Performance Portability across Diverse Computer Architectures

Tom Deakin*, Simon McIntosh-Smith*, James Price*, Andrei Poenaru*,
Patrick Atkinson*, Codrin Popa* and Justin Salmon*
*High Performance Computing Research Group
University of Bristol, Bristol, UK
Email: {tom.deakin, S.McIntosh-Smith}@bristol.ac.uk

Abstract—**Previous studies into performance portability have typically analysed a single application (and its various implementations) in isolation. In this study we explore the wider landscape of performance portability by considering a number of applications from across the space of dwarfs, written in multiple parallel programming models, and across a diverse set of architectures. We apply rigorous performance portability metrics, as defined by Pennycook et al [1]. We believe this is the broadest and most rigorous performance portability study to date, representing a far reaching exploration of the state of performance portability that is achievable today. We will present a summary of the performance portability of each application and programming model across our diverge range of twelve computer architectures, including six different server CPUs from five different vendors, five different GPUs from two different vendors, and one vector architecture. We will conclude with an analysis of the performance portability of key programming models in general, across different application spaces as well across differing architectures, allowing us to comment on more general performance portability principles.**

Index Terms—**performance portability, productivity, mini-app, programming models**

I. Introduction

In recent years, there has been significant disruption from an ever increasing diversity in HPC processors. Traditional CPU architectures have gained large numbers of cores and wide SIMD units. As of today, 32-core sockets are available in Marvell ThunderX2 and AMD Naples processors, with up to 64-cores per socket due to arrive in the coming year. There is renewed competition in the CPU space, with multiple CPU vendors designing processors for high-performance computing, employing a variety of different instruction sets, ensuring a healthy ecosystem and yielding improvements in the rate of innovation and in performance per dollar. Additionally, accelerators such as GPUs are being adopted in increasing numbers to achieve the highest levels of performance, with most of the Top 10 supercomputers in the world now using accelerators [2]. This increased diversity in computer architecture is perhaps most keenly observed in the accelerator field, with GPUs from NVIDIA, AMD and most recently Intel, NEC's SX-Aurora TSUBASA vector engine, FPGAs and specialised processors such as the emerging architectures for AI (e.g. Google's TPU, GraphCore's IPU and Intel's Nervana).

This diverse set of architectures are rapidly being adopted in the race for Exascale systems. The 'Fugaku' (Post-K) system at RIKEN will use Fujitsu's A64FX Arm processors; the 'Aurora' system at Argonne National Laboratory will use heterogeneous processors from Intel including Xeon CPUs, Xe GPUs, AI Nervana processors and Stratix FPGAs; the 'Frontier' system at Oak Ridge National Laboratory will use AMD CPUs and GPUs; while the 'El Capitan' system at Lawrence Livermore National Laboratories will be another heterogeneous system. It is clear that there will be a significant challenge for application developers to achieve performance portability over such a more diverse range of vendors and architectures than has been seen in many years.

This study seeks to assess the performance portability landscape using representative applications, programming models and a wide range of processors available today which capture the future direction of technology. We select from a range of mini-apps, implemented in a range of parallel programming models selected either because they are already widely used, or because they represent likely future directions. These include directive based models (OpenMP and OpenACC), high-level C++ abstraction layers (Kokkos and SYCL) and close to the metal offload models (OpenCL and CUDA). No programming model is inherently performance portable. However, all the models we have selected support writing performance portable programs, to one degree or another. Some of the models in our list are less portable than others, due to their vendor-specific nature, but we include these in order to capture the best achievable performance of an application on each platform, regardless of the choice of implementation, in order to provide a baseline. Indeed, a key part of performance portability is that close to the best achievable performance is attainable, in a portable manner (this is what is known as "application efficiency" in the performance portability metric defined by Pennycook et al in [1]). Both the directive-based and high-level abstractions we use offer mechanisms for parallelisation within each computational node (on the 'host' CPUs or via offload to an accelerator); we do not consider multi-node programming models where MPI is the predominant model chosen.

For this study, we use the following definition of performance portability: A code is performance portable if it can achieve a similar performance efficiency on a range of different target architectures, where performance efficiency is as defined in [1]. That is, performance efficiency is either:

1) Architectural efficiency: Achieved performance as a fraction of peak theoretical hardware performance, or
2) Application efficiency: Achieved performance as a fraction of best observed performance.

Additionally, we require not only that we achieve a similar performance efficiency across our target architectures, but that we also achieve a *good* performance efficiency. We define this to mean within 20% of the best achievable performance, which often corresponds to hand-optimised OpenMP or CUDA/OpenCL code. The size of the set of target architectures in any performance portability study is of course dependent on perspectives and motivations; our set is chosen to represent the current diversity in CPUs and GPUs, and to represent potential future architectural developments.

In this study we make the following contributions:

- We present a broad reaching set of performance portability results across five mini-apps running on up to twelve hardware platforms. Each mini-app is implemented in up to six different programming models, representing 360 different combinations of code, language and platform.
- We show a rigorous analysis of the portability of each application using the Performance Portability Metric as defined by Pennycook et al. [1].
- We discuss in depth the performance portability of the OpenMP and Kokkos programming models with respect to the performance achievable over a range of architectures *and* applications. In this manner, we can measure how performance portable an application may be given the choice of programming model for the implementation.

A. Related Work

The authors of this paper have a long history in examining the performance portability of a range of codes. In 2013 they performed one of the first performance portability studies, investigating the compute-bound molecular docking code BUDE, using an optimised OpenCL implementation initially targetted at NVIDIA GPUs [3]. Extensive work enabled this code to achieve similar percentages of peak floating-point performance on other GPUs (from NVIDIA *and* AMD) and on CPUs including Intel Xeon and Xeon Phi (Knights Corner). To the best of our knowledge, this was the first result to demonstrate successful performance portability across different CPUs and GPUs. A similar approach was used for the memory bandwidth bound codes D3Q9-BGK Lattice-Boltzmann, ROTORSIM and CloverLeaf [4], where it was shown that similar percentages of peak memory bandwidth can be achieved across diverse hardware platforms.

The portability of programming models across a range of architectures was explored in the BabelStream benchmark [5], and we include the latest results in this paper. The performance of a number of parallel programming models on GPUs was explored using the TeaLeaf mini-app [6], showing that each model can achieve similar performance.

Pennycook et al invented a metric to assess performance portability, and it is this metric which we will use in our study [1]. This original study applied the metric to a number of different applications to demonstrate its use and effectiveness at characterising performance portability. Only a single programming model was used for each application of the metric; in this study we will apply the metric across all our data, which includes codes, architectures and programming models.

Sedova et al. expand this metric to consider architecture specialisation of certain key routines of applications [7]. In this study, we aim to use the same code base on all platforms, but we do discuss two mini-apps which require some algorithmic differences between different classes of architecture. The metric of Pennycook also requires an efficiency metric, and one such alternative is proposed by Yang et al based on the Roofline model in order to apply the metric to an optimisation strategy [8]. We favour the original application and architectural efficiencies proposed by Pennycook et al for this study as they provide the simplest rating across our wide stable of codes.

We do not consider domain specific languages (DSLs) in our study, although portability is a key motivation in their design. DSLs do allow isolation of the application from the changes in hardware and the performance of such approaches is commonplace (e.g. [9], [10]). In this study, we use parallel programming models in which a wide range of applications may be written, and so exclude DSLs.

Many previous performance portability studies usually focus on just a few applications, platforms and parallel programming models (as clearly demonstrated by the topics of presentations at a 2019 meeting on performance portability[1]). It is a key contribution of the work presented in this paper that we push this horizon significantly further, considering many more applications, more platforms and more parallel programming models simultaneously, in a consistent and rigorous manner. As such, our analysis comments on the current state of performance portability in the general sense.

II. Systematic Evaluation of Performance Portability

Motivated by the previous work which demonstrated that performance portability can be obtained, our goal here is to initiate a wide-ranging evaluation of performance portability across many codes, many programming languages and many architectures. This will provide the community with useful data, along with an example study showing how performance portability can be evaluated in other contexts, for example starting to include performance portability as a formal metric during continuous integration, with a regression in performance portability being treated the same way as a regression in functionality or performance would be. Our hope is that this work will also contribute to the fundamental understanding of performance portability.

This systematic study of performance portability faced a number of challenges. In particular, each architecture is often hosted in its own system, distinct from any other system,

[1] https://doep3meeting2019.lbl.gov/agenda

and as such comes with differences in software environments, such as the compilers and math libraries that were available. Additionally, in order to explore the portability of programming models, each application must be written in each model of interest, significantly increasing the effort required ensure consistency across those different implementations. To aid others in the rigorous study of performance portability, and to support reproducibility, we have created an open repository of the scripts we used to generate our results, in order to describe how we obtained each result, taking into account the choice of compiler and compiler flags on each platform for each code[2].

The definition of performance portability can be a contentious issue. We recognise that there is no one correct definition. Instead, in this paper we consider a code as performance portable if it can achieve a similar fraction of performance efficiency on a desired set of target architectures, where performance efficiency is defined in [1]. Two aspects of this definition require further clarification: firstly we expect that a good fraction of peak performance is obtained (after all this is high performance computing); and secondly, the range of target architectures should be wide enough to capture current hardware requirements, while also anticipating future architectural developments. The performance portability metric introduced to the community by Pennycook et al is well aligned to this definition [1]. The metric, Ƥ, is quoted below from their paper for use in our evaluation, where $e(a,p)$ captures the performance efficiency of application a on platform p.

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

The metric is defined as the harmonic mean of the performance efficiency $e(a,p)$ of application a on a given platform p on a set of platforms H. Two methods of measuring the performance efficiency are defined:

- The performance efficiency is measured as a percentage of theoretical peak hardware achieved by the application where it is possible to obtain appropriate performance models. This method is known as *architectural efficiency*.
- The performance efficiency is measured as a percentage of the best performance observed of that application on a given platform, eliminating other factors such as programming model, etc. This method is known as *application efficiency*.

We use both definitions in our study as appropriate.

For this study, we have selected five mini-apps to represent critical workloads on many of the largest supercomputers in the world: BabelStream [5], CloverLeaf [4], TeaLeaf [11], Neutral [12] and MiniFMM [13]. A short description of each mini-app will be given in Section III. These mini-apps are written in a number of programming models: OpenMP and OpenACC, SYCL and Kokkos, OpenCL and CUDA. These programming models represent different levels of abstraction over the underlying hardware: low level, or close to the metal expressions of parallelism (CUDA and OpenCL); loop-level parallelism expressed through pragmas (OpenMP and OpenACC); and higher-level abstractions expressed through C++ lambdas (Kokkos and SYCL).

We run each application implemented in each programming model on a wide range of target hardware platforms, where possible. We have selected twelve hardware platforms, sampling the latest CPUs from Intel, AMD, IBM, Marvell and Ampere, four generations of NVIDIA GPUs, a relatively new GPU from AMD, and the NEC SX-Aurora TSUBASA vector processor. This selection includes the best in class from each processor vendor available on the market at the time of writing. Each processor demonstrates trade-offs in the theoretical peak performance of floating-point operations and main memory throughput. In order to asses the efficiency of the application (so as to calculate Ƥ), we require the theoretical peak performance of each device. Details about the processors are presented in Table I, along with the machine balance (GFLOP/s / GWord/s) which represents the average number of floating-point operations for every 64-bit word of memory accessed [14]. This balance shows the number of operations that can be performed for every word loaded from memory. Please refer to the Reproducibility Appendix for further details on the hosting systems for each processor.

III. Results

In this section we present the raw results for each application running in each programming model on each hardware platform. The results are presented as heatmaps, where the colouring signifies yellow is "better" and green is "worse" in a consistent manner. Where throughput or Ƥ is used as the metric, higher numbers are coloured yellow, and for runtime the lower numbers are coloured yellow, in both cases indicating "better" results. Due to the current maturity of these models it is not always possible to collect each combination of model and platform, either due to lack of a robust implementation of a compiler or runtime, or else simply that a platform is not supported by the model or the vendor's implementation of it (such as CPUs in CUDA). We will detail missing results for each mini-app in the following sections.

For each application, we calculate the performance portability using the metric Ƥ. A simple application of the portability metric to our results would yield $\text{Ƥ} = 0$ in the majority of cases due to at least one combination of platform and model unable to produce a result. This clearly highlights the challenges in achieving a minimum level of portability of applications across a diverse range of hardware.

Therefore, for each mini-app we show a graph of performance portability calculated for a sequence of subsets of architectures. We use application efficiency to calculate Ƥ. The graphs start with computing Ƥ for all the architectures. In order to prevent any preconceived bias to requiring support for any one platform in our set, we use the following heuristic

[2] https://github.com/UoB-HPC/benchmarks/tree/doe-p3-2019

TABLE I
PROCESSOR CONFIGURATIONS AND SYSTEM BALANCE

Architecture	Sockets	Cores	Clock Speed GHz	FP64 TFLOP/s	Memory Bandwidth GB/s	STREAM Balance
Skylake	2	28	2.1	3.76	256	117.5
KNL	1	64	1.3	2.66	490	43.4
Power 9	2	20	3.2	1.02	340	24
Naples	2	32	2.5	1.28	288	35.6
ThunderX2	2	32	2.5	1.28	288	35.6
Ampere	1	32	3.3	0.21	159	10.6
NEC Aurora	1	8	1.4	2.15	1,200	14.3
K20	1	13	0.71	1.18	208	45.4
P100	1	56	1.13	4.04	732	44.2
V100	1	80	1.37	7.01	900	62.3
Turing	1	68	1.25	0.37	616	4.8
Radeon VII	1	60	1.4	3.5	1,000	28

to remove one platform from each set in turn; the x-axis on our figure shows the removed architecture from the previous subset. Firstly we remove the architecture which has the most missing results. In the case of a tie, we remove the platform which results in the largest change in Ꝑ for all architectures, calculated as a L_2-norm.

The resultant graphs show Ꝑ for each programming model for a given mini-app, and show how the performance portability changes over a range of target architectures. It is useful to present some intuition as to interpretation at this stage:

- The further to the left a model has a non-zero Ꝑ result, the more *portable* it is.
- The area under the curve is intuitively related to how performance portable it is over the widest range of architectures.

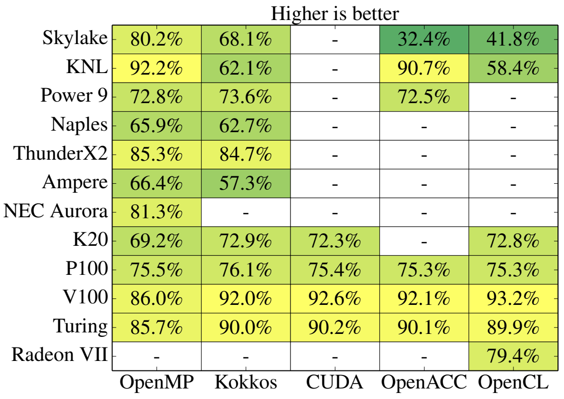
We will later apply the performance portability metric in novel ways across the programming models in Section IV.

A. *BabelStream*

The BabelStream benchmark is a re-implementation of the classic McCalpin STREAM benchmark in many parallel programming models [5]. It also includes a dot-product kernel, and employs best-practice implementations for each model. BabelStream also captures commonplace programming patterns which are not captured by STREAM, such as only making the problem size available at runtime and allocating memory on the heap, as opposed to the stack, where it is subject to unrealistic compiler optimisation.

The sustained memory bandwidth as output by the application is shown as a percentage of the theoretical peak memory bandwidth in Figure 1. This is equivalent to the architectural efficiency (as defined in Section II). For the peak numbers, please refer to Table I.

BabelStream provides the greatest coverage of all the applications in our study; it should be noted that it is by far the simplest of the applications but if results are viable here then other main memory bound codes should (in theory) behave similarly. However, there are still results which we were not able to obtain and no one programming model is able to run across all processors in our study. Only OpenMP is supported on the NEC Aurora, and clearly CUDA is unsupported on architectures not from NVIDIA. We were unable to obtain OpenACC results on Arm platforms and on the K20m due to compatibility with the `glibc` version on that system. We were not able to collect OpenCL results on Power 9 and Arm due to lack of a compiler. The AMD GPU OpenMP compiler and Kokkos ROCm backend is still in development, and the other missing models are unsupported on the Radeon VII.



	OpenMP	Kokkos	CUDA	OpenACC	OpenCL
Skylake	80.2%	68.1%	-	32.4%	41.8%
KNL	92.2%	62.1%	-	90.7%	58.4%
Power 9	72.8%	73.6%	-	72.5%	-
Naples	65.9%	62.7%	-	-	-
ThunderX2	85.3%	84.7%	-	-	-
Ampere	66.4%	57.3%	-	-	-
NEC Aurora	81.3%	-	-	-	-
K20	69.2%	72.9%	72.3%	-	72.8%
P100	75.5%	76.1%	75.4%	75.3%	75.3%
V100	86.0%	92.0%	92.6%	92.1%	93.2%
Turing	85.7%	90.0%	90.2%	90.1%	89.9%
Radeon VII	-	-	-	-	79.4%

Fig. 1. Percentage of peak memory bandwidth achieved for BabelStream

Figure 2 shows Ꝑ calculated using application efficiency for decreasing subsets for each programming model. OpenMP is the first model to appear, showing it has the greatest portability running on all but one of the platforms with Ꝑ $= 97.5\%$, and maintains a similar result across all subsets. This says that for the Triad kernel written in OpenMP 4.5 (using the `target` directives as appropriate), we can expect to achieve a high percentage of theoretical peak across the hardware in our set; this is a fantastic example of where performance portability is indeed achieved.

If we also remove the NEC Aurora, Kokkos runs on all other platforms with Ꝑ $= 89.3\%$, and similar to OpenMP remains at this level. This again shows that in writing BabelStream in Kokkos, we were able to achieve portable performance which we would expect to be around 90% of the best seen on our platforms; this highlights the small overhead of less than 10% on average for the higher level abstraction Kokkos provides over OpenMP and CUDA.

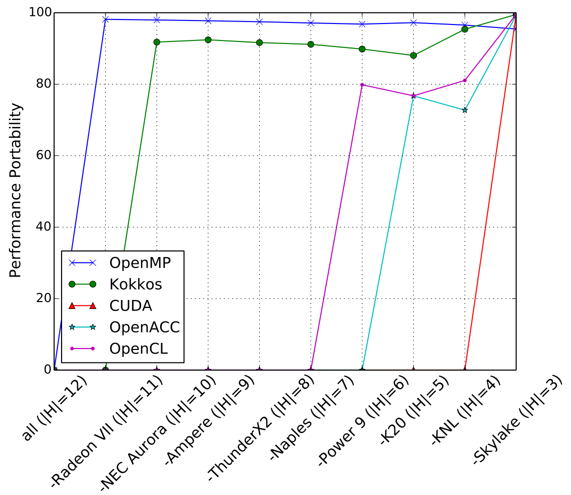


Fig. 2. Performance Portability for BabelStream

Excluding all the non-Intel CPUs, we see OpenCL achieves Ꝑ = 76.7% on the remaining platforms (Skylake, KNL, K20, P100, V100 and Turing). Once CPU platforms are removed entirely (leaving only the NVIDIA GPUs), we observe Ꝑ = 99.2% indicating that the performance of the Intel OpenCL runtime on CPUs is holding back the achievable portable performance (we previously observed this for architectures with NUMA effects [5]); inspection of Figure 1 also indicates this; however, the impact is clearer to identify using Ꝑ.

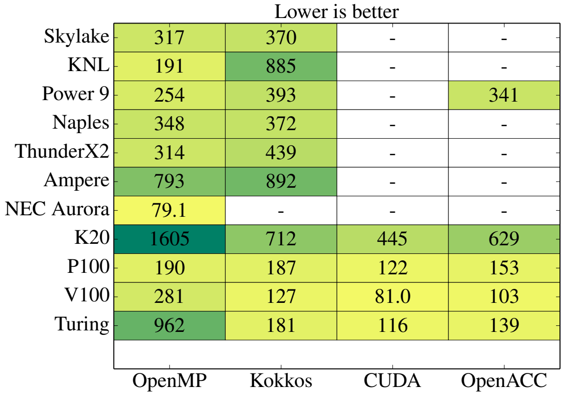
We observe that OpenACC achieves a similar Ꝑ to OpenCL, and indeed suffers similar NUMA effects on CPUs [5]. Note that the heuristic we use to remove platforms causes the K20m to remain in the subset of architectures beyond the Power 9 (a supported OpenACC platform); we had two missing models on Power 9 but only one for K20. The subset of platforms at this stage is however rather limited compared to our initial set, and so the cross-vendor support of such a model is somewhat lacking; again we observe this in Figure 2 with the first non-zero result being far to the right.

In summary, the BabelStream benchmark is simple enough for us to showcase the best achievable coverage of the programming models across our diverse set of architectures. Despite this, many results were unobtainable due to lack of support for all programming models on all platforms. The performance portability metric shows that across the largest subsets of architectures, both OpenMP and Kokkos achieve close to the best possible performance across the most architectures, demonstrating that high performance of 90 – 98% of the best observed is portable.

B. TeaLeaf

The TeaLeaf mini-app solves the heat diffusion equation on a 2D structured grid using a Conjugate Gradient linear solver [11]. The code is main memory bandwidth bound on each node, and at scale becomes communication bound by the reductions in dot-product operations. As such in this study it should demonstrate similar performance characteristics to BabelStream.

The runtimes of TeaLeaf across the combinations of model and architecture are shown in Figure 3. One can see that the NEC Aurora which offers the highest memory bandwidth of all platforms in our study yields the fastest runtime. Aside from the issues already identified for BabelStream, we could not collect OpenACC results on x86 CPUs due to runtime segmentation faults in access to members of Fortran derived types with PGI 18.10.


Lower is better

	OpenMP	Kokkos	CUDA	OpenACC
Skylake	317	370	-	-
KNL	191	885	-	-
Power 9	254	393	-	341
Naples	348	372	-	-
ThunderX2	314	439	-	-
Ampere	793	892	-	-
NEC Aurora	79.1	-	-	-
K20	1605	712	445	629
P100	190	187	122	153
V100	281	127	81.0	103
Turing	962	181	116	139

Fig. 3. Runtime (s) for the TeaLeaf mini-app

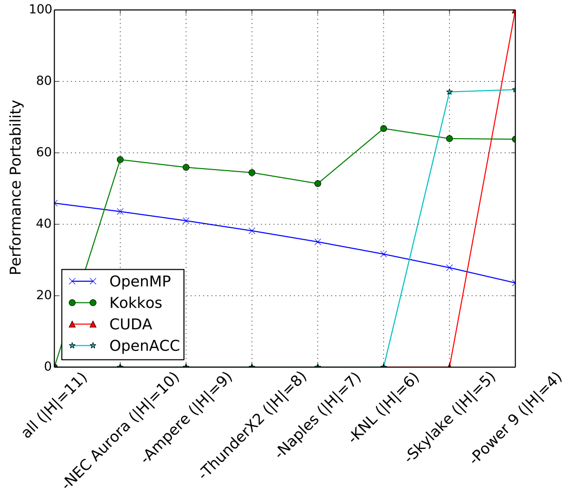


Fig. 4. Performance Portability for TeaLeaf

We will use application efficiency to calculate Ꝑ, with the results shown in Figure 4. In comparison with the equivalent graph for BabelStream in Figure 2, it is clear that Ꝑ is lower here for all programming models. OpenMP achieves Ꝑ = 45.9% for all platforms here (excluding Radeon VII). The final data point shown is for just NVIDIA GPU platforms; observe that the Ꝑ for OpenMP reduces as the subsets become dominated by these GPUs. We see in Figure 3 that the OpenMP on GPUs results have poor application efficiency with the exception of P100. Indeed, the P100 is the only platform where the Cray compiler was available, with the other results collected using the Clang compiler for which OpenMP `target` is still in development and performance is expected to improve over time.

For the Kokkos implementation of TeaLeaf, we see that $\text{Ꝑ} = 57.4\%$ for the largest supported subset and maintains a similar level for subsequent subsets. We see an improvement when the KNL is removed, and note that this result has an efficiency of 22% which we believe is due to vectorisation issues of the loops by the Intel compiler on this processor. On the GPU dominant platform subsets, we see Kokkos achieves a greater performance portability than OpenMP ($\text{Ꝑ} = 63.8\%$ and 23.6% respectively for the final subset). Kokkos uses the low level CUDA (which performs well here) rather than OpenMP to target NVIDIA GPUs and so can insulate the developer from the observed performance discrepancies.

For OpenACC on the NVIDIA and Power platforms alone, $\text{Ꝑ} = 77.1\%$ and $|H| = 5$; however, this is a narrow set of platforms and so we cannot draw many conclusions.

In summary, the performance portability of a larger application like TeaLeaf begins to show more discrepancies in the attainable performance. OpenMP and Kokkos again do well across the largest platform subsets, however OpenMP is limited today by the maturity of compilers for GPUs. The metric and our presentation in the figures highlights where performance issues in the implementation of the model occur due to immature compilers or performance bugs, and the breadth of the results here helps discover these cases.

C. CloverLeaf

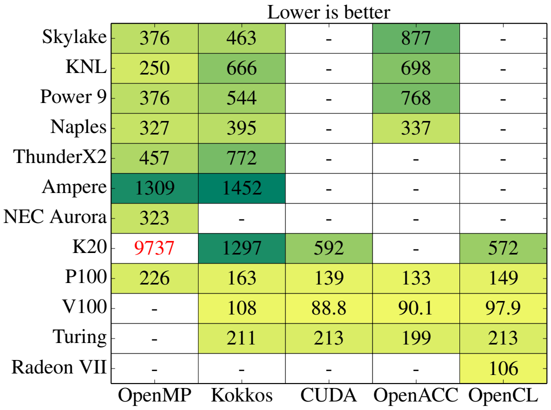
The 2D structured grid Lagrandian-Eulerian hydrodynamics mini-app, CloverLeaf, has been well studied and forms part of the Mantevo suite of benchmarks [4], [15]. The kernels are primarily stencil updates or element-wise updates, and so are typically main memory bandwidth bound. Figure 5 shows the runtime of CloverLeaf. We have refrained from colouring the K20 OpenMP result as it is very poor; as we found with TeaLeaf, the Clang compiler is currently lacking in comparison with the Cray compiler for OpenMP `target`. Note too that the K20 GPU is rather old, and we expect that little tuning of the current Clang compiler has been performed for this architecture. Additionally, we found that the OpenMP `target` implementation of CloverLeaf did not work correctly on the V100 and Turing GPUs, however we believe these are issues with the mini-app rather than the model and plan to investigate this further.

The NEC Aurora result presented here removes the file I/O, which had an enormous overhead under the reverse offload environment. This I/O has negligible time on other the other platforms. Note too that with some other minor changes to the source to ensure the immature NEC compiler vectorises correctly, the runtime can improve from 323s to 188s. We do not use this improved result in our analysis, however note that the compilers for other platforms apply these automatically.

We plot Ꝑ for decreasing subsets in Figure 6. We find that Kokkos provides the best platform coverage here, running on all platforms except the Radeon VII and the NEC Aurora and demonstrates $\text{Ꝑ} = 66.2\%$ on the latest set of platforms. Note that this is a similar percentage to that achieved for Kokkos TeaLeaf; we will discuss such matters further in Section IV. For CloverLeaf, the Ꝑ of the Kokkos implementation remains relatively stable across the subsets too.

OpenMP is noticeably absent, and this is due to our issues with collecting results on NVIDIA GPUs as detailed above. Our heuristic for removing least supported platforms removes the CPU platforms first which have two missing results before removing the GPUs where we could not obtain OpenMP results. It is difficult to determine an appropriate ordering for subsets of our wide range of platforms in a fair, systematic and robust way. On the nine platforms where we have results for OpenMP, $\text{Ꝑ} = 35.0\%$; this is somewhat low due to the poor K20 result; excluding this substantially raises Ꝑ to 91.9% over eight platforms.

Figure 5 shows that on NVIDIA GPUs, CUDA and OpenCL achieve similar performance and this is also seen in Figure 6


Lower is better

	OpenMP	Kokkos	CUDA	OpenACC	OpenCL
Skylake	376	463	-	877	-
KNL	250	666	-	698	-
Power 9	376	544	-	768	-
Naples	327	395	-	337	-
ThunderX2	457	772	-	-	-
Ampere	1309	1452	-	-	-
NEC Aurora	323	-	-	-	-
K20	9737	1297	592	-	572
P100	226	163	139	133	149
V100	-	108	88.8	90.1	97.9
Turing	-	211	213	199	213
Radeon VII	-	-	-	-	106

Fig. 5. Runtime (s) for the CloverLeaf mini-app

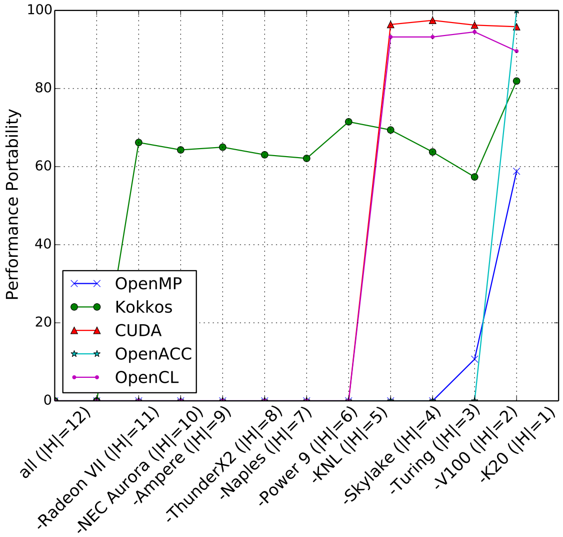


Fig. 6. Performance Portability for CloverLeaf

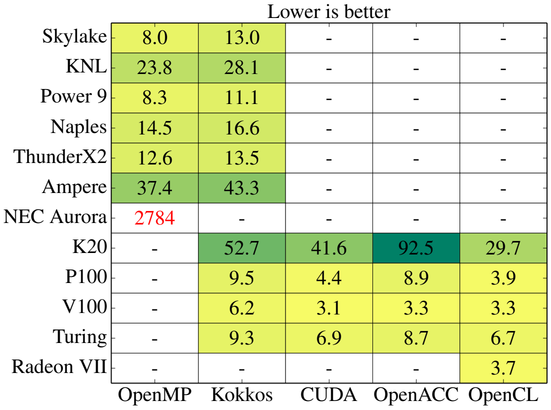
where the Ᵽ for both models is similar on this subset.

As with TeaLeaf, we find that the platform support for OpenACC limits the portability and so it is difficult to accurately comment on performance portability.

In summary, our implementations of CloverLeaf suffer from a lack of portability due in part to issues with platform support or bugs in the implementation; the maintenance of even mini-apps in multiple programming models is somewhat burdensome and CloverLeaf is the oldest application in this study. The Kokkos implementation provides the best coverage, running on ten of our twelve platforms, and does demonstrate reasonable performance portability too. On the eight platforms supported by OpenMP with mature compiler support, OpenMP achieves the best Ᵽ, however this is not displayed in Figure 6 due to lack of robust GPU support in comparison with the other programming models.

D. Neutral

Neutral is a Monte Carlo neutral particle transport mini-app, developed to explore the properties and performance of on-node parallelism in this algorithm [12]. One finding of this work was that the most optimal data layout and parallel scheme differed on CPU and GPU architectures, with the former requiring an array-of-structures (AoS) and "Over Particles" scheme, and the latter requiring a structure-of-arrays (SoA) and "Over Events" scheme. The mini-app has versions of each kernel implemented with both approaches, and so we follow this convention in our results. Note that the OpenACC and OpenCL versions are missing the AoS implementation at this time, and so we omit the CPU results. The aforementioned study also demonstrated that Neutral was often bound by the latency of memory access, and the architectures which are latency tolerant gave the best performance. Unfortunately, the OpenMP `target` implementation causes internal compiler errors with the Cray 8.6 compiler and the application crashes immediately with the latest version of the Clang compiler; therefore we cannot present results here. The runtimes of Neutral are shown in Figure 7.



Lower is better

	OpenMP	Kokkos	CUDA	OpenACC	OpenCL
Skylake	8.0	13.0	-	-	-
KNL	23.8	28.1	-	-	-
Power 9	8.3	11.1	-	-	-
Naples	14.5	16.6	-	-	-
ThunderX2	12.6	13.5	-	-	-
Ampere	37.4	43.3	-	-	-
NEC Aurora	2784	-	-	-	-
K20	-	52.7	41.6	92.5	29.7
P100	-	9.5	4.4	8.9	3.9
V100	-	6.2	3.1	3.3	3.3
Turing	-	9.3	6.9	8.7	6.7
Radeon VII	-	-	-	-	3.7

Fig. 7. Runtime (s) for Neutral mini-app

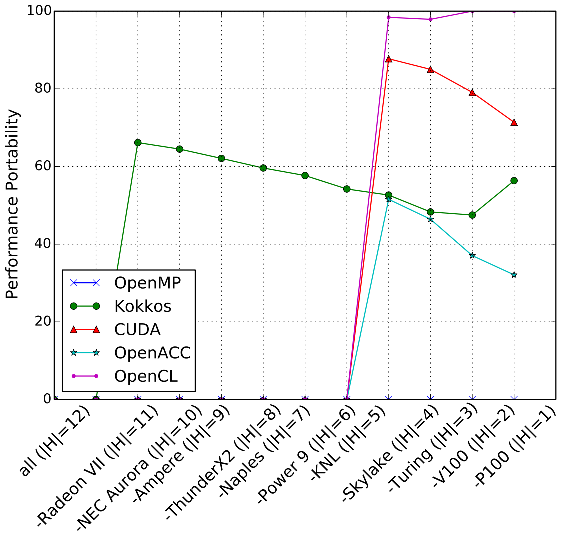


Fig. 8. Performance Portability for Neutral

Our Ᵽ calculations are shown in Figure 8. As with CloverLeaf, Kokkos achieved the best coverage running on ten of our twelve platforms with a similar metric where $Ᵽ = 66.0\%$. On removing the V100 platform, the performance portability improves, and as seen in Figure 7 this result was indeed rather poor — a similar observation was made for KNL running the Kokkos implementation of TeaLeaf.

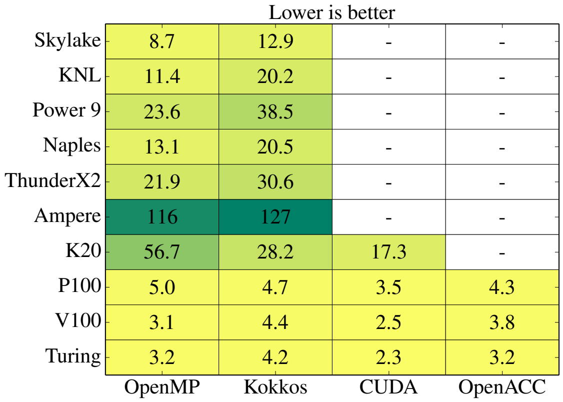
The algorithm for selecting platform subsets always contains at least one GPU and as we do not yet have OpenMP results for these devices, OpenMP remains with $Ᵽ = 0$ for all subsets. On CPU architectures, OpenMP gets the best performance and the other base case for the Ᵽ metric is observed: $Ᵽ = 100\%$.

On NVIDIA GPUs alone, we see that OpenCL gives the best performance; indeed the results are better than CUDA due to low efficiency on the K20. The Ᵽ reflects this, showing the CUDA line below that of OpenCL. OpenCL achieves the best performance on the NVIDIA GPUs (excepting CUDA on the V100) and this renders $Ᵽ = 98.4\%$. We can see that on this subset, the OpenACC implementation attains $Ᵽ = 51.5\%$, which is similar to that demonstrated by Kokkos.

In summary, Neutral shows the complexity involved in measuring performance portability where different algorithms are required to achieve good performance on a given class of platforms. Each programming model is able to implement the required changes, however we do not have complete coverage of implementations for our chosen set of models and platforms. As such, our choice of subsets becomes much more limited in general. Kokkos here has the best coverage of platforms, and shows similar levels of performance portability to other other mini-apps in this study.

E. MiniFMM

MiniFMM is a fast multipole mini-app for N-body applications, using a linear time tree-based approximation for computing the attractive forces between the bodies rather than the naive quadratic time complexity algorithm [13]. The mini-app is unique in our study in that it uses complex numbers and is naturally expressible as a recursive tree of tasks. It also is much more compute bound, uses single-precision FP32 complex numbers, and has a higher FLOP/byte ratio than the other mini-apps in this study. Also included is an iterative version which negates the need for a sophisticated device-side tasking model, by pre-computing the dependency tree and traversing it in parallel. Note that tasks are still used on the host side however; the NEC OpenMP runtime does not support tasks and so we were able to run the mini-app there at all. The iterative version was used for implementing the method on the GPU as no robust and efficient task-parallel programming model exists for entirely on-GPU execution. The runtimes are presented in Figure 9. Aside from the issues already identified, we found that the OpenACC implementation crashed quickly on CPU platforms and so we do not have these results.



Lower is better

	OpenMP	Kokkos	CUDA	OpenACC
Skylake	8.7	12.9	-	-
KNL	11.4	20.2	-	-
Power 9	23.6	38.5	-	-
Naples	13.1	20.5	-	-
ThunderX2	21.9	30.6	-	-
Ampere	116	127	-	-
K20	56.7	28.2	17.3	-
P100	5.0	4.7	3.5	4.3
V100	3.1	4.4	2.5	3.8
Turing	3.2	4.2	2.3	3.2

Fig. 9. Runtime (s) for MiniFMM mini-app

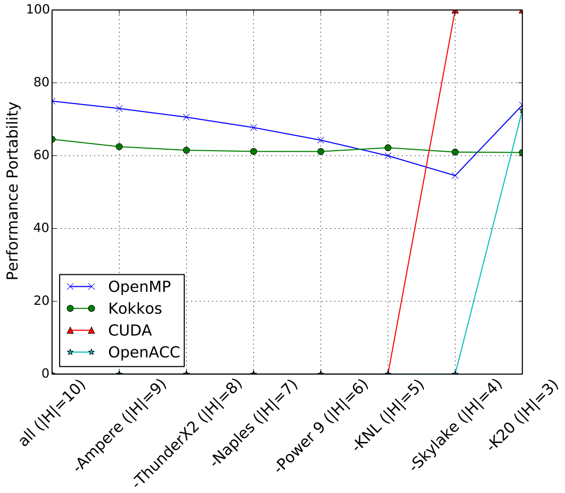


Fig. 10. Performance Portability for MiniFMM

We consider the Φ for MiniFMM in Figure 10. Both OpenMP and Kokkos show consistent levels of performance portability: the line is relatively flat across the subsets and consistent around 70%. The Φ for OpenMP is in general around 10% higher than that for Kokkos. Again we notice that one architecture, in this case K20, has an inefficient implementation and reduces the metric until it is removed from the platform list. Indeed, we only achieve 20% application efficiency here due to the use of the development LLVM compiler which is relatively untested on K20.

We must note that we used the Clang builtin for leveraging shared memory on the GPUs, which is available via OpenMP 5.0 however is not implemented in the compiler yet. The Kokkos implementation used this GPU version with explicit shared memory allocation and was run without modification on the CPU, a platform with no programmable shared memory. As the Φ shows, good efficiency is still achieved with this Kokkos implementation assisting in applying memory abstractions appropriately to various target architectures.

For this mini-app, we find that both Kokkos and OpenMP show a high level of performance portability across the range of architectures, with Φ scores higher than many of the other mini-apps in the study. Kokkos shows a more consistent level of performance across all the architectures than OpenMP, however the performance is lower than that which OpenMP achieves in most cases.

IV. Performance Portability of OpenMP and Kokkos

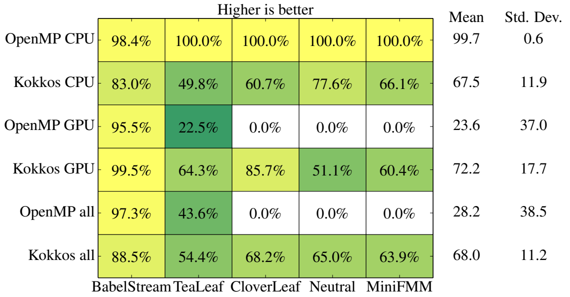
The analysis in Section III considered the performance portability of each mini-app in turn. In this section, we collate the performance portability metric Φ results for a consistent subset of platforms across each mini-app. We focus on the OpenMP and Kokkos programming models as these provided the most portability across the most platforms for each mini-app; recall that where platforms are unsupported $\Phi = 0$ which is of little interest. They also demonstrate two different paradigms in programming model abstractions: the directive based OpenMP and the C++ abstraction framework (parallel execution of lambda functions) Kokkos. We take the following three subsets of platforms:

- CPUs: Skylake, KNL, Power 9, Naples, ThunderX2.
- GPUs: P100, V100, Turing.
- All: the two categories above.

We have excluded the less mature and least well covered platforms (NEC SX-Aurora, Ampere and AMD Radeon VII). We have also removed the NVIDIA K20 GPU which is currently being phased out of the large systems in production today (such as OLCF's Titan). These excluded platforms would distort the calculation of portability due to reasons of immaturity in the software stack and tell us little about the portability of our mini-app implementations in the programming models.

We calculate Φ for OpenMP and Kokkos across these three sets of architectures for each mini-app, and present the results in Figure 11. Application efficiency was used for all mini-apps to ensure the numerical values are directly comparable across mini-apps.

We can examine trends along each row of this figure by considering how each model fares on a given set of platforms



	BabelStream	TeaLeaf	CloverLeaf	Neutral	MiniFMM	Mean	Std. Dev.
OpenMP CPU	98.4%	100.0%	100.0%	100.0%	100.0%	99.7	0.6
Kokkos CPU	83.0%	49.8%	60.7%	77.6%	66.1%	67.5	11.9
OpenMP GPU	95.5%	22.5%	0.0%	0.0%	0.0%	23.6	37.0
Kokkos GPU	99.5%	64.3%	85.7%	51.1%	60.4%	72.2	17.7
OpenMP all	97.3%	43.6%	0.0%	0.0%	0.0%	28.2	38.5
Kokkos all	88.5%	54.4%	68.2%	65.0%	63.9%	68.0	11.2

Fig. 11. Performance portability of OpenMP and Kokkos

for *different applications*. On CPU platforms, OpenMP gets the best performance with Φ scores at (or very close to) 100%. For BabelStream, OpenMP does not get the highest performance on Naples (OpenACC) and Power 9 (Kokkos), however the performance different is very slight. Kokkos on CPU platforms does show some overhead, with the expectation that performance is reduced by 17–50%. One would expect some overhead, as the Kokkos abstraction is itself implemented using OpenMP.

On the GPU platforms, the support for a robust OpenMP offload across all platforms in lacking, and so renders $\Phi = 0$ in some cases where one platform was unable to correctly produce a result. Kokkos on GPU architectures on the other hand does well, again with overheads of between 0.5–49%, and around 14–49% excluding the BabelStream result. Note we only consider GPUs from one vendor, as unfortunately we were unable to collect sufficient results from other vendors.

Considering the "all" subset of architectures encompassing both CPUs and GPUs, the performance portability metric shows that the lack of widespread support for OpenMP on GPUs limits the portability of OpenMP as of today. These results show that where support is in place, the TeaLeaf mini-app achieved on average 45% application efficiency. Kokkos fares better thanks to the GPU support, and the final row of Figure 11 shows that performance portability *is* possible.

Figure 11 also shows the average (mean) and standard deviation of the performance portability metric results. As shown, for the Kokkos results on all platforms considered in the section, we see see $\overline{\Phi} = 68\%$, and so one would expect that for an application written in Kokkos we would expect to achieve within 32% of the "best" performance for a given platform. The standard deviation here is also fairly low at 11%, indicating that one would expect results to differ from the mean by only this much, leading us to expect that on average Kokkos should achieve 57–79% of the best performance. Indeed, this comes with no source changes at all, whereas OpenMP 4.5 requires different directives for CPU and GPU; the meta-directives in OpenMP 5.0 look like a very promising solution for this.

V. Productivity

An important part of the performance portability picture is the productivity of writing and maintaining applications. As mentioned in Section II, we required implementations of each of our mini-apps in each programming model used in this study. This results in lots of code to maintain in a mixture of Fortran, C and C++ as appropriate for each model.

The lines of code for each implementation of each mini-app, normalised to the smallest implementation for that code is shown in Figure 12 to give some indication for productivity. We used the simple UNIX tool `wc -l` to count all lines of the source, including comments which form an important part of the mini-apps. For CloverLeaf we only counted Fortran files (the implementation also include C kernels). Where the mini-app had multiple implementations, we selected only the source files required for each programming model in turn (including drivers/infrastructure each time). The figure highlights the verbosity of some of the low level models (such as OpenCL and CUDA). We also see that the original reference versions of CloverLeaf and TeaLeaf are rather long in comparison to the more recent ports to other models. Using lines of code is often a crude metric (as surveyed by Harrell et al [16]), and there are often more sophisticated methods, such as that recently proposed by Pennycook [17].

These mini-apps have been developed over a period of many years and as such we do not have raw data relating to productivity collected during the development of each mini-app (using a process such as one suggested by Harrell et al [18]). For want of this however, in our experience, it has typically taken about two weeks to port an application from any one model to another. This includes language translation, bug fixing, etc. As such, the verbosity of the language, represented in lines of code, does not capture the similar levels of effort to program in any of these models. The time to express a single parallel loop (concisely in a C++ abstraction, or else a separately compiled kernel like in OpenCL) is greatly amortised after the first is written, as the code can be copied and altered for the other loops saving a great deal of time.

The collection of the results for this study alone is worthy of mention of productivity. Given our choice of 12 platforms, 5 programming models and 5 mini-apps, there were up to 300 individual runtimes to collect for inclusion in this study. We have left out some results as not all are valid combinations; this in and of itself is an important observation. The majority of platforms are configured differently, with different software and programming environments, and so therefore required different paths for the various compilers, runtimes and libraries to be configured correctly.

To assist us, we developed a set of scripts which we make public on GitHub [3]. These scripts are structured in a way which allows us to share with the community how our codes were built and run on each of the systems required for this study. In this way we hope that the labour required to reproduce our results and expanding our efforts to include more codes and models is minimal.

[3] https://github.com/UoB-HPC/benchmarks/tree/doe-p3-2019

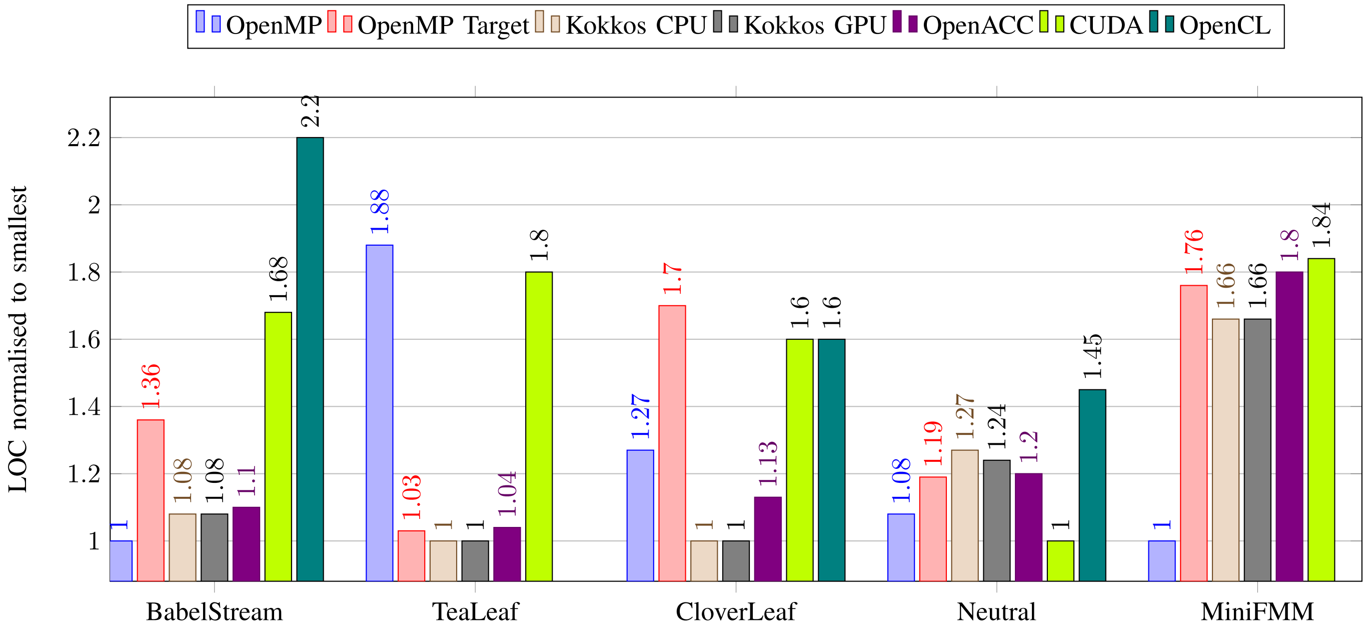


Fig. 12. Normalised lines of code for each implementation of each mini-app

VI. Conclusion

This paper represents the first broad study into performance portability of different applications across a diverse range of processors and programming models. The performance portability metric, originally proposed by Pennycook et al [1], was used to rigorously analyse the performance data and provide empirical evidence for how performance portable an application may be. A key part of our work was to analyse the performance portability of a range of applications written in OpenMP and Kokkos as well as a range of architectures. Our results show that as of today, it is possible to achieve performance portability in some cases, however in other instances we see quite variable results. Some programming models may do well on some platforms but perform poorly on others. Often this is a result of the compiler producing large differences in runtime, particularly in the case of OpenMP `target`. Both Kokkos and OpenMP do well in coverage of platforms to provide at least portability. The non-open and non-standard programming models, such as OpenACC, worked only on a very limited set of platforms.

We saw that Kokkos does well in allowing applications to achieve portable performance across the greatest range of both applications and architectures. The definition of performance portability presented in Section I asks for 20% of best performance; we found that Kokkos fared best and got close to our goal, achieving within 32% of the best performance on average.

Our results here, combined with our previous work described in Section I-A, show that a number of lessons can be learnt in how best to approach writing a performance portable code:

- Use open (standard) programming models supported by multiple vendors across multiple hardware platforms.
- Expose maximal parallelism at all levels of the algorithm and application, and thus allowing the programming model to map the work to appropriate hardware resources.
- Avoid over-optimising for any one platform, and develop and improve codes on multiple platforms simultaneously so as to demand portable performance.
- Although not discussed in this study, multi-objective auto-tuning can help find suitable parameters in a flexible code base to achieve good performance on all platforms [19].

Such techniques will allows us as a community to improve on the current state of performance portability. It is important to mandate performance portability when developing applications, and require that a minimum level of performance portability is maintained. This paper addressed the systematic measurement performance portability, and demonstrates a way to quantify the performance portability of a suite of codes.

This study will form the basis of our future work in this area, where we plan to increase our coverage by including additional mini-apps in this study from our partners. Additionally, when new architectures become available, such as the Arm-based Fujitsu A64FX and AMD Rome CPUs, we hope to add such results to our existing analysis. Also, the development of open standard programming models, such as OpenMP 5.0 and SYCL, provide new opportunities for developing portable codes. As the community continues to develop ways to analyse programmer productivity, in particular with an existing source code, we hope to apply such techniques to our performance portability study in the future.

Acknowledgment

Many thanks to Matt Martineau (NVIDIA) for his assistance with this study. This work used the Isambard UK National Tier-2 HPC Service (http://gw4.ac.uk/isambard/) operated by GW4 and the UK Met Office, and funded by EPSRC (EP/P020224/1). This work was also part funded by the EPSRC ASiMoV project (EP/S005072/1). Access to the Cray XC50 supercomputer 'Swan' was kindly provided through Cray Inc.'s Marketing Partner Network. Thanks extend to AWE for their financial support. This work was carried out using the computational facilities of the Advanced Computing Research Centre (ACRC), University of Bristol — http://www.bristol.ac.uk/acrc/. Many thanks to Chris Edsall from the ACRC at Bristol University for providing access to AMD Naples on the Oracle Cloud.

References

[1] S. J. Pennycook, J. D. Sewall, and V. W. Lee, "A Metric for Performance Portability," in *Programming Models, Benchmarking and Simulation (PMBS) workshop at SC*, 2016, pp. 1–7.

[2] E. Strohmaier, H. Simon, J. Dongarra, and M. Meuer, "Top 500 - November 2018," http://www.top500.org, 2018. [Online]. Available: www.top500.org

[3] 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, pMID: 25972727. [Online]. Available: https://doi.org/10.1177/1094342014528252

[4] S. McIntosh-Smith, M. Boulton, D. Curran, and J. Price, "On the Performance Portability of Structured Grid Codes on Many-Core Computer Architectures," in *Supercomputing*, ser. Lecture Notes in Computer Science, J. M. Kunkel, T. Ludwig, and H. W. Meuer, Eds. Cham: Springer International Publishing, 2014, vol. 8488, pp. 53–75. [Online]. Available: http://link.springer.com/10.1007/978-3-319-07518-1http://link.springer.com/10.1007/978-3-319-07518-1{_}4

[5] T. Deakin, J. Price, M. Martineau, and S. McIntosh Smith, "Evaluating attainable memory bandwidth of parallel programming models via BabelStream," *International Journal of Computational Science and Engineering*, vol. 17, no. 3, pp. 247–262, 2018. [Online]. Available: http://www.inderscience.com/link.php?id=10011352

[6] 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, aug 2017. [Online]. Available: http://doi.wiley.com/10.1002/cpe.4117

[7] 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, nov 2018, pp. 1–13. [Online]. Available: https://ieeexplore.ieee.org/document/8639943/

[8] C. Yang, R. Gayatri, T. Kurth, P. Basu, Z. Ronaghi, A. Adetokunbo, B. Friesen, B. Cook, D. Doerfler, L. Oliker, J. Deslippe, and S. Williams, "An Empirical Roofline Methodology for Quantitatively Assessing Performance Portability," in *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, nov 2018, pp. 14–23. [Online]. Available: https://ieeexplore.ieee.org/document/8639946/

[9] T. Zhao, S. Williams, M. Hall, and H. Johansen, "Delivering Performance-Portable Stencil Computations on CPUs and GPUs Using Bricks," in *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, nov 2018, pp. 59–70. [Online]. Available: https://ieeexplore.ieee.org/document/8639931/

[10] G. R. Mudalige, M. B. Giles, I. Reguly, C. Bertolli, and P. H. J. Kelly, "OP2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures," *2012 Innovative Parallel Computing, InPar 2012*, no. 2, 2012.

[11] S. McIntosh-Smith, M. Martineau, T. Deakin, G. Pawelczak, W. Gaudin, P. Garrett, W. Liu, R. Smedley-Stevenson, and D. Beckingsale, "TeaLeaf: A Mini-Application to Enable Design-Space Explorations for Iterative Sparse Linear Solvers," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, sep 2017, pp. 842–849. [Online]. Available: http://ieeexplore.ieee.org/document/8049027/

[12] M. Martineau and S. McIntosh-Smith, "Exploring On-Node Parallelism with Neutral, a Monte Carlo Neutral Particle Transport Mini-App," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, vol. 2017-Septe. IEEE, sep 2017, pp. 498–508. [Online]. Available: http://ieeexplore.ieee.org/document/8048962/

[13] P. Atkinson and S. McIntosh-Smith, "On the performance of parallel tasking runtimes for an irregular fast multipole method application," in *Scaling OpenMP for Exascale Performance and Portability*, B. R. de Supinski, S. L. Olivier, C. Terboven, B. M. Chapman, and M. S. Müller, Eds. Cham: Springer International Publishing, 2017, pp. 92–106.

[14] J. D. McCalpin, "Memory Bandwidth and Machine Balance in Current High Performance Computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, dec 1995.

[15] M. a. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," *Sandia National ...*, no. September, pp. 1–38, 2009. [Online]. Available: http://207.211.63.18/MantevoOverview.pdf

[16] S. L. Harrell, J. Kitson, R. Bird, S. J. Pennycook, J. Sewall, D. Jacobsen, D. N. Asanza, A. Hsu, H. C. Carrillo, H. Kim, and R. Robey, "Effective Performance Portability," in *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, nov 2018, pp. 24–36. [Online]. Available: https://ieeexplore.ieee.org/document/8639933/

[17] J. Pennycook, J. Sewall, and D. Jacobsen, "Quantifying productivity/maintenance cost with code base investigator," Presentation at Deparment of Energy Performance Portability and Productivity Meeting, April 2019.

[18] S. L. Harrell, J. Kitson, R. Bird, S. J. Pennycook, J. Sewall, D. Jacobsen, D. N. Asanza, A. Hsu, H. C. Cabada, H. Kim, and R. Robey, "Towards effective productivity: The case for in-situ tracking of performance portability development," Presentation at Deparment of Energy Performance Portability and Productivity Meeting, April 2019.

[19] J. Price and S. McIntosh-Smith, "Exploiting auto-tuning to analyze and improve performance portability on many-core architectures," in *High Performance Computing*, J. M. Kunkel, R. Yokota, M. Taufer, and J. Shalf, Eds. Cham: Springer International Publishing, 2017, pp. 538–556.

Appendix A
Artifact Description Appendix: Performance Portability across Diverse Computer Architectures

A. Abstract

Capturing the performance of implementations of five codes in up to five parallel programming models across twelve platforms required a systematic and reproducible approach. Many different programming environments, systems and compilers were required in order to collect these results. We describe here the scripts which we developed to build and run each code in a consistent manner across platforms. The scripts were made flexible to allow us to test multiple compilers where a choice was available.

B. Description

1) Check-list (artifact meta information):

- **Program:** BabelStream, TeaLeaf, CloverLeaf, Neutral, MiniFMM.
- **Compilation:** Variety of compilers, detailed in scripts.
- **Data set:** Input files detailed in scripts.
- **Run-time environment:** We used a wide range of compilers (and version numbers) for this study: Cray, GCC, LLVM, XL and PGI. We used the default environment (OS, etc) on each system.
- **Hardware:** We run on a number of systems in order to have large coverage of platforms. The Cray XC50 supercomputer 'Swan' was used for the Intel Intel Xeon Platinum 8176 (Skylake) and Intel Xeon Phi 7210 (Knights Landing) processors. The University of Bristol HPC 'Zoo' was hosted the Ampere Arm CPU, the NVIDIA GTX 2080 Ti (Turing) GPU, and the AMD Radeon VII GPU. The Oracle Cloud was used for access to AMD Naples. We used two of the University of Bristol's Advanced Computing Resource Centre's systems: BlueCrystal Phase 3 for access to the NVIDIA K20m (Tesla) GPU, and the BlueGem system for access to the NVIDIA V100 (Volta) GPU with an Intel x86 host. We used GW4's 'Isambard' Cray XC-50 system for access to Marvell ThunderX2 Arm processors, and 'Isambard Phase 1' (a Cray CS400) for access to NVIDIA P100 (Pascal) GPUs and IBM Power 9 CPUs with NVIDIA V100 (Volta) GPUs.
- **Execution:** Detailed in scripts.
- **Experiment workflow:** Codes are built and run via a set of scripts written in a common format.
- **Publicly available?:** Yes

2) How software can be obtained (if available): The scripts which download, build and run the software on each system are available on GitHub: https://github.com/UoB-HPC/benchmarks/tree/doe-p3-2019. We call this the benchmarks repository. The source code for the mini-apps themselves are all available on GitHub. The location of these may be viewed in the corresponding `fetch.sh` script in the benchmarks repository.

3) Hardware dependencies: The mini-apps used in this study are designed to run on different architectures, and in general there is a version of each code which runs on the hardware listed in the checklist above. Please see the main body of the paper (detailed throughout Section III) for currently unsupported or unavailable combinations.

4) Software dependencies: Each system has a unique set of compilers and programming environments. We installed additional compilers as required. The Kokkos versions were built using Kokkos 2.8.00 compiled on each system. The combinations of system and compiler is detailed in the options available for each `benchmark.sh` script in our repository.

In total, we used this collection of compilers in this study:

- GCC: 4.8, 4.9, 6.1, 7.2, 7.3, 8.1, 8.2, 9.1.
- Intel: 2018, 2019.
- XL: 16.1.
- PGI: 18.4, 18.10, 19.4.
- CCE: 8.7.
- LLVM: trunk.

CloverLeaf and TeaLeaf both require MPI implementations, however we only ran with a single MPI rank and so the choice of MPI library is unimportant. We used a combination of Cray's MPICH and OpenMPI as appropriate depending on what was available on the system.

5) Datasets: We detail the input deck or problem parameters for each mini-app. Each input deck is a standard one which ships with the source code of the mini-app.

- BabelStream: The default problem of 2^{25} FP64 elements per array.
- TeaLeaf: `tea_bm5.in`.
- CloverLeaf: `clover_bm16.in`.
- Neutral: `csp.params`
- MiniFMM: `large.in`

C. Installation

On each system, we clone the benchmark repository:

```
git clone -b doe-p3-2019 \
  https://github.com/UoB-HPC/benchmarks
cd benchmarks
```

D. Experiment workflow

The scripts are designed to download, build and run the mini-app, setting the correct paths so it can be build and run against the correct software versions.

Change to the mini-app and platform/system subdirectory, for example:

```
cd babelstream/tx2-isambard
```

Then to download and build the code with the default compiler and programming model, execute:

```
./benchmark.sh build
```

A different choice of compiler and model can be supplied to this command. To build the Kokkos version of BabelStream with GCC, one might execute:

```
./benchmark.sh build gcc-8.2 kokkos
```

The code is run (possibly by submitting a job to the system queue) as follows:

```
./benchmark.sh run
```

Any choices of compiler and model when building must also be supplied when running. For the previous example, the command to run the code following building is:

```
./benchmark.sh run gcc-8.2 kokkos
```

To see the supported combinations of compiler and programming model, run the script without any arguments:

```
./benchmark.sh
```

E. Evaluation and expected result

Once the code has finished running on the system (directly or via a job submission queue), any output is placed in a directory named with the following convention:

```
<platform>_<compiler>_<model>
```

The `stdout` output is captured in an output file in this directory. Any output files created by the application are also placed here.

For our previous example of Kokkos BabelStream on ThunderX2, the following directory would contain the built binary and any output files:

```
benchmarks/babelstream/tx2-isamabard/tx2_gcc-8.2_kokkos
```

F. Experiment customization

The `benchmark.sh` script and corresponding `run.sh` script are both designed to be easily customisable to add additional compilers and models.