An Investigation into the Performance and Portability of SYCL Compiler Implementations

Wageesha R. Shilpage and Steven A. Wright(✉)

University of York, York, UK
steven.wright@york.ac.uk

Abstract. In June 2022, Frontier became the first Supercomputer to "officially" break the ExaFLOP/s barrier on LINPACK, achieving a peak performance of 1.1×10^{18} floating-point operations per second using AMD Instinct accelerators. Developing high performance applications for such platforms typically requires the adoption of vendor-specific programming models, which in turn may limit portability. SYCL is a high-level, single-source language based on C++17, developed by the Khronos group to overcome the shortcomings of those vendor-specific HPC programming models. In this paper we present an initial study into the SYCL parallel programming model and its implementing compilers, to understand its performance and portability, and how this compares to other parallel programming models. We use three major SYCL implementations for our evaluation – Open SYCL (previously hipSYCL), DPC++, and ComputeCpp – on a range of CPU and GPU hardware from Intel, AMD, Fujitsu, Marvell, and NVIDIA. Our results show that for a simple finite difference mini-application, SYCL can offer competitive performance to native approaches, while for a more complex finite-element mini-application, significant performance degradation is observed. Our findings suggest that development work is required at the compiler- and application-level to ensure SYCL is competitive with alternative approaches.

Keywords: SYCL · High-Performance Computing · Performance Portability

1 Introduction

In the seven decades since the UNIVAC-I digital computer, computing has evolved enormously, fuelled by developments in both hardware and software. As computing power has increased, so too has our reliance on computing as a primary tool in scientific research. The fastest computers in the world today are being employed to help us solve many fundamental questions in the science and engineering disciplines. The High Performance Computing (HPC) research field

A. Bienz et al. (Eds.): ISC High Performance 2023 Workshops, LNCS 13999, pp. 605–619, 2023.
https://doi.org/10.1007/978-3-031-40843-4_45

is concerned with how these systems, and the software running on them, can be better engineered to provide increased accuracy and decreased time-to-solution.

Historically, the primary metric for assessing the performance of an HPC system has been the number of floating-point operations that can be completed each second (FLOP/s). By this measure, the Exascale-barrier was broken by the Frontier system in June 2022, using nearly 38,000 GPU accelerators to achieve 1.1 ExaFLOP/s[1].

Like many of the recent systems to achieve the #1 ranking, Frontier is a heterogeneous system, with each compute node comprising of both CPUs and GPU accelerators; in the case of Frontier, one 64-core AMD "Trento" CPU, and four AMD Instinct MI250X GPUs. Extracting the maximum level of performance from such systems requires that data is efficiently moved between the host CPU and connected accelerator devices, that algorithms are effectively parallelised and that computation is appropriately distributed across the available hardware. Achieving this is no mean feat, and in some cases might require the use of a vendor-specific parallel programming model.

In order to avoid issues of vendor lock-in and increase developer productivity, a number of language-like tools and frameworks have been developed that are capable of providing the programming semantics that allow us to target heterogeneous architectures from a single codebase. Some of the most commonly used ones are OpenMP [14], OpenACC [13], OpenCL [21], Kokkos [6] and RAJA [1].

The SYCL parallel programming model was developed by the Khronos group in 2014, as another such tool to assist heterogeneous programming [22]. One of the key design goals of SYCL is portability. However, there have been discussions about how performant it is across platforms [4,8]. A 2021 study by Lin et al. addressed this and some other concerns by evaluating historical performance of three major SYCL implementations across a range of platforms [12]. Their study shows the increasing maturity of the compilers, but highlights remaining potential for further improvements.

In this paper, we further evaluate the performance portability of the Open SYCL, DPC++, and ComputeCpp compilers with a focus on mini-applications of interest to the plasma physics community. Our evaluation is motivated by Project NEPTUNE (NEutrals & Plasma TUrbulance Numerics for the Exascale), a UK project to develop a new simulation code to aid in the design of a future nuclear fusion power plant. Specifically, we make the following contributions:

- We evaluate SYCL against OpenMP and CUDA on a simple finite difference heat diffusion code. This serves as a baseline of performance and portability we can expect from SYCL and its implementing compilers;
- We then evaluate SYCL against MPI, OpenMP, Kokkos, CUDA and HIP on a mini-application implementing a finite element method. This evaluation is based on a simple conversion to SYCL and therefore this provides us with an indication of how much optimisation might be required for SYCL to provide performance that is competitive with other approaches;

[1] https://www.top500.org/lists/top500/2022/06/.

– Finally, we analyse the performance portability of these two mini-applications using visualisations developed by Sewall et al. [20], showing that for simple codes, SYCL can provide equivalent performance to OpenMP with minimal developer effort, but that for more complex cases, a basic code conversion is not sufficient and additional developer effort is required to bridge the gap.

The remainder of this paper is structured as follows: Section 2 provides an overview of the background and related work; Sect. 3 outlines the methodology of our study; Sect. 4 provides the results of our study; finally, Sect. 5 concludes this paper.

2 Background and Related Work

Since the introduction of IBM Roadrunner in 2008 there has been a shift towards heterogeneous architectures within HPC. However, programming systems with multiple architectures can be challenging, and often relies on vendor-led programming models specifically developed for each architecture (e.g. CUDA on NVIDIA, HIP/ROCm on AMD). Adopting these programming models for large HPC applications can lead to vendor lock-in. To combat this, there are a number of programming models that have been developed that are able to target multiple host and accelerator architectures from a single codebase.

The typically stated goal of these programming models is to achieve "the three Ps", *performance*, *portability*, and *productivity* [18]. Notable examples are the compiler directive-based approaches OpenMP [3] and OpenACC [13], the C++ template-based approaches Kokkos [6] and RAJA [1], and language extensions such as OpenCL [21]. Many of these have been the target of studies looking at *performance portability* across heterogeneous platforms [5,7,9,10,15,17,23].

Another approach that is beginning to see widespread adoption in HPC is the SYCL parallel programming model [22]. SYCL is a high-level, single-source programming model based on ISO C++17. It was introduced by the Khronos group in 2014, and takes inspiration from, though is independent of, OpenCL. In particular, SYCL sits at a higher level of abstraction to OpenCL, removing much of the "boiler-plate" code that was previously required.

Since its inception, multiple SYCL compilers have been developed, each implementing different subsets of the standard, and targeting different architectures or execution approaches. The programming model has been the subject of a number of recent studies examining its performance portability and the maturity of its implementing compilers [2,5,8,12,19]. In this paper, we build on these previous studies, with a focus on three mainstream SYCL compilers and algorithms of interest to the plasma physics domain.

Open SYCL (previously known as hipSYCL) is an open-source library or SYCL compiler developed at the University of Heidelberg, by Aksel et al.[2]. It is based on the LLVM compiler framework, and one of its defining features is that it

[2] https://opensycl.github.io.

is not built on OpenCL. Instead, Open SYCL uses other low-level backends to target different platforms. Open SYCL currently supports an OpenMP backend for CPUs, CUDA and HIP backends for NVIDIA and AMD GPUs, and an experimental Level-Zero backend to support Intel's Level-Zero hardware.

DPC++ is a C++ and SYCL compiler developed by Intel, that forms part of their OneAPI project. They provide two versions of their compiler, one a pre-compiled proprietary implementation[3] and one an open-source fork of the LLVM compiler framework[4]. The compiler can target host CPUs directly, or through an OpenCL runtime, and can target GPUs through CUDA, HIP and Level-Zero.

ComputeCpp was the first fully compliant SYCL 1.2.1 implementation, developed by Codeplay[5]. The compiler is built on the open-source Clang 6.0 compiler, but is distributed as a proprietary compiler with no open-source implementation available. ComputeCpp relies on an OpenCL driver for compilation of kernels and therefore has limited platform support. With the announcement in June 2022 that Intel has acquired Codeplay Software, it is likely that future development effort will instead be focused on Intel's DPC++ compiler.

In addition to these three mainstream implementations there are other projects, like triSYCL and neoSYCL, that are not included in this study.

3 Methodology

This paper seeks to answer three questions regarding the SYCL programming model and its implementing compilers:

1. How does SYCL's performance compare to other parallelising frameworks?
2. How does each SYCL compiler perform relative to other implementations?
3. How much portability is offered by SYCL and by each SYCL compiler?

To address these questions, we evaluate two applications across eleven platforms.

3.1 Benchmarks

Evaluating new parallel programming frameworks is difficult on production-grade applications that often consist of tens of thousands, or hundreds of thousands, of lines of code. Instead, mini-applications are typically used to rapidly investigate performance, portability and productivity, prior to extensive porting efforts. These mini-applications usually implement key kernels or algorithms that are found in production applications, but in only a few thousand lines of code. In this paper we focus our effort on two applications that implement computational methods typically used in the simulation of fluids and plasma.

[3] https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html.
[4] https://github.com/intel/llvm.
[5] https://developer.codeplay.com/products/computecpp/ce/home/.

Heat is a simple mini-application that solves a heat diffusion equation using a finite-differencing scheme, with a 5-point stencil [24]. It was developed at the University of Bristol as part of an OpenMP tutorial course, and therefore consists of only a few hundred lines of code. This limits the optimisation space and gives us a good baseline for the potential performance of any particular parallel programming model. The mini-application operates on a two-dimensional structured grid, and in our study we use a fixed problem size of 10000^2. For our study, implementations are available in OpenMP, CUDA and SYCL.

miniFE also solves a heat diffusion equation, but does so on an unstructured brick-shaped domain, using a finite-element method [11]. miniFE is significantly more complex than Heat, and the SYCL variant used in this paper is a conversion from an OpenMP 4.5 implementation. The relative performance of this simplistic conversion to SYCL will provide us with insight into the optimisation effort that might be required to port an application to SYCL in a performance portable manner. In this paper, we use a 256^3 problem size, and focus our efforts only on the conjugate gradient kernel (since this kernel dominates the performance). We present results for the MPI, OpenMP (with and without target directives), CUDA, HIP, Kokkos and SYCL implementations.

3.2 Evaluating Performance and Portability

The focus of this paper is in evaluating the maturity of the SYCL programming model and its implementing compilers in terms of both the performance of SYCL applications when compared to alternative programming models, and also the performance portability of SYCL applications.

To evaluate comparative performance, we use runtime as the figure of merit; to evaluate the performance portability, we apply the Pennycook metric [18].

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

where H is the set of platforms, a is the application of interest, p is the chosen problem, and $e_i(a,p)$ is the efficiency of an application a solving problem p on platform i (i.e. the ratio of the achieved performance against the best known implementation on the given platform). In this paper we focus on *application efficiency*, where $\Phi(a,p,H)$ represents the *harmonic mean* of the observed efficiency of an application across the selected set of platforms.

However, rather than present the raw performance portability, we instead use the cascade plot visualisations outlined by Sewall et al. [20], and generated using Intel's P3 Analysis Library [16].

3.3 Evaluation Platforms

The results in this paper have been collected using Isambard, at the University of Bristol, and the Intel DevCloud. The CPU platforms are detailed in Table 1, while the GPU platforms are detailed in Table 2. Our evaluation includes the

Intel HD Graphics P630 GPU, which is a mid-range integrated GPU provided on some Intel Xeon Coffee Lake and Kaby Lake CPUs. It is included in our evaluation to demonstrate the portability to Intel *Xe*-HPC GPUs, but we do not expect its performance to be competitive with discrete GPUs.

Table 1. Summary of CPUs used for the platform setup

Name	Clock	Cores (Threads)	Sockets	Memory (GB/s)	Vector type
CLX	2.10 GHz	20 (40)	2	131 (DDR4, 6 ch)	SSE, AVX512
(Intel Xeon Gold 6230, Cascade Lake)					
Rome	2.25 GHz	64 (128)	2	205 (DDR4, 8 ch)	AVX2
(AMD EPYC 7742, Zen 2)					
Milan	2.0 GHz	64 (128)	2	205 (DDR4, 8 ch)	AVX2
(AMD EPYC 7713, Zen 3)					
KNL	1.30 GHz	64 (256)	1	102 (DDR4, 6 ch)	AVX512
(Intel Xeon Phi 7210, Knights Landing)					
ThunderX2	2.10 GHz	32 (128)	2	252 (DDR4, 8 ch)	128 NEON
(Cavium CN9980, ARMv8.1)					
A64FX	1.80 GHz	48 (48)	1	1024 (HBM2)	128-512 SVE
(Fujitsu A64FX, ARMv8.2-A)					

Table 2. Summary of GPUs used for the platform setup

Name	Cores	Memory (GB)	Bandwidth (GB/s)
P100	3840	16	732
(NVIDIA P100 Pascal, CUDA Capability 6.0)			
V100	5120	16	900
(NVIDIA V100 Volta, CUDA Capability 7.0)			
A100	6912	40	1555
(NVIDIA A100 Ampere, CUDA Capability 8.0)			
MI100	120	32	1200
(AMD Instinct MI100, CDNA 1.0)			
HD P630	24	(System-shared) 64	(System-shared) 41.6
(Intel HD Graphics P630, Gen 9.5)			

For each of our evaluations on Isambard, we use version 11.0 of the Clang/LLVM compiler environment. We use a custom-build of the compiler infrastructure, to ensure all required features are available (e.g. OpenMP target offload directives, CUDA, HIP). All of our results are collected with `-O3` and other performance relevant compiler flags. We use OpenMPI version 4.1, except on the ThunderX2 platform, where we use version 3.1. We use version 11.2 of the

CUDA Toolkit, specifying the correct architecture each time. For Kokkos, we use the OpenMP backend for CPU platforms and the CUDA and HIP backends for GPU platforms. The results presented in this paper are the best runtime achieved on each platform, regardless of maximum parallelism achievable, using the best discovered combination of runtime parameters.

For Open SYCL, we build version 0.9.4 of the compiler from source, enabling it to target CPUs and GPUs through OpenMP and CUDA/HIP, respectively.

We use Intel's proprietary DPC++ compiler for the CLX, KNL and HD P630 platforms, and we build DPC++ version 16.0 from source for the AMD and NVIDIA platforms. For ThunderX2 and A64FX, we were able to compile benchmarks using DPC++ but we encountered linking errors that we were unable to resolve and so we omit results from these platforms.

We use version 2.10.0 of the ComputeCpp compiler, which is distributed as a pre-built executable. It is only compliant up to the SYCL 1.2.1 standard, and therefore is dependent on an OpenCL driver for each architecture; because of this, our platform set is limited to CLX, KNL, Rome and Milan CPUs.

4 Results and Analysis

We begin our investigation with the "Heat" mini-application. Since this application is implemented in only a few hundred lines of code, it serves as a good starting point to show the potential performance and portability of the SYCL programming model. We use the OpenMP, CUDA and SYCL implementations present in the HeCBench benchmark repository[6].

We then analyse the performance of miniFE. This application implements a finite element method on an unstructured grid, using 8-point hex elements. The application is implemented in approximately 5000 lines of code, and the SYCL port is based on simplistic conversion from the OpenMP 4.5 implementation of miniFE.

For each application we first present the raw runtime data, and we then analyse the performance portability using visualisations from Sewall et al. [20].

4.1 Heat

Figure 1 depicts the runtime for Heat on eight of the platforms surveyed. This simple evaluation reveals valuable information on platform coverage for SYCL. Crucially, there is at least one SYCL compiler that is able to target each architecture, and SYCL appears to provide performance comparable to OpenMP 4.5. The two most striking features of the data are perhaps the superior performance of the two NVIDIA GPU platforms and the relatively poor performance of the two ARM-based systems (ThunderX2 and A64FX). The V100 is approximately 10× faster than the fastest CPU execution observed, and importantly that performance improvement is seen in both CUDA and Open SYCL/DPC++. The runtimes observed on both ARM platforms are much worse than on the Cascade Lake and Rome CPU systems, likely due to using a custom-build of LLVM

[6] https://github.com/zjin-lcf/HeCBench.

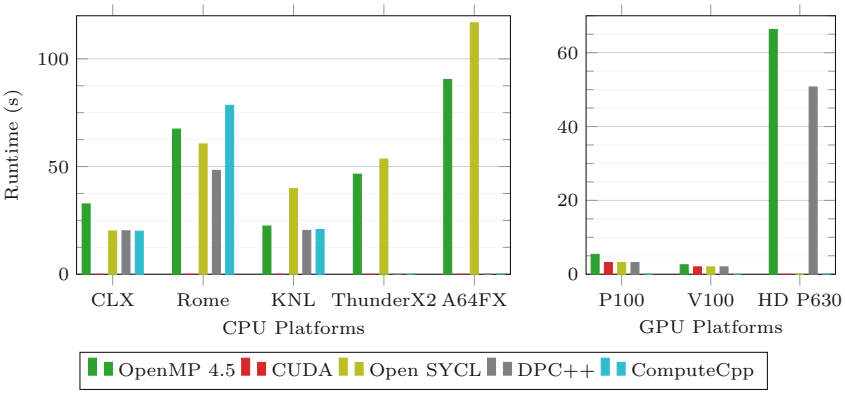
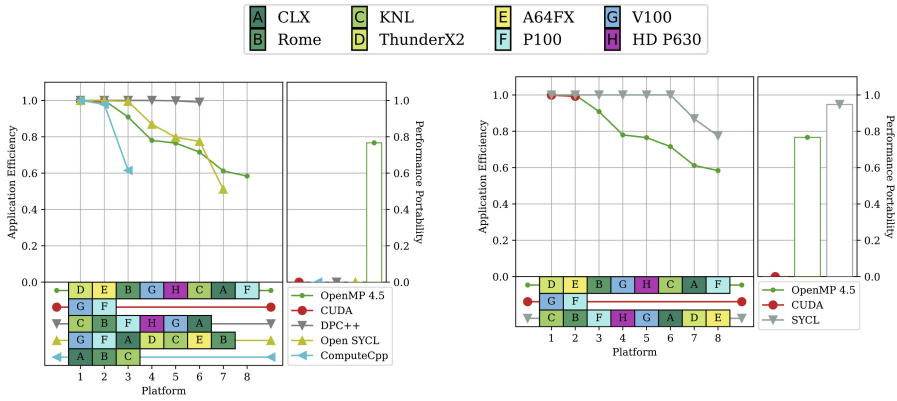


Fig. 1. Raw runtime data for Heat on five CPU and three GPU platforms.



(a) Complete Dataset (b) SYCL Compressed

Fig. 2. A cascade plot for the Heat application with (a) the complete dataset, and (b) the SYCL data compressed to a single value.

rather than the vendor supplied compiler (which did not support target offload semantics or SYCL).

Figure 2(a) shows how the performance efficiency changes for each programming model as new platforms are added to the evaluation set (in order of decreasing efficiency). For six of the eight platforms evaluated, DPC++ achieves almost perfect efficiency. Both Open SYCL and OpenMP 4.5 follow a similar trajectory, with Open SYCL maintaining a marginally higher efficiency up to the addition of the AMD Rome system. That SYCL is able to outperform OpenMP 4.5 (in particular on GPU platforms) can perhaps be explained by the richer semantics available in the programming model, allowing a greater scope for customisation and optimisation.

To compare the programming models in isolation (away from concerns about individual SYCL implementations), Fig. 2(b) shows a cascade plot where the SYCL data point is taken as the minimum runtime achieved by Open SYCL, DPC++, and ComputeCpp. This analysis further shows the potential of the SYCL programming model, where it is consistently able to achieve equivalent or better performance, and is portable to all of the architectures evaluated. Across the eight platforms, SYCL achieves $\Phi \approx 0.95$; OpenMP 4.5 achieves $\Phi \approx 0.77$ and lags SYCL after just three platforms are added to its evaluation set.

4.2 miniFE

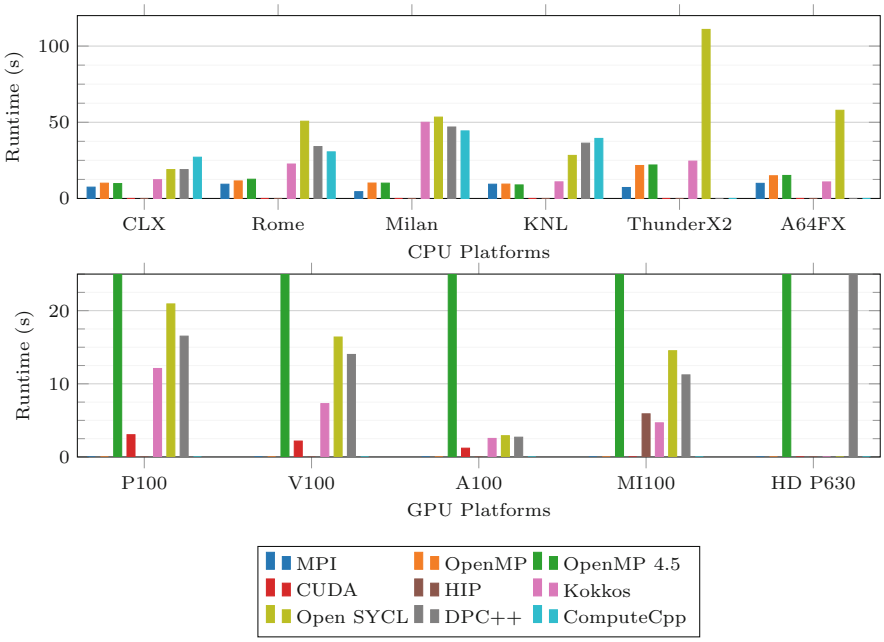
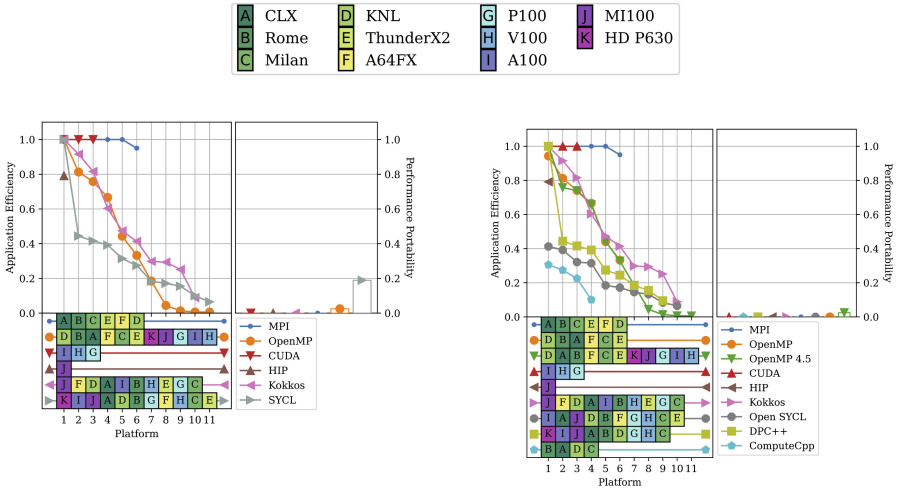


Fig. 3. Raw runtime data for miniFE on six CPU and five GPU platforms.

While Heat provides a good benchmark for the potential of the SYCL programming model and its compilers, its simplicity belies the effort that may be required for larger, more complex applications. The miniFE SYCL port used in this paper is also provided as part of the HeCBench benchmark suite, and is based on the OpenMP 4.5 implementation of miniFE. To provide a more thorough analysis, we compare this against the MPI reference implementation of miniFE, two OpenMP implementations (one with target offload semantics and one without), a CUDA implementation, a HIP implementation, and a Kokkos implementation. Similar to Heat, our evaluation includes the Intel HD Graphics P630 integrated

GPU; while this is not an HPC GPU, it does allow us an insight into the level of support for the Intel *Xe* product line.

Figure 3 shows the runtime achieved by miniFE running on the eleven evaluation platforms. On the six CPU platforms, the reference MPI implementation is the most performant; on the NVIDIA and Intel GPU platforms, the vendor-specific implementations are the fastest (i.e. CUDA on NVIDIA, DPC++ on Intel); Kokkos is the fastest implementation on the AMD Instinct MI100 platform. In contrast to Heat, no portable programming model is as performant as the MPI and CUDA non-portable programming models. However, there is at least one SYCL implementation able to execute on each of the eleven platforms, while OpenMP 4.5 (with target offload directives) is able to target all eleven platforms. As with Heat, there is a performance degradation present on the ARM platforms when using the SYCL programming model, and for miniFE this degradation is exaggerated further (particularly when compared to the reference MPI implementation). Only the DPC++ and OpenMP 4.5 variants have been executed on the Intel GPU, and the runtime achieved by the OpenMP 4.5 implementation (371.3 s) is approximately 5.4× slower than the DPC++ runtime (68.7 s). On each of the GPUs, the OpenMP 4.5 runtime is significantly slower than all other implementations (> 100 s) and so, along with the data for the HD P630, they have been cropped from Fig. 3.



(a) Complete Dataset (b) SYCL Compressed

Fig. 4. A cascade plot for the miniFE application with (a) the complete dataset, and (b) the SYCL data compressed to a single value.

The performance portability of each miniFE implementation is visualised in Fig. 4(a). The two "native" programming models, MPI and CUDA, both follow

the 1.0 efficiency line before abruptly stopping as they reach GPU and non-NVIDIA platforms, respectively. Only the OpenMP 4.5 programming model is able to target all eleven platforms (though the SYCL programming model can achieve this with different compilers for each platform, as shown in Fig. 4(b)). Kokkos and Open SYCL extend to ten of the eleven platforms, respectively, with Kokkos providing better efficiency throughout. DPC++ follows a similar trend to the Open SYCL compiler, but its ability to target the Intel HD Graphics P630 GPU means that its efficiency is generally higher (since it is the most performant implementation on this architecture). For each of the portable approaches, the GPUs and ARM platforms are typically the source of decreased efficiency.

As before, Fig. 4(b) provides the same data but with the SYCL data point taken as the best (minimum) result achieved by either of Open SYCL, DPC++ and ComputeCpp, and the OpenMP data point taken as the best result achieved by OpenMP with or without target offload directives. We can now see that both OpenMP 4.5 and SYCL are able to target every platform in our evaluation set. SYCL's efficiency rapidly drops below 0.5, as soon as the MI100 is added to its evaluation set, but it achieves a $\Phi \approx 0.19$. OpenMP is similarly portable, but the addition of the GPU platforms (platforms 7-11 in the evaluation set) push its efficiency to near zero, ultimately achieving $\Phi \approx 0.03$. The Kokkos variant generally achieves a higher application efficiency, but achieves $\Phi = 0$ as we were unable to collect a data point for the Intel HD Graphics P630 GPU.

Our findings for miniFE run counter to the data seen for Heat in Fig. 2(b). The "performance portability gap" between these two applications is likely not a result of the programming model chosen, but instead an indication that additional effort may be required to optimise the application for heterogeneous architectures. In the case of Heat, the simplicity of the application means that the kernel likely translates reasonably well for each of the target platforms without much manual optimisation effort, regardless of programming semantics. For a significantly more complex application like miniFE, the target architectures must be much more carefully considered in order to optimise memory access patterns and minimise unnecessary data transfers [19].

Figure 5 shows a simplified cascade plot containing only the three portable programming models considered in this study (i.e. OpenMP, Kokkos and SYCL). In this figure, both OpenMP and Kokkos follow the 1.0 efficiency line up to the addition of GPU and CPU platforms, respectively. On CPUs, SYCL is typically less performant than OpenMP; and on GPUs, SYCL is typically less performant than Kokkos, with the exception of the Intel HD Graphics P630 (for which we do not have a Kokkos data point). The platform ordering in Fig. 5 shows that Kokkos and SYCL are typically better at targetting GPU platforms than CPU platforms, while the reverse is true for OpenMP. Overall, when compared only to other portable programming models, SYCL achieves $\Phi \approx 0.36$, while OpenMP only achieves $\Phi \approx 0.06$. Although Kokkos achieves $\Phi = 0$ (due to no result on one of the platforms), if we remove the Intel HD Graphics P630 from our evaluation set, it achieves $\Phi \approx 0.64$.

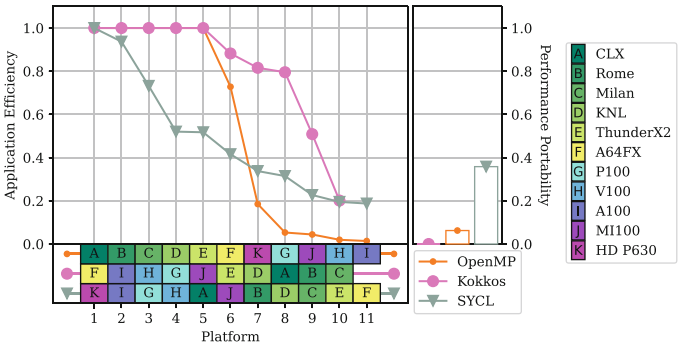


Fig. 5. A cascade plot for the miniFE application considering only the three *portable* programming models (OpenMP, Kokkos and SYCL).

At the most basic level, SYCL provides a similar abstraction to Kokkos, i.e., a parallel-for construct for loop-level parallelism, and a method for moving data between host and device. For this reason we believe that SYCL should be able to provide competitive performance portability to Kokkos. That SYCL achieves approximately half of the performance portability of Kokkos is therefore likely due to limited optimisation efforts at the application-level, and possibly lack of maturity at the compiler-level; this performance gap is likely to close in time.

5 Conclusion

This paper details our initial investigation into the current status of compilers implementing the SYCL programming model in terms of performance and performance portability. Our study is motivated by the growing rate of SYCL adoption within HPC, fuelled by its adoption by Intel for their new *Xe* line of GPUs. Our evaluation is based on three SYCL compilers and two mini-applications that implement methods commonly found in plasma physics simulation applications: one a finite difference method, the other a finite element method.

For a simplistic finite difference heat diffusion mini-application, our results show that SYCL is able to offer performance that is comparable to other performance portable frameworks such as OpenMP 4.5 (with target offload). For well optimised, simplistic kernels, SYCL is able to achieve high performance across a range of different architectures with little developer effort.

On a significantly more complex finite element method application, SYCL leads to a significant loss of efficiency when compared to native approaches such as CUDA and MPI. When compared against other portable programming models, such as OpenMP 4.5 and Kokkos, SYCL fares better, achieving $\Phi \approx 0.36$. Kokkos is arguably the most performance portable approach considered in this study, achieving $\Phi \approx 0.64$ (without the Intel HD Graphics P630). It is likely that a focused optimisation effort would improve the performance of the SYCL variant across every platform and reduce the gap between Kokkos and SYCL.

Overall our results (and those of previous studies [4,8,12,19]) show that the SYCL programming model can provide performance and portability across platforms. However, our experience with the miniFE application shows that this performance does not come "for free" and likely requires careful consideration of compilers and compiler options, and a SYCL-focused optimisation effort. As the language and compiler infrastructure are further developed, the burden on developers should decrease considerably.

5.1 Future Work

The work presented in this paper shows an initial investigation into the SYCL programming model using two mini-applications. The most significant performance issues highlighted concern the SYCL implementation of miniFE used in this paper. Since it is a conversion from an OpenMP 4.5 implementation, it has not been subject to the same optimisation efforts as the other ports evaluated. It would therefore be prudent to re-evaluate the application following a focused optimisation effort. Nonetheless, the work in this paper highlights the probable performance gap between a simplistic conversion and a focused porting effort.

Acknowledgements. Many of the results in this paper were gathered on 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 Intel HD Graphics P630 GPU was provided by Intel through the Intel Developer Cloud.

The ExCALIBUR programme (https://excalibur.ac.uk/) is supported by the UKRI Strategic Priorities Fund. The programme is co-delivered by the Met Office and EPSRC in partnership with the Public Sector Research Establishment, the UK Atomic Energy Authority (UKAEA) and UKRI research councils, including NERC, MRC and STFC.

References

1. Beckingsale, D.A., et al.: RAJA: portable performance for large-scale scientific applications. In: IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), pp. 71–81 (2019)
2. Breyer, M., Van Craen, A., Pflüger, D.: A comparison of SYCL, OpenCL, CUDA, and OpenMP for massively parallel support vector machine classification on multi-vendor hardware. In: International Workshop on OpenCL (IWOCL), pp. 1–12 (2022)
3. Dagum, L., Menon, R.: OpenMP: an industry standard API for shared-memory programming. IEEE Comput. Sci. Eng. **5**(1), 46–55 (1998)
4. Deakin, T., McIntosh-Smith, S.: Evaluating the performance of HPC-style SYCL applications. In: International Workshop on OpenCL (IWOCL). ACM (2020)
5. Deakin, T., et al.: Performance portability across diverse computer architectures. In: IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), pp. 1–13 (2019)
6. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: enabling manycore performance portability through polymorphic memory access patterns. J. Parallel Distrib. Comput. (JPDC) **74**(12), 3202–3216 (2014)

7. Herdman, J.A., et al.: Accelerating hydrocodes with OpenACC, OpenCL and CUDA. In: SC Companion: High Performance Computing, Networking Storage and Analysis, pp. 465–471 (2012)
8. Joo, B., et al.: Performance portability of a Wilson Dslash stencil operator mini-app using Kokkos and SYCL. In: IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), pp. 14–25 (2019)
9. Kirk, R.O., Mudalige, G.R., Reguly, I.Z., Wright, S.A., Martineau, M.J., Jarvis, S.A.: Achieving performance portability for a heat conduction solver mini-application on modern multi-core systems. In: IEEE International Conference on Cluster Computing (CLUSTER), pp. 834–841 (2017)
10. Law, T.R., et al.: Performance portability of an unstructured hydrodynamics mini-application. In: IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), pp. 0–12 (2018)
11. Lin, P.T., Heroux, M.A., Barrett, R.F., Williams, A.B.: Assessing a mini-application as a performance proxy for a finite element method engineering application. Concurrency Comput. Pract. Experience **27**(17), 5374–5389 (2015)
12. Lin, W.C., Deakin, T., McIntosh-Smith, S.: On Measuring the Maturity of SYCL Implementations by Tracking Historical Performance Improvements. In: International Workshop on OpenCL (IWOCL). ACM (2021)
13. OpenACC-Standard.org: The OpenACC Application Program Interface Version 3.3 (2022). https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.3-final.pdf
14. OpenMP Architecture Review Board: OpenMP API Version 4.5 (2015). https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
15. Pennycook, S.J., Jarvis, S.A.: Developing performance-portable molecular dynamics kernels in opencl. In: 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, pp. 386–395 (2012)
16. Pennycook, S.J., Sewall, J., Jacobsen, D., Deakin, T., Zamora, Y., Lee, K.L.K.: Performance, portability and productivity analysis. Library (2023). https://doi.org/10.5281/zenodo.7733678
17. Pennycook, S.J., Hammond, S.D., Wright, S.A., Herdman, J.A., Miller, I., Jarvis, S.A.: An investigation of the performance portability of OpenCL. J. Parallel Distrib. Comput. (JPDC) **73**(11), 1439–1450 (2013)
18. Pennycook, S., Sewall, J., Lee, V.: Implications of a metric for performance portability. Futur. Gener. Comput. Syst. **92**, 947–958 (2019)
19. Reguly, I.Z., Owenson, A.M.B., Powell, A., Jarvis, S.A., Mudalige, G.R.: Under the hood of SYCL – an initial performance analysis with an unstructured-mesh CFD application. In: Chamberlain, B.L., Varbanescu, A.-L., Ltaief, H., Luszczek, P. (eds.) ISC High Performance 2021. LNCS, vol. 12728, pp. 391–410. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78713-4_21
20. Sewall, J., Pennycook, S.J., Jacobsen, D., Deakin, T., McIntosh-Smith, S.: Interpreting and visualizing performance portability metrics. In: IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), pp. 14–24 (2020)
21. Stone, J.E., Gohara, D., Shi, G.: OpenCL: a parallel programming standard for heterogeneous computing systems. Comput. Sci. Eng. **12**(3), 66 (2010)

22. The Khronos SYCL Working Group: SYCL 2020 Specification (2023). https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf
23. Truby, D., Wright, S.A., Kevis, R., Maheswaran, S., Herdman, J.A., Jarvis, S.A.: BookLeaf: an unstructured hydrodynamics mini-application. In: IEEE International Conference on Cluster Computing (CLUSTER), pp. 615–622 (2018)
24. University of Bristol HPC Group: Programming Your GPU with OpenMP: A Hands-On Introduction (2022). https://github.com/UoB-HPC/openmp-tutorial