Taking GPU Programming Models to Task for Performance Portability

Anonymous Author(s)

***Abstract*—Portability is critical to ensuring high productivity in developing and maintaining scientific software as the diversity in on-node hardware architectures increases. While several programming models provide portability for diverse GPU systems, they don't make any guarantees about performance portability. In this work, we explore several programming models – CUDA, HIP, Kokkos, RAJA, OpenMP, OpenACC, and SYCL, to assess the consistency of their performance across NVIDIA and AMD GPUs. We use five proxy applications from different scientific domains, create implementations where missing, and use them to present a comprehensive comparative evaluation of the programming models. We provide a Spack scripting-based methodology to ensure reproducibility of experiments conducted in this work. Finally, we attempt to answer the question – to what extent does each programming model provide performance portability for heterogeneous systems in real-world usage?**

***Index Terms*—performance portability, heterogeneous systems, programming models**

I. Introduction

Heterogeneous CPU-GPU architectures have come to dominate the design of high performance computing (HPC) systems. Nine of the top ten systems in the June 2024 TOP500 list, and ~39% of the systems on the complete list, employ co-processors or accelerators [1]. Further, a diverse set of specific architectures are in use, supplied by a range of vendors, as the current top ten includes GPUs from AMD, NVIDIA, and Intel. A similarly-diverse range of programming models has emerged, which aim to allow application developers to write their code once and run it on any system. Programming models such as OpenMP [2], RAJA [3], and Kokkos [4] act as *portability layers*, bridging the gap between high-level implementation of an algorithm and low-level execution on a given target architecture. Yet running scientific applications efficiently on HPC systems requires more than just *functional* portability, which refers to program correctness. Codes must also perform well on a range of target systems, ideally without incurring the technical debt of system-specific implementations. This is often referred to as *performance* portability.

Application developers would benefit from a deeper understanding of the performance portability provided by different programming models on modern GPU systems before porting their application to a particular model. Choosing a programming model for porting a CPU-only application to GPUs is a major commitment, requiring significant time for developer training and programming. If a programming model delivers unacceptable performance, then that investment is wasted.

Nevertheless, each programming model's effectiveness at enabling performance portability, as well as the definition of performance portability itself, remain open questions. Although developers' experiences comparing the performance portability of several models on a single application are valuable, we have observed that open-source applications or even proxy applications implemented in a several different programming models are uncommon and difficult to find. Further, a single smaller application or benchmark implemented in most programming models is unlikely to be representative of the diverse and complex production applications typically run on HPC systems. Finally, conducting exhaustive combinatorial studies of programming model, compiler, system, and application combinations is a significant undertaking, as each programming model usually requires unique combinations of compilers flags and libraries for any given system.

In this paper, we provide a comprehensive empirical study of the performance portability of several programming models on GPU-based leadership-class supercomputers. We use a variety of proxy applications that are representative of production codes, and using them, we enable realistic comparisons of the performance portability of GPU kernels written in several programming models across different architectures. We study five proxy applications from different scientific domains, create implementations where missing, and comprehensively evaluate differences between these programming models.

We present a Spack-based [5] environment and scripting system to significantly lower the barrier for performance portability studies. This system encapsulates our methodology for systematically building, running and benchmarking a suite of applications in several programming models, in a manner which can be adapted for future studies. Our comparative evaluation of model performance includes specific insights into why certain programming models perform well or poorly for particular applications on different target systems. To our knowledge, this is one of the most comprehensive performance portability studies to date, in terms of the breadth of programming models and applications studied and the detail provided in the analysis of results.

To summarize, our contributions include the following:

- We evaluate the performance portability enabled by seven different programming models using a diverse set of five proxy applications benchmarked across NVIDIA and AMD GPUs in production supercomputers.
- We create several additional implementations of existing proxy applications in new programming models to ensure full coverage of programming models across applications.
- We describe a methodology employing Spack scripting and environment tools [5] to easily manage the process

of building and running all $7 \times 5 = 35$ versions across four supercomputing systems, each with unique software stacks. We provide this software to the community in order to substantially reduce the effort required to reproduce or extend our study.
- We conduct a thorough analysis of the reasons for key outliers in the performance portability cases studied, and describe and test optimizations that improve performance portability in some cases.

II. Background on Portable Programming Models

In this section we provide relevant background information on the various programming models we evaluate. Table I displays key information about each programming model. HIP and CUDA act as our baselines in this study, as they are the native programming model for AMD and NVIDIA devices, respectively. Below, we describe the key characteristics of each category of programming model.

TABLE I
Summary of programming models used in this study. Vendor support may be subject to change in the future.

Prog. Model	Category	GPU Vendors Supported
CUDA	Language extension	NVIDIA
HIP	Language extension	NVIDIA, AMD
SYCL	Language extension	NVIDIA, AMD, Intel
Kokkos	C++ abstraction lib.	NVIDIA, AMD, Intel
RAJA	C++ abstraction lib.	NVIDIA, AMD, Intel
OpenMP	Directive-based	NVIDIA, AMD, Intel
OpenACC	Directive-based	NVIDIA, AMD

A. *Language extensions*

SYCL, HIP, and CUDA are language extensions, which add features to the base language (C++, C, and/or Fortran) for programming GPUs. SYCL and HIP are open standards, while CUDA is proprietary. The language extensions we consider are more verbose than the other programming models. Users call runtime functions to manage memory and write functions that they then invoke as kernels to offload execution. SYCL provides multiple methods of memory management, including the *explicit USM (unified shared memory)* API, which uses CUDA or HIP style runtime calls to move and allocate data, or the *buffer/accessor* API, which is more implicit, allowing the compiler and runtime to schedule data movement but does not allow explicit access to valid device pointers.

B. *C++ abstraction libraries*

Kokkos and RAJA are C++ abstraction libraries. These are template-based C++ libraries that provide high-level functions and data types. Users write their code directly employing these data types and typically structure GPU code as lambdas to pass into library function calls. The library translates the user code to a device backend such as CUDA, HIP, or OpenMP at compile-time or runtime. Note that Kokkos provides both memory and compute abstractions, while RAJA provides compute abstractions and users must employ the related Umpire or CHAI libraries to abstract memory management.

C. *Directive-based models*

OpenMP and OpenACC are directive-based models. They provide compiler directives, or *pragmas*, to parallelize or offload code. They are typically standard specifications implemented by a compiler front-end and a runtime library to implement parallel or offloaded execution that abstracts the underlying hardware architecture. Directive-based models are usually less verbose and less intrusive, as users can often annotate existing code with minimal refactoring. This facilitates incremental development. These models provide clauses and standalone directives to schedule data movement, which are then carried out by the compiler and device runtime.

III. Related Work

Several studies on programming language extensions, models, and libraries have been designed to assist developers achieve performance portability [2]–[4], [6], [7]. Additionally, several studies have assessed the portability of certain frameworks. We categorize the related work on empirical performance portability studies into three groups: metric studies, application or programming model studies, and broader studies that are not scoped to a particular model or app. In this section, we provide an overview of recent work in each category.

A. *Studies of performance portability metrics*

Pennycook et al. propose the metric Φ for performance portability, defining it as the harmonic mean of the performance efficiencies of an application across different systems [8]–[12]. Daniel et al. propose an alternative metric, P_D, which accounts for problem size, and Marowka compares Φ with $\overline{\Phi}$, a similar metric that uses the arithmetic mean instead of the harmonic mean [13]–[15].

B. *Studies examining the portability of individual application categories or programming models*

A number of studies evaluate performance portability in specific applications with multiple programming models or a single programming model model [6], [16]–[28]. For instance, Dufek et al. compare Kokkos and SYCL for the Milc-Dslash benchmark, while Rangel et al. examine the portability of CRK-HACC in SYCL [21], [24]. Other studies investigate performance portability across applications using specific programming models. Brunst et al. benchmark the 2021 SPEChpc suite, which contains nine mini applications in OpenMP and OpenACC, on Intel CPUs and NVIDIA and AMD GPUs [26]. Kuncham et al. evaluate the relative performance of SYCL and CUDA on the NVIDIA V100 using BabelStream, Mixbench, and Tiled Matrix-Multiplication [27].

While these studies provide useful information to developers working on similar applications or those interested in specific programming models, making more general statements about programming models themselves requires a more comprehensive evaluation of a diverse set of case studies.

C. Broader performance portability studies

Deakin et al. present performance portability studies of five programming models across a wide range of hardware architectures, using BabelStream, TeaLeaf, CloverLeaf, Neutral, and MiniFMM [29], [30]. More recent papers by Deakin et al. focus on more specific problems such as reductions and GPU to CPU portability [31], [32]. Lin et al. evaluate implementations of C++17 StdPar against five models on AMD devices [33]. While these studies provide performance portability comparisons across systems, applications, and models, they do not include RAJA and sometimes omit HIP and OpenACC. Furthermore, they do not provide extensive analysis of the reasons for performance differences between programming models or ways to address differences.

Several other studies are similar in scope but different in focus. Kwack et al. evaluate portability development experiences for three full applications and three proxy applications across GPUs from multiple vendors [34]. Harrell et al. study performance portability alongside developer productivity [35]. However, in these studies each application is only ported to a single portable programming model. This makes it difficult to draw conclusions about each programming model's relative suitability to particular applications. Koskela et al. provide six principles for reproducible portability benchmarking, along with a demonstration of these principles in a Spack+Reframe CI infrastructure for a study of BabelStream on some CPU architectures and an NVIDIA V100 [36].

Studies on various aspects of performance portability abound, but all are limited in at least one manner. Some are limited to a single application or benchmark, preventing comparisons of programming models across applications. Others are focused on a single programming model, disallowing comparisons between different programming models. Our work aims to provide a comprehensive analysis of performance portability across multiple applications and libraries, each implemented in several different programming models and executed on production supercomputers. Additionally, unlike prior work, we conduct a detailed investigation of the performance of the most notable outliers we identify among our results, providing users with a better understanding of how application characteristics impact the performance portability of each model as well as potential workarounds to avoid portability pitfalls. Finally, prior studies do not provide a comprehensive description of the build and run infrastructure used to collect their results, leaving the task of consistently building applications on a wide range of systems with complex library and compiler flag dependencies to the reader. Our study is the first to apply the principles of reproducible benchmarking [36] in a comprehensive study of performance-portable programming models.

IV. Methodology for Evaluating Performance Portability on GPU Platforms

In this section, we describe our approach to comprehensively compare programming models that provide portability on GPU systems. We also justify for our choices of programming models, proxy applications, systems, and metrics.

A. Choice of programming models

Our goal in this work is to empirically compare the performance portability provided by popular programming models. In Section II, we describe three categories of programming models with a few examples in each category. We identified those representative models by surveying a broad range of proxy applications in order to determine how common existing implementations in each model were. We surveyed a variety of sources for proxy applications, including the ECP Proxy Apps suite [37], the NERSC Proxy suite [38] and the Mantevo Applications Suite [39]. Armed with that knowledge, we have decided to focus on CUDA, HIP, SYCL, Kokkos, RAJA, OpenACC, and OpenMP, as they were most commonly found in the proxy applications we surveyed. Together, these models cover the three categories of models mentioned earlier.

B. Choice of proxy applications

Based on the survey of proxy applications mentioned above, we identify five applications that represent the range of typical scientific computing workloads on GPU clusters. These include a pure memory bandwidth benchmark as well as four other proxy applications. They range from highly compute-intensive (miniBUDE) to highly memory-intensive (BabelStream), and also include one representative from each of the three large proxy application suites we surveyed. The scientific domains covered by them include hydrodynamics (CloverLeaf), molecular dynamics (miniBUDE), nuclear physics (XSBench), and particle physics (su3_bench), and computational methods include structured grid (CloverLeaf and su3_bench), dense linear algebra (su3_bench), n-body (miniBUDE) and Monte Carlo (XSBench) methods.

CloverLeaf, miniBUDE, and XSBench are missing implementations in some programming models compared in this work. So, we develop these missing implementations to obtain full coverage of the space of application and model combinations. Table II summarizes the key details of each proxy application, and identifies the implementations that were either created or modified by us for this study. Here, modifications refers to small changes to the memory management library or style to ensure portability and consistency of gathering execution times across implementations. Below, we describe in brief the five proxy applications that we use in this study:

BabelStream is a memory bandwidth benchmark with five kernels: `copy`, `add`, `mul`, `triad`, and `dot` [22]. The dot kernel includes a reduction operation, known to be a challenging operation for some programming models [40].

XSBench [41] is a proxy for OpenMC, a Monte Carlo transport code [42]. XSBench runs one kernel, OpenMC's macroscopic cross-section lookup kernel, with a large number of lookups. We use the event-based transport method with a hash-based grid as it is preferred for GPUs.

TABLE II
SUMMARY OF PROXY APPLICATIONS AND BENCHMARKS USED IN THIS STUDY AS WELL AS WHICH PROGRAMMING MODEL PORTS AND SPACK PACKAGES ARE UPDATED OR CREATED BY THE AUTHORS. HERE, E = ALREADY EXISTS, M = MODIFIED BY US, C = CREATED BY US.

Proxy Application	Scientific Domain	Method(s)	Suite	CUDA	HIP	SYCL	Kokkos	RAJA	OpenMP	OpenACC	Spack Pkg.
BabelStream	N/A	Bandwidth benchmark	N/A	E	E	E	E	**M**	E	E	**M**
XSBench	Nuclear physics	Monte Carlo	ECP	E	E	**M**	**C**	**C**	E	**C**	**M**
CloverLeaf	Hydrodynamics	Structured grid	Mantevo	E	E	**M**	E	**C**	E	**C**	**M**
su3_bench	Particle physics	Structured grid, dense lin. alg.	NERSC	E	E	E	E	**C**	E	E	**C**
miniBUDE	Molecular dynamics	N-body	N/A	E	E	**M**	E	**M**	E	E	**C**

TABLE III
ARCHITECTURAL DETAILS OF THE SYSTEMS USED IN THIS PAPER.

System	CPU Architecture	CPU Cores/node	CPU Memory (GB)	GPU Model	GPU Memory (GB)	Hosting Facility
Summit	IBM POWER9	44	512	NVIDIA V100	32*	OLCF (ORNL)
Perlmutter	AMD EPYC 7763	64	256	NVIDIA A100	40	NERSC (LBL)
Corona	AMD EPYC 7401	48	256	AMD MI50	32	LC (LLNL)
Frontier	AMD EPYC 7713	64	512	AMD MI250X	$64^{\dagger}$	OLCF (ORNL)

* We use the high-memory GPUs on Summit.
† This is for one GCD of an MI250X on Frontier.

CloverLeaf is a 2D structured compressible Euler equation solver, with 14 kernels [43]. The `advec_mom`, `advec_cell`, `PdV`, and `calc_dt` kernels are typically the most time-intensive, and `calc_dt` contains a reduction.

su3_bench [44] is a proxy application for MILC, a lattice quantum chromodynamics code [45]. It implements the SU(3) matrix-matrix multiply routine in its lone kernel.

miniBUDE is a proxy for Bristol University Docking Engine (BUDE), a molecular dynamics code which simulates molecular docking for drug discovery [46]. miniBUDE computes the energy field for a single configuration of a protein repeatedly.

C. Choice of systems

Evaluating performance portability requires selecting a range of systems with diverse architectures. One of the main goals of this study is to evaluate performance portability on production GPU-based supercomputers, given the rising prominence of GPUs in new systems [1]. We select four different supercomputers for our experiments: Summit and Frontier at ORNL, Perlmutter at NERSC, and Corona at LLNL (architectural details in Table III). These systems cover the majority of the GPU architectures in the top ten systems. Frontier and Summit are in the top ten, and Perlmutter is in the top fifteen. Additionally, we include Corona (AMD MI50) to provide additional context with older AMD hardware. Note that, for Frontier's MI250X GPUs, we run on one Graphics Compute Die (GCD) but refer to the GCD as a GPU in consistency with the system's documentation[1].

[1]https://docs.olcf.ornl.gov/systems/frontier_user_guide.html#frontier-compute-nodes.

D. Measurement and evaluation strategy

In this study, we modify applications where needed to consider both the efficiency of GPU kernel(s) and that of data movement between host and device needed to run the application. However, as will be discussed in Sec. VII, the impact of data movement on overall performance is minimal for these applications and not presented in detail. We add a runtime option to all the applications to specify a number of warmup iterations at the start of the simulation which we exclude from timing. XSBench normally runs only for only a single iteration, so we add a loop that repeatedly runs the kernel a user-specified number of times to ensure consistency across applications. As will be mentioned in Sec. VI, variability across runs is low, with runs of a given setup differing by at most 3.3%.

Having determined how to consistently define performance for each application, we can also derive additional higher-level metrics about performance portability for each combination of application and programming model. In this work, we use Φ **with application efficiency** proposed by Pennycook et al. [8]. Φ is defined, for some application a, problem p, set of systems H, and measure of application efficiency e, as:

$$\Phi(\underbrace{a}_{\text{application}}, \underbrace{p}_{\text{problem}}, \overset{\text{set of systems}}{H}) = \begin{cases} \dfrac{|H|}{\sum_{i \in H} \dfrac{1}{e_i(a, p)}} & \text{if } i \text{ is supported } \forall i \in H \\ 0 & \text{otherwise.} \end{cases}$$

This is the harmonic mean of the efficiencies of an application running the same input problem across a set of systems.

The application efficiency $e_i(a, p)$ of an application a solving problem p is the ratio $\frac{t_{min}}{t}$, where t is the runtime of a solving p on the particular hardware i, and t_{min} is the best observed runtime across all variants of a solving p on i. Φ ranges from 0 to 1, where 1.0 is perfect portability in which the application runs at the best observed performance on all systems.

E. Automation and reproducibility strategy

In our experiments, we ensure that compilers, dependency versions, and flags are used consistently across applications and systems. We accomplish this with Spack [5], a popular HPC package manager. We create a single Spack environment file for each system which specifies the exact compiler, application, and library dependency versions along with any needed flags. As listed in Table II, we have created or updated Spack package files for each proxy app, and these updates will be provided to the community. Our Spack environments for this project can be easily adapted to any new system, allowing for easy reproduction of our experiments, and significantly reducing the extremely time-consuming effort of building every combination of application and programming model.

We further employ Spack's Python scripting tools[2] to develop robust automation for our experiments — we can create jobs with a single-line invocation leveraging Spack's spec syntax to adjust which application, models, or compilers are used, and save profile data to disk to be directly read by our plotting scripts. These scripts and environments will be published to allow the community to use our portability study methodology. These infrastructural contributions dramatically reduce the effort required to reproduce our results and create new studies of portable programming models.

V. Porting to New Programming Models

The proxy applications we choose have implementations in most of the evaluated programming models. In these existing ports, we make minor modifications to consistently align timing measurements across different programming models. We also update the RAJA ports of BabelStream and miniBUDE to use Umpire for portable memory allocations.

When creating new ports, we apply the same level of effort for all of them in order to avoid granting an unfair advantage to any particular implementation arising from excess optimization. We spend similar amounts of time implementing each new port, and keep the structure of the code between new and existing ports as similar as possible. Further, we specifically do not tune kernel grid size, block size, and shared memory per block. For programming models that require the user to specify these values (CUDA, HIP, RAJA, SYCL), we use the default values provided by the respective proxy application developers. For programming models that can select their own default parameter values (OpenMP, OpenACC, Kokkos), we allow the model to do so if compatible with the existing application code. Our results reflect "out of the box" performance that a user would encounter with minimal porting effort.

[2]https://spack-tutorial.readthedocs.io/en/latest/tutorial_spack_scripting.html

In the following subsections, we discuss our experiences working with the programming models as applicable. Table II summarizes our development efforts. We plan to merge these contributions to their respective upstream repositories.

A. Porting to Kokkos

Porting the XSBench code to Kokkos requires converting the existing `for` loop to be a lambda function passed into a `Kokkos::parallel_for` call and converting the data structures to be used in Kokkos calls to `Kokkos:Views`. For example, XSBench's `SimulationData` struct contains several dynamic arrays which need to be Views in order to work on the GPU. In this situation, there are two options available to a developer: 1) rewrite all of the application code to use Views from the beginning, including any CPU-side setup or initialization; or 2) avoid rewriting the any setup code by constructing Views out of pointers to any ordinary C++ arrays after initialization but before copying them to the device and launching kernels.

We opted for the second of these methods to minimize changes to this existing application code. Listing 1 provides an example of this approach as we implemented it. In summary, we construct an unmanaged View in the `HostSpace` called `u_cocns` using the heap memory of the `SD.concs` array, construct a new View in the device space called `SD.d_concs`, and finally `deep_copy` the unmanaged host View to the new device View. While Kokkos requires developers to use its memory abstraction, the View, in order to make use of its portable kernel abstraction, we demonstrate how an application developer looking to work incrementally can minimize changes to application code while gaining the portability benefits of Kokkos.

```
View<double*, LayoutLeft, HostSpace,
     MemoryTraits<Unmanaged>>
    u_concs(SD.concs, SD.length_concs);
SD.d_concs = new View<double*>("d_concs",
                                SD.length_concs);
deep_copy(*SD.d_concs, u_concs);
```

Listing 1. Example of converting a C++ dynamic array to a device View for incremental development, where SD is a struct containing XSBench simulation data.

B. Porting to RAJA

In contrast to Kokkos, the RAJA portability ecosystem uses multiple libraries to provide portability. Briefly, the RAJA library itself provides C++ lambda-capturing to allow developers to express portable computation. For memory management, the developer can either write or use a custom portable memory management library, or use the related Umpire [47] library, which provides portable memory allocation primitives and memory pools. This separation of concerns in the RAJA ecosystem provides greater capability for incremental porting of an existing codebase (i.e., portable compute first, then portable data structures), avoiding more extensive refactoring.

In our case, we opt to take advantage of Umpire for CloverLeaf and XSBench, which both have extensive existing code for managing and initializing data structures. However,

TABLE IV

COMPILERS AND VERSIONS USED FOR BUILDING DIFFERENT PROGRAMMING MODEL IMPLEMENTATIONS ON EACH SYSTEM.

Prog. Model	Summit	Perlmutter	Corona	Frontier
CUDA	GCC 12.2.0	GCC 12.2.0	N/A	N/A
HIP	N/A	N/A	ROCmCC 5.7.0	ROCmCC 5.7.0
SYCL*	DPC++ 2024.01.20	DPC++ 2024.01.20	DPC++ 2024.01.20	DPC++ 2024.01.20
Kokkos	GCC 12.2.0	GCC 12.2.0	ROCmCC 5.7.0	ROCmCC 5.7.0
RAJA	GCC 12.2.0	GCC 12.2.0	ROCmCC 5.7.0	ROCmCC 5.7.0
OpenMP†	NVHPC 24.1	NVHPC 24.1	LLVM 17.0.6	LLVM 17.0.6
OpenACC	NVHPC 24.1	NVHPC 24.1	Clacc 2023-08-15	Clacc 2023-08-15

* We use AdaptiveCpp 23.10.0 for SYCL CloverLeaf due to performance improvement.
† We use ROCmCC 5.7.0 for OpenMP su3_bench on AMD GPUs due to performance improvement.

we encounter several challenges building the RAJA applications. Relying on multiple independent libraries increases the expertise required and frequency of errors in setting up build systems, a process that is already complicated for a single library containing device kernels. Package managers such as Spack [5] can mitigate these problems for end users, although this solution pushes the responsibility of ensuring the libraries build and install correctly onto the package maintainers.

C. Porting to OpenACC

OpenMP ports already exist for all applications, so creating similar OpenACC ports where needed just requires a one-to-one conversion of the relevant OpenMP pragmas to OpenACC. For example, `omp target teams distribute parallel for` becomes `acc parallel loop`. This rote method makes our experience with porting XSBench and CloverLeaf from OpenMP to OpenACC very productive. In contrast to Kokkos and RAJA, working with existing data structures is highly transparent in OpenACC, so long as the structures are plain old data (POD) and do not contain pointers to CPU memory internally. In those more advanced cases, which we do not encounter in this work, users must write more complex directives to handle such data structures, convert them to simpler formats, or use automatically managed memory if provided by the GPU device [**?**].

VI. EXPERIMENTAL SETUP

In this section, we describe the setup for the experiments conducted in this work. We run all the applications on all four systems selected (listed in Table III).

Table IV lists the compilers used with each programming model alongside their versions. We use GCC 12.2.0 as the host compiler on NVIDIA systems and ROCmCC 5.7.0 on AMD. We use CUDA version 12.2 on NVIDIA systems, and HIP 5.7.0 on AMD systems, as well as Kokkos version 4.2.00 and RAJA v2023.06.1. OpenACC, OpenMP, and SYCL all have different implementations provided by multiple compilers on the systems where we perform our experiments. We test all the available compilers for these models[3] and choose the best-performing compiler for each application, model, and system. We perform this compiler-choice tuning to reflect the fact that applications using these programming models will likely test their code with all working compilers, and use in practice the best-performing option. In all models except SYCL and OpenMP, the best-performing compiler is consistent across applications on each system. For the SYCL port of CloverLeaf, AdaptiveCpp is consistently superior, so we present AdaptiveCpp results for that application and DPC++ for all others. For OpenMP, ROCmCC wins on AMD systems for su3_bench and Clang wins for all other applications. Note also that we are unable to build CloverLeaf with Clacc due to lack of support for the `host_data` clause, and hence we cannot run CloverLeaf on AMD systems with OpenACC.

We select input decks and command line inputs for each proxy application based on recommended settings from their respective developers. When given a choice of problem size, we select the largest representative problem available that fits on all tested GPUs. We also choose the number of iterations for each application to ensure about a minute of execution time, so as to reduce variability. Section IV-D describes how we modify the proxy applications to ensure consistent timings. We present the final command line arguments in Table V.

TABLE V

INPUT PARAMETERS TO THE PROXY APPLICATIONS.

Application	Input parameters
BabelStream	`-n 1500 -w 150 -s $((1<<29))`
XSBench	`-s large -m event -G hash -n 150 -w 15`
CloverLeaf	`--in clover_bm64_mid.in -w 52`
su3_bench	`-l 32 -i 100000 -w 10000`
miniBUDE	`--deck bm2 -p 2 --wgsize 128 -i 10 --warmups 1`

Note that for all cases tested the time spent in data movement is negligible (less than 2%) compared to time spent in device kernels, so our result figures present **only** GPU kernel time. For all performance results presented we run the application three times and present the average result. Variability is low; the largest range of times recorded as a percentage of mean runtime for a case is 3.3%, and the mean is 0.1%. We report total runtime for BabelStream kernels rather than memory bandwidth in order to ensure that "lower is better" across all performance results we present. The values

[3]OpenMP: Clang, GCC, ROCmCC, NVHPC, CCE; OpenACC: Clacc, GCC, NVHPC; SYCL: DPC++, AdaptiveCpp

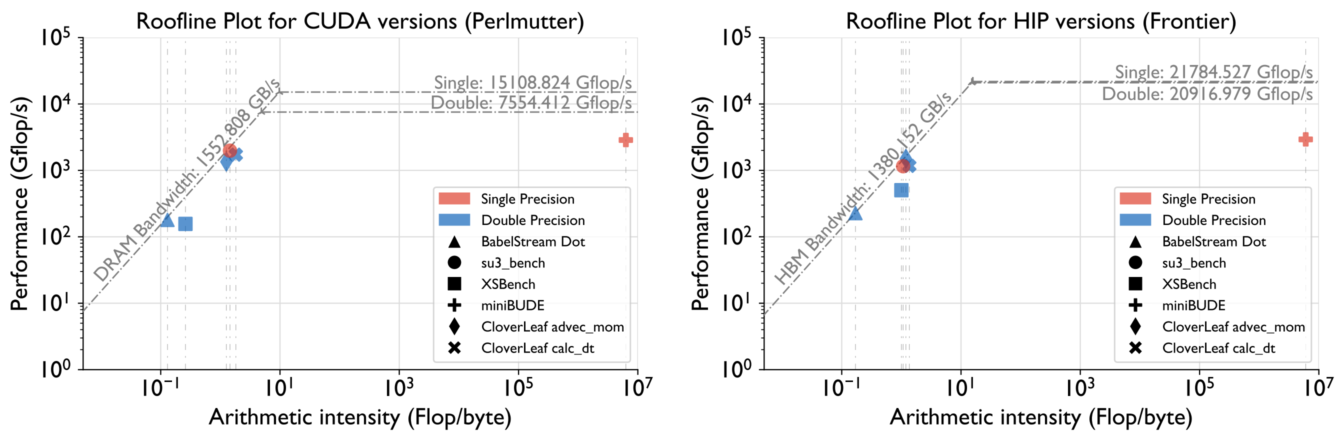

Fig. 1. Roofline plot for the most time-consuming kernel in the CUDA (left) and HIP (right) versions of each application, run on Perlmutter (NVIDIA A100) and Frontier (AMD MI250X) respectively. Red points are single precision, and blue points are double precision. For each application we plot the predominant precision used.

collected can be converted to bandwidth (GB/s) by dividing the total data moved by the time.

VII. Results and Discussion

We first present a roofline analysis of the native port implementations of each application to understand their compute and memory behavior. Next, we present the results of our detailed performance comparison across programming models, systems, and applications, first in summary and then in depth.

A. Roofline analysis

Figure 1 provides the empirical rooflines for the NVIDIA A100 GPU on Perlmutter and AMD MI250X GPU on Frontier. It also plots the positions of the most time-consuming kernels in the CUDA and HIP implementations of the five proxy applications. For BabelStream, this is the `dot` kernel, and for CloverLeaf, these are the `advec_mom` and `calc_dt` kernels. miniBUDE, XSBench, and su3_bench contain a single computational kernel each. We plot each kernel for the predominant floating-point precision used. We can quickly observe that all kernels evaluated are memory-bound except for miniBUDE, which is highly compute-bound, on both architectures. Among the memory-bound apps, on both systems BabelStream `dot` is the most memory-bound (i.e., furthest to the left). This is expected given that BabelStream is a memory bandwidth benchmark. CloverLeaf and su3_bench are much closer to the knee point on both systems, while XSBench has substantially different arithmetic intensity on both systems — 0.26 on Perlmutter, 1.00 on Frontier. It is possible that XSBench heavily utilizes some instruction types that are accounted differently between NVIDIA and AMD's counters used for roofline plotting. All of these kernels are relatively close to the roofline, suggesting these CUDA and HIP versions are relatively close to optimal for the algorithms they implement.

Next, we present performance results for BabelStream `dot`, XSBench, CloverLeaf, su3_bench, and miniBUDE in Figure 2. We omit the BabelStream Copy, Add, Triad, and Mul kernels due to the high degree of consistency across programming models for those kernels. Each heatmap cell represents the total **GPU kernel execution time** averaged over three runs of the application, as described in Section IV. Note that while we do measure data movement time, we do not report it here, as it is consistently negligible ($<2\%$) compared to the time spent in the GPU kernels. Additionally, all values represent the mean of three separate runs of each case, with maximum difference between any run and the three-run mean at 3.3%. The "Native Port" row in each plot represents CUDA performance on Summit and Perlmutter (the NVIDIA systems) and HIP performance on Corona and Frontier (the AMD systems). We organize our initial insights into these results into five observations.

B. Performance of native ports

Observation 1: *On NVIDIA systems, CUDA almost always performs at or near the best observed performance.*

CUDA is the best or within 3% of the best performing model in eight out of ten cases. For these applications, this is a useful validation of the maturity of the CUDA baseline for each application, and confirms our expectation that the low-level vendor model would be the most performant and portable across GPUs from the same vendor. In one notable exception, for RAJA BabelStream `dot` on Perlmutter, we observe that RAJA takes advantage of warp-level primitives in addition to shared memory to perform the reduction, maximizing utilization of hardware-specific features for such operations.

Observation 2: *On AMD systems, HIP does not always guarantee the best performance.*

For most cases on AMD systems, including CloverLeaf, BabelStream `dot`, and su3_bench on Frontier, AMD's HIP programming model achieves the best performance, as expected. However, in multiple instances, HIP does not achieve the best performance, particularly for XSBench. Using Omniperf to profile XSBench, we observe that the HIP port achieves lower Gflop/s and lower L1 cache bandwidth, while Kokkos uses a larger workgroup size and arranges L1 cache read

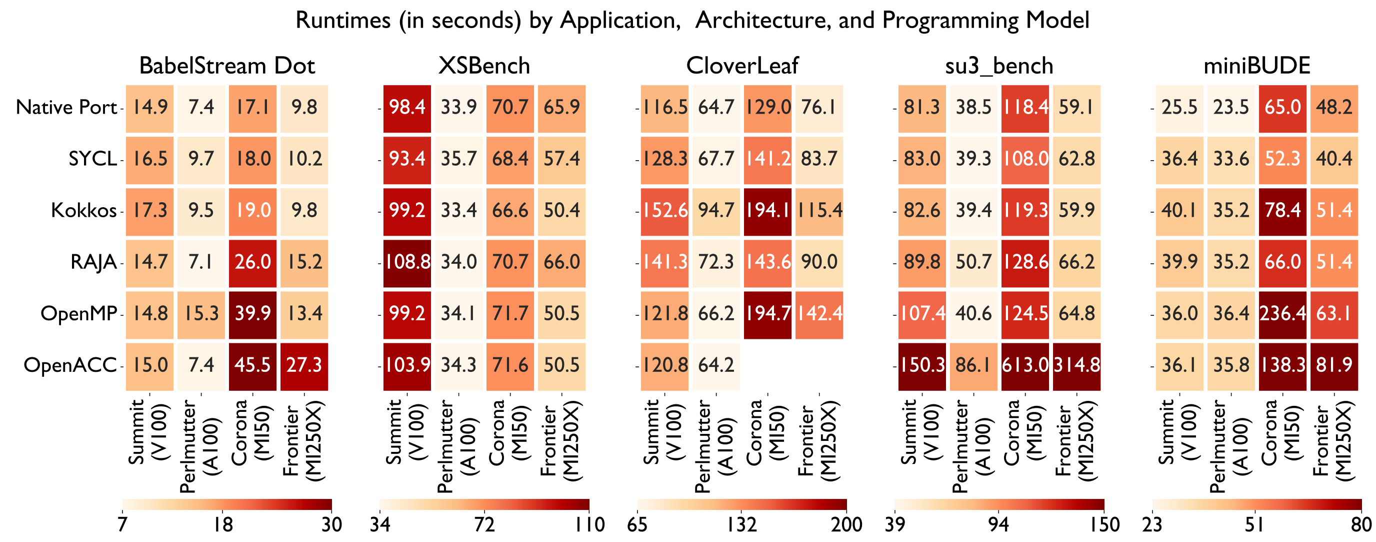

Fig. 2. Average execution time of all proxy applications across all systems and programming models. Lower is better.

requests in a larger number of smaller requests for a similar number of bytes. This suggests Kokkos is selecting a more ideal workgroup size and arranges data access patterns more efficiently for AMD GPUs in XSBench. Meanwhile, OpenMP appears to be able to take advantage of Local Data Share (LDS) implicitly, reducing stalls for accesses to memory, while HIP is not able to.

XSBench is a performance test case used in the development of LLVM OpenMP offloading, which Clacc also uses for OpenACC on Frontier, helping explain why both directive-based models perform so well with XSBench. However, given that Kokkos is a C++ abstraction over HIP code, it is surprising that it can outperform HIP. We note that HIP XSBench performance on Frontier is only slightly better than HIP XSBench on Corona, suggesting that the XSBench HIP implementation is not a fully optimized and mature baseline.

Documentation for XSBench indicates that developers used the Hipify tool to create the XSBench HIP port, and in comparing the HIP and CUDA versions it is clear that they are identical aside from simple substitution of CUDA syntax for HIP syntax. We observe that HIP kernels translated directly from CUDA without additional optimization may not guarantee optimal performance on AMD hardware. Portable programming models are able to achieve superior performance in some cases with a similar level of effort.

C. Portability of SYCL

Observation 3: *SYCL performance is competitive with HIP and relatively stable across system and application pairs.*

In five out of ten cases on AMD systems, SYCL performs better than HIP. As a lower-level language extension, similar to CUDA or HIP, this is not necessarily surprising. In some cases, SYCL is able to improve on CUDA or HIP performance, and even where SYCL is more than 3% slower than a native port, is is never the worst-performing port except in XSBench on Perlmutter, where is is only 5.3% slower. SYCL is the fastest non-native programming model in more cases than any other model, at nine out of twenty total application and system pairs, and six of these are on AMD systems.

D. Portability of C++ abstraction libraries

Observation 4: *Kokkos and RAJA are competitive with CUDA and HIP on many system and application pairs.*

Kokkos and RAJA compare favorably with CUDA and HIP on NVIDIA and AMD systems, with one of the two ports either nearing or exceeding the native port's performance on every combination of system and app, besides those involving CloverLeaf on any system or miniBUDE on an NVIDIA system. However, which model is more performant is very application-dependent. With these very mixed results it is hard to pick a clear portability winner between Kokkos and RAJA, but we can observe that RAJA tends to perform more competitively for NVIDIA systems, and Kokkos tends to have an advantage on AMD systems.

Kokkos performance in CloverLeaf is a notable exception. We observe that the Kokkos port of CloverLeaf spends longer in the `calc_dt` reduction kernel relative to other ports. In Nsight Compute, we find that the Kokkos port achieves fewer eligible warps on average, mostly due to barrier warp stalls, which we do not observe in the other ports. In Sec. VIII we identify a fix for these issues in CloverLeaf Kokkos.

For su3_bench, we observe with RAJA substantially lower arithmetic intensity in L1 and L2 cache compared to HIP, suggesting the RAJA port loads unnecessary data from memory more often. Additionally, in miniBUDE RAJA is not making use of shared memory, which we address in Sec. VIII.

E. Portability of directive-based models

Observation 5: *OpenMP is slower than other implementations in roughly half our cases, and OpenACC struggles with AMD systems.*

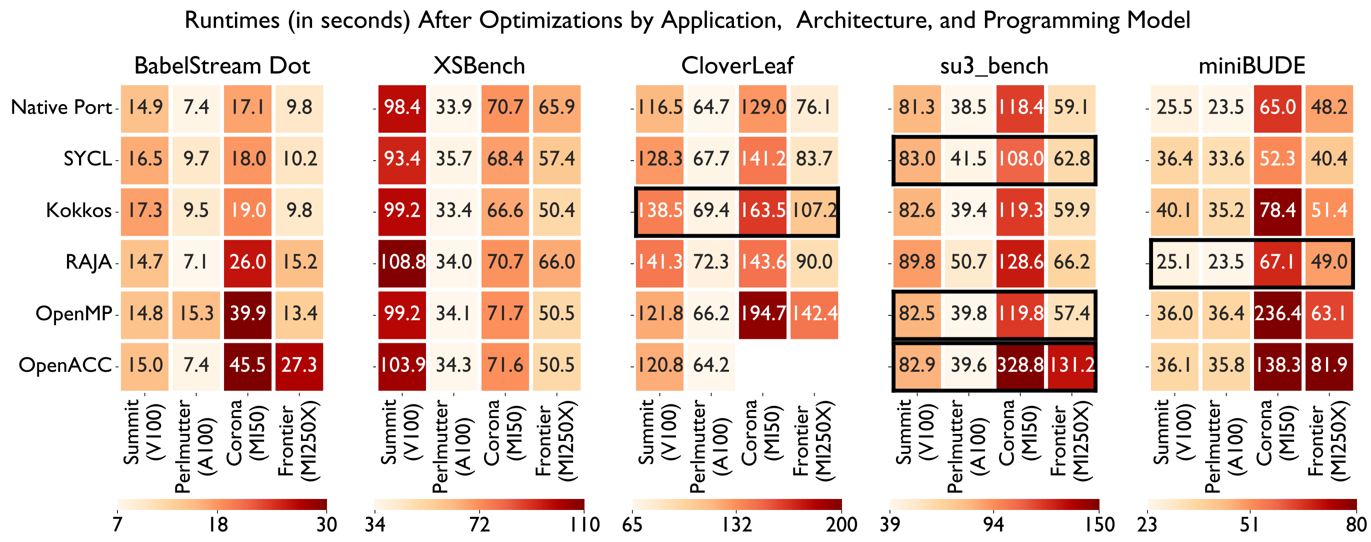

Fig. 3. Average performance of all proxy applications across all systems and programming models, after applying optimizations. Boxes indicates where our optimizations are applied. Lower is better.

OpenMP performance can be slower than the native baseline, achieving significantly better performance than the baseline only for XSBench on Frontier. OpenMP is able to achieve rough parity with the native baseline in exactly half the cases tested. CloverLeaf performance for OpenMP is a notable outlier. We find that compared to HIP the OpenMP port spends significantly more time in the `PdV` kernel. OpenMP achieves less than half the L1 cache bandwidth in this kernel, as well as a roughly 40% lower L2 cache hit rate and 30% higher rate of stalls on L2 cache data, relative to HIP. Meanwhile, in miniBUDE, the OpenMP port appears to allocate an order of magnitude more Local Data Share (LDS) bytes than HIP does, limiting the number of active compute units.

On NVIDIA systems, OpenACC generally achieves more consistent performance with the baseline, but is consistently worse than OpenMP and further worse than HIP on AMD systems, likely because it is employing the same LLVM OpenMP offloading runtime through the Clacc compiler. Per Clacc developers, there is some overhead due to suboptimal translation of OpenACC to OpenMP within Clacc which will be addressed in a future release. The OpenACC port for su3_bench in particular suffers from insufficient exposed parallelism, even on NVIDIA. This is caused by a small fixed number of iterations being distributed to a single block, leading to fewer active threads per block. In Sec. VIII we identify a portability improvement for the su3_bench OpenACC port.

VIII. Optimizations

Here, we present performance optimizations for a few chosen outliers from our broader results. These optimizations include rearranging directives, changing the level of parallelism exposed, and improving use of hardware features.

A. Adjusting Kokkos CloverLeaf reduction parallelization

Kokkos CloverLeaf encounters a relatively high number of barrier stalls. Comparing the implementations of the `calc_dt` between ports, we find that Kokkos is the only one to use a 2D reduction instead of collapsing the kernel into a 1D reduction. We adjust the Kokkos port to use a 1D scheme, bringing Kokkos `calc_dt` performance closer to the native port on all studied systems, and no longer observe barrier warp stalls in the new profile. As shown in Figure 3, Kokkos CloverLeaf performance improves on all systems with this change. The benefit is greater on Perlmutter and Corona, where Kokkos's performance on the other three significant kernels compares more favorably with the native ports.

B. Improving parallelism and alignment in OpenACC and OpenMP su3_bench

The su3_bench OpenACC port originally generates code with only 36 threads per block, despite iterations being assigned to blocks of size 128. This limits the parallelism available on the device. We address this issue by collapsing all four loops, exposing more parallelism.

We find that both OpenMP and OpenACC generated twice as many global loads and stores as CUDA, due to a misaligned complex number struct. OpenMP, OpenACC, and SYCL do not provide a native complex type for GPUs. We declare this struct aligned to sizeof(T) * 2, resulting in a single load and store for each complex number in the array. On AMD this optimization has no effect. As presented in Figure 3, OpenACC benefits strongly from this combination of optimizations, whereas OpenMP achieves modest speedups.

C. Utilizing shared memory in RAJA miniBUDE

In comparing the Kokkos, RAJA, SYCL, and CUDA versions of miniBUDE, we notice that the RAJA version is not making use of shared memory, while the Kokkos, SYCL, and CUDA ports are. RAJA recently added features for dynamically allocating shared memory inside a kernel, a feature needed in miniBUDE since the forcefield data is input-dependent in size, so we modify RAJA miniBUDE to use shared memory for this data.

This optimization improves RAJA performance on NVIDIA systems, with little impact on AMD, leading to an overall increase in portability (see Fig. 3). After the change RAJA performance comes very close to the CUDA performance on Perlmutter, an impressive gain since other models already using shared memory do not get this close on NVIDIA systems. At the time of writing we are unable to add dynamic shared memory allocation inside the kernel for the OpenMP and OpenACC ports due to lack of support.

D. Evaluating performance portability after optimizations

Figure 4 displays the Φ metric for each programming model and proxy application combination after applying the optimizations described above. The "Native Port" column provides context, indicating what the metric would report if a team decided to maintain both a HIP and CUDA version of the application. We are unable to run CloverLeaf with OpenACC on AMD systems, so that cell is zero per the official formulation of the metric[4]. According to Φ, we observe a moderate preference for SYCL, RAJA, and Kokkos as performance portable programming models, and for OpenMP over OpenACC within directive-based models.

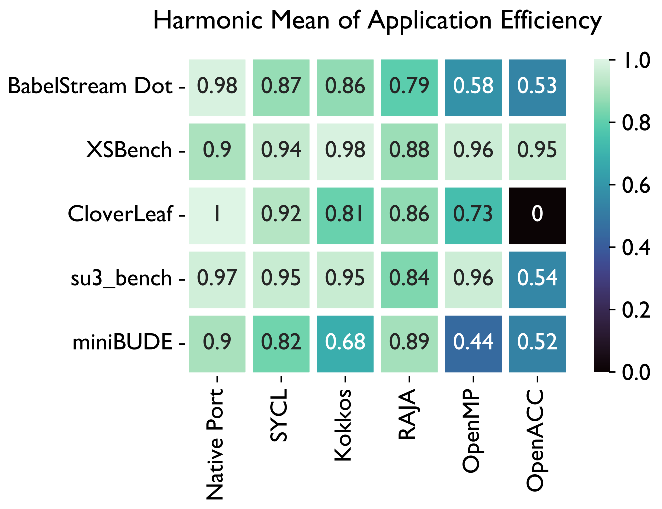

Fig. 4. Φ of GPU kernel performance for each programming model and application combination, after optimizations. Applications are listed in ascending order of arithmetic intensity from top to bottom. Note for OpenACC we are unable to compile CloverLeaf on AMD systems.

IX. Conclusion

In this paper, we empirically evaluated seven GPU programming models and directly compared their capabilities for enabling performance portability. We performed this evaluation on some of the fastest supercomputers in the world using proxy application codes that represent real scientific workloads. We developed a Spack-based methodology to substantially lower the barrier for future experiments comparing portable programming models. We invested significant effort in ensuring each proxy application's implementations in each model can be easily built and run on additional systems, and we plan to open-source these efforts, sharing them with the broader HPC community.

After our optimizations, a few broad outliers remain in the performance portability cases we studied which may be of interest to developers looking to choose a programming model. We highlight the frequent gap between OpenACC and OpenMP performance on AMD systems, the challenges with miniBUDE and CloverLeaf that Kokkos, OpenACC, and OpenMP all face on AMD systems, generally poor reduction performance in OpenACC and OpenMP, and poor reductions on AMD systems with RAJA. For application, compiler, and programming model developers, we present several insights from our experiences as well as suggestions for future investment of effort towards performance portability:

- Successfully building all of these applications across systems is not trivial, especially for RAJA as a multi-library portability suite. Robustness and documentation in the build process may enable app developers to more easily test competing programming models.
- Our ability to identify bottlenecks depended heavily on profiling tools. Improving the quality of these tools for new programming models and hardware architectures will be critical to enabling performance portability. Line-level stall attribution is a crucial capability missing from Omniperf at the time of writing.
- Reduction operations continue to be a major bottleneck, as observed in prior studies, and work on improving compiler handling of reductions would close some of the major remaining performance outliers between portable models and native baselines.
- The ability to separate correctness and performance concerns in these models was critical in identifying the optimizations we describe, as it allowed us to tune ports without invalidating scientific results. Exposing and documenting more semantic-preserving performance "knobs" within each model may provide developers with a wider space to explore to fine-tune performance portability.

[4]For the subset of systems (Summit and Perlmutter) we are able to run OpenACC on, the value is 0.98.

References

[1] TOP500.org, "June 2024 top500," 2024. [Online]. Available: https://www.top500.org/lists/top500/2024/06/

[2] "OpenMP Application Program Interface. Version 4.0. July 2013," 2013.

[3] R. D. Hornung and J. A. Keasler, "The RAJA Portability Layer: Overview and Status," Lawrence Livermore National Laboratory, Tech. Rep. LLNL-TR-661403, Sep. 2014.

[4] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2022.

[5] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral, "The spack package manager: bringing order to hpc software chaos," in *SC15: International Conference for High-Performance Computing, Networking, Storage and Analysis*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2015. [Online]. Available: https://doi.ieeecomputersociety.org/10.1145/2807591.2807623

[6] A. Sabne, P. Sakdhnagool, S. Lee, and J. S. Vetter, "Evaluating performance portability of openacc," in *Languages and Compilers for Parallel Computing: 27th International Workshop, LCPC 2014, Hillsboro, OR, USA, September 15-17, 2014, Revised Selected Papers 27*. Springer, 2015, pp. 51–66.

[7] T. Ben-Nun, J. de Fine Licht, A. N. Ziogas, T. Schneider, and T. Hoefler, "Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–14.

[8] S. J. Pennycook, J. D. Sewall, and V. W. Lee, "A metric for performance portability," in *Proceedings of the 7th International Workshop in Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, 2016. [Online]. Available: https://arxiv.org/abs/1611.07409

[9] ——, "Implications of a metric for performance portability," *Future Generation Computer Systems*, vol. 92, pp. 947–958, 2019.

[10] J. Sewall, S. J. Pennycook, D. Jacobsen, T. Deakin, and S. McIntosh-Smith, "Interpreting and visualizing performance portability metrics," in *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2020, pp. 14–24.

[11] S. J. Pennycook, J. D. Sewall, D. W. Jacobsen, T. Deakin, and S. McIntosh-Smith, "Navigating performance, portability, and productivity," *Computing in Science & Engineering*, vol. 23, no. 5, pp. 28–38, 2021.

[12] S. J. Pennycook and J. D. Sewall, "Revisiting a metric for performance portability," in *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2021, pp. 1–9.

[13] D. F. Daniel and J. Panetta, "On applying performance portability metrics," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2019, pp. 50–59.

[14] A. Marowka, "A comparison of two performance portability metrics," *Concurrency and Computation: Practice and Experience*, p. e7868, 2023.

[15] ——, "Toward a better performance portability metric," in *2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 2021, pp. 181–184.

[16] M. Martineau, S. McIntosh-Smith, and W. Gaudin, "Assessing the performance portability of modern parallel programming models using tealeaf," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 15, p. e4117, 2017.

[17] I. Z. Reguly and G. R. Mudalige, "Productivity, performance, and portability for computational fluid dynamics applications," *Computers & Fluids*, vol. 199, p. 104425, 2020.

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

[19] A. Sedova, J. D. Eblen, R. Budiardja, A. Tharrington, and J. C. Smith, "High-performance molecular dynamics simulation for biological and materials sciences: Challenges of performance portability," in *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 2018, pp. 1–13.

[20] S. Boehm, S. Pophale, V. G. Vergara Larrea, and O. Hernandez, "Evaluating performance portability of accelerator programming models using spec accel 1.2 benchmarks," in *High Performance Computing: ISC High Performance 2018 International Workshops, Frankfurt/Main, Germany, June 28, 2018, Revised Selected Papers 33*. Springer, 2018, pp. 711–723.

[21] A. S. Dufek, R. Gayatri, N. Mehta, D. Doerfler, B. Cook, Y. Ghadar, and C. DeTar, "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, 2021, pp. 57–67.

[22] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, "Evaluating attainable memory bandwidth of parallel programming models via babelstream," *Int. J. Comput. Sci. Eng.*, vol. 17, no. 3, p. 247–262, jan 2018.

[23] V. Artigues, K. Kormann, M. Rampp, and K. Reuter, "Evaluation of performance portability frameworks for the implementation of a particle-in-cell code," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 11, p. e5640, 2020.

[24] E. M. Rangel, S. J. Pennycook, A. Pope, N. Frontiere, Z. Ma, and V. Madananth, "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*, 2023, pp. 1114–1125.

[25] R. Gayatri, C. Yang, T. Kurth, and J. Deslippe, "A case study for performance portability using openmp 4.5," in *Accelerator Programming Using Directives: 5th International Workshop, WACCPD 2018, Dallas, TX, USA, November 11-17, 2018, Proceedings 5*. Springer, 2019, pp. 75–95.

[26] H. Brunst, S. Chandrasekaran, F. M. Ciorba, N. Hagerty, R. Henschel, G. Juckeland, J. Li, V. G. M. Vergara, S. Wienke, and M. Zavala, "First experiences in performance benchmarking with the new spechpc 2021 suites," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2022, pp. 675–684.

[27] G. K. Reddy Kuncham, R. Vaidya, and M. Barve, "Performance study of gpu applications using sycl and cuda on tesla v100 gpu," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, 2021, pp. 1–7.

[28] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. H. Still, "Exploring traditional and emerging parallel programming models using a proxy application," in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '13. IEEE Computer Society, May 2013.

[29] T. Deakin, S. McIntosh-Smith, J. Price, A. Poenaru, P. Atkinson, C. Popa, and J. Salmon, "Performance portability across diverse computer architectures," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2019, pp. 1–13.

[30] T. Deakin, A. Poenaru, T. Lin, and S. McIntosh-Smith, "Tracking performance portability on the yellow brick road to exascale," in *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2020, pp. 1–13.

[31] T. Deakin, S. McIntosh-Smith, S. J. Pennycook, and J. Sewall, "Analyzing reduction abstraction capabilities," in *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 2021, pp. 33–44.

[32] T. Deakin, J. Cownie, W.-C. Lin, and S. McIntosh-Smith, "Heterogeneous programming for the homogeneous majority," in *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2022, pp. 1–13.

[33] W.-C. Lin, S. McIntosh-Smith, and T. Deakin, "Preliminary report: Initial evaluation of stdpar implementations on amd gpus for hpc," *arXiv preprint arXiv:2401.02680*, 2024.

[34] J. Kwack, J. Tramm, C. Bertoni, Y. Ghadar, B. Homerding, E. Rangel, C. Knight, and S. Parker, "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)*, 2021, pp. 45–56.

[35] S. L. Harrell, J. Kitson, R. Bird, S. J. Pennycook, J. Sewall, D. Jacobsen, D. N. Asanza, A. Hsu, H. C. Carrillo, H. Kim *et al.*, "Effective performance portability," in *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 2018, pp. 24–36.

[36] T. Koskela, I. Christidi, M. Giordano, E. Dubrovska, J. Quinn, C. Maynard, D. Case, K. Olgu, and T. Deakin, "Principles for automated and reproducible benchmarking," in *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, 2023, pp. 609–618.

[37] "Ecp proxy applications," https://proxyapps.exascaleproject.org/, accessed: 2023-09-30.

[38] "Nersc proxy suite," https://www.nersc.gov/research-and-development/nersc-proxy-suite/.

[39] M. A. Heroux, R. F. Barrett, J. M. Willenbring, S. D. Hammond, D. Richards, J. Mohd-Yusof, and A. Herdman, "Mantevo suite 1.0." Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2013.

[40] J. H. Davis, C. Daley, S. Pophale, T. Huber, S. Chandrasekaran, and N. J. Wright, "Performance assessment of openmp compilers target-

ing nvidia v100 gpus," in *Accelerator Programming Using Directives*, S. Bhalachandra, S. Wienke, S. Chandrasekaran, and G. Juckeland, Eds. Cham: Springer International Publishing, 2021, pp. 25–44.

[41] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "Xsbench-the development and verification of a performance abstraction for monte carlo reactor analysis," *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, 2014.

[42] P. K. Romano, N. E. Horelik, B. R. Herman, A. G. Nelson, B. Forget, and K. Smith, "Openmc: A state-of-the-art monte carlo code for research and development," *Annals of Nuclear Energy*, vol. 82, pp. 90–97, 2015.

[43] J. Herdman, W. Gaudin, S. McIntosh-Smith, M. Boulton, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis, "Accelerating hydrocodes with openacc, opencl and cuda," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, 2012, pp. 465–471.

[44] D. Doerfler and C. Daley, "su3_bench: Lattice qcd su (3) matrix-matrix multiply microbenchmark (su3_bench) v1. 0," Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), Tech. Rep., 2020.

[45] C. Bernard, M. C. Ogilvie, T. A. DeGrand, C. E. DeTar, S. A. Gottlieb, A. Krasnitz, R. L. Sugar, and D. Toussaint, "Studying quarks and gluons on mimd parallel computers," *The International Journal of Supercomputing Applications*, vol. 5, no. 4, pp. 61–70, 1991.

[46] S. McIntosh-Smith, J. Price, R. B. Sessions, and A. A. Ibarra, "High performance in silico virtual drug screening on many-core processors," *The international journal of high performance computing applications*, vol. 29, no. 2, pp. 119–134, 2015.

[47] D. A. Beckingsale, M. J. McFadden, J. P. S. Dahm, R. Pankajakshan, and R. D. Hornung, "Umpire: Application-focused management and coordination of complex hierarchical memory," *IBM Journal of Research and Development*, vol. 64, no. 3/4, pp. 00:1–00:10, 2020.