A Performance Analysis of Modern Parallel Programming Models Using a Compute-Bound Application

Andrei Poenaru(✉), Wei-Chen Lin, and Simon McIntosh-Smith(✉)

Department of Computer Science, University of Bristol, Bristol, UK
{andrei.poenaru,cssnmis}@bristol.ac.uk

Abstract. Performance portability is becoming more-and-more important as next-generation high performance computing systems grow increasingly diverse and heterogeneous. Several new approaches to parallel programming, such as SYCL and Kokkos, have been developed in recent years to tackle this challenge. While several studies have been published evaluating these new programming models, they have tended to focus on memory-bandwidth bound applications. In this paper we analyse the performance of what appear to be the most promising modern parallel programming models, on a diverse range of contemporary high-performance hardware, using a compute-bound molecular docking mini-app.

We present miniBUDE, a mini-app for BUDE, the Bristol University Docking Engine, a real application routinely used for drug discovery. We benchmark miniBUDE on real-world inputs for the full-scale application in order to follow its performance profile closely in the mini-app. We implement the mini-app in different programming models targeting both CPUs and GPUs, including SYCL and Kokkos, two of the more promising and widely used modern parallel programming models. We then present an analysis of the performance of each implementation, which we compare to highly optimised baselines set using established programming models such as OpenMP, OpenCL, and CUDA. Our study includes a wide variety of modern hardware platforms covering CPUs based on ×86 and Arm architectures, as well as GPUs.

We found that, with the emerging parallel programming models, we could achieve performance comparable to that of the established models, and that a higher-level framework such as SYCL can achieve OpenMP levels of performance while aiding productivity. We identify a set of key challenges and pitfalls to take into account when adopting these emerging programming models, some of which are implementation-specific effects and not fundamental design errors that would prevent further adoption. Finally, we discuss our findings in the wider context of performance-portable compute-bound workloads.

Keywords: Programming models · Performance portability · Performance analysis · Compute-bound benchmark

B. L. Chamberlain et al. (Eds.): ISC High Performance 2021, LNCS 12728, pp. 332–350, 2021.
https://doi.org/10.1007/978-3-030-78713-4_18

1 Introduction

In High-Performance Computing (HPC), the majority of programs utilise established, long-running parallel programming frameworks. OpenMP and MPI are widely adopted [1], generally on top of the C and Fortran programming languages, and other frameworks are sometimes used in more specific situations, such as task-based parallelism libraries, C++-native applications, or programming models that can target GPUs [2].

Recent developments in parallel programming frameworks—be it frameworks developed from scratch, such as Kokkos, or additions and improvements to existing ones, such as tasking and offload support in OpenMP 4.5 and later—have all shared a number of common goals: performant support for a wide range of hardware platforms, interoperability with modern versions of the C++ language, and a focus on programmers' productivity. These goals have organically arisen as a result of the shortcomings in established programming models, and together contribute to the wider endeavour in the field of HPC towards achieving *performance portability* [3].

In order to support both GPU and CPU platforms, high-level programming frameworks manage underlying data structures automatically: the programmer expresses what data needs to be computed on, and the framework arranges it in a format suitable for the target hardware. Because this process is desirable but not always optimal, it is one of the key focuses of previous analyses of high-level parallel programming models [4]. However, the portability and productivity advantages alone brought by these frameworks may be enough to also justify their usage on applications that are not memory-bandwidth-bound, provided that they can deliver performance comparable to established frameworks.

One suitable application for such heterogeneous frameworks is the Bristol University Docking Engine (BUDE), a molecular docking code that is heavily compute-bound [5]. BUDE is routinely used for *in silico* drug discovery, and out of a need to support both CPUs and GPUs, it is comprised of two parallel implementations: an OpenMP version for CPUs and an OpenCL version for GPUs. In this paper, we used a mini-app created from the core computation kernel for BUDE to analyse the performance of emerging parallel programming models compared to that of traditional models.

This paper makes the following contributions:

– Highly optimised implementations of the miniBUDE mini-app in six parallel programming models: OpenMP (for CPU and offload), OpenCL, CUDA, OpenACC, Kokkos, and SYCL;
– An analysis of the performance of each implementation on contemporary high-performance processors from Intel, AMD, Fujitsu, Marvell, and NVIDIA;
– A discussion of the lessons learned from developing the miniBUDE benchmark, and their implications for other compute-bound workloads and the wider HPC field.

2 Background

2.1 High-Performance Molecular Docking

BUDE is an application for *in silico* molecular docking, a computational technique for predicting the structure of a complex formed between two molecules and estimating the strength of their interaction [5]. Docking is computationally challenging because of the many different ways in which two molecules may be arranged together to form a complex (three translational and three rotational degrees of freedom). Indeed, interacting all patches of the surface of one protein molecule with all patches of a second molecule requires on the order of 10^7 trials, each one of which is a computationally expensive operation [6].

The application includes several modes of operation, of which the most commonly used—and the most computationally intensive—is *virtual screening*. In this mode, molecules of drug candidates, known as *ligands*, are generated using a genetic algorithm and are bonded to a *target* protein molecule. BUDE uses a tuned empirical free-energy forcefield to predict the binding energy of the ligand with the target. There are many ways in which this bonding could occur, so a variety of positions and rotations of the ligand relative to the protein are attempted; these are known as *poses*. For each pose, the energy, i. e. the strength, of the bond is evaluated.

2.2 Modern Parallel Programming Models

In the previous decade, low-level programming models that offered the programmer great control over the hardware saw a rise in their usage. Their appeal of low overhead and extensive tuning options made them popular with Graphics Processing Units (GPU) programmers [7], but over the years the HPC community has learned the cost these frameworks incur: they require extensive knowledge of the hardware, and they steer towards over-optimisations for one target, up to the point where a significant fraction of the code needs to be rewritten when moving to a new system [8–10]. The latter observation is particularly relevant in the context of the upcoming exascale systems Frontier, El Capitan, Aurora, and Perlmutter, which together utilise combinations of CPUs from two vendors and GPUs from three vendors [11,12]. It is, thus, not feasible to use platform- or vendor-specific programming models, and a portable approach is needed.

In moving to new programming models, the C++ language has particular appeal: it can achieve the same zero-overhead performance compared to C and Fortran, but it also offers modern features to write more expressive and safer code. Programmers writing parallel C++ hope to outweigh any lost performance with time gained through easier-to-write and easier-to-debug code. This is the core selling point of modern parallel programming frameworks [13].

Two modern, single-source frameworks with a focus on performance, portability, and productivity have emerged: Kokkos [14] and SYCL [15]. Kokkos is a new framework developed natively for C++, while SYCL builds on previous OpenCL toolchains and integrates them with modern C++ code. Both of these frameworks can generate machine code for both CPUs and GPUs without any change to the high-level source code.

These frameworks solve the same problem in different ways. Kokkos is distributed as source code that needs to be integrated into the application's build process. This means that every application using Kokkos needs to build Kokkos itself—a relatively quick process—but it also avoids the pitfalls of system-wide libraries; a C++ compiler is all that is needed to compile a Kokkos application.

In contrast, SYCL applications rely on a SYCL compiler. At the time of writing, there are three major SYCL compilers: Data-Parallel C++ (DPC++) [16], ComputeCpp [17], and hipSYCL [18]. Each implementation can use different backends: DPC++ can use OpenCL, CUDA, or Intel's Level Zero; ComputeCpp relies on OpenCL; and hipSYCL supports OpenMP to target CPUs, CUDA to target NVIDIA GPUs, and ROCm to target AMD GPUs.

2.3 Performance Portability

In recent years, the HPC community has made efforts to understand how to quantify performance portability. Although some formal metrics have been developed and are commonly applied in portability studies [19], the results are not always trivial to interpret correctly [20]. One attempt to solve this challenge relies on carefully designed visualisations [21].

Portability is a common concern for developers—and users—of modern programming models. These make it more feasible to target several kinds of compute devices simultaneously, which has led to a diverse landscape of architectures being investigated in contemporary HPC research. As such, significant attention to portability and programmer productivity is also given in recent studies that evaluate the applicability of novel parallel frameworks [22,23].

3 Evaluation Methodology

3.1 A BUDE Mini-App

We have implemented a mini-app for BUDE virtual-screening runs, with kernels written in a range of widely used parallel programming models. The baseline implementation is written in OpenCL and is virtually identical to the core kernel of the full-scale BUDE application. There is a CUDA port with minimal changes, a CPU OpenMP version that restructures the computation in the OpenCL kernel to make it easier for compilers to vectorise, and similar implementations for GPUs using OpenACC and OpenMP `target` offload. We chose SYCL and Kokkos for implementations in novel programming models because of their relative popularity and compatibility with a wide range of platforms, covering both CPUs and GPU.

The focus of the mini-app is on the core computation, and so most of the plumbing around it, such as flexible I/O and custom file formats, has been removed. Instead of using a genetic algorithm to generate ligands, a procedure which takes negligible time in a full-scale BUDE run, the mini-app uses pre-generated

Table 1. Hardware platforms used for evaluation.

Platform	Abbrev.	Type	Cores	Clock speed	Peak SP performance
Intel Skylake 8176	SKL	CPU	2 × 28	2.1 GHz	5,734 GFLOP/s
Intel Cascade Lake 6230	CXL	CPU	2 × 20	2.1 GHz	4,096 GFLOP/s
AMD Rome 7742	Rome	CPU	2 × 64	2.25 GHz	9,216 GFLOP/s
Marvell ThunderX2	TX2	CPU	2 × 32	2.5 GHz	2,560 GFLOP/s
Fujitsu A64FX	A64FX	CPU	48	1.8 GHz	5,530 GFLOP/s
NVIDIA V100	—	GPU	80	1.13 GHz	15,700 GFLOP/s
AMD Radeon VII	—	GPU	60	1.4 GHz	13,800 GFLOP/s
Intel Iris Pro 580	—	GPU	72	0.95 GHz	1,094 GFLOP/s

molecules obtained from the full BUDE application. The main advantages of this approach are that it simplifies the mini-app logic, it makes the results easier to reproduce, and it allows for a built-in validation procedure by comparing mini-app output against reference output from the full application. Thus, the mini-app simply reads in a protein and a ligand, computes the bonding energies over a user-defined number of poses, and compares them against a reference set. To enable custom-length benchmarks, the mini-app can run several iterations of the same ligand–protein combination instead of requiring a new ligand each time. The result is a benchmark consisting of a few hundred lines of code for each implementation, which is easy to understand, feasible to profile and analyse, has built-in validation, requires no external libraries, and with a performance profile that maintains the same important characteristics of the full application.

3.2 Performance Analysis

We analysed the performance of our mini-app on a range of modern HPC platforms; Table 1 shows the systems used and their specifications. Where several compilers could be used for the same programming model, we tested all the options and picked the best-performing one in each case. We used aggressive compiler optimisation flags to the level of `-march=native -Ofast`. Table 2 lists the compilers used, the parallel programming frameworks supported by each, and any platform targetting restrictions they have.

We collected performance data using industry-standard tools. On CPU platforms, we accessed hardware counters through the built-in Linux `perf` tool, and collected application-level profiles with Cray Perftools; on GPUs, we used the NVIDIA CUDA profiler and the OpenCL Intercept Layer. We obtained peak memory bandwidth figures using BabelStream [24] and the University of Bristol's HPC Group's cache-bandwidth measurement tool [25].

We used two input decks to benchmark the application: a small input set, consisting of 26 ligands, and a large set, with 2672 ligands. The former takes around 0.5 s to run on a contemporary dual-socket-CPU HPC system, while the latter takes around 1.5 min. In both cases, we ran 8 iterations of the algorithm and we computed 2^{16} poses per iteration. We utilised all the available cores on

Table 2. Compilers used and their programming model and target platform support.

Compiler	CPUs	GPUs	Frameworks
AOCC 2.3	X		m k s
AOMP 11.0		M	m
Arm Compiler 21.0	R		m k s
ComputeCpp 2.1.1	X	I	m k s
Cray Compiler 10.0	R X	N	a[1] m k s
Fujitsu Compiler 4.3	R		m k s
GCC 10.3	R X	M N	a l m k s
Intel ICX 2019	X		m k[2] s
Intel DPC++ 2021.1	X	N	m k s
LLVM 11.0	R X	N	m k s
NVCC 10.2		N	c
PGI 19.10		N	a

CPUs: A**R**M, **X**86; GPUs: A**M**D, **N**VIDIA, **I**NTEL
Frameworks: **c**uda, open**a**cc, open**cl**, open**mp**, **k**okkos, **s**ycl

[1] Version 9.0 only; [2] With the experimental `INTEL_GEN` backend.

each platform, using a single thread per core on all the CPU platforms; where available, using more than a single thread per core did not improve performance. A warm-up iteration was always run before the timers were started.

There was very little run time variability in miniBUDE. Even on the small input set, when individual iterations take less than 100 ms, variance was only fractions of a percent. This was true for both CPU and GPU implementations, as long as care is taken to bind threads correctly, especially when two interacting systems are present, e. g. OpenMP's `OMP_PROC_BIND` and Cray's `aprun`. There was one exception to this observation, which we addressed in Sect. 5.

4 Results and Performance Analysis

4.1 CPUs

OpenMP. The OpenMP implementation was written in plain C, without any higher-level framework, and was optimised for CPU platforms. We expected thisversion to incur the least overhead and thus perform fastest on the CPU. As we will see in this section, OpenMP did offer the best performance on CPUs in most cases, but higher-level implementations were sometimes able to match it.

Parallelism is exposed through OpenMP at two levels: poses are distributed between threads, and the calculations for each pose take advantage of each thread's SIMD lanes. Thread-level parallelism is achieved by dividing the poses into groups and then distributing the groups over threads; this creates an execution model similar to OpenCL workgroups, where each thread iterates over its assigned poses. The size of the group of poses is specified as a compile-time parameter.

We found that the group size had significant impact on the performance of the OpenMP implementation of miniBUDE. On each platform, this parameter should be at least as large as the native vector length, such that all the SIMD lanes are utilised for computation, but we found that most platforms achieved the best performance at group sizes several times larger than the native vector length. This happened because compilers were able to fully unroll the inner thread loops. As such, the group size is not only a vectorisation factor, but also an unroll factor, and higher values allowed platforms to fully exploit their out-of-order resources by interleaving several (unrolled) loop iterations. Furthermore, a small part of the arithmetic can be factored out and computed only once per work group, resulting in additional computation time savings. Figure 1 shows the impact of the group size parameter on performance for each platform.

The other defining factor for the performance of the OpenMP implementation is vectorisation. In order to maintain portability, no architecture-specific intrinsics are used; we rely on compiler auto-vectorisation. The code is structured such that vectorisation is required at the innermost level, which allowed all compilers tested to vectorise the main computation. The Cray and Intel compilers successfully vectorised *all* the loops in the code, while GCC and the Arm compiler did not understand the structure of one `do-while` loop and so did not vectorise it. This last loop, however, is not critical for performance.

The compilers further differed in their instruction choice and scheduling. On the Intel platforms, only the Cray compiler generated 512-bit vector code by default. Because this code is compute-heavy, long vectors greatly benefit performance, and forcing the Intel and GNU compilers to generate 512-bit operations—instead of their 256-bit default—significantly reduced the run time. In addition, the Intel and Cray compilers automatically interleaved the loop bodies, thus overlapping arithmetic and memory operations from different iterations.

On the other hand, GCC only unrolled the loops, without interleaving, and so instructions for each iteration were scheduled sequentially. This lowered the achieved performance on the platforms with fewer out-of-order resources, such as the A64FX, which performed slower than a ThunderX2, even though the former has 4× the vector width of the latter. The Fujitsu compiler, which has a good cost model of the A64FX and performs aggressive software pipelining and division optimisation, generates the fastest code in this case. Figure 2 shows the performance of the OpenMP implementation on the CPU platforms across the compilers tested.

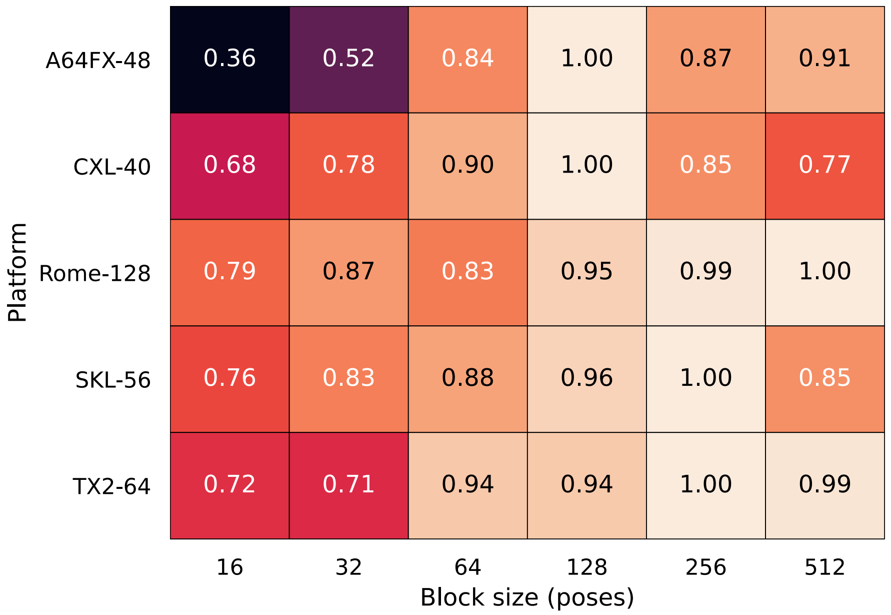

Fig. 1. Performance of the OpenMP implementation at different group sizes, normalised to the best result on each platform. Platforms are labelled using the abbreviations in Table 1 and the number of cores. Higher numbers, shown here in brighter colours, correspond to higher performance.

Figure 3 shows a roofline chart of the Cascade Lake platform. The OpenMP implementation of miniBUDE has an operational intensity of 0.3 and achieved a performance of 2301 GFLOP/s, which represents 56.2% of the platform's peak. The application sits directly below the arithmetic roof and above the memory bandwidth bound, confirming the code is compute-bound. For the purposes of the roofline model, FLOPs and memory traffic (assuming caching as per the cache-aware roofline model) were manually counted in the application's source code and corroborated using hardware counters.

Kokkos. The Kokkos implementation is a direct port of the OpenMP version, with parallelism expressed via the idiomatic `Kokkos::parallel_for` function. We retained the group size parameter to investigate the effects of unrolling, and we found that it had the same effect as in the case of OpenMP, and the same values were optimal on each platform. Like the OpenMP version, Kokkos does not offer built-in types for vectors and functions to use with them. From a productivity standpoint, it may be preferable for the framework and runtime to provide optimised versions of common math types and functions, so that compilers can better optimise code with the correct constraints. This is especially important for parallel frameworks that can target different backends—as Kokkos

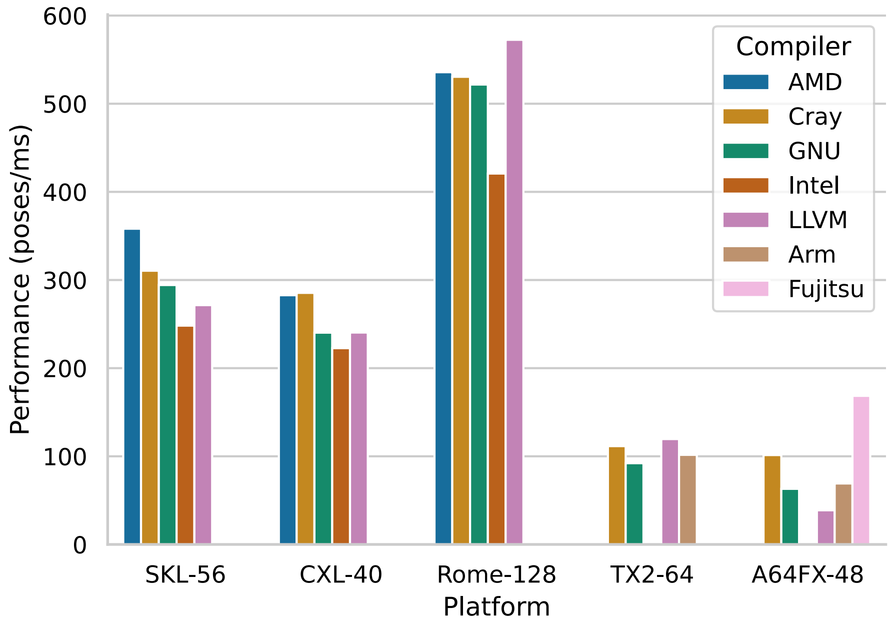

Fig. 2. Performance of the OpenMP implementation across systems and compilers. Higher numbers represent faster execution.

does—where each platform can have its own unique requirements, e. g. alignment on specific boundaries.

Kokkos was able to provide complete platform support in our study by virtue of being able to utilise many different programming frameworks as backends. Because a C++ compiler is the only requirement to build a Kokkos application, and because Kokkos itself is built as part of the same process, we can compare the relative performance on the platforms studied when using different compilers. Figure 4 shows a performance comparison on each CPU platform, where Kokkos uses the OpenMP backend, normalised to the fastest result. The results shows a strong correlation compared to the OpenMP implementation results described in Sect. 4.1, which shows Kokkos is using OpenMP efficiently on all the architectures.

SYCL. The SYCL implementation was written in idiomatic SYCL 1.2.1. The kernel is a direct port of the OpenCL version, utilising workgroup-based parallelism (`sycl::nd_range`) with few changes required. We retained the existing GPU-friendly optimisations from the OpenCL kernel where data is first copied to local memory via OpenCL's `async_work_group_copy`. Due to SYCL's roots in OpenCL, the APIs used for implementing these operations are identical both in name and semantics. We were even able to retain the use of 3d vector types which corresponds to the `cl_vec3` in OpenCL.

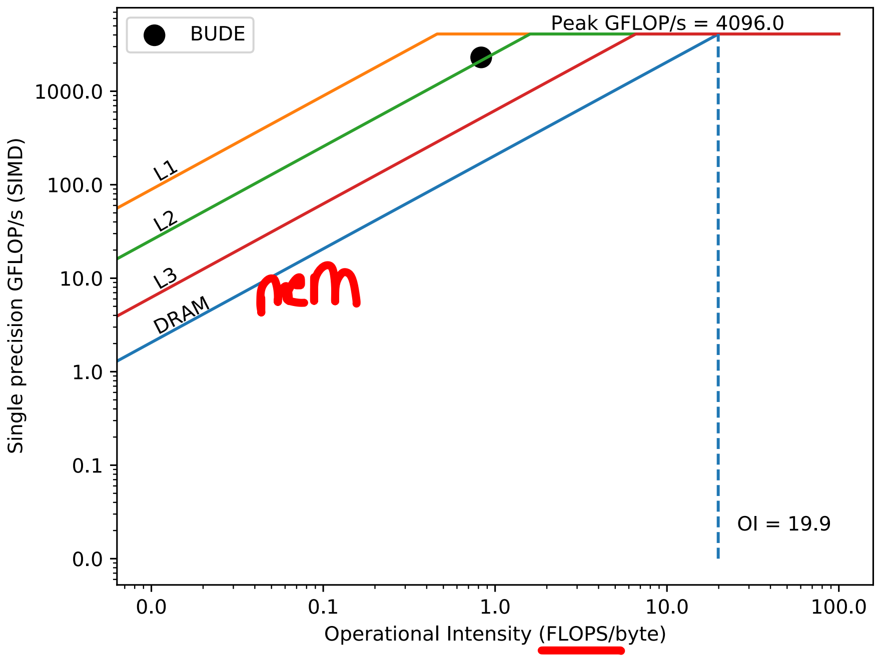

Fig. 3. Cache-aware roofline for the Cascade Lake platform showing the achieved performance for miniBUDE.

For comparison, we also implemented a separate kernel that is closer to the OpenMP implementation, where parallelism is achieved with flat `parallel_for` calls based on `sycl::range`. Although in theory plain `range` may be easier to map onto the hardware than `nd_range`, we found the performance difference between the two implementations to be negligible (below 2%).

Figure 5 shows the performance of all SYCL implementations on the platforms tested where at least two implementations were supported. On each platform, performance is normalised to the fastest implementation. For hipSYCL on the x86-based platforms, we tried all the compilers available and picked the one that produced the fastest binary, which was Cray on both Cascade Lake and Rome. The Skylake platform is missing from these results because an incorrect interaction between the Intel OpenCL driver installed on the system and the Cray `aprun` launcher resulted in all threads being pinned to a single core, effectively invalidating the results obtained with the two implementations that reply on OpenCL, OneAPI and ComputeCpp. On the V100 and the Radeon VII, hipSYCL is the only usable SYCL implementation.

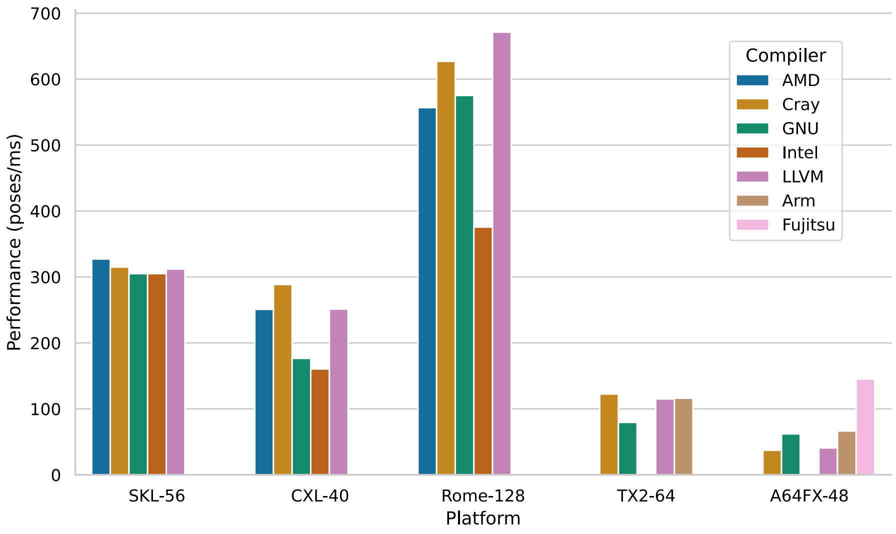

Fig. 4. Performance of Kokkos with the OpenMP backend on the test platforms. Higher numbers represent faster execution.

4.2 GPUs

Low-Level: OpenCL and CUDA. The OpenCL implementation is a close representation of the main kernel in the full-scale application, with the modifications presented in Sect. 3.1; the CUDA implementation is a direct port of the OpenCL version. The two versions performed similarly on the NVIDIA V100 GPU: the CUDA implementation was 18% faster than the OpenCL code, on both the small and the large input decks. The performance difference was evenly spread across the execution of the program: all the kernels were slightly slower when using OpenCL. Memory transfers are not timed for the purposes of the benchmark, and they take negligible time (<1% of the total run time). All of the benchmarks were run on CUDA Toolkit 10.2 running on NVIDIA driver version 440.64, so the difference likely came from more optimisation on the CUDA side of the NVIDIA library.

Both versions also ran on the AMD Radeon VII, converting the CUDA version through HIP, but OpenCL was 1.6× faster on this platform. Since the kernel code for both implementations was very similar, we attributed the performance difference to inefficiencies in AMD's HIP compiler. CUDA and HIP cannot be used on the Intel GPU.

Directives-Based: OpenMP Offload and OpenACC. The directive-based GPU implementations run the same kernel code in the OpenCL implementation, but expressed in the same C file as the host application and without any of the explicit OpenCL platform set-up and clean-up code. This is a significant advantage for productivity: given host code, only three `pragma` directives are

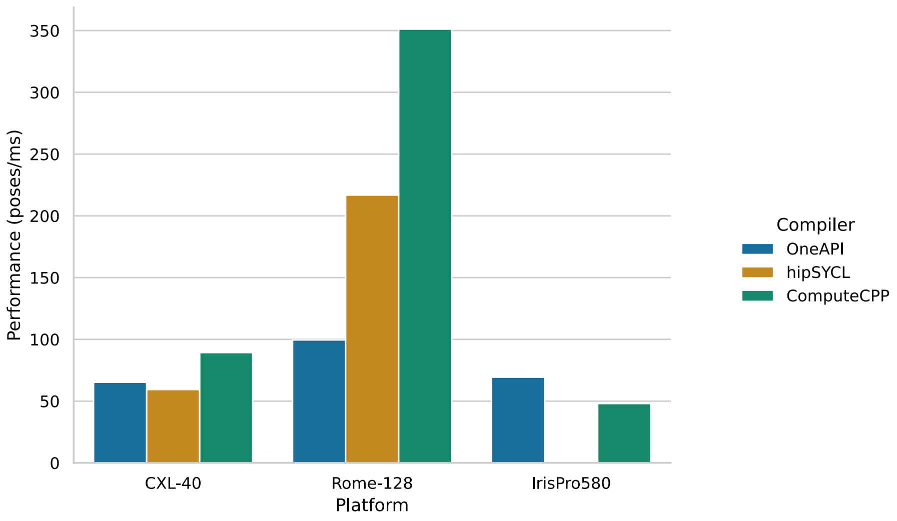

Fig. 5. Relative performance of SYCL implementations, on the platforms where more than one was available. Higher numbers represent faster execution.

used to transfer the data to the GPU and generate GPU kernel code. The main difference from the OpenCL version is that the global and local sizes aren't set by the programmer, but are controlled by the runtime. To control the amount of computation per workgroup, the directives-based implementations include a macro to control loop unrolling, similarly to the CPU OpenMP implementation.

The implementations achieved virtually identical performance on the V100. This was expected, because the same CUDA-based backend is used to generate code for both frameworks. Compiler support, however, differs between the two: the OpenMP code can use the latest versions of the Cray and GNU compilers, but the OpenACC version could only be compiled with an older version of the Cray compiler (9.0). The GNU and PGI compilers produced non-working code for OpenACC, and newer versions of CCE have dropped support for it.

On the V100, the directives-based approach showed about 0.4× the performance of the optimised CUDA code. This is the combined result of inefficiencies we identified in two places: 1) high register usage in the kernels generated by the compiler limits the maximum achievable GPU occupancy; 2) lower performance of library functions. This difference is higher than what has been observed in previous studies [4], and is likely exacerbated by the heavily compute-bound nature of miniBUDE.

On the Radeon, OpenACC can be compiled with GNU, but the resulting code was two orders of magnitude slower than OpenMP, which in turn only reached 0.3× the performance of the fastest model, OpenCL. The low-level nature of OpenCL allowed the code to map very well onto the target hardware, a performance which the GNU offload maths libraries could not match.

On the Intel GPU, OpenMP `target` reached only $0.2-0.3\times$ the performance of the fastest model, which in this case was SYCL. Although SYCL uses the same drivers as OpenCL on this platform, in this case the OneAPI compiler was better able to extract performance from the hardware when starting from higher-level, more expressive programming model. The OpenCL implementation was developed with HPC GPUs in mind, and while with code changes specific to the Intel GPU architecture it should be possible to reach the same performance with a low-level OpenCL implementation, this result highlights the productivity benefit of the higher-level programming model when targetting several platforms simultaneously, as long as the model is well-supported on all the targets.

High-Level: Kokkos and SYCL. Kokkos and SYCL both run on all the GPUs studied, but only one implementation, hipSYCL, runs on AMD and NVIDIA. The code run on the GPU platforms was unchanged from the version run on CPUs, not even to define different parallelism, as was the case when moving from CPU OpenMP to OpenMP `target` offload.

Figure 6 shows the results on the GPU platforms for all programming models studied. The three GPUs each target different segments: the V100 is a top-end HPC GPU, the Radeon VII is a high-end consumer GPU, and the Iris Pro is a mobile chip designed for a very constrained power and transistor budget. A direct performance comparison between such different platforms is not useful; instead, we present programming model performance normalised to the fastest result on each platform. In absolute figures, the best result on the V100 (CUDA) was twice as fast than the best on the Radeon VII (OpenCL) and $14\times$ faster than the best Iris Pro 580 result (OneAPI SYCL).

5 Towards Portable High-Performance Code

Section 4 has analysed the performance of the miniBUDE implementations on the platforms studied, but the implications of these results are further-reaching. Figure 7 aggregates the performance results over all the platforms and programming models and highlights that no programming model can currently achieve optimal performance on *all* platforms.

This effect is more pronounced on GPUs: each of the three platforms studied achieved the highest performance using a *different* programming model, and they relied on parameter tuning to do so. This immediately imposes a penalty when moving to a new platform, at which point at the very least tuning needs to be redone. In the worse case, low-level frameworks can trap users into code so specific to one platform that a major rewrite is needed when changing targets. However, OpenCL was the fastest model on the Radeon VII and a close second on the other two GPUs studied, suggesting that is may still be the best choice for good performance portability.

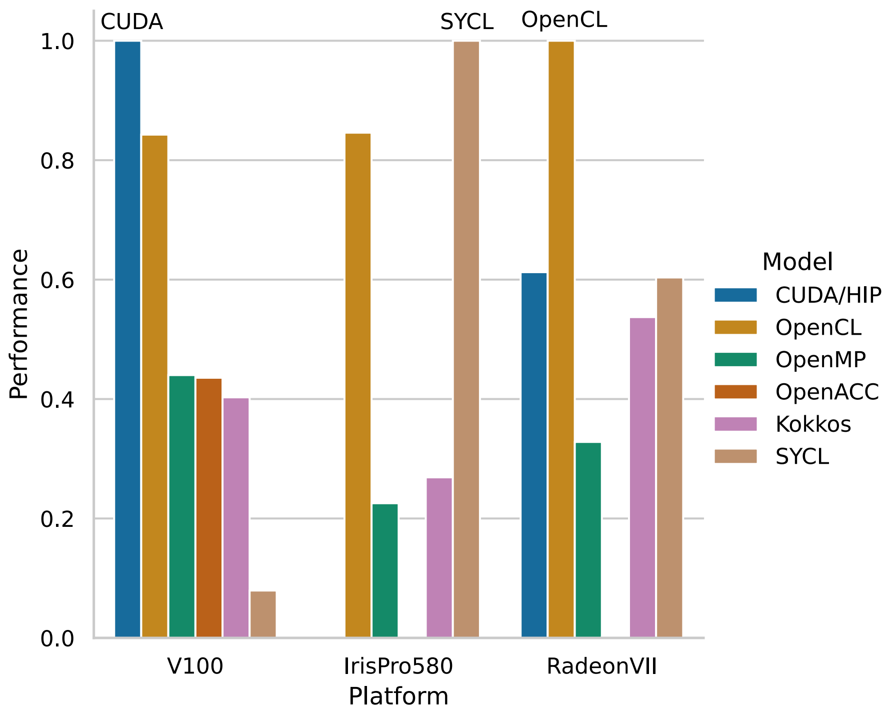

Fig. 6. Performance of the GPU implementations, normalized to the fastest result on each platform. The fastest model on each platform is labelled explicitly.

Higher-level programming models avoid this issue of over-specialisation of the code, instead relying on being able to translate the high-level code to efficient machine code as part of the framework. Kokkos is a good example of this: on the CPU platforms it achieves performance close to that of OpenMP, and both frameworks require similarly small amounts of framework-specific code, which consists mostly of loop annotations. The same Kokkos code is able to run on both CPUs and GPUs, and on the platforms studied it again achieved performance similar to that of OpenMP, but *without any source changes*; with OpenMP, a *different* version of the code was written for GPUs. Kokkos was the only framework that was able to support all CPU and GPU platforms in one package.

The SYCL landscape is rapidly evolving, and indeed the new SYCL 2020 standard—which is already being adopted by the three main implementations—brings much-needed productivity improvements such as built-in reduction support and alignment with the newer C++17 standard [26]. However, at the time of writing there are still rough edges to the current SYCL compilers, mainly around platform support fragmentation. First, support for non-GPU or non-x86 platforms is experimental, or even missing from some implementations. Even for GPUs, there is no single implementation that works across all the hardware from the major vendors.

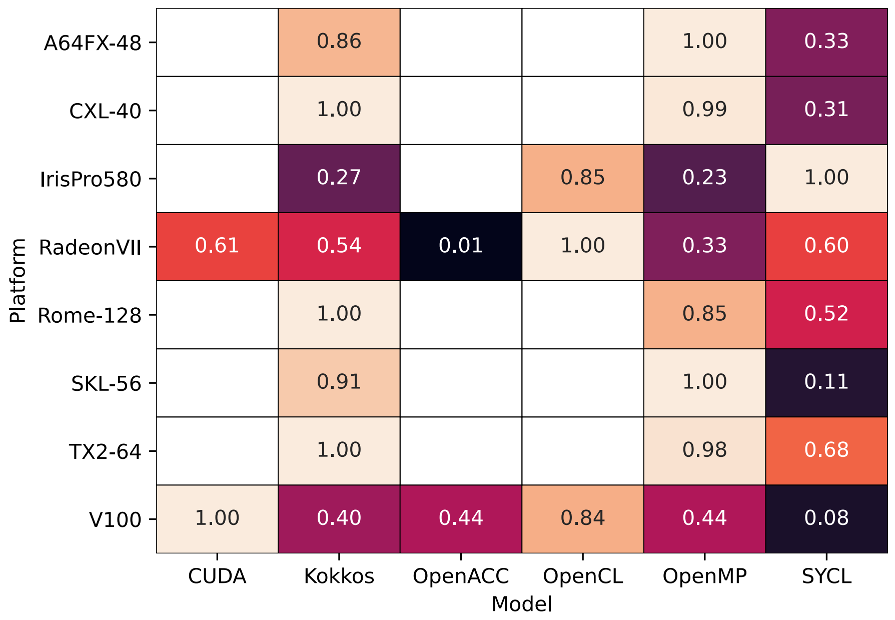

Fig. 7. Achieved performance across all programming models, normalised to the fastest result on each platform. Lighter colours correspond to higher relative performance; blank cells are impossible results.

The open-source hipSYCL implementation is the most portable of the set, being able to run on CPUs, as well as on NVIDIA and AMD GPUs. Both ComputeCpp and OneAPI provide experimental NVIDIA GPU support, but there are still blocking issues such as missing built-in function implementations, which prevent miniBUDE from compiling. Finally, running SYCL on Intel GPUs requires Intel's OpenCL-based ComputeRuntime, but only ComputeCpp and OneAPI support this mode of operation.

The situation on CPUs is similarly complicated. For x86-based platforms, both ComputeCpp and OneAPI run on top of the Intel OpenCL runtime, similar to the situation on Intel GPUs. The OpenCL runtime achieves parallelism via Intel's OneAPI Threading Building Blocks (OneTBB), which provides an optimised abstraction for managing logical threads. Such runtime approaches limit the extent of SYCL implementations to what the underlying runtime supports, from platform coverage to features it can provide; this currently prevents the use of ComputeCpp or OneAPI on Arm-based platforms.

On the other hand, hipSYCL translates SYCL abstractions to OpenMP code, which can then take advantage of existing compiler optimisations natively. This approach results in wide platform support for hipSYCL, but it also means, in principle, that parallelism abstractions are mapped to straightforward OpenMP equivalents. In practice, we found that performance was lower with hipSYCL compared to Kokkos or plain OpenMP, and code changes such as using different parallelism abstractions made little difference for miniBUDE. On platforms not explicitly supported by hipSYCL, as was the case of the A64FX at the time of writing, the additional layer of abstraction also prevented optimal code from being generated, despite having used the correct C++ compiler target flag.

Portability between CPUs and GPUs remains a concern, as SYCL has inherited the same set of problems seen when running OpenCL on the CPU: it is problematic to map workgroup-based parallelism onto a CPU intuitively and efficiently, and it suffers from unexpected setup costs compared to the OpenMP implementation. To work around potentially inefficient mapping, we implemented a compile-time tuning parameter to adjust the amount of work performed by each workgroup, though we found no common setting that provided the best performance on all platforms. On platforms that use Intel's OpenCL runtime, i. e. ComputeCpp and OneAPI, we found the kernel runtime to have large variations, and no functionality was provided to address or mitigate this. In particular, investigations revealed that initialisation of the SYCL context—the `queue`—took upwards of 800 ms in certain cases, even for a simple benchmark that itself ran in half that time.

We also discovered that when running several iterations of a benchmark back-to-back, the first run was usually up to 2× slower than subsequent runs. It was essential to implement a "warm-up" run, which is completely discarded, before starting the timer on the benchmark. Once the warm-up run was completed, the remaining iterations showed consistent run times, as with the other programming models. Both ComputeCpp and OneAPI compile SYCL kernels ahead-of-time, and neither give any indication why initialisation imposes such a large overhead; it is most likely an interaction with the underlying driver. Implementations that do not use the Intel OpenCL runtime, e. g. hipSYCL, did not incur this performance penalty.

6 Future Work

This study opens the path to additional work on the full-scale BUDE application. Instead of maintaining separate implementations in OpenCL for GPUs and OpenMP for CPUs, the code could incorporate a framework like SYCL or Kokkos to reduce divergence. Of course, embracing a new programming model for a scientific application is bound to encounter additional challenges, but in solving those the boundary of performance portability will be pushed further. A higher-level language undoubtedly benefits the ease of maintaining an application, but the higher the price that needs to be paid in terms of performance, the less eager developers are be to adopt it. A targeted investigation using the full application, one with more focus on productivity and software development practices, could reveal if this trade-off would be beneficial for BUDE.

In addition, Kokkos is constantly expanding its support for existing programming models as parallelism backends, thus further increasing its reach on platforms: a SYCL backend is being added, while the existing—but experimental—OpenMP `target` and HIP backends begin to mature. A future study could revisit the performance of hand-tuned, low-level kernels versus implementations using future Kokkos versions.

7 Reproducibility

The source code for all the miniBUDE implementations used in this study, as well as build and run instructions and benchmark input cases, can be found online[1]. A set of scripts is also provided to build and run the benchmark on the platforms used in this study[2].

8 Conclusion

In this paper we have explored performance portability through the lens of a simple, yet realistic, compute-bound benchmark. We have implemented the benchmark in several programming models, including low- and high-level, both well-established and up-and-coming. We have shown that modern programming models can perform on-par with traditional ones, and with constant work done to improve them, their platform support continues to grow.

On the other hand, we have seen that true performance portability is still out of reach: no single version of the code achieved the best performance—or a high fraction of it—on all the platforms studied. Even for a small kernel, platform-specific optimisations and empirical tuning of parameters accounted for more than 30% of the performance and that was enough to differentiate the best-performing implementation from the rest. On GPUs, low-level APIs continue to provide the highest possible performance, and on CPUs, the still-immature driver and implementation ecosystem around SYCL presents an obstacle to the wide adoption of this programming model as a true cross-platform, cross-architecture framework. Of the frameworks studied, Kokkos emerged as a reliable choice, with its lightweight, optimised implementation, and OpenMP remains in a strong position due to it widespread support, although different code paths are still needed for optimal CPU and GPU implementations at the time of writing.

Acknowledgement. The authors would like to thank Si Hammond at Sandia National Laboratories for providing short-notice results for the A64FX platform. Thank you to James Price and Matt Martineau for their original contributions towards optimised OpenMP, OpenCL, and CUDA implementations of the BUDE kernel. This study would not have been possible without previous work by the developers of the Bristol University Docking Engine: Richard Sessions, Deborah Shoemark, and Amaurys Avila Ibarra.

This work used the Isambard UK National Tier-2 HPC Service (https://gw4.ac.uk/isambard/) operated by GW4 and the UK Met Office, and funded by EPSRC (EP/T022078/1). Access to the Cray XC50 supercomputer Swan was kindly provided

[1] https://github.com/UoB-HPC/miniBUDE.

[2] https://github.com/UoB-HPC/performance-portability/tree/2021-benchmarking/benchmarking/2021/bude.

through the Cray Marketing Partner Network. Work in this study was carried out using the HPC Zoo, a research cluster run by the University of Bristol HPC Group (https://uob-hpc.github.io/zoo/).

References

1. Laguna, I., et al.: A large-scale study of MPI usage in open-source HPC applications. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2019. Association for Computing Machinery, Denver (2019). https://doi.org/10.1145/3295500.3356176. ISBN 9781450362290
2. Bernholdt, D.E., et al.: A survey of MPI usage in the US exascale computing project. Concurr. Comput. Pract. Exp. **32**(3), e4851 (2020)
3. Deakin, T., et al.: Performance portability across diverse computer architectures. In: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC). IEEE, Denver, pp. 1–13, November 2019. https://doi.org/10.1109/P3HPC49587.2019.00006. ISBN 978-1-72816-003-0
4. Deakin, T., et al.: Tracking performance portability on the yellow brick road to exascale. In: 2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), Atlanta, GA, USA, p. 13. In press
5. McIntosh-Smith, S., et al.: High performance in silico virtual drug screening on many-core processors. Int. J. High Perf. Comput. Appl. **29**(2), 119–134 (2015). https://doi.org/10.1177/1094342014528252
6. Cherfils, J., Janin, J.: Protein docking algorithms: simulating molecular recognition. Current Opinion Struct. Biol. **3**(2), 265–269 (1993). https://doi.org/10.1016/S0959-440X(05)80162-9. ISSN 0959–440X
7. Fuchs, A., Wentzla, D.: The accelerator wall: limits of chip specialization. In: 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 1–14 (2019). https://doi.org/10.1109/HPCA.2019.00023
8. Price, J., McIntosh-Smith, S.: Exploiting auto-tuning to analyze and improve performance portability on many-core architectures. In: Kunkel, J.M., Yokota, R., Taufer, M., Shalf, J. (eds.) ISC High Performance 2017. LNCS, vol. 10524, pp. 538–556. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67630-2_38
9. Katz, M.P., et al.: Preparing nuclear astrophysics for exascale. In: The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 2020), Atlanta, GA, USA, November 2020, in press
10. Siegel, A.: ECP: lessons learned in porting complex applications to accelerator-based systems. Presentation, Atlanta, GA, USA (2020)
11. Heroux, M.A., et al.: ECP software technology capability assessment report-public. Technical report, NNSA, p. 200 (2020)
12. Lambert, J., et al.: CCAMP: an integrated translation and optimization framework for OpenACC and OpenMP. In: The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 2020), Atlanta, GA, USA, November 2020, in press
13. Mills, R.T., et al.: Toward performance-portable PETSc for GPU-based exascale systems. In: arXiv preprint arXiv:2011.00715 (2020)
14. Carter Edwards, H., Trott, C.R.: Kokkos: enabling performance portability across manycore architectures. In: Extreme Scaling Workshop (XSW 2013). IEEE, pp. 18–24 (2013)

15. Hammond, J.R., Kinsner, M., Brodman, J.: A comparative analysis of Kokkos and SYCL as heterogeneous, parallel programming models for C++ applications. In: Proceedings of the International Workshop on OpenCL, IWOCL 2019. Association for Computing Machinery, Boston (2019). https://doi.org/10.1145/3318170.3318193. ISBN 9781450362306
16. Intel: Intel® oneAPI: A Unied X-Architecture Programming Model (2020). https://software.intel.com/content/www/us/en/develop/tools/oneapi.html. Accessed 16 Dec 2020
17. Codeplay Software: ComputeCPP. https://developer.codeplay.com/products/computecpp/ce/home. Accessed 16 Dec 2020
18. Alpay, A., Heuveline, V.: SYCL beyond OpenCL: the architecture, current state and future direction of HipSYCL. In: Proceedings of the International Workshop on OpenCL. Association for Computing Machinery, Munich (2020). https://doi.org/10.1145/3388333.3388658. ISBN 9781450375313
19. Harrell, S.L., et al.: Effective performance portability. In: 2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), pp. 24–36 (2018). https://doi.org/10.1109/P3HPC.2018.00006
20. Pennycook, S.J., Sewall, J.D., Lee, V.W.: Implications of a metric for performance portability. Future Gener. Comput. Syst. **92**, 947–958 (2019). https://doi.org/10.1016/j.future.2017.08.007. ISSN 0167–739X
21. Sewall, J., et al.: Interpreting and visualizing performance portability metrics. In: 2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), Atlanta, GA, USA (2020, in Press)
22. Deakin, T., McIntosh-Smith, S.: Evaluating the performance of HPCStyle SYCL applications. In: Proceedings of the International Workshop on OpenCL, IWOCL 2020. Association for Computing Machinery, Munich (2020). https://doi.org/10.1145/3388333.3388643. ISBN 9781450375313
23. Lin, W.-C., Deakin, T., McIntosh-Smith, S.: On measuring the maturity of SYCL implementations by tracking historical performance improvements. In: Proceedings of the International Workshop on OpenCL, IWOCL 2020. Association for Computing Machinery (2021, in Press)
24. Deakin, T., Price, J., Martineau, M., McIntosh-Smith, S.: GPU-STREAM v2.0: benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models. In: Taufer, M., Mohr, B., Kunkel, J.M. (eds.) ISC High Performance 2016. LNCS, vol. 9945, pp. 489–507. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46079-6_34
25. Martineau, M., Atkinson, P., McIntosh-Smith, S.: Benchmarking the NVIDIA V100 GPU and tensor cores. In: Mencagli, G., et al. (eds.) Euro-Par 2018. LNCS, vol. 11339, pp. 444–455. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-10549-5_35
26. Reyes, R., et al.: SYCL 2020: more than meets the eye. In: Proceedings of the International Workshop on OpenCL, IWOCL 2020. Association for Computing Machinery, Munich (2020). https://doi.org/10.1145/3388333.3388649. ISBN 9781450375313