of deformable particles immersed in a fluid using a combined immersed boundary lattice Boltzmann finite element method. *Computers & Mathematics with Applications* 61, 12 (2011), 3485–3505.

[Krüger et al.(2016)] Timm Krüger, Halim Kusumaatmaja, Alexander Kuzmin, Orest Shardt, Goncalo Silva, and Erlend Magnus Viggen. 2016. *The Lattice Boltzmann Method - Principles and Practice*. https://doi.org/10.1007/978-3-319-44649-3

[Louhivuori(2023)] Martti Louhivuori. 2023. HOP - Header only porting. Github Repository. https://github.com/cschpc/hop

[Martin et al.(2024)] Aristotle Martin, Geng Liu, Balint Joo, Runxin Wu, Mohammed Shihab Kabir, Erik W Draeger, and Amanda Randles. 2024. Designing a GPU-Accelerated Communication Layer for Efficient Fluid-Structure Interaction Computations on Heterogeneous Systems. In *2024 SC24: International Conference for High Performance Computing, Networking, Storage and Analysis SC*. IEEE Computer Society, 1483–1502.

[Pennycook et al.(2016)] Simon J Pennycook, Jason D Sewall, and Victor W Lee. 2016. A metric for performance portability. *arXiv preprint arXiv:1611.07409* (2016).

[Peskin(1977)] Charles S Peskin. 1977. Numerical analysis of blood flow in the heart. *Journal of Computational Physics* 25, 3 (1977), 220–252.

[Randles et al.(2013)] Amanda Peters Randles, Vivek Kale, Jeff Hammond, William Gropp, and Efthimios Kaxiras. 2013. Performance analysis of the lattice Boltzmann model beyond Navier-Stokes. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 1063–1074.

[Rangel et al.(2023)] Esteban Miguel Rangel, Simon John Pennycook, Adrian Pope, Nicholas Frontiere, Zhiqiang Ma, and Varsha Madananth. 2023. A Performance-Portable SYCL Implementation of CRK-HACC for Exascale. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. 1114–1125.

[Reguly(2019)] István Z Reguly. 2019. Performance portability of multi-material kernels. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 26–35.

[Reguly(2023)] Istvan Z Reguly. 2023. Evaluating the performance portability of SYCL across CPUs and GPUs on bandwidth-bound applications. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. 1038–1047.

[Ruzicka et al.(2024)] Josef Ruzicka, Christian Asch, Esteban Meneses, Markus Rampp, and Erwin Laure. 2024. A Study of Performance Portability in Plasma Physics Simulations. *arXiv preprint arXiv:2411.05009* (2024).

[Sengupta et al.(2011)] Shubhabrata Sengupta, Mark J Harris, Michael Garland, and John D Owens. 2011. *Efficient parallel scan algorithms for many-core gpus*. eScholarship, University of California.

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