Tracking Performance Portability on the Yellow Brick Road to Exascale

Tom Deakin*, Andrei Poenaru*, Tom Lin* and Simon McIntosh-Smith*
*Department of Computer Science, University of Bristol, UK
Email: {tom.deakin, andrei.poenaru, s.mcintosh-smith}@bristol.ac.uk

***Abstract*—With Exascale machines on our immediate horizon, there is a pressing need for applications to be made ready to best exploit these systems. However, there will be multiple paths to Exascale, with each system relying on processor and accelerator technologies from different vendors. As such, applications will be required to be portable between these different architectures, but it is also critical that they are efficient too. These double requirements for portability and efficiency begets the need for performance portability. In this study we survey the performance portability of different programming models, including the open standards OpenMP and SYCL, across the diverse landscape of Exascale and pre-Exascale processors from Intel, AMD, NVIDIA, Fujitsu, Marvell, and Amazon, together encompassing GPUs and CPUs based on both x86 and Arm architectures. We also take a historical view and analyse how performance portability has changed over the last year.**

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

I. Introduction

Exascale-class supercomputers are on the immediate horizon, and are proving to be built using a diverse range of high-performance architectures. The United States systems are predominately heterogeneous GPU-accelerated nodes, whilst the Fugaku supercomputer in Japan uses a homogeneous many-core CPU design, which exploits high bandwidth memory technology similar to GPUs. The GPU accelerators are provided by different vendors, with three GPU vendors (AMD, Intel, and NVIDIA) now in this space, along with multiple CPU vendors, including AMD, Fujitsu (Arm) and Intel. This range of different classes of architectures, with a choice from multiple vendors, demonstrates the acute need for developing performance-portable applications.

Parallel programming models are developed to target these different systems. Although some of these models are proprietary to specific vendors, the more attractive approaches are the open and standardised models which allow for a wide support base and cross-platform ecosystem. OpenMP and SYCL are two flagship examples of standardised models which enable us — as developers — to write performance-portable programs. The Kokkos abstraction framework offers another approach, adding another layer on top of the choice of programming model, to insulate users from this choice, at the expense, of course, of choosing an open-source, but not industry-standard, model.

This work was supported through ASiMoV under EPSRC grant number EP/S005072/1.

To further enable the development of performance-portable programs, in this study we update and greatly expand our earlier, wide-reaching study on performance portability [1]. We include the latest and greatest architectures, including for the first time the Arm-based Fujitsu A64FX processor, the NVIDIA Ampere GPU, and Intel GPUs. Thus, this study spans the processor architecture design space of the first Exascale machines.

As this work is an expansion and update of the 2019 study, we are able to begin to explore the historical perspective for how performance portability changes over time. The ecosystems surrounding each of the processors have had time to expand and mature, and therefore by refreshing many of the results from the original study in 2019 we can track the progress of support, performance, and performance portability.

In this update, for the first time we include results from applications written in SYCL. The applications we include are all open source and were ported for the purposes of this study, thus representing a contribution to the community in and of itself. SYCL is a key part of the Exascale ecosystem for Argonne National Laboratory [2], and is a critical part of the performance portability landscape today.

The contributions of this study can be summarised as:

- We expand the hardware coverage of our previous study by including the latest and greatest processors from all major HPC vendors. This study includes results from the key architecture tracks that are being used at Exascale.
- Applications developed in SYCL are included in this study, thus broadening the scope to include the key on-node programming models that will be used at Exascale.
- We provide a retrospective update on the performance portability landscape, including how the software ecosystem has evolved to demonstrate improved coverage.

II. Methodology

We take a similar approach to our prior peer-reviewed rigorous study into performance portability [1]. To this end, we consider a number of codes, each written in multiple programming models. This allows us to target a wide range of processors, and use a variety of compilers. Where more than one compiler is available, we test them all and select the best result[1]. As before, we measure the efficiency of each implementation on

[1]See Appendix A for the link to the repository which contains details on which compilers and implementations were used and the results from all tested combinations.

978-0-7381-1088-2/20/$31.00 ©2020 IEEE
DOI 10.1109/P3HPC51967.2020.00006

each architecture, with the primary focus that a consistent and high efficiency is measured across all platforms for any implementation. This motivates our description of performance portability:

Definition 1. A code is performance-portable if it can achieve a similar fraction of performance efficiency on a desired set of target architectures.

- The efficiency is calculated as either application or architectural efficiency, as defined by Pennycook et al. [3].
- We expect that efficiency to be high (ideally 80% or more of peak or best performance).

The performance portability metric, Φ, is defined as the harmonic mean of efficiencies over a set of platforms, or else zero if one of more platforms in the set is not supported. For readers unfamiliar with this metric we recommend they consult the paper by Pennycook et al. [3]. The Φ represents the *expected* performance of the application on the set of platforms. Differences in Φ occur due to the efficiency data, but as an aggregate value sheds little insight onto the underlying distribution [4].

To this end, we once again set about a broad reaching survey of platforms and programming models. Each code is implemented in six parallel programming models relevant to Exascale: OpenMP, OpenCL, OpenACC, CUDA, Kokkos, and, in addition for this updated study, SYCL. This collection represents a range of approaches for performance portability: compiler directives, modern C++ abstractions, as well as low-level APIs. A key feature of many of these models is to expose the programmer to the dual memory spaces of the host and the attached device, which may be a GPU or indeed the host CPU itself expressed through this abstraction.

The **OpenMP** standard is a collection of compiler directives for programming shared memory CPUs with optional attached devices which have their own separate memory spaces [5]. There are implicit rules and explicit controls defined for how data should be moved between the host and device memory spaces. It is up to the programmer to write directives to control the data movement between the two memory spaces, although the compiler (and features in OpenMP 5.0) can assist with using common directives for expressing parallelism in both cases. Because of this difference in memory models, the OpenMP codes we use are not strictly single source; however, we present unified results for OpenMP as the model itself is able to target both devices. It is beyond the scope of this paper to delve into this topic further.

OpenACC began as a trailblazer for the OpenMP standard, but remains to this day due to the commercial support of NVIDIA [6]. Compiler directives are used to show which loops can be parallelised and also to provide rules for mapping data between the host and device memory spaces. As with OpenMP, compiler support is provided to selectively ignore the directives when compiling simply for the host.

OpenCL is an open-standard parallel programming model from Khronos for programming heterogeneous devices [7]. Parallelism is expressed through kernel programming, where kernels contain essentially the loop bodies, with multiple copies executed on the device in parallel. It is a fairly low level model and as such has historically provided wide platform support. **CUDA** follows a similar kernel programming model to OpenCL, and is a commercial product for programming only NVIDIA GPUs [8]. HIP from AMD follows a similar model to CUDA, which we will consider including in a future study.

Kokkos is a C++ library abstraction layer which enables programmers to express parallel patterns and data spaces in an abstract manner, relying on the Kokkos library to map these to an underlying model such as OpenMP, CUDA or HIP [9]. This approach insulates the programmer from vendor-locked programming models by providing a path from common code to a variety of backends, increasing the portability. As an abstraction layer, compiler support for Kokkos is not required, which therefore places the burden of writing and maintaining backeneds on Kokkos itself, rather than shared between the compiler community. Kokkos supports a kernel style of programming, where lambda functions are written to describe the execution of a single loop iteration (similar to the kernels of OpenCL) and are executed over a range in parallel. RAJA and the associated libraries provide a very similar approach to performance portable programming and so we only show Kokkos results in this study; the differences between them have been explored previously [10], [11].

SYCL is a single-source C++ parallel programming model for heterogeneous systems [12]. Much of the original design of SYCL stemmed from the ideas in OpenCL, but it shares many similarities with C++ abstraction layers such as Kokkos. In particular, kernels are defined using lambda functions, where multiple instances are then instantiated on a device, and a `Buffer` abstraction is used for data. Based on how buffers are used in kernels, a dependency graph is constructed for execution, providing a powerful and convenient tasking model on top of a kernel-based programming approach. There is significant momentum behind the SYCL programming model at present following the announcement by Intel of their support for it as part of their oneAPI, which is providing a crucial path to Exascale for the Aurora supercomputer due to be installed at Argonne National Laboratory [2].

The hardware platforms we used for this study along with key performance details are summarised in Table I. This selection covers a wide sweep of the commonly available processors at the time of writing. Importantly, this list covers processors from the major hardware vendors in both the CPU and GPU spaces. For the most part, these architectures fall somewhere on the roadmap to Exascale-class architectures, and some are already used in pre-Exascale systems, such as the Fujitsu A64FX processor. It is important to note where CPUs are configured in single- or dual-socket systems, since this can often determine the total memory bandwidth available on each system, because the number of memory controllers is typically configured per-socket.

What is common in all these processors (both CPUs and GPUs alike) is that they contain a large number of cores, and

TABLE I: Tehcnical specifications of the processors used in this study

Processor	**Type***	**Vendor**	**Details**	**Cores/ Compute Units**	**Clock** (GHz)	**Peak memory bandwidth** (GB/s)	**Peak FP64 FLOPs** (TFLOP/s)
Cascade Lake	2S CPU	Intel	Xeon Gold 6248	2 x 20	2.50	282	3.2
Skylake	2S CPU	Intel	Xeon Platinum 8176	2 x 28	2.10	256	3.8
Knights Landing	CPU	Intel	Xeon Phi 7210	64	1.30	490	2.7
Rome	2S CPU	AMD	EPYC 7742	2 x 64	2.25	410	4.6
Power 9	2S CPU	IBM	—	2 x 20	3.20	340	1.0
ThunderX2	2S CPU	Marvell	—	2 x 32	2.50	288	1.3
Graviton 2	CPU	Amazon	Arm Neoverse N1	64	2.50	205	0.6
A64FX	CPU	Fujitsu	—	48 (+4)	2.20	1024	3.4
P100	GPU	NVIDIA	PCIe	56	1.13	732	4.0
V100	GPU	NVIDIA	SXM2	80	1.37	900	7.0
A100	GPU	NVIDIA	—	108	1.40	1555	9.7
RTX 2080 Ti	Consumer GPU	NVIDIA	—	68	1.25	616	0.4
Radeon VII	Consumer GPU	AMD	—	60	1.40	1000	3.5
MI50	GPU	AMD	—	60	1.73	1024	6.6
Iris Pro Gen 9	Integrated GPU	Intel	i7-6770HQ/Iris Pro 580	72	2.6/0.4	34	0.9

* "2S" indicates a dual-socket system

each of these cores typically provides some form of SIMD (or vector) parallelism. As such, there is a high degree of concurrency that needs to be exploited. As the performance of many HPC applications is bound by the main memory bandwidth, the trends to increase this resource can also be seen. GPU accelerators have typically provided an advantage here by utilising High Bandwidth Memory (HBM), although with the Intel Xeon Phi (Knights Landing) and now the Fujitsu A64FX we are seeing CPU-based architectures using the same technology and offering comparable levels of memory bandwidth. A key difference here is that such a CPU does not require heterogeneous programming.

This range of systems also highlights the diversity in vendors in both CPU and GPU technology. We include three Arm-based processors alongside four x86-based CPUs. We also include GPUs from three vendors: AMD, Intel, and NVIDIA. Whilst not all of these processors are destined for announced Exascale machines, the specific processors here all lie directly on a path to Exascale-era processors. This diversity of architectures, and in particular the diversity in attached accelerators, shows the need for writing performance-portable programs. It is no longer possible to write vendor-specific code, as there is now a wider landscape of vendors and processors to program for.

In this study, by focusing on portable programming models, we can provide insights into the current performance portability landscape. Our previous study provided a wealth of results [1], and having this historical data allows us to compare and assess how the landscape is changing. We restrict this historical analysis to the subset of platforms common between the two studies; see Section V for this analysis. In future years we hope to extend this analysis and begin to observe and track changes to the performance portability ecosystem over time.

III. BabelStream

BabelStream is an implementation of the traditional McCalpin STREAM kernels with a number of key differences in order to better align with modern HPC software styles [11], [13]. Arrays are allocated on the heap as opposed to the stack, and the size of the arrays is known only at run-time rather than at compile-time. This is in keeping with HPC codes which might read their input in from a file, for instance. BabelStream also has wide support for different platforms by providing implementations in many different parallel programming models in a common framework.

In this study we explore the performance portability of the Triad and Dot kernels. These kernels are both memory-bandwidth-bound, and are highly relatable building blocks to patterns found in many codes. For the Triad kernel we consider two problem sizes so as to explore performance portability of different inputs. We include this in response to the large last-level cache sizes of some of the processors in this study, so as to ensure the data is streamed from main memory rather than held in cache. However some devices do not have sufficient memory capacity for the larger problem size. The dot-product kernel is shown to explore the performance portability potential of reduction kernels.

A. Triad

Triad is the typical kernel used to measure the sustained memory bandwidth attainable on a given processor. It reads two large arrays and sums a multiple of one to the other, storing the result in a third array. Every update to the array is independent, with no sharing of data between array entries (unlike in a stencil update), and so it is easy to parallelise across the entire length of the array. The arrays are large, and should be held in main memory. As the kernel has no reuse of array elements, data must be moved from main memory, through the cache hierarchy, and then written back to main memory.

Figure 1a shows the achieved memory bandwidth in GBytes/s (where 1 GByte is 10^9 Bytes) for arrays of 2^{25} FP64 elements. As a throughput metric, the higher values indicate better performance. In the figure, an 'X' represents an

impossible combination of platform and programming model (such as CUDA on anything not from NVIDIA) for our representative systems in Table I, and 'E' represents a value which is possible but we encountered an error whilst collecting the result. The colour map range is shown in the colour bar which accompanies each heatmap.

For the first time we see Arm-based CPU architectures with HBM technology: the A64FX processor. This offers memory bandwidth similar to GPUs such as NVIDIA's V100. Note that on A64FX with the Fujitsu compiler we had to ensure the pointers were correctly labelled with `__restrict` and `const` qualifiers in order to attain this bandwidth; we expect this will not be required as the compiler matures. The AMD Rome system uses traditional DDR memory technology and has the highest bandwidth of all the CPUs with this constraint. Note that care must be taken to recall which CPU systems are dual- and single-socket when comparing bandwidth: the Graviton 2 processor is single socket, and as such shows highly competitive per-socket bandwidth.

The NVIDIA A100 GPU achieves the highest bandwidth of all our platforms, in line with the highest peak performance as listed in Table I. The Intel IrisPro GPU is an integrated GPU within the CPU package and shares access to the same physical memory as the CPU. As such, we do not expect the peak performance to be competitive with the other HPC GPUs; however, it is important to include Intel GPU results in order to have sufficient coverage of Exascale technologies. The AMD MI50 GPU likewise represents the best available GPU from AMD at the present time and has similar performance to NVIDIA's V100 GPU.

In contrast to our previous study [1], three models show almost complete coverage of all the platforms: OpenMP, Kokkos and SYCL. Indeed, we were able to obtain a complete set of results for OpenMP. This indicates that those models are portable across a wide range of processors; portability is a prerequisite to performance portability.

The architectural efficiency is computed based on the results in Figure 1a as a fraction of the theoretical peak in Table I. We plot this efficiency in Figure 1b. Consistent high efficiency indicates good performance portability under our definition in Section II.

Plotting efficiency shows the effects of the substantial last level cache on the AMD Rome CPU: $2 \times 256 = 512$ MiB on the dual-socket system. The input arrays are 256 MiB in size, yielding a total application footprint of 768 MiB, only 1.5 times larger than the cache size. According to the STREAM run rules[2], this is not a valid configuration as much of the arrays can be held in cache. We see this for the Kokkos result, where the efficiency exceeds 100% of the main memory bandwidth, indicating cache effects are in play. This motivates the need to consider a larger problem size.

1) Performance portability of BabelStream Triad: Although the heatmaps can provide intuitive information about the distribution of efficiencies, and it looks like the three portable models identified above (OpenMP, Kokkos and SYCL) are doing well in also providing good performance in general, we can be more precise by using the performance portability metric $\mathcal{P}$ as defined by Pennycook et al. [3]. Table II shows the performance portability metric results for each programming model calculated from the architectural efficiencies shown in Figure 1b. For many models as there are some platforms where we were not able to generate a result, we observe $\mathcal{P}$ values of zero for all platforms according to the definition of the metric.

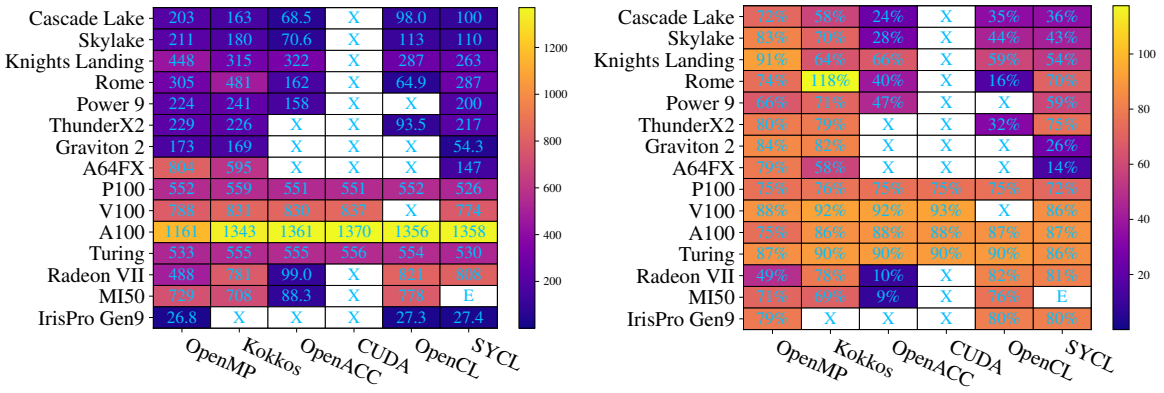
We restrict the platforms for each model to only those where results were generated, then recompute the $\mathcal{P}$. For those platforms where results do not exist, the ecosystem needs enrichment from (primarily) vendors in order to bring support for these models to their platforms: we remind the readers of the dangers of using proprietary models where such support can only be provided from one source. OpenMP covers all platforms and we see a $\mathcal{P}$ showing the expected performance is 75% of architectural efficiency. Kokkos shows a similar $\mathcal{P} = 75\%$. OpenACC is low, although this was clear from the low values on many platforms in Figure 1b; there are also only two-thirds of the platforms with results. CUDA likewise has very low portability, but does attain close to the peak performance on the NVIDIA GPUs it will run on (note architectural efficiency of 100% is unachievable).

The $\mathcal{P}$ for OpenCL combined with the absent results from Figure 1b is disappointing. OpenCL is demonstrating high performance on GPUs, but on CPUs we find that either the runtime is lacking performance, or on Arm platforms there is not official support (we used POCL on Arm and IBM to generate the results we could, but not all platforms were able to build POCL). This issue bleeds into SYCL a little, which on Intel CPUs uses the Intel OpenCL runtime for support. The Figure 1b (and [14]) shows the tightly coupled performance between OpenCL and SYCL on these platforms. We find in general that SYCL performance is close to that of the model used for the underlying implementation.

In order to expand on the differences between $\mathcal{P}$ for OpenMP, Kokkos and SYCL, we considered the performance portability over CPUs and GPUs in isolation in order to see if support for a class of processors are limiting the performance of any given model. We again compute $\mathcal{P}$ over the supported CPUs and GPUs for each model, presenting the numbers in the final two rows of the first section of Table II.

For Kokkos, we find that similar $\mathcal{P}$ values are attained on both classes of processor, with $\mathcal{P}$ on CPUs a little lower than on GPUs. This shows that Kokkos is successfully providing isolation from the underlying implementations required to run on the hardware from different vendors. This abstraction comes at a cost however, for Kokkos is not a standard but an open-source project and so support for the various backends comes from the core Kokkos team at Sandia National Laboratories and other collaborators to the project. As such we could not run on an Intel GPU with Kokkos at this time as such devices are not supported by the Kokkos build system. As a library abstraction layer too, Kokkos must maintain this layer in isolation; many vendors' compilers and parallel

[2]https://www.cs.virginia.edu/stream/ref.html#size



(a) Sustained memory bandwidth in GBytes/s, higher is better

(b) Architectural efficiency, higher is better

Fig. 1: BabelStream Triad results for arrays of length 2^{25} FP64 elements

TABLE II: Performance portability metric for BabelStream Triad

Platform set	Φ by programming model					
	OpenMP	Kokkos	OpenACC	CUDA	OpenCL	SYCL
All platforms	75.1	0	0	0	0	0
All non-zero platforms	75.1	75.4	27.3	86.1	46.6	47.4
Supported CPUs	77.9	71.6	35.9	0	30.8	36.1
Supported GPUs	72.2	81.2	22.8	86.1	81.4	81.7
Large input non-zero platforms	57.9	69.5	27.2	85.9	44.8	55.1

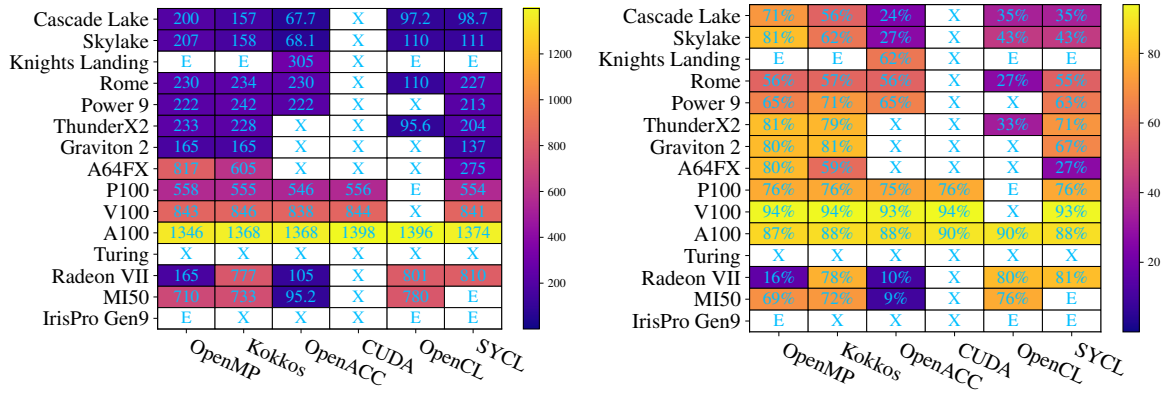
runtimes are collaborating around the LLVM project as a common infrastructure helping mature the wider ecosystem in partnership. Kokkos is a very real pragmatic solution for providing performance portability but with a high cost of ownership; standards based programming models on the other hand have active support directly from the vendors.

We find the two Φ values for OpenMP on CPU and GPUs shows promise, and the gap between performance on CPUs and GPUs is similarly small as it is for Kokkos. The GPU performance is much improved from our earlier studies, however still trails a little behind the CPU performance. The community has much improved over the last year, with GCC 10 bringing support for NVIDIA and AMD GPUs, and Intel's C++ compiler as shipped under oneAPI (note this is distinct from both the DPC++ compiler and their usual production compiler) support OpenMP on Intel GPUs. Combined with the support in Cray's CCE and LLVM's Clang, the base for support is very wide. On CPUs however, OpenMP continues to provide the leading performance of the models tested.

The situation for SYCL shows that it is the CPU performance that is holding back the Φ value. On GPUs, we see $\Phi = 82\%$ which is close to the underlying models of OpenCL and CUDA, yet on CPUs we see a much lower value. This is visible in the heatmap in Figure 1b by the low pink-coloured values particularly on Intel and some Arm-based CPUs. SYCL has a wide support base, however there are primarily two backends used for SYCL: OpenMP and OpenCL. For the OpenCL backends, such as those on Intel, we see the performance again limited by the issues surrounding thread spawning that has long plagued these runtimes (we first saw this for BabelStream in 2015 [15] and it remains through to 2018 [11], 2019 [1] and 2020 [14]). The OpenMP backends have other limitations: they typically use a library-based implementation, rather than compiler-based, and the current mainline approaches map SYCL work-items to OpenMP threads[3]. This means that threads are accessing adjacent memory locations which produces poor parallel memory access patterns. As BabelStream is main memory bandwidth bound, this limitation is sometimes less visible in the results as the bottleneck of moving memory from memory into the caches is sufficient to mask this problem; the problem can be observed for other simple patterns [14]. Therefore, the community needs to focus around improving the support for SYCL on CPUs. This is highly likely with planned improvements to hipSYCL[4] and Intel's support of SYCL as part of oneAPI.

2) Effects of using a larger input size for Triad: We supplement these results with a larger input of 2^{29} FP64 elements in order to mitigate the size of the large last-level cache on Rome. This has a memory footprint of 12 GiB which far exceeds the cache size. The peak bandwidth results are shown in Figure 2a with the architectural efficiency shown in Figure 2b. As expected, the Rome result is now lower than 100% indicating the cache residency effects are mitigated.

[3]https://github.com/illuhad/hipCPU#implementation
[4]https://github.com/illuhad/hipSYCL/pull/289



(a) Sustained memory bandwidth in GBytes/s, higher is better

(b) Architectural efficiency, higher is better

Fig. 2: BabelStream Triad results for arrays of length 2^{29} FP64 elements

BabelStream allocates some host vectors in the driver code for results checking; by default these are allocated before the programming models allocate the memory used for execution. Due to the strong NUMA effects on A64FX, this means that only for this large problem, the memory in the first NUMA domain is used up by these vectors, resulting in the memory used for execution to be allocated in other NUMA regions. For the result presented we moved this driver allocation until after the benchmark has run, and we are working to redesign the benchmark execution to take this into account universally.

We would like to note that although the Knights Landing has sufficient memory for this problem, many of the models ran out of memory.

The $\mathcal{P}$ values for this large problem are shown as the final row in Table II. For both OpenMP and Kokkos, we see the expected performance efficiency is around 23% and 8% lower respectively (a raw reduction of 17% and 6%). Such changes in $\mathcal{P}$ with problem size are in line with the changes previously seen by Daniel and Panetta [16]. As such, as Exascale technologies are enabling larger problems to be solved, it is important to consider how performance portability might be effected by the changes in input problem.

B. Dot

The BabelStream application contains an extension to the traditional STREAM benchmark in the form of a dot-product kernel. This read-only kernel takes two arrays and sums the element-wise product to produce a single value. This requires a parallel reduction, which is a key parallel pattern in many HPC codes. Unlike Triad, the Dot product requires some communication between the units of parallel work (threads, or equivalent) in order to produce just a single result. Many programming models embed first-class support for reductions in the language; of the programming models we use here this is the case for OpenMP, Kokkos and OpenACC. The other models require users to write their own reduction kernel. We would like to highlight that first-class reduction support has been added to SYCL in the 2020 provisional version however BabelStream is currently implemented with SYCL 1.2.1.

The architectural efficiency of the BabelStream Dot kernel for arrays of length 2^{25} FP64 elements is shown in Figure 3. It is clear that in comparison with the Triad kernel, it is rare to attain high efficiency for a great many of the combinations of platform and programming model. This is seen in Table III showing the $\mathcal{P}$. Kokkos fares best here, supporting the greatest number of platforms with the best efficiencies. Reductions in OpenMP on CPUs show similar expected efficiencies to the Triad kernel, but OpenMP on GPUs shows much room for improvement. We know that the reductions on GPUs in the open-source LLVM OpenMP implementation are in need for optimisation, and these results provide clear evidence for this. SYCL on GPUs shows little overhead over OpenCL, which is as we would expect as both use the same manual implementation of a reduction. However, there is less success on CPUs despite tuning the number of work-groups based on device preferences as reported by the OpenCL and SYCL runtimes. This is due to the challenges of implementing `ndrange` parallelism in a library [17]. The performance of OpenACC is similar to Triad.

Therefore, the reduction parallel pattern highlights the need to improvements in the support from many of the parallel programming model runtimes. The key focus in our opinion should be on improving the support for OpenMP reductions on GPUs and to implement the latest SYCL 2020 provision specification which provides first-class support for reductions in SYCL. As with Triad, it is the CPU performance which is limiting the numerical results for SYCL at this time.

IV. CloverLeaf

CloverLeaf is a 2D hydrodynamics mini-app for solving Euler equations [18]. It operates on a structured grid, with a around a dozen kernels either looping over the vertices or the central values performing either stencil or local updates.

TABLE III: Performance portability metric for BabelStream Dot

Platform set	$\mathcal{P}$ by programming model					
	OpenMP	Kokkos	OpenACC	CUDA	OpenCL	SYCL
All platforms	9.3	0	0	0	0	0
All non-zero platforms	9.3	60.5	15.6	86.1	7.4	0.7
Supported CPUs	79.7	57.1	42.0	0.0	3.5	0.4
Supported GPUs	4.6	65.7	10.3	86.1	78.3	74.6

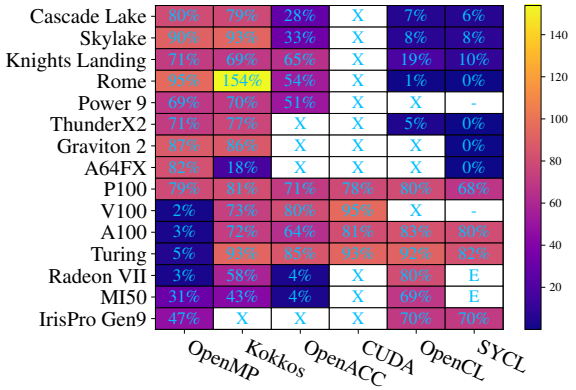


Fig. 3: BabelStream Dot architectural efficiency results for arrays of length 2^{25} FP64 elements, higher is better

We use the typical `bm_16` input which has a mesh size of 3840-by-3840 cells. This has a memory footprint of a few hundred MB, and so it exceeds the size of the last level cache on many processors, and the low operational intensity means CloverLeaf is principally main-memory-bandwidth-bound. The trend for increasingly large caches means that for the AMD Rome processor this problem size is well within the 256 MiB last level cache capacity per socket. We will show that for CloverLeaf, as was the case for the BabelStream results in Figure 1a, Rome was able to attain performance close to those processors with HBM due to the problem fitting in cache. It is important to note that CloverLeaf is a mini-app, and as such the input sizes provided with the code are designed to be representative, and so it is not necessarily feasible to simply increase the mesh size to exceed cache sizes.

We show the total runtime, in seconds, for CloverLeaf in Figure 4a. Any combinations of platform and programming model which are impossible to collect are shown with an 'X'. For combinations for which we encountered an error at build time which we could not resolve have been marked as 'B'. Any runtime errors, which range from execution completing but with erroneous results to execution finishing early due to a crash, we denote with 'E'.

The Fujitsu A64FX processor, which has HBM, achieves performance similar to that of the NVIDIA GPUs, which use similar memory technology, although as with any new system we expect to see improvements in the software stack of the A64FX in the future. Indeed, the OpenMP result for A64FX is lower than MPI due to the NUMA effects of the Core Memory Group (CMG) design, and for a production-style run of a hybrid code where one MPI rank is launched per CMG with OpenMP used across cores in the CMG we see the performance improve to levels very close to flat MPI performance. Still, these are promising results, as close-to-GPU performance can be obtained without needing to program with a heterogeneous programming model.

We include MPI results in our study, augmenting the approach taken with our earlier study [1]. We can see that on x86 processors, MPI and OpenMP attain very similar levels of performance. However, on two of the three Arm-based processors we observe that OpenMP performance is reduced compared to MPI. On A64FX we have already identified this as occurring due to NUMA. On the Amazon Graviton 2 processor, however, we observe this trend reversed. This therefore requires future investigation and collaborations with the various compiler and software teams to understand this difference going forwards.

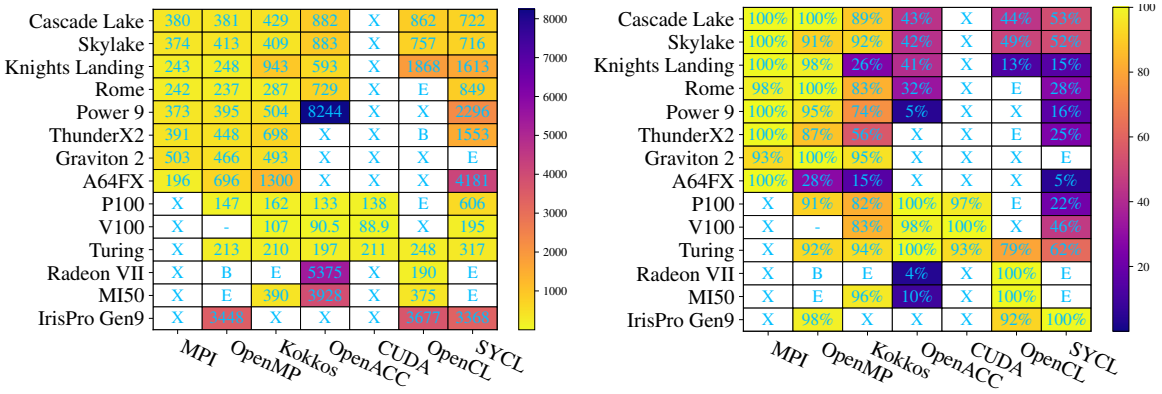
The Kokkos build used for Cascade Lake simply targets Skylake, and similarly Kokkos does not yet know about the A64FX processor, with only a generic ARMv8.1 target being the most specific option in this case. This means that the compiler flags as selected by the Kokkos library may not be optimal for these processors, as reflected in the larger performance overhead compared to the underlying OpenMP compared to the other platforms. The reduced performance on Knights Landing due to the lack of vectorisation is an outstanding issue from 2019[5].

We contribute a new version of CloverLeaf using OpenMP target, written in OpenMP 4.5 to supersede the version previously available which used OpenMP 4.0 and was written principally for the Intel Xeon Phi (Knights Corner). The OpenCL version is written using C++98, and so for the non-Intel CPUs where we were using the POCL library for support, we found an incompatibility as POCL requires a newer version of C++.

For the SYCL results on NVIDIA GPUs, much of the slowdown is attributed to the timestep reduction [14].

Comparisons using Figure 4a are useful for determining which processor was able to run the problem the fastest, but that is not the focus of this work. Indeed, some processors such as the IrisPro GPU do not currently have comparable specifications to the HPC GPUs (see Table I), but we include them here because they lay the foundational stones on the path to Exascale. A more meaningful comparison can be drawn by computing the application efficiency [3], and we show these

[5]https://github.com/kokkos/kokkos/issues/2216



(a) Total runtime in seconds, lower is better

(b) Application Efficiency, higher is better

Fig. 4: CloverLeaf results with `clover_bm_16.in` input

results in Figure 4b. This shows the performance relative to the fastest runtime observed on that architecture. We can use these efficiencies to calculate $\mathcal{P}$, shown in Table IV.

As with BabelStream, we find that there is still at least one platform where there is no result for each programming model, thus giving a $\mathcal{P}$ value of zero. We again recalculate on the supported platforms, but note that for CloverLeaf there are fewer platforms contributing than for BabelStream. CloverLeaf is a significantly larger application and many of the versions are hybrid in base language (Fortran plus C and/or C++), meaning a correct compilation for some of the programming models can be challenging.

MPI shows close-to-optimum performance portability on CPUs and, when comparing to OpenMP, we see how the performance loss on some of the non-x86 platforms reduces the $\mathcal{P}$ there. Still, on CPU platforms OpenMP still has the highest $\mathcal{P}$ of the solely-intra-node programming models. The other models on CPUs do not yet provide compelling results due to the variability in efficiency values. As with BabelStream, we find the OpenACC performance on CPUs is limited, and there are very few choices in compilers with which to explore this in detail.

For SYCL, we rely on hipSYCL, which at the time of writing has non-optimal memory access patterns on CPUs due to the use of the hipCPU library (recall our comments in Section III-A1). On Intel platforms, we use DPC++, which unlike hipCPU, uses OpenCL for its implementation. The results in this paper extend our work on other systems exploring a SYCL implementation of CloverLeaf [14]. In both cases, the performance of SYCL is limited by issues in Intel's OpenCL runtime, the parallelism in hipCPU, and ROCm stack issues on AMD platforms. However, platforms support for SYCL is strong, with compilers building our code on all platforms, although three platforms had runtime errors. We expect the situation here to improve in the very near future as compilers and drivers mature.

On GPUs, we find that OpenMP, OpenCL and CUDA all provide the best performance on the devices they support. Again, OpenACC shows little overhead to CUDA on NVIDIA GPUs, but on AMD GPUs we find the performance from the GCC compiler significantly lower, which is shown in the low $\mathcal{P}$ value despite the contributions from platforms with close to 100% efficiency.

Kokkos is showing a fair level of performance portability for CloverLeaf, although its total $\mathcal{P}$ is not as high as OpenMP. The $\mathcal{P}$ on the CPU platforms is reduced from the lower efficiency values on primarily Knights Landing, but also the Arm platforms. Excluding just Knights Landing, we see $\mathcal{P} = 78.7\%$ on CPUs, and $\mathcal{P} = 82.3\%$ also including the supported GPUs, which shows that where Kokkos has been sufficiently tuned to the underlying architecture and underlying model, it succeeds in providing good opportunities for achieving performance portability.

CloverLeaf, as a larger more complex code, stretches the performance portable programming models beyond BabelStream. Some of the implementations have remained untouched for a number of years and as compilers, systems and platforms have been upgraded, updating the build systems and base language versions has become a necessary task which we will explore in the future. Overall we find that Kokkos helps insulate from the underlying models and platforms, and where Kokkos itself has built in support for the platform the performance is both good and consistent, aligning with our definition of performance portability in Section II.

V. Historical Comparisons

Systematic evaluation of performance portability requires a consistent approach to benchmarking. This study continues our previous effort in running different applications in different programming models on a wide variety of diverse architectures [1]. As a result of this crucial data, we can begin to track the historical changes in performance portability of these codes over time. We begin this study in earnest here, focusing on the Triad kernel from BabelStream. We choose this as it has the

TABLE IV: Performance portability metric for CloverLeaf

Platform set	Φ by programming model						
	MPI	OpenMP	Kokkos	OpenACC	CUDA	OpenCL	SYCL
All platforms	0	0	0	0	0	0	0
All non-zero platforms	98.8	78.1	53.0	13.5	96.5	42.9	20.1
Supported CPUs	98.8	73.5	44.1	15.5	0	25.1	24.9
Supported GPUs	0	93.6	88.4	12.0	96.5	91.9	42.8

most complete results over the generations of benchmarking this particular code, and is often highly insightful into how other memory bandwidth bound codes behave.

In addition to the results from Deakin et al. from 2019 [1], we also use BabelStream results from our earlier work in 2018 [11], where there are a number of platforms in common between these studies. We select the processors common to these two studies and consider the changes in efficiency for the various programming models. We also include results from an Intel Xeon E2699 v4 (Broadwell) dual-socket system from the 2018 study which is still available today, and so we were able to run the application there as part of this study.

Architectural efficiencies for the subset of platforms common between these three studies are plotted in Figure 5. Each programming model is shown separately. Each line tracks the efficiency for one processor from the three studies between the years 2018 and 2020.

The notation Φ_x is used to represent the performance portability metric Φ computed on the platforms in the year x.

OpenMP shows good progress, with improved performance for Knights Landing and most GPUs, and in particular the support for the AMD GPU thanks to the GCC compiler. Crucially, $\Phi_{2020} = 74.7\%$ is non-zero, because all platforms are supported, in contrast to last year, where support was lacking for the AMD GPU. The Φ_{2020} value is very similar to $\Phi_{2018} = 65.3\%$ but over a much increased platform set. Note that if we exclude the AMD GPU from 2019, we find $\Phi_{2019\setminus AMD} = 82.1\%$ indicating that some platforms have reduced efficiency in 2020, notably the Power 9 and the ThunderX2. This latter result is lower because in 2019 not all cores were used, because higher efficiency was attainable in that smaller configuration, but this year we always collect all results using all available cores. Most of the results are tightly clustered around 80% efficiency (with $\Phi_{2020\setminus AMD} = 80.5\%$), showing that high consistent performance is on the near horizon with OpenMP.

Kokkos has showed steady levels of efficiency over these three studies. With the support available for AMD GPUs in the latest results, the figure shows that the efficiency on this platform is better than what is achievable with the current implementation of OpenMP in GCC. We find $\Phi_{2019\setminus AMD} = 76.7\%$ (excluding the AMD GPU; including it has $\Phi_{2019} = 0$) and $\Phi_{2020} = 74.3\%$, which shows a sustained level efficiency over time, even when new platforms are added. On the three platforms in 2018, $\Phi_{2018} = 65.7\%$ which shows the small improvements on those platforms observed by the slight upward trend on the graph. What is most noticeable here is that most of the efficiency data is fairly clustered, and exceeds 60% on all these platforms. Notice that Φ_{2020} for Kokkos is 3.8 percentage points lower than the majority of the OpenMP data when we exclude the Radeon VII, indicating that Kokkos does show a very small overhead in its abstractions, but it more than makes up for this when all platforms are considered.

For OpenACC, Figure 5 shows that on NVIDIA GPUs, the performance is stable in this time period. However, it is clear that this model suffers from the limited support on other architectures. On all processors from other vendors, the performance has fallen. Note too that both Φ_{2019} and Φ_{2020} are zero due to the lack of support for Arm. Despite the progress with GCC supporting OpenACC offload to AMD GPUs, Cray have removed support for OpenACC in their latest compiler, leaving only one commercial compiler (PGI) that supports this framework, which itself is due to be overhauled as part of the NVIDIA HPC SDK which was made available after these results were collected. The results are spread out over the entire efficiency range, in contrast to the clustered results of OpenMP and Kokkos.

The efficiencies for OpenCL seem consistent, but they remain spread out. The Φ is zero for both 2019 and 2020 as there are some platforms for which is it not possible to run OpenCL. The 2019 study had access to a V100 with an x86 host which was not available for this study, and the IBM Power host for the V100 in this study does not have an OpenCL driver. As such we can no longer include a result for the V100. Additionally, although we were able to use POCL to bring OpenCL support to Arm CPUs, this did not work on IBM Power so there was no way to run OpenCL on an IBM-based system. We find that for the non-zero platforms, $\Phi_{2019} = 67.6\%$ and $\Phi_{2020} = 56.0\%$. This reduction is felt due to the high efficiency of the V100 from 2019 keeping the Φ value buoyant no longer being included in this statistic. If we compute Φ on the two Intel platforms, the P100 and Turing and Radeon VII GPUs we see a very stable change from $\Phi_{2019} = 64.1$ to $\Phi_{2020} = 65.6$. Figure 5 also confirms the observations made in Section III-A1 regarding the fact that the performance on CPUs is limiting the overall performance portability of OpenCL.

Both the clustering of the data and the historical trends observable in plots such as those shown in Figure 5 provide valuable insight into measuring changes in performance portability. Going forward, we hope to be able to extend this plot to track performance portability over future studies. This will augment this contribution which already begins to track the continental drifts in the landscape of programming HPC

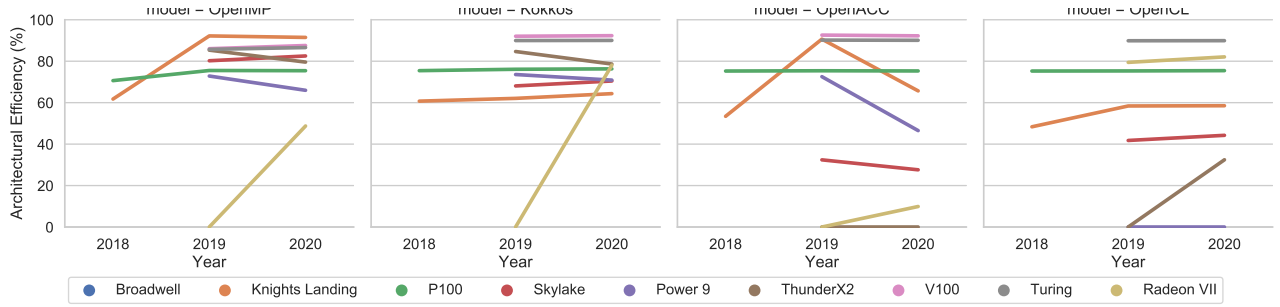


Fig. 5: Tracking changes of architectural efficiency in 2018–2020

processors. In addition, as we collect more historical data from other programming models, notably SYCL, we can track their progress in their enablement of performance portability.

VI. Conclusion

This study provides a wealth of performance results for codes written in lots of different parallel programming models. These models all allow for performance-portable programs to be written, and by benchmarking applications written in those models on the latest hardware on the path towards Exascale, we can assess how successfully they achieve that aim. It is a multi-faceted problem, however, for although in theory all such programs can be portable and performance-portable across system, the reality can differ based on system and platform support. As such, surveying this landscape is an important endeavour and we hope that the results we present here can help improve the ecosystem and allow developments in the fundamental understanding of what performance portability means. In addition, the results can help build a historical picture of how the performance portability landscape is changing as we are propelled towards ever increasing levels of parallelism.

We include SYCL in this study. SYCL is a crucial part of the path to Exascale programming, and is one viable choice for using cross-platform open standards for programming highly parallel heterogeneous systems. The situation is promising, but on CPU platforms there needs to be significant investment from vendors in order to ensure that performance and support are suitable. We plan to continue to observe how support for SYCL progresses over the coming years.

Kokkos continues to provide isolation from the underlying programming models by hiding vendor-specific models behind a common abstraction. This means that where Kokkos has support for a platform, we can observe good levels of performance portability. For BabelStream Triad, we found that Kokkos and OpenMP had similar levels of performance portability. For the reductions required in the Dot kernel, the performance on GPUs is lower for OpenMP, but this was not the case for Kokkos, as they can provide optimised implementations for such patterns. This clearly comes at a huge cost of ownership, and this can be observed as the Kokkos project has expanded to include contributions of backends from the wider community.

The support for OpenMP is considerably more robust that in our study last year, observed though our tracking of the historical changes in performance portability of all the models in Section V. We were able to show OpenMP running on all our platforms. The support for OpenMP GPUs in particular has improved, where the efforts of the open-source communities of GCC and LLVM have paid dividends. In addition, with the Intel compiler now supporting Intel GPUs, OpenMP is the first model we explored which has platform support from all classes of processor. The $\mathcal{P}$ values for OpenMP confirm performance portability for this open-standard parallel programming model.

Acknowledgment

This work is funded by the EPSRC ASiMoV project (EP/S005072/1). We extend our thanks for Fujitsu for access to the FX1000 system under the Early Access Program. 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). Access to the Cray XC50 supercomputer 'Swan' was kindly provided through the Cray Marketing Partner Network. 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/. We also used their 'Cluster in the Cloud' tools (https://cluster-in-the-cloud.readthedocs.io/en/latest/) for configuring Amazon Web Services. This work used the HPC Zoo, a research cluster run by the High-Performance Computing Group at the University of Bristol — https://uob-hpc.github.io/zoo/. This work used the DiRAC@Durham facility managed by the Institute for Computational Cosmology on behalf of the STFC DiRAC HPC Facility (www.dirac.ac.uk). The equipment was funded by BEIS capital funding via STFC capital grants ST/P002293/1, ST/R002371/1 and ST/S002502/1, Durham University and STFC operations grant ST/R000832/1. DiRAC is part of the National e-Infrastructure. The NVIDIA A100 results have been provided by Paul Graham from NVIDIA.

References

[1] T. Deakin, S. McIntosh-Smith, J. Price, A. Poenaru, P. Atkinson, C. Popa, and J. Salmon, "Performance Portability across Diverse Computer Architectures," in *2019 IEEE/ACM International Workshop on*

Performance, Portability and Productivity in HPC (P3HPC). Denver, CO: IEEE, nov 2019, pp. 1–13.

[2] "Khronos Steps Towards Widespread Deployment of SYCL with Release of SYCL 2020 Provisional Specification," 2020. [Online]. Available: https://www.khronos.org/news/press/khronos-releases-sycl-2020-provisional-specification

[3] 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.

[4] J. Sewall, S. J. Pennycook, D. Jacobsen, T. Deakin, and S. McIntosh-Smith, "Interpreting and Visualizing Performance Portability Metrics," in *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, nov 2020.

[5] *OpenMP Application Programming Interface*, OpenMP Architecture Review Board, 2015, Version 4.5.

[6] *The OpenACC Application Programming Interface*, The OpenACC Community, 2019, Version 3.0.

[7] *The OpenCL Specification*, Khronos, 2012, Version 1.2.

[8] *CUDA Toolkit Documentation v11.0.3*, NVIDIA, https://docs.nvidia.com/cuda/.

[9] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[10] M. Martineau, S. McIntosh-Smith, W. Gaudin, M. Boulton, and D. Beckingsale, "A Performance Evaluation of Kokkos and RAJA using the TeaLeaf Mini-App (poster)," in *Supercomputing*, Austin, Texas, 2015.

[11] 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.

[12] *SYCL Specification*, Khronos, 2020, Version 1.2.1, Document Revision 7.

[13] 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.

[14] T. Deakin and S. McIntosh-Smith, "Evaluating the performance of HPC-style SYCL applications," in *Proceedings of the International Workshop on OpenCL*. New York, NY, USA: ACM, apr 2020, pp. 1–11.

[15] ——, "GPU-STREAM: Benchmarking the achievable memory bandwidth of Graphics Processing Units (poster)," in *Supercomputing*, Austin, Texas, 2015.

[16] D. F. Daniel and J. Panetta, "On Applying Performance Portability Metrics," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, nov 2019, pp. 50–59.

[17] A. Alpay and V. Heuveline, "Sycl beyond opencl: The architecture, current state and future direction of hipsycl," in *Proceedings of the International Workshop on OpenCL*, ser. IWOCL '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3388333.3388658

[18] 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.

APPENDIX A
ARTIFACT DESCRIPTION APPENDIX: TRACKING PERFORMANCE PORTABILITY ON THE YELLOW BRICK ROAD TO EXASCALE

A. Abstract

Capturing the performance of implementations of codes in different parallel programming models across multiple 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): Fill in whatever is applicable with some informal keywords and remove the rest

- **Program:** BabelStream, CloverLeaf.
- **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, PGI, Fujitsu and the Arm compilers.
- **Hardware:** We run on a number of systems in order to have large coverage of platforms. See the Acknowledgements section for system details.
- **Execution:** Detailed in scripts.
- **Output:** We have saved the output for all results in the repository.
- **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/performance-portability. The source code for the mini-apps themselves are all available on GitHub. The location of these may be viewed in the corresponding `common.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 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 3.1 or the devlop branch compiled on each system. The combinations of system and compiler is detailed in the options available for each `benchmark.sh` script in our repository.

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} and 2^{29} FP64 elements per array.
- CloverLeaf: `clover_bm16.in`.

C. Installation

On each system, we clone the benchmark repository:

```
git clone \
  https://github.com/UoB-HPC/\
    performance-portability
cd benchmarking
```

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-9.2 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-9.2 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 saved in a file named with the following convention:

```
<code>-<platform>_<compiler>_<model>.out
```

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:

```
benchmarking/2020/babelstream/\
  tx2-isamabard/BabelStream-tx2_gcc-8.2_kokkos
```

and output would be saved in the current working directory:

```
BabelStream-tx2_gcc-8.2_kokkos.out
```

F. Experiment customization

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