

Taking GPU Programming Models to Task for Performance Portability

Joshua H. Davis

Department of Computer Science,
University of Maryland
College Park, Maryland, USA
jhdavis@umd.edu

Pranav Sivaraman

Department of Computer Science,
University of Maryland
College Park, Maryland, USA
psivaraman@umd.edu

Joy Kitson

Department of Computer Science,
University of Maryland
College Park, Maryland, USA
jkitson@umd.edu

Konstantinos Parasyris

Lawrence Livermore National
Laboratory
Livermore, California, USA
parasyris1@llnl.gov

Harshitha Menon

Lawrence Livermore National
Laboratory
Livermore, California, USA
gopalakrishn1@llnl.gov

Isaac Minn

Department of Computer Science,
University of Maryland
College Park, Maryland, USA
iminn@umd.edu

Giorgis Georgakoudis

Lawrence Livermore National
Laboratory
Livermore, California, USA
georgakoudis1@llnl.gov

Abhinav Bhatele

Department of Computer Science,
University of Maryland
College Park, Maryland, USA
bhatele@cs.umd.edu

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 performance portability of these programming models. We provide a Spack scripting-based methodology to ensure reproducibility of experiments conducted in this work. Finally, we analyze the reasons for why some programming models underperform in certain scenarios and in some cases, present performance optimizations to the proxy applications.

CCS Concepts

• **Computing methodologies** → **Parallel programming languages**; • **General and reference** → **Empirical studies**; **Performance**.

Keywords

performance portability, programming models, GPGPUs

ACM Reference Format:

Joshua H. Davis, Pranav Sivaraman, Joy Kitson, Konstantinos Parasyris, Harshitha Menon, Isaac Minn, Giorgis Georgakoudis, Abhinav Bhatele. 2025. Taking GPU Programming Models to Task for Performance Portability. In *2025 International Conference on Supercomputing (ICS '25)*, June 8–11, 2025, Salt Lake City, UT, USA. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/3721145.3730423>

1 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 November 2024 TOP500 list, and ~42% of the systems on the complete list, employ co-processors or accelerators [46]. 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 have emerged, which all aim to allow application developers to write their code once and run it on any system. Programming models such as OpenMP [32], RAJA [23],



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

ICS '25, June 8–11, 2025, Salt Lake City, UT, USA

© 2025 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-1537-2/2025/06

<https://doi.org/10.1145/3721145.3730423>

and Kokkos [48] 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 [18] 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 on production HPC systems.
- We create several additional implementations of existing proxy applications in previously unsupported programming models to ensure full coverage of programming models across applications.
- We describe a methodology employing Spack scripting and environment tools [18] to easily manage the process of building and running all $7 \times 5 = 35$ versions across five supercomputing systems, each with unique software stacks. We open-source these recipes for 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.

2 Background: Portable Programming Models

In this section, we provide relevant background information on the various programming models we evaluate. HIP and CUDA act as our baselines in this study, as they are the native models for AMD and NVIDIA devices, respectively. Below, we describe the key attributes of each category of programming model. All programming models used in this study support both NVIDIA and AMD devices except CUDA.

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 not allowing explicit access to valid device pointers.

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.

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 is carried out by the compiler and device runtime.

3 Related Work

Several studies on programming language extensions, models, and libraries have been designed to assist developers achieve performance portability [5, 23, 32, 42, 48]. 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. Below, we provide an overview of recent work in each category.

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 [33–36, 44]. Daniel et al. propose an alternative metric, P_D , which accounts for problem size [9], and Marowka compares Φ with $\bar{\Phi}$, a similar metric that uses the arithmetic mean instead of the harmonic mean [28, 29].

Studies involving 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 [3, 7, 8, 15, 17, 19, 24, 30, 37–40, 42, 43]. For instance, Dufek et al. compare Kokkos and SYCL for the Milc-Dslash benchmark [17], while Rangel et al. examine the portability of CRK-HACC in SYCL [37]. Other studies investigate

performance portability across applications using specific programming models. Brunst et al. benchmark the 2021 SPEC_{hpc} suite, which contains nine mini-applications in OpenMP and OpenACC, on Intel CPUs and NVIDIA and AMD GPUs [8]. Kuncham et al. evaluate the relative performance of SYCL and CUDA on the NVIDIA V100 using BabelStream, Mixbench, and Tiled Matrix-Multiplication [38].

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 broader evaluation of a diverse set of case studies.

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 [13, 14]. More recent papers by Deakin et al. focus on more specific problems such as reductions and GPU to CPU portability [11, 12]. Lin et al. evaluate implementations of C++17 StdPar against five models on AMD devices [27]. 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 [26]. Harrell et al. study performance portability alongside developer productivity [20]. 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 [25]. Other studies uniformly fail to follow these principles, making reproducing them an arduous task.

4 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.

Table 1: Summary of proxy applications and benchmarks used in this study along with which of their ports the authors changed. 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

4.1 Choice of programming models

Our goal in this work is to empirically compare the performance portability provided by popular programming models. In Section 2, we describe three categories of programming models with a few examples in each category. We identify those representative models by surveying a broad range of proxy applications in order to determine how common existing implementations in each model are. We survey a variety of sources for proxy applications, including the ECP Proxy Apps suite [1], the NERSC Proxy suite [2] and the Mantevo Applications Suite [22]. Armed with that knowledge, we decide to focus on CUDA, HIP, SYCL, Kokkos, RAJA, OpenACC, and OpenMP, as they are the most popular models found in the proxy applications we surveyed. Together, these models cover the three categories of models mentioned earlier.

4.2 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. The scientific domains represented cover three of the four most common disciplines found in INCITE awardees of the last three years — physics, engineering, and biology.¹

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 1 summarizes the key details of each proxy application, and identifies the implementations that are either created or modified by us for this study. Our modifications consist of small changes to the memory management library or style to ensure portability and consistency of gathering execution times across implementations. Below, we describe the five proxy applications used in this study:

BabelStream is a memory bandwidth benchmark with five kernels: copy, add, mul, triad, and dot [15]. The dot kernel includes a reduction operation, known to be a challenging operation for some programming models [10].

XSbench [47] is a proxy for OpenMC, a Monte Carlo transport code [41]. 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.

CloverLeaf is a 2D structured compressible Euler equation solver, with 14 kernels [21]. The advect_mom, advect_cell, PdV, and calc_dt kernels are typically the most time-intensive, and calc_dt contains a reduction.

su3_bench [16] is a proxy application for MILC, a lattice quantum chromodynamics code [6]. 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 [31]. miniBUDE computes the energy field for one configuration of a protein repeatedly.

4.3 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

¹<https://doeleadershipcomputing.org/awardees/>

prominence of GPUs in new systems [45]. We select five different supercomputers for our experiments (architectural details in Table 2): Summit and Frontier at ORNL, Perlmutter at NERSC, Corona at LLNL, and Zataran at the University of Maryland (UMD). 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. We include Corona (AMD MI50) and Zataran (NVIDIA H100) for context with older AMD and newer NVIDIA hardware, respectively. For Frontier’s MI250X GPUs, we run on one Graphics Compute Die (GCD) which is an independent unit of allocation.

Table 2: Architectural details of the GPUs in each system used in this paper. Flop/s and bandwidth values are theoretical peaks provided by device manufacturers.

System	GPU Model	Peak Tflop/s*	DRAM bandwidth	DRAM size
Summit [†] (ORNL)	NVIDIA V100	14.0/7.0	900 GB/s	32 GB
Perlmutter (LBL)	NVIDIA A100	9.5/9.7	1555 GB/s	40 GB
Zataran (UMD)	NVIDIA H100	7.0/34.0	3350 GB/s	80 GB
Corona (LLNL)	AMD MI50	3.3/6.6	1000 GB/s	32 GB
Frontier [‡] (ORNL)	AMD MI250X	23.9/23.9	1600 GB/s	64 GB

* Single-precision/double-precision.

[†] We use the high-memory GPUs on Summit.

[‡] Details for a single GCD of one MI250X.

4.4 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 discussed in Sec. 7, 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 mentioned in Sec. 6, 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. [35]. Φ is defined, for some application a , problem p , set of systems H , and measure of application efficiency e , as:

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

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 indicates the application runs at the best observed performance on all systems.

4.5 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 [18], 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 1, 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 time-consuming effort of building every combination of application and programming model.

We further employ Spack’s Python scripting tools² 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.

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

5 Porting to Unsupported 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 seek to 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.

In the following subsections, we discuss our experiences working with the programming models as applicable. Table 1 summarizes our development efforts. We plan to merge these contributions to their respective upstream repositories.

5.1 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 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_concs` 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.

```
1 View<double*, LayoutLeft, HostSpace,
2     MemoryTraits<Unmanaged>>
3     u_concs(SD.concs, SD.length_concs);
4 SD.d_concs = new View<double*>("d_concs",
5                               SD.length_concs);
6 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.

5.2 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 [4] library, which provides portable memory allocation primitives and memory pools. This separation of concerns in the RAJA ecosystem provides facilitates 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, 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 [18] can mitigate these problems for end users, although this solution pushes the work of ensuring the libraries build and install correctly onto the package maintainers.

5.3 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 [49].

6 Experimental Setup

In this section, we describe the setup for the experiments conducted in this work. We run all the applications on all five systems selected (listed in Table 2).

Table 3 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 models³ 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.

Table 3: Compilers and versions used for building each programming model implementation, by system type.

Prog. Model	NVIDIA	AMD
CUDA	GCC 12.2.0	N/A
HIP	N/A	ROCmCC 5.7.0
SYCL*	DPC++ 2024.01.20	DPC++ 2024.01.20
Kokkos	GCC 12.2.0	ROCmCC 5.7.0
RAJA	GCC 12.2.0	ROCmCC 5.7.0
OpenMP [†]	NVHPC 24.1	LLVM 17.0.6
OpenACC	NVHPC 24.1	Clacc 2023-08-15

* We use AdaptiveCpp 23.10.0 for SYCL CloverLeaf.

[†] We use ROCmCC 5.7.0 for OpenMP su3_bench on AMD GPUs.

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.

³For OpenMP we test Clang, GCC, ROCmCC, NVHPC, CCE; for OpenACC we test Clacc, GCC, NVHPC; for SYCL we test DPC++, AdaptiveCpp

We compile all proxy applications with ‘-O3’ as well as fast math flags and hardware specific instructions for approximate sqrt and division operations. For AMD systems we also add ‘-munsafe-fp-atomics’ as we found this to be broadly beneficial to performance. Finally, for the Clacc compiler, we provide the flag ‘-fopenacc-implicit-worker=vector-outer’ at the recommendation of a developer, as this flag will soon be enabled by default for Clacc.

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 4.4 describes how we modify the proxy applications to ensure consistent timings. We present the final command line arguments in Table 4.

Table 4: Input parameters to the proxy applications.

Application	Input parameters
BabelStream	-n 1500 -w 150 -s $\$(1 \ll 29)$
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 -wgsz 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 collected can be converted to bandwidth (GB/s) by dividing the total data moved by the time.

7 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 model comparisons for individual applications and then across all systems and applications.

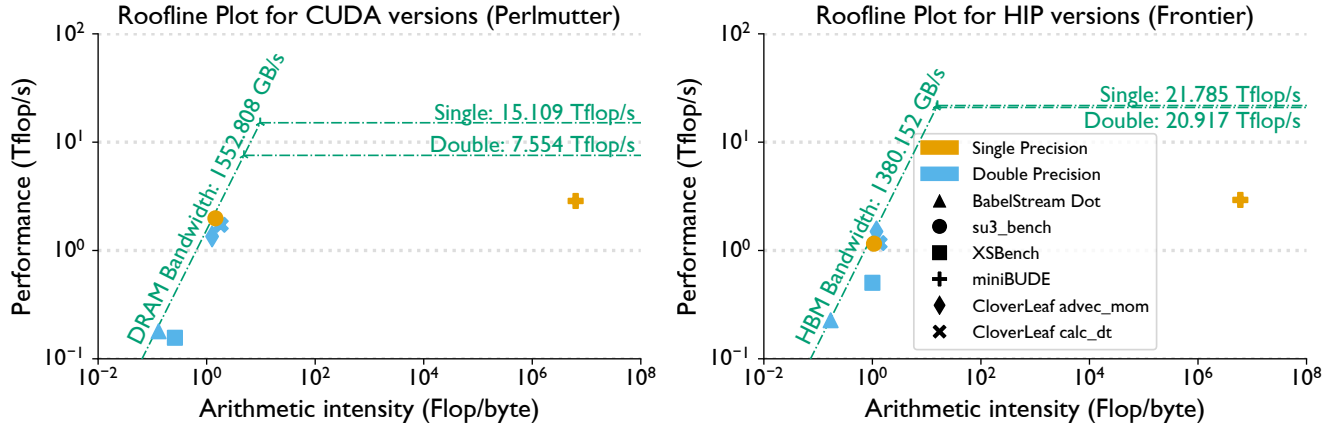


Figure 1: Roofline plots for the most time-consuming kernel in the CUDA (left) and HIP (right) versions of each application, from runs on Perlmutter (NVIDIA A100) and Frontier (AMD MI250X GCD) respectively. Orange points are single precision, and blue points are double precision. We plot each application’s predominant precision.

Table 5: Key details of the major kernels in each proxy application used in the paper. CC is cyclomatic complexity and LN indicates the level of nesting in the kernel’s main loop. Theoretical and achieved occupancy, shared memory per block, static instruction count, total DRAM traffic, and registers per thread are from NCU profiles of the CUDA implementations on Perlmutter.

Application	Kernel	Cyclomatic complexity	Main loop nesting level	Grid, block sizes	Theoretical, achieved occupancy	Shared memory per block	Total DRAM traffic	Static instruction count	Registers per thread
BabelStream	copy	1	1	(524288, 1024)	100%, 81.8%	0 B	8.6 GB	12	16
BabelStream	dot	5	2	(432, 1024)	100%, 99.9%	8.2 KB	8.6 GB	48	16
XSbench	xs_lookup	39	3	(66407, 256)	50%, 46.4%	0 B	156 GB	670	49
CloverLeaf	advec_mom	4	2	(230491, 256)	62.5%, 56.9%	0 B	1.5 GB	405	43
CloverLeaf	calc_dt	8	3	(256, 256)	62.5%, 29.3%	2.1 KB	4.6 GB	643	47
su3_bench	k_mat_nn	3	4	(294912, 128)	100%, 89.7%	0 B	624 MB	59	26
miniBUDE	fasten	29	4	(256, 128)	50%, 14.5%	0.7 KB	1.8 MB	357	62

7.1 Roofline analysis

Figure 1 provides the empirical rooflines for the NVIDIA A100 GPU on Perlmutter and AMD MI250X GCD 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, this is advec_mom. advec_cell and PdV are also highly time-consuming, but have similar positions on the roofline. We also plot calc_dt, as it has the longest per-invocation execution time in CloverLeaf. miniBUDE, XSbench, and su3_bench contain a single computational kernel each. We plot each kernel for the predominant floating-point precision used. We 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. Except for XSbench and miniBUDE, 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.

Table 5 provides additional details about the kernels compared. We provide cyclomatic complexity (CC) to reflect control flow complexity in the kernels and the number of loops nested in the main loop to reflect dimensionality of potential parallelism. The theoretical and achieved occupancy, shared

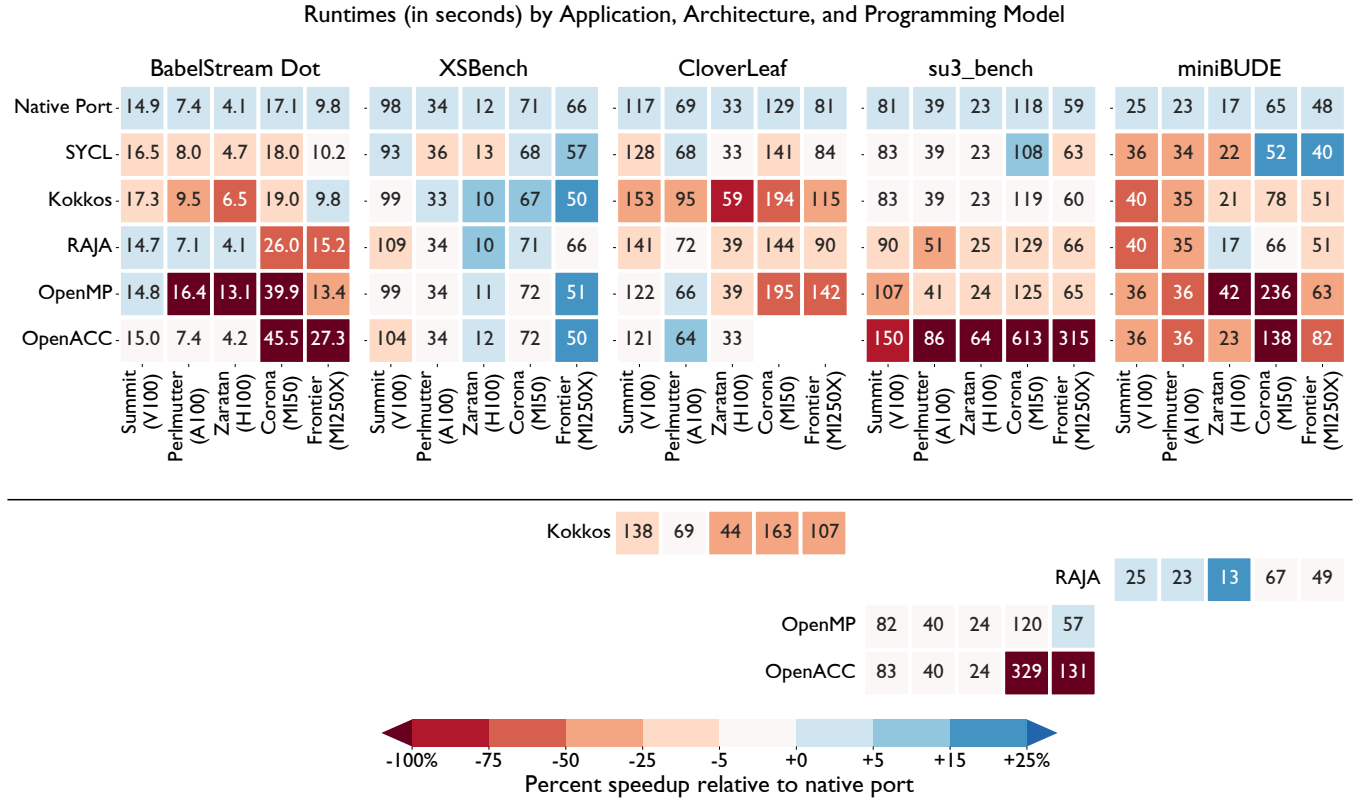


Figure 2: Execution time (indicated as raw numbers in each cell) over three trials of all proxy applications across all systems and programming models (lower is better). The color of each cell indicates performance improvement or degradation relative to the native port for that system and application (blue is faster and red is slower than the native port). Execution times for optimized implementations are provided below the horizontal line.

memory per block, static instruction count, total DRAM traffic, and registers per thread are taken from Nsight Compute profiles of each kernel on Perlmutter.

Observation 1

Most major kernels examined in this work are memory-bound with the exception of miniBUDE, which is strongly compute-bound.

7.2 Analysis of individual applications

Next, we present performance results for BabelStream dot and all four mini-apps in Figure 2. Each heatmap cell represents the **total execution time** across all kernel invocations in each application. Note that each value is the mean of three separate runs, with the maximum difference between any run and this mean being 3.3% (as described in Section 4). Also, while we do measure data movement time, we do not report it here, as it is consistently negligible (<2% of run-time) compared to the time spent in the GPU kernels. 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 discuss observations derived from each mini-app in turn.

7.2.1 BabelStream. BabelStream contains five kernels: copy, add, mul, triad, and dot. All of these kernels are simple one-line memory-bound operations. dot is unique in that it utilizes a reduction operation to compute a dot product, making it a useful simple benchmark of reduction operations across programming models. We observe that for all kernels except dot, performance across programming models is highly consistent with the native port. Figure 3 displays the performance of the add and copy kernels across implementations on Frontier and Perlmutter as a demonstration. OpenACC on AMD systems is the only significant outlier, likely arising from overhead of the Clacc compiler translation approach.

Performance in dot is in contrast much more variable, which is why it is the only one reported in Figure 2. Usually, the portable programming models offer similar or worse

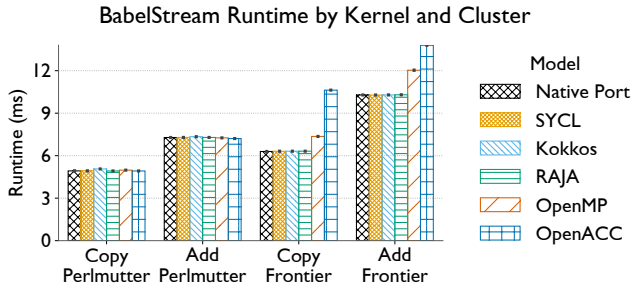


Figure 3: Execution time by model for Copy and Add kernels in BabelStream on Frontier (MI250X GCD) and Perlmutter (A100).

performance than the native port. Kokkos performs moderately worse than the CUDA on Perlmutter and Zaratan, while RAJA performs moderately worse than HIP on Corona and Frontier. Notably, OpenMP performs significantly worse than the native port on Perlmutter, Zaratan, and Corona, and moderately worse than HIP on Frontier. OpenACC performs significantly worse than HIP on AMD systems.

In one notable exception, for RAJA BabelStream dot on Perlmutter, we observe that RAJA takes advantage of warp-level primitives and shared memory to perform the reduction, maximizing utilization of hardware-specific features for such operations. This allows RAJA to moderately out-perform CUDA on all NVIDIA systems.

Observation 2

Simple BabelStream kernels (other than dot) are dominated by memory-bound operations, and all programming models can provide performance portability for them, except OpenACC on AMD.

Observation 3

For dot, a reduction kernel, SYCL comes close to providing performance portability across all systems, and RAJA and OpenACC provide good performance on NVIDIA systems. OpenMP and OpenACC struggle to provide performance portability for dot.

7.2.2 XSBench. XSBench runs a single, long kernel which performs a large quantity of binary searches. In this case, all programming models achieve near or moderately better performance than the native port. In some cases, particularly on Frontier for all models except RAJA, the portable programming models outperform HIP. 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 requests

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

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. The XSBench kernel is also notably more cyclomatically complex and longer than other kernels we examine (Table 5). Together, these observations suggest that HIP kernels with more complex control flow translated directly from CUDA without additional optimization may not guarantee optimal performance on AMD GPUs, which have significantly smaller cache capacity per thread workgroup relative to NVIDIA GPUs. More broadly, the case of XSBench demonstrates how portable programming models are able to achieve matching or even superior performance for more complex kernels with a similar level of development effort as compared to vendor programming models.

Observation 4

All programming models can achieve competitive performance portability for XSBench, a kernel with relatively complex control flow where the implementation style between portable and native ports is highly similar.

7.2.3 CloverLeaf. As a larger proxy application with many kernels including structured stencil operations as well as a reduction operation in `calc_dt`, CloverLeaf encompasses multiple types of GPU kernels. Nevertheless, several programming models achieve broadly consistent performance compared to native ports on both NVIDIA and AMD GPUs. SYCL in particular achieves slightly better performance than CUDA on Zaratan. OpenACC slightly outperforms CUDA on both Perlmutter and Zaratan, mostly due to improved performance in smaller kernels like `calc_dt`, but unfortunately we are unable to compile CloverLeaf with OpenACC on AMD systems at this time due to lack of support for the

host_data clause. RAJA performance is slightly worse than native ports across systems.

Figure 4 breaks down CloverLeaf performance into major constituent kernels to illustrate where performance differences arise for outlier cases. First, CloverLeaf performance for OpenMP on Frontier is a notable outlier. We find that compared to HIP the OpenMP port spends significantly more time in the `advec_*` and `PdV` kernels. 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.

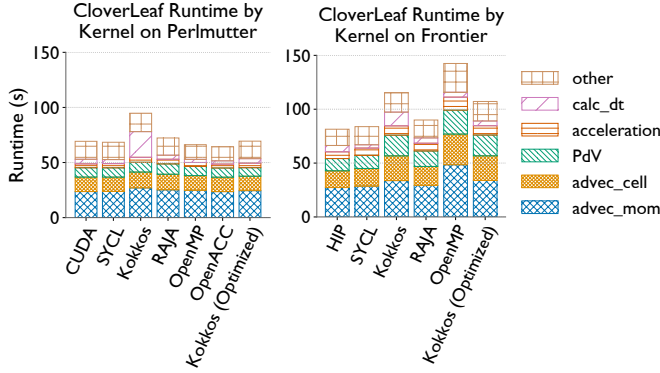


Figure 4: Execution time by model broken down by kernel for CloverLeaf runs on Perlmutter (left, A100) and Frontier (right, MI250X GCD).

Kokkos performance in CloverLeaf is also a notable exception. We observe that the Kokkos port of CloverLeaf spends longer in the `calc_dt` reduction kernel relative to other ports, particularly on NVIDIA systems, like Perlmutter. 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. 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 presented in Figure 2, Kokkos CloverLeaf performance improves on all systems with this change. The benefit is greater on NVIDIA, where Kokkos’s performance on the other three significant kernels nears that of the native ports.

Observation 5

For CloverLeaf’s memory-bound kernels, which employ stencil operations, SYCL and RAJA can consistently provide performance portability and OpenMP, Kokkos and OpenACC struggle with providing portable performance.

7.2.4 *su3_bench*. *su3_bench* is a single-kernel proxy application that computes a matrix multiplication on complex numbers with a relatively deep nested loop hierarchy (Table 5). Generally, performance across programming models is fairly consistent, with the obvious exception of OpenACC before our improvements.

The OpenACC port for *su3_bench* in particular suffers from insufficient exposed parallelism, even on NVIDIA GPUs. The *su3_bench* OpenACC port originally generates code with only 36 threads per block, despite iterations being assigned to blocks of size 128. This leads to fewer active threads per block. We address this issue by collapsing all four loops, exposing more parallelism.

We also 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 GPU-native complex type. 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 2, OpenACC benefits strongly from this combination of optimizations, whereas OpenMP achieves modest speedups.

For RAJA, we observe substantially lower arithmetic intensity in L1 and L2 cache compared to native ports, suggesting the RAJA port loads unnecessary data from memory more often, although the overall impact on performance portability of this limitation is relatively low.

Observation 6

For *su3_bench*, a memory-bound kernel with a deep loop nest, almost all programming models can achieve approximate performance portability as long as sufficient parallelism is exposed and struct declarations are aligned, with the moderate exception of RAJA and stronger exception of OpenACC.

7.2.5 *miniBUDE*. *miniBUDE*, a single-kernel app, is by the most challenging proxy application for the programming models we test. *miniBUDE* is unusual compared to our other proxy applications in that it is highly compute-bound, and leverages thread coarsening to hide memory latencies using instruction-level parallelism within the kernel. It also exerts the highest register pressure in the native CUDA implementation relative to other kernels we examine (Table 5). No programming model is able to achieve consistent performance with the native port, except for RAJA after our improvements. We note that SYCL does achieve superior performance compared to HIP on AMD devices.

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 Figure 2). After the change RAJA performance comes very close to the CUDA performance on Perlmutter and 27% faster than CUDA on Zaratan, 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.

The OpenMP port of miniBUDE appears to allocate an order of magnitude more Local Data Share (LDS) bytes than HIP does, limiting the number of active compute units and thus reducing the degree to which memory access latency can be hidden. Comparing the OpenMP port to CUDA, we observe a significant increase in registers used per thread (86 vs. 62) and dramatically more static instructions (1463 vs. 357). Other models encounter similar, but less pronounced, issues: for example, Kokkos uses 69 registers per thread and generates a 797-instruction kernel. For both Kokkos and OpenMP these overheads correspond to a 500% increase in cycles spent in L2 cache activity as well as 50% increases in DRAM and L1 cache cycles.

Observation 7

For miniBUDE, a highly compute-bound kernel relying on shared memory, RAJA (after adding shared memory support) can provide very competitive performance portability where most other models, especially OpenMP and OpenACC, struggle due to register pressure and increased generated instruction counts. SYCL is also highly competitive, but only on AMD systems.

7.3 Evaluating performance portability across applications after optimizations

From analysis of the lower portion of Figure 2, containing results after our optimizations, we can make several general observations about the performance portability enabled by each programming model.

7.3.1 Language extensions. CUDA is the best or within 3% of the best performing model in eleven out of fifteen 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 that vendor.

Meanwhile, 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, as discussed under that application.

Finally, SYCL performs better than HIP in five out of ten cases on AMD systems. 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, it is never the worst-performing port except in XSBench on Perlmutter and Zaratan, where it is only 5.3% and 10% slower, respectively. SYCL is the fastest non-native programming model in more cases than any other model, at eleven out of twenty-five total application and system pairs, and six of these are on AMD systems.

Observation 8

On NVIDIA systems, CUDA almost always performs at or near the best observed performance, whereas on AMD systems there are some cases, in particular for XSBench, where other models are significantly faster than HIP.

Observation 9

SYCL performance is often competitive with CUDA and HIP, and relatively stable across system and application pairs, with the exception of miniBUDE on NVIDIA GPUs.

7.3.2 C++ abstraction libraries. 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 Summit and Perlmutter. While which model is more performant is very application-dependent, we can observe that RAJA tends to perform more competitively for NVIDIA systems, and Kokkos tends to have an advantage on AMD systems.

Observation 10

Kokkos and RAJA are competitive with CUDA and HIP on many system and application pairs, with a slight preference for RAJA on NVIDIA GPUs and Kokkos on AMD GPUs.

7.3.3 Directive-based models. 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 twelve out of twenty-five cases.

On NVIDIA systems, OpenACC generally achieves more consistent performance with the baseline, but is often 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.

Observation 11

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

7.4 Performance portability metric evaluation

Figure 5 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 metric⁴. According to Φ , we observe a moderate preference for SYCL, RAJA, and Kokkos among the portable programming models, in roughly that order, and for OpenMP over OpenACC within directive-based models.⁵

Observation 12

Summarizing our Φ results, we find that SYCL most consistently achieves performance portability for our tests, followed closely by RAJA and Kokkos.

8 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. Overall, compared to prior

⁴When only considering NVIDIA systems, the value is 0.98.

⁵We also compared Φ to $\bar{\Phi}$ [29], which uses the arithmetic mean instead of the harmonic mean (results not shown). This penalizes low outliers much less, but the overall ordering of results remains the same with both metrics.

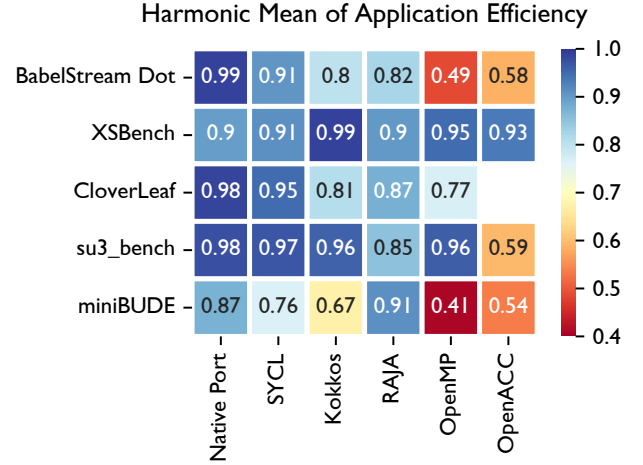


Figure 5: Φ of GPU kernel performance for each programming model and application combination, after optimizations. Applications are listed in ascending order of arithmetic intensity. Note for OpenACC we are unable to compile CloverLeaf on AMD systems.

comparative studies [13, 14] we find improved performance portability across models, particularly for SYCL.

After our optimizations, a few broad outliers remain in the performance results 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, generally poor reduction performance in OpenACC and OpenMP, poor reductions on AMD systems with RAJA, and consistent difficulty with the compute-bound and register-intensive miniBUDE for all programming models and systems. 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 a multi-library portability suite like RAJA. Additional robustness in – and documentation for – this 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 Omniprperf 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 gaps between portable models and native baselines.

- The example of miniBUDE demonstrates that performance in a compute-bound kernel with high register pressure can be highly sensitive to the choice of a portable programming model. Identifying techniques to reduce spilling of registers to memory and instruction count bloat when adopting a programming abstraction may help users maximize arithmetic bandwidth.
- 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 range of options to improve the performance portability of their applications.

Acknowledgments

This material is based upon work supported in part by the National Science Foundation (NSF) under Grant No. 2047120, the NSF Graduate Research Fellowship Program under Grant No. DGE 2236417, and the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research, DOE Computational Science Graduate Fellowship under Award No. DE-SC0021. This work was performed in part under the auspices of the U.S. DOE by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-855581).

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. DOE under Contract No. DE-AC05-00OR22725, and of the National Energy Research Scientific Computing Center (NERSC), a U.S. DOE Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231 using NERSC awards DDR-ERCAP0025593 and DDR-ERCAP0029890.

References

- [1] [n. d.]. ECP Proxy Applications. <https://proxyapps.exascaleproject.org/>. Accessed: 2023-09-30.
- [2] [n. d.]. NERSC Proxy Suite. <https://www.nersc.gov/research-and-development/nersc-proxy-suite/>.
- [3] Victor Artigues, Katharina Kormann, Markus Rampp, and Klaus Reuter. 2020. Evaluation of performance portability frameworks for the implementation of a particle-in-cell code. *Concurrency and Computation: Practice and Experience* 32, 11 (2020), e5640.
- [4] D. A. Beckingsale, M. J. McFadden, J. P. S. Dahm, R. Pankajakshan, and R. D. Hornung. 2020. Umpire: Application-focused management and coordination of complex hierarchical memory. *IBM Journal of Research and Development* 64, 3/4 (2020), 00:1–00:10. <https://doi.org/10.1147/JRD.2019.2954403>
- [5] Tal Ben-Nun, Johannes de Fine Licht, Alexandros N Ziogas, Timo Schneider, and Torsten Hoefler. 2019. 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*. 1–14.
- [6] Claude Bernard, Michael C Ogilvie, Thomas A DeGrand, Carleton E DeTar, Steven A Gottlieb, A Krasnitz, Robert L Sugar, and Doug Tous-saint. 1991. Studying quarks and gluons on MIMD parallel computers. *The International Journal of Supercomputing Applications* 5, 4 (1991), 61–70.
- [7] Swen Boehm, Swaroop Pophale, Verónica G Vergara Larrea, and Oscar Hernandez. 2018. 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, 711–723.
- [8] Holger Brunst, Sunita Chandrasekaran, Florina M Ciorba, Nick Hagerty, Robert Henschel, Guido Juckeland, Junjie Li, Veronica G Mellesse Vergara, Sandra Wienke, and Miguel Zavala. 2022. 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, 675–684.
- [9] Daniela F. Daniel and Jairo Panetta. 2019. On Applying Performance Portability Metrics. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 50–59. <https://doi.org/10.1109/P3HPC49587.2019.00010>
- [10] Joshua Hoke Davis, Christopher Daley, Swaroop Pophale, Thomas Huber, Sunita Chandrasekaran, and Nicholas J. Wright. 2021. Performance Assessment of OpenMP Compilers Targeting NVIDIA V100 GPUs. In *Accelerator Programming Using Directives*, Sridutt Bhalachandra, Sandra Wienke, Sunita Chandrasekaran, and Guido Juckeland (Eds.). Springer International Publishing, Cham, 25–44.
- [11] Tom Deakin, James Cownie, Wei-Chen Lin, and Simon McIntosh-Smith. 2022. Heterogeneous Programming for the Homogeneous Majority. In *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 1–13. <https://doi.org/10.1109/P3HPC56579.2022.00006>
- [12] Tom Deakin, Simon McIntosh-Smith, S John Pennycook, and Jason Sewall. 2021. Analyzing Reduction Abstraction Capabilities. In *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 33–44.
- [13] Tom Deakin, Simon McIntosh-Smith, James Price, Andrei Poenaru, Patrick Atkinson, Codrin Popa, and Justin Salmon. 2019. Performance Portability across Diverse Computer Architectures. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 1–13. <https://doi.org/10.1109/P3HPC49587.2019.00006>
- [14] Tom Deakin, Andrei Poenaru, Tom Lin, and Simon McIntosh-Smith. 2020. Tracking Performance Portability on the Yellow Brick Road to Exascale. In *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 1–13. <https://doi.org/10.1109/P3HPC51967.2020.00006>
- [15] Tom Deakin, James Price, Matt Martineau, and Simon McIntosh-Smith. 2018. Evaluating Attainable Memory Bandwidth of Parallel Programming Models via BabelStream. *Int. J. Comput. Sci. Eng.* 17, 3 (jan 2018), 247–262.
- [16] Douglas Doerfler and Christopher Daley. 2020. *su3_bench: Lattice QCD SU(3) matrix-matrix multiply microbenchmark (su3_bench) v1.0*. Technical Report. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States).
- [17] Amanda S Dufek, Rahul Kumar Gayatri, Neil Mehta, Douglas Doerfler, Brandon Cook, Yasaman Ghadar, and Carleton DeTar. 2021. Case study

- of using Kokkos and SYCL as performance-portable frameworks for Milc-Dslash benchmark on NVIDIA, AMD and Intel GPUs. In *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 57–67.
- [18] T. Gambin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral. 2015. The Spack package manager: bringing order to HPC software chaos. In *SC15: International Conference for High-Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, Los Alamitos, CA, USA. <https://doi.org/10.1145/2807591.2807623>
- [19] Rahul Kumar Gayatri, Charlene Yang, Thorsten Kurth, and Jack Deslippe. 2019. 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, 75–95.
- [20] Stephen Lien Harrell, Joy Kitson, Robert Bird, Simon John Pennycook, Jason Sewall, Douglas Jacobsen, David Neill Asanza, Abigail Hsu, Hector Carrillo Carrillo, Hesso Kim, et al. 2018. Effective performance portability. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 24–36.
- [21] JA Herdman, WP Gaudin, Simon McIntosh-Smith, Michael Boulton, David A Beckingsale, Andrew C Mallinson, and Stephen A Jarvis. 2012. Accelerating hydrocodes with OpenACC, OpenCL and CUDA. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, 465–471.
- [22] Michael Allen Heroux, Richard Frederick Barrett, James Michael Wilenbring, Simon David Hammond, David Richards, Jamal Mohd-Yusof, and Andrew Herdman. 2013. *Mantevo Suite 1.0*. Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).
- [23] Rich D. Hornung and Jeff A. Keasler. 2014. *The RAJA Portability Layer: Overview and Status*. Technical Report LLNL-TR-661403. Lawrence Livermore National Laboratory.
- [24] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L. Chamberlain, Jonathan Cohen, Zachary DeVito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, David Richards, Martin Schulz, and Charles H. Still. 2013. Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium (IPDPS '13)*. IEEE Computer Society.
- [25] Tuomas Koskela, Ilektra Christidi, Mosè Giordano, Emily Dubrovskaya, Jamie Quinn, Christopher Maynard, Dave Case, Kaan Olgu, and Tom Deakin. 2023. 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*. 609–618.
- [26] JaeHyuk Kwack, John Tramm, Colleen Bertoni, Yasaman Ghadar, Brian Homerding, Esteban Rangel, Christopher Knight, and Scott Parker. 2021. Evaluation of Performance Portability of Applications and Mini-Apps across AMD, Intel and NVIDIA GPUs. In *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 45–56. <https://doi.org/10.1109/P3HPC54578.2021.00008>
- [27] Wei-Chen Lin, Simon McIntosh-Smith, and Tom Deakin. 2024. Preliminary report: Initial evaluation of StdPar implementations on AMD GPUs for HPC. *arXiv preprint arXiv:2401.02680* (2024).
- [28] Ami Marowka. 2021. Toward a better performance portability metric. In *2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 181–184.
- [29] Ami Marowka. 2023. A comparison of two performance portability metrics. *Concurrency and Computation: Practice and Experience* (2023), e7868.
- [30] Matthew Martineau, Simon McIntosh-Smith, and Wayne Gaudin. 2017. Assessing the performance portability of modern parallel programming models using TeaLeaf. *Concurrency and Computation: Practice and Experience* 29, 15 (2017), e4117.
- [31] Simon McIntosh-Smith, James Price, Richard B Sessions, and Amaury A Ibarra. 2015. High performance in silico virtual drug screening on many-core processors. *The international journal of high performance computing applications* 29, 2 (2015), 119–134.
- [32] OpenMP4 2013. OpenMP Application Program Interface. Version 4.0. July 2013.
- [33] S. John Pennycook and Jason D. Sewall. 2021. Revisiting a Metric for Performance Portability. In *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 1–9. <https://doi.org/10.1109/P3HPC54578.2021.00004>
- [34] S John Pennycook, Jason D Sewall, Douglas W Jacobsen, Tom Deakin, and Simon McIntosh-Smith. 2021. Navigating performance, portability, and productivity. *Computing in Science & Engineering* 23, 5 (2021), 28–38.
- [35] Simon J Pennycook, Jason D Sewall, and Victor W Lee. 2016. A metric for performance portability. In *Proceedings of the 7th International Workshop in Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. <https://arxiv.org/abs/1611.07409>
- [36] Simon J Pennycook, Jason D Sewall, and Victor W Lee. 2019. Implications of a metric for performance portability. *Future Generation Computer Systems* 92 (2019), 947–958.
- [37] 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.
- [38] Goutham Kalikrishna Reddy Kuncham, Rahul Vaidya, and Mahesh Barve. 2021. Performance Study of GPU applications using SYCL and CUDA on Tesla V100 GPU. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. <https://doi.org/10.1109/HPEC49654.2021.9622813>
- [39] 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.
- [40] István Z Reguly and Gihan R Mudalige. 2020. Productivity, performance, and portability for computational fluid dynamics applications. *Computers & Fluids* 199 (2020), 104425.
- [41] Paul K Romano, Nicholas E Horelik, Bryan R Herman, Adam G Nelson, Benoit Forget, and Kord Smith. 2015. OpenMC: A state-of-the-art Monte Carlo code for research and development. *Annals of Nuclear Energy* 82 (2015), 90–97.
- [42] Amit Sabne, Putt Sakdhnagool, Seyong Lee, and Jeffrey S Vetter. 2015. 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, 51–66.
- [43] Ada Sedova, John D Eblen, Reuben Budiardja, Arnold Tharrington, and Jeremy C Smith. 2018. 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, 1–13.
- [44] Jason Sewall, S. John Pennycook, Douglas Jacobsen, Tom Deakin, and Simon McIntosh-Smith. 2020. Interpreting and Visualizing Performance Portability Metrics. In *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 14–24. <https://doi.org/10.1109/P3HPC51967.2020.00007>
- [45] TOP500.org. 2024. June 2024 TOP500. <https://www.top500.org/lists/top500/2024/06/>

- [46] TOP500.org. 2024. November 2024 TOP500. <https://www.top500.org/lists/top500/2024/11/>
- [47] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014. 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).
- [48] Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan Madsen, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. 2022. Kokkos 3: Programming Model Extensions for the Exascale Era. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2022), 805–817. <https://doi.org/10.1109/TPDS.2021.3097283>
- [49] Michael Wolfe, Seyong Lee, Jungwon Kim, Xiaonan Tian, Rengan Xu, Barbara Chapman, and Sunita Chandrasekaran. 2018. The OpenACC data model: Preliminary study on its major challenges and implementations. *Parallel Comput.* 78 (2018), 15–27.