Evaluating the performance of HPC-style SYCL applications

Tom Deakin
Simon McIntosh-Smith
tom.deakin@bristol.ac.uk
S.McIntosh-Smith@bristol.ac.uk
University of Bristol
Bristol, UK

ABSTRACT

SYCL is a parallel programming model for developing single-source programs for running on heterogeneous platforms. To this end, it allows for one code to be written which can run on a different architectures. For this study, we develop applications in SYCL which are representative of those often used in High-Performance Computing. Their performance is benchmarked on a variety of CPU and GPU architectures from multiple vendors, and compared to well optimised versions written in OpenCL and other parallel programming models.

CCS CONCEPTS

• **General and reference → Performance**; • **Computing methodologies → Parallel programming languages**; **Massively parallel and high-performance simulations**; • **Software and its engineering → Parallel programming languages**.

KEYWORDS

SYCL, GPGPUs, performance portability, benchmarking

ACM Reference Format:
Tom Deakin and Simon McIntosh-Smith. 2020. Evaluating the performance of HPC-style SYCL applications. In *International Workshop on OpenCL (IWOCL '20), April 27–29, 2020, Munich, Germany*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3388333.3388643

1 INTRODUCTION

SYCL is an open standard parallel programming model allowing the development of codes which can run on heterogeneous systems containing devices from a range of hardware vendors. This means a single-source code base can be written (in C++11) which will run key parts of the application on accelerator devices, or in parallel on the host CPU. To this end, it provides a unified programming model for programming whatever device is present in the system: an attractive proposition for the endeavour for performance portability.

Although SYCL was first released in 2014, it has recently gained traction as hardware vendors release their own implementations alongside those from the open source community. As such, it is now possible to run a SYCL application on a wider range of HPC and consumer (desktop) grade hardware than ever before. The breadth of support is only now similar to those of other established models such as OpenMP and OpenCL.

This study therefore explores the performance of three SYCL applications from a High-Performance Computing domain across a range of CPU and GPU architectures. Importantly, we utilise multiple SYCL implementations in order to test devices from multiple hardware vendors, testing both the portability and performance portability of the application. The applications we use have a heritage in this pursuit and so we can use the prior evaluation of these codes in order to fairly assess the current state of the SYCL ecosystem. As such, it is possible to know a priori the expected performance of the codes on a given architecture independent of the choice of programming model used for the implementation. This approach is extremely valuable in benchmarking these SYCL applications fairly.

In particular, the following contributions are made:

- We introduce two open source mini-apps which have been ported to SYCL. These applications, along with BabelStream, are benchmarked on a number of CPU and GPU architectures.
- The performance of SYCL is assessed with respect to peak hardware expectations and other portable parallel programming models including OpenMP and OpenCL.

1.1 Related Work

Some studies have looked at the performance of SYCL applications across a range of hardware. The study by Reguly looks at the performance of one particular application across a range of devices in a number of models, including SYCL [10]. The study focuses on the performance portability of each kernel across multiple devices implemented in a variety of programming models and show findings similar to our work [2]. Reguly does not include OpenCL in their study as we do here as it is highly pertinent for the SYCL community.

Similarly the study by Joo et al. looks at the performance of a single kernel implemented in SYCL and Kokkos [7]. They focus primarily on GPU performance only and the reference shows few CPU results. The CPU results used an experimental OpenCL driver from Intel for which the support position is unclear according to the Intel driver webpage[1].

Silva et al. compare the performance of two kernels written in SYCL, OpenCL, and OpenMP on CPU devices [11]; GPUs were not considered in their study.

IWOCL '20, April 27–29, 2020, Munich, Germany

ACM ISBN 978-1-4503-7531-3/20/04.
https://doi.org/10.1145/3388333.3388643

[1] https://software.intel.com/en-us/articles/opencl-drivers

Trigkas looked at the performance of OpenCL and SYCL on the Intel Xeon Phi Coprocessor [12]. At this stage, SYCL was very new and as such the performance of SYCL is shown with a large overhead of OpenCL, contrary to what we demonstrate in this paper today. The test architecture (Intel Xeon Phi Knights Corner) is also now defunct.

The study by Hammond et al. compared the SYCL and Kokkos programming models with respect to semantics and parallelism, but does not present performance results directly [5].

2 SYCL APPLICATIONS

We use three applications all implemented in, amongst other models, SYCL. The applications are capable of supporting a heterogeneous memory model and as much of the data as possible remains fully resident on device memory for the duration of the application; such an approach is a basic but crucial requirement of a high performance code. The applications are all main memory bandwidth bound but vary in complexity.

The applications presented all capture characteristics of HPC-style codes. The software engineering is idiomatic of HPC applications [9], likewise is the use of the chosen programming models. In some cases these codes are proxy applications (or mini-apps) which represent closely performance characteristics of a parent code. In others the code expresses a parallel pattern which is extremely common within HPC.

Two of the applications have inbuilt performance models to capture the aggregate useful memory bandwidth attained. As such they themselves provide an insight into the expected performance. The performance of the final application is well known in the literature and has been studied extensively and so the expected performance can be estimated accurately in advance. In this way, all the applications here should achieve close to the peak performance and if they do not this is clear. Such knowledge is exceedingly important when quantifying performance and it is this which has led to our choice of applications for this study into the performance of HPC-style applications written in SYCL.

2.1 BabelStream

The BabelStream benchmark is an implementation of the infamous STREAM benchmark in many parallel programming models. In this way, it is a highly useful application for benchmarking the achievable performance of a very wide range of CPU and GPU architectures. Additionally, it covers the spectrum of programming models from high-level abstractions (such as SYCL and Kokkos), directive based models (such as OpenMP and OpenACC) and lower level offload models (such as OpenCL and CUDA). As such, BabelStream has been a stalwart of our work in investigating performance portability (e.g. [2, 4]).

BabelStream implements the four main STREAM kernels (Copy, Mul, Add and Triad) along with a Dot product to test reduction performance. Such simple kernels are all memory bandwidth bound. Our expectation is that any parallel programming model should be able to leverage a high fraction of peak theoretical hardware memory bandwidth for these kernels, and BabelStream therefore allows for a consistent benchmarking effort to test this. In this work we focus on SYCL performance and will compare to the peak performance of the devices. Results from some other models will be included for perspective to show that SYCL is highly competitive.

Although SYCL results for BabelStream have been presented in the past [3], this work contains new and updated results on a wider and more up to date range of architectures. In particular, a broader choice of SYCL implementations and compilers from both vendors and the open source community are now available for us to use which greatly expands the range of architectures supported by SYCL today in comparison to our prior work.

This application requires a simple one-dimensional `cl::sycl::parallel_for` kernel, and each work-item operates on a unique part of the buffer with no sharing. The kernels are enqueued and run in order as the output of one kernel is used as in put to the next. This is a trick from STREAM to ensure that the hardware memory caches are fully overwritten by each kernel. The dependencies from the accessors mean this ordering remains true.

The dot product kernel requires a reduction to be implemented, and as of SYCL 1.2.1 there is no first-class support for this as there is in other models including OpenMP, RAJA, Kokkos, etc. A commutative tree reduction is therefore implemented inside each work-group, with the final value from each work-group reduced on the host. The number of work-groups is chosen differently for CPUs and GPUs according to a simple rule based on device properties.

2.2 Heat

A great many simulation codes can be categorised as requiring what is known as a *stencil* update to each point in the domain. This pattern updates each cell or value with some average of neighbouring values. This type of kernel is found in image processing, linear solver libraries and in many finite difference modelling-simulation codes.

The Heat code is a simple explicit finite difference solve for a simple differential equation, used primarily for teaching parallel programming of GPUs in OpenMP [2]. The Method of Manufactured Solutions is applied so that the code produces a simple analytically known answer. The main kernel is a simple 5-point stencil, averaging the value in each cell with those in the four axial neighbouring cells.

Although a simple code, and as with BabelStream, it allows us to test robustly how a typical 5-point stencil performs in different parallel programming models on a wide range of architectures. The performance limiting factor of such a kernel is again main memory bandwidth, and so in this study we compare to the achievable memory bandwidth on each platform.

The data is stored in a two-dimensional buffer, allocated with extent `cl::sycl::range<2>(nx,ny)`. The kernel is a two-dimensional `cl::sycl::parallel_for`, with the global number of work-items defined with the same range.

The nature of a stencil code requires offsetting the work-item IDs to access neighbouring elements. The components of `cl::sycl::id` are therefore used to memory access through the accessor as follows:

```
size_t j = id[0];
size_t i = id[1];
u_tmp[j][i] =  r2 * u[j][i] + r * u[j-1][i] + ...;
```

[2] https://github.com/uob-hpc/openmp-tutorial

Note that this unpacking of the `cl::sycl::id` conforms to the `get_linear_id()` ordering of work-items *and* memory layout in a consistent manner as specified in the SYCL 1.2.1 standard. By this we mean that the following notations are equivalent and should ensure a stride-one access pattern in memory:

```
u_tmp[id]
u_tmp[id.get_linear_id()]
u_tmp[id[0]][id[1]]
```

Further discussion on the accessor notation will be presented in Section 3.3.

A minor point in this implementation which may be of interest to those with large legacy applications is the `get_pointer()` function belonging to a host accessor of a buffer. This returns a host accessible pointer to the memory, which here could be passed to an existing routine as a `double *` without modification of that routine. We use this to our advantage here when checking the final solution.

2.3 CloverLeaf

The CloverLeaf mini-app is a performance proxy for a 2D structured grid Lagrangian-Eulerian hydrodynamics code [8]. It contains around a dozen kernels consisting of both point-wise and stencil-like update of grid values. Whilst primarily main memory bandwidth bound, some kernels do require math library functions such as exponentials and square roots. As part of the Mantevo benchmark suite [6] and our other work (most recently [2]), the performance portability of CloverLeaf has been well studied.

For this work, we present the first SYCL port of CloverLeaf along with results across CPU and GPU architectures. The wealth of existing implementations and performance data provide an ideal comparison for exploring the performance of such a SYCL implementation.

CloverLeaf is by far the largest application in this study; the SYCL port is nearly 8,000 lines of code (excluding comments, blank lines, etc.).

CloverLeaf is primarily formed of a number of kernels which consist of two tightly nested loops which iterate over the spatial domain: this is the source of parallelism whence we use a two-dimensional `cl::sycl::parallel_for`. The Fortran origins of this code mean that halo (ghost) cells are allocated surrounding the main grid data. The offset mechanism in the parallel dispatch is used to adjust the global IDs of the launched work-items appropriately so that the work-item `id` can be passed directly to the accessors. As such this application explores additional features of the SYCL programming model that BabelStream and Heat do not use.

Two kernels require a reduction operation: one on a single FP64 value, the other on five FP64 values. For those readers familiar with OpenMP, the latter could be very simply expressed as follows:

```
#pragma omp parallel for collapse(2) \
  reduction(+:vol,mass,ie,ke,press)
```

As we saw with BabelStream, the current lack of reduction support in SYCL 1.2.1 means a reduction operation must be implemented. To this end, the Dot product reduction from BabelStream was brought into CloverLeaf and updated to support the summation of multiple values for our needs in this mini-app. Some additional details on the implementation can be found online at [1].

3 RESULTS

3.1 Platforms and SYCL compilers

Table 1 details the devices used in this study which capture three consumer GPU devices from different vendors along with a HPC server CPU. The memory bandwidth is quoted from the tech sheets where available[3]. This is a selection from the devices hosted in the University's of Bristol HPC Zoo where SYCL compilers and runtimes are made available.

The range of devices from different vendors requires us to use various SYCL compilers in order to collect results. We use three SYCL compiles/runtimes which each support one or more of our selected platforms:

- ComputeCpp 1.2.0
- LLVM/SYCL (an open source development effort lead by Intel)
- hipSYCL (an open source implementation lead by Heidelberg University)

As a set, these compilers provide support for all our devices.

At the time of writing for our selected platforms, ComputeCpp supports the Xeon processor and the NUC, however the support for NVIDIA is only experimental and we have found severe host side performance degradation. The offloaded kernels themselves achieve runtimes close to that which we expect, however we notice significant inexplicable idle time is present between kernel submissions. Codeplay recently announced updated support for NVIDIA GPUs will be available later in 2020 and so this issue should be resolved shortly. The AMD GPU uses the ROCm driver stack which does not support SPIR, which is required by ComputeCpp.

The LLVM/SYCL compiler at present supports only Intel OpenCL platforms. This compiler is under heavy development and so the results represent a recent snapshot of the very much in development compiler.

On the Intel platforms (where we have two compiler/runtimes to choose from) we present the most favourable result in the figures and highlight the choice of ComputeCpp and LLVM/SYCL and any performance differences observed in the text. The Intel NUC result shown will always be for running onthe GPU; we omit the CPU result.

hipSYCL supports AMD GPUs (via HIP and the ROCm software stack) and NVIDIA GPUs (via CUDA). Therefore for NVIDIA and AMD GPUs we can only present results from hipSYCL.

We use the following drivers:

- The Intel NUC has the Intel OpenCL Graphics Driver 19.44.14658 installed.
- The Intel Xeon (Skylake) uses the Intel OpenCL runtime 18.1.0.0920.
- The NVIDIA RTX 2080 Ti uses the NVIDIA driver 418.39.
- The AMD Radeon VII uses AMD's ROCm driver 2833.0 (HSA1.1,LC).

Our OpenMP results were built with Intel Parallel Studio 2019.4. Our CUDA results were built with CUDA 10.1. Our HIP results were built with version 1.5.19055.

[3]The Intel ARK does not publish memory bandwidth for Xeon Scalable Processors. We quote the value here from https://en.wikichip.org/wiki/intel/xeon_gold/6126.

Table 1: Platform details

Name	Architecture	Device Type	Mem. BW (GB/s)
Intel Xeon Gold 6126 (12-core)	Skylake	HPC CPU (1 socket)	119.21
Intel NUC i7-6770HQ with Iris Pro 580 Graphics	Skylake/Gen9	CPU + Integrated GPU	34.1
AMD Radeon VII	Vega 20	Discrete GPU	1024
NVIDIA RTX 2080 Ti	Turing	Discrete GPU	616

3.2 BabelStream

We present results from three BabelStream kernels (Triad, Dot and Copy) for SYCL and OpenCL. The default problem size is used of arrays of 2^{25} FP64 elements and 100 iterations.

3.2.1 Triad. Triad is the canonical STREAM-style kernel and so we begin by showing these results in Figure 1. The values are shown as a percentage of the theoretical peak bandwidth detailed in Table 1. As with all STREAM results, we do not expect to reach 100% of theoretical peak.

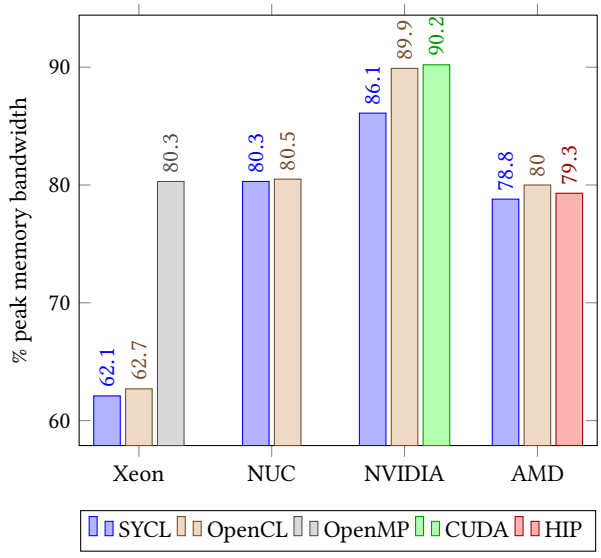

Figure 1: BabelStream Triad results

On the Intel NUC the figure shows the result using ComputeCpp; the LLVM/SYCL result is very similar at 79.7%. For the Intel Xeon we show the ComputeCpp result and were unable to successfully build with LLVM/SYCL against the Intel OpenCL 18.1 runtime library.

In all cases we find there is a slight overhead of between 0.25–4.26% for SYCL over OpenCL. The largest difference is seen on the NVIDIA platform, but do recall we use hipSYCL here as there is no other official way to target NVIDIA GPUs with SYCL currently.

OpenMP is very well supported on the Intel Xeon platform, and so it is useful to include the result of 80.3% here for comparison. This shows there is a clear performance penalty from OpenCL on this true HPC platform despite the usual precautions of pinning threads with `taskset`.

We also show CUDA and HIP results, the vendor-choice APIs for programming NVIDIA and AMD GPUs. This results highlight there is a small overhead from hipSYCL where it is implemented on top of CUDA. These results show that the hipSYCL implementation can indeed provide portable performance very close to vendor-tied languages.

3.2.2 Dot. The Dot product kernel has equivalent implementations in the OpenCL and SYCL versions as detailed in [4]. The number of work-groups launched differs on CPUs and GPUs, however the number is the same for both SYCL and OpenCL on a given platform. Figure 2 shows the percentage of theoretical peak bandwidth for the Dot kernel for SYCL and OpenCL.

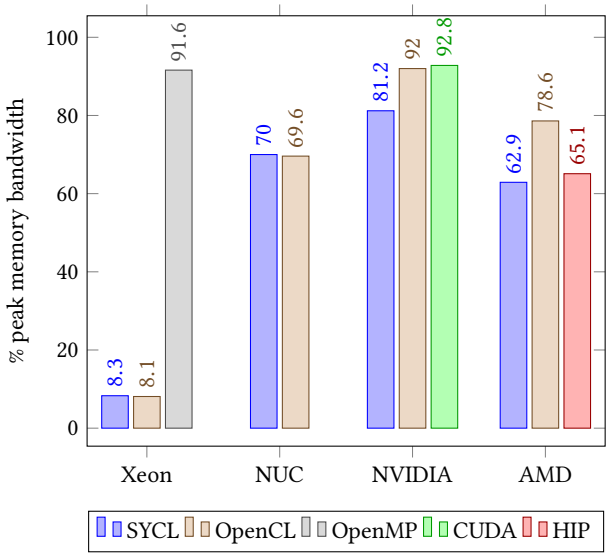

Figure 2: BabelStream Dot results

We show the LLVM/SYCL result for the Intel NUC, noting that ComputeCpp achieves a similar 69.6%.

Despite the implementations being equivalent we observe a greater overhead on the discrete GPUs using hipSYCL of 11.7–20.0%. The reduction requires work-group barriers, local memory and a device to host transfer of an array of reduced values for each work-group; any of which may have overheads. The lack of proper profiling tools for SYCL makes the job of identifying this intractable at present. hipSYCL does not currently implement SYCL event profiling[4]; as it does not use OpenCL as a backend (technically rendering it not SYCL 1.2.1 compliant), event timings are not obtainable either. As such we cannot use the manual event

[4] An example of event profiling for SYCL can be found at https://codeplay.com/portal/08-27-19-optimizing-your-sycl-code-using-profiling

profiling technique built into the SYCL standard. Vendors do not provide SYCL tools to enable us to identify where the source of the performance degradation may be.

The native HIP implementation on AMD GPUs does show similar performance to the SYCL version, which indicates that it is the underlying HIP rather than overheads in hipSYCL itself which causes this performance degradation we observe in comparison with the OpenCL result.

Neither ComputeCpp nor LLVM/SYCL show any variation over OpenCL on the Intel platforms, however the Xeon performance is significantly reduced compared to Triad. The OpenMP version of BabelStream achieves 91.6% peak memory bandwidth for the Dot kernel. It is possible that our CPU reduction implementation is not as well optimised as that inside OpenMP. As we found in our performance portability work [2, 4], it is a significant challenge to have a wide support for OpenCL on HPC processors. It is a limitation of both OpenCL and SYCL that the user is required to implement a highly performant reduction for such a key parallel programming pattern; other models including OpenMP do allow manual reduction implementation where the provided one is not suitable or sufficient.

3.2.3 Copy. The Copy kernel provides some interesting counterpoint to the other kernels. This kernel is simply copying from one array to another with no floating point operations. Note too that in contrast to Triad which has two input and one output array, Copy moves less data overall.

The results are shown in Figure 3. Again, we present the LLVM/SYCL result for the Intel NUC; ComputeCpp achieved a similar 90.8%.

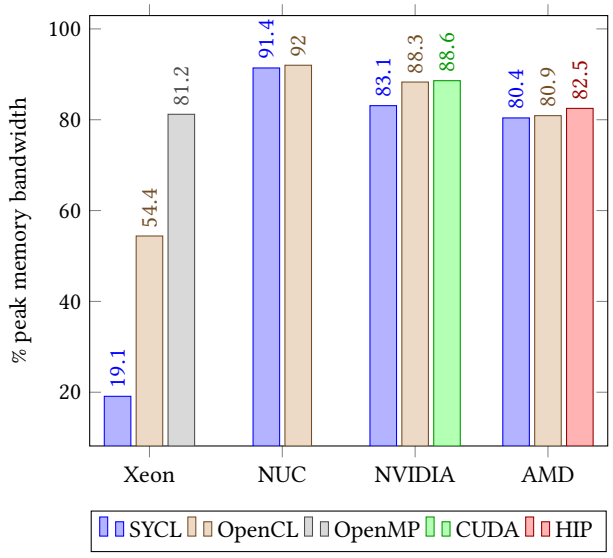

Figure 3: BabelStream Copy results

The overheads of hipSYCL over native OpenCL are reduced a little for this kernel, and similar bandwidth to the other kernels was attainable. The Intel NUC manages to demonstrate 10% more memory bandwidth compared to Triad and this is likely due to the change in the number of data streams. This behaviour was previously observable in [4].

As with the Triad kernel, there is little difference in attainable performance between OpenCL and the vendor-specific CUDA and HIP programming models.

Although SYCL and OpenCL are similar for Triad and Dot, there is a stark difference for Copy on the Xeon platform. This platform shows the largest disparity in performance for these three kernels in both models: 8–63% of peak bandwidth. As SYCL 1.2.1 relies on an OpenCL implementation, this variability that comes from OpenCL will clearly impact SYCL performance.

Additionally, there is clearly a missed optimisation for this Copy kernel in the LLVM/SYCL compiler as this is the only result for BabelStream where there is such a large difference between OpenCL and SYCL. On Intel Xeon SYCL is only achieving 22.8 GB/s where as OpenCL was able to attain 64.9 GB/s. Here OpenMP attained 81.16% (96.7 GB/s) highlighting the issues here.

3.3 Heat

A problem size of 8000 by 8000 spatial cells is used, with 1000 timesteps. This ensures the benchmark runs for long enough and that the arrays are larger than caches so that the application should be limited by the performance of main memory bandwidth.

The main kernel reads from one array and writes to a different array (overwriting), and so is most similar to the Copy kernel from BabelStream. We therefore present the performance results as a percentage of the Copy bandwidth as presented in Figure 3. We compute this ratio with respect to each programming model independently so as to factor out the performance variation already discussed in Section 3.2. For example, we show the Heat *OpenCL* performance as a percentage of the BabelStream Copy *OpenCL* performance, and the Heat *SYCL* performance as a percentage of the BabelStream Copy *SYCL* performance on each platform.

We show the runtime and bandwidth results for Heat in Table 2. We note that the bandwidth is proportional to the runtime according to the simple aggregate bandwidth model calculation. This table shows the raw results which we then compare against the BabelStream Copy results in Figure 4.

As noted in Section 3.2.3, the SYCL Copy kernel performs poorly on the Intel Xeon, however for consistency we use the attainable Copy bandwidth — the performance of Heat should be similar to that of Copy after all. Indeed, what we observe is that both OpenCL and OpenMP attain around 65 GB/s of bandwidth for Heat.

The hipSYCL implementation does not currently correctly support the `A[j][i]` buffer accessor notation and so we therefore had to replace all such accesses with `A[cl::sycl::id<2>{j,i}]`. On platforms where both notations were supported we noticed no performance difference between the notations. The results we present here therefore use the later notation for 2D access.

The source code was edited to use `cpu_selector` instead of `gpu_selector` on the Intel Xeon platform.

Figure 4 shows the performance of the SYCL version of Heat on all the architectures in our study. We present the average (mean) of five runs, with the minimum and maximum values shown with error bars. For the Intel NUC we present LLVM/SYCL results; the ComputeCpp kernel time was similar however had a larger overall runtime from the host code parts of the application (host code and solution checking).

Table 2: Heat average runtime and bandwidth results

Platform	Model	Runtime (s)	Mem. BW (GB/s)
Xeon	SYCL (2D range)	77.17	13.27
	SYCL (1D range)	87.64	11.68
	OpenCL	15.71	65.04
	OpenMP	15.52	65.99
NUC	SYCL (2D range)	38.34	26.71
	SYCL (1D range)	39.44	25.97
	OpenCL	38.31	26.73
NVIDIA	SYCL (2D range)	2.28	449.50
	SYCL (1D range)	2.27	450.23
	OpenCL	4.06	252.13
	CUDA	3.97	257.80
AMD	SYCL (2D range)	2.23	461.13
	SYCL (1D range)	1.74	588.20
	OpenCL	2.26	460.32
	HIP	2.09	490.17

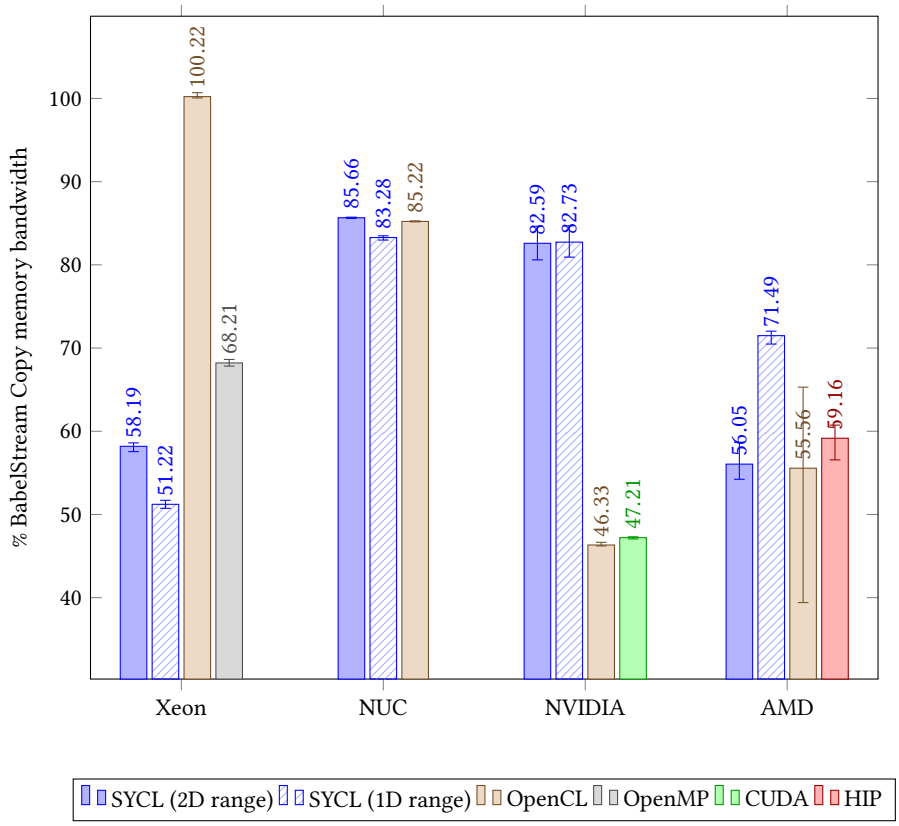

Figure 4: Heat results, showing average percentage of achievable memory bandwidth of five runs with min/max range

The error bars show that there is more performance variability with the discrete GPU platforms using hipSYCL compared to the Intel platforms. The difference is still fairly small at around 2% for the SYCL versions.

In general we see that SYCL is achieving good performance across all the processors. The performance on the AMD GPU is however much reduced. An early issue with some SYCL implementations (which has been addressed by many implementations) was

the mismatch between mappings of SYCL work-items to OpenCL work-items. SYCL 1.2.1 specifies the ordering of work-items via the `get_linear_id()` API call where first dimension in the `range` is the *slowest* with the largest stride. Unfortunately this ordering is opposite that chosen by OpenCL implementations. Note that OpenCL does not stipulate an ordering and so it is simply convention that the first NDRange dimension was the *fastest* moving with a stride of one.

We therefore also test a manual linearisation of the range. Buffers are changed to be allocated with a one-dimensional `cl::sycl::range` and the `parallel_for` dispatch functions also iterate with a one-dimensional range. Buffers accessors are indexed as follows which ensures stride-one access:

```
int j = idx[0] / n;
int i = idx[0] % n;
A[i+j*n] = ...
```

The memory access pattern here should be no different to that assumed by the `get_linear_id()` access pattern, however it is clear there are performance differences even with runtimes which have addressed this issue. The bars labelled "SYCL (1D range)" in Figure 4 show this manual 1D access. It is clear that the AMD performance has much improved. The HIP backend to hipSYCL is exhibiting this inconsistency with the CUDA backend as the manual change does not cause any variation on the NVIDIA GPU.

The performance on Xeon looks much reduced in Figure 4 with the 1D access pattern, however the bandwidth achieved for 2D is 13 GB/s which is reduced to 12 GB/s for the 1D access. The performance on Xeon is poor in both cases as it was for the Copy kernel in BabelStream and so this is likely the primary factor rather than the change in accessor index pattern.

Figure 4 also shows the performance of the OpenCL version of Heat. The performance is calculated as achieved memory bandwidth as a percentage of that attained by the OpenCL version of the BabelStream Copy kernel. The Intel NUC GPU shows very close alignment with the SYCL version as expected. However the other platforms each exhibit some differences. The performance of the Intel Xeon is actually inline with what we would expect as it is close to the Copy bandwidth (recall Section 3.2.3).

The OpenCL and CUDA results on the NVIDIA GPU highlights some issues with the particular combination of driver and GPU, for the same code achieves 93.3% on a NVIDIA P100 GPU. The lack of good double precision on our 2080 Ti is unlikely to be the issue for a memory bandwidth kernel, noting too that BabelStream is also double precision. Indeed, both OpenCL and CUDA show this phenomenon so we with high confidence we can discount an implementation issue in the Heat application itself.

There are two aspects to the difference observed with the OpenCL run on the AMD GPU. Firstly there is a very large amount of variability from run to run and we have been unable to identify the root cause of. Secondly, a 1D NDRange kernel does not improve the performance as it did for the SYCL version. The 1D version attains around 46% of Copy bandwidth, however does seem to more consistently reach this performance.

The results of this Heat mini-app therefore highlight that portable performance relies in part on the quality of the software stack for each platform of interest, and so we stress that ensuring that the wide SYCL ecosystem explored in this study reaches maturity quickly is a matter of priority.

3.4 CloverLeaf

The SYCL port of CloverLeaf is the newest and most substantial mini-app in this study. As such, we present these early results which are the first to be collected for this code across this many different SYCL devices. As such, these present the first encouraging snapshot and we plan to extend the study further in the near future.

As with the other mini-apps, in order to build with the version hipSYCL used previously we would need to change the buffer accessor notation. This is a significant change however, and so instead we opt to use the latest hipSYCL version (Git commit `5296ee5`) which supports the accessor notation required. Unfortunately we were unable to run CloverLeaf on the AMD GPU due to runtime errors with both approaches.

Figure 5 shows the runtime of the `clover_bm16_short.in` input file for the SYCL and OpenCL implementations. The results were stable so we present the minimum value of five runs in each case. On the Intel NUC we show the result from LLVM/SYCL noting that the ComputeCpp result was within around 2%. On the Intel Xeon we show the ComputeCpp result.

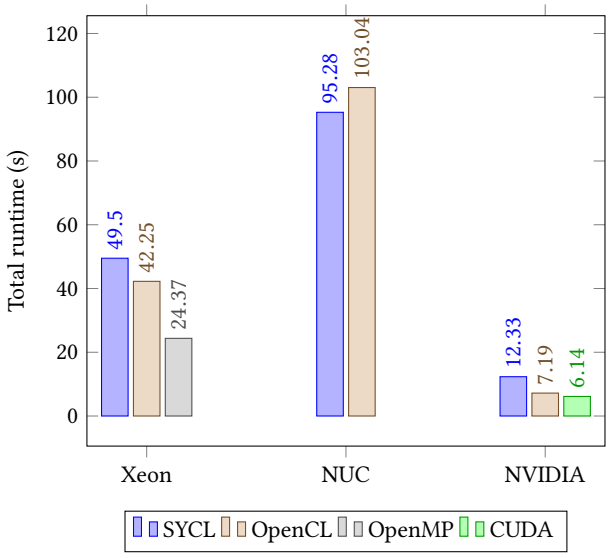

Figure 5: Latest CloverLeaf results

Here it is clear that the SYCL performance is similar to that of OpenCL (within 10%). It is slightly faster on the Intel NUC due to the differing performance of the advection kernels, as shown in Figure 6. This routine consists of a large number of small kernels, and the implementations look superficially fairly similar.

A challenge with studying performance portability is the maintenance requirements of a large number of different implementations of mini-apps developed over a number of years. We will of course investigate this further as future work but expect that the OpenCL version can be improved. For our purposes in this study we see that the SYCL performance is indeed similar to OpenCL for all other kernels.

The OpenMP result is again faster than both OpenCL and SYCL, a consistent pattern for all the benchmarks in this study.

On the NVIDIA GPU we again see the OpenCL and CUDA both achieve similar results as expected. The SYCL result is somewhat slower, and on inspection of the built in profiler this extra runtime is associated with the reduction operation (the timestep kernel). All the other kernels are close to parity with CUDA. Each queue submission is followed by a wait operation to ensure the timing is correctly apportioned to the routines. This situation differs to the runtime on the NUC as described above.

4 CONCLUSION

This study highlights that it is often possible to write SYCL applications which achieve high performance across a number of architectures. Our results show that across each of the different codes the SYCL implementation achieves similar performance to a direct OpenCL implementation. Additionally, for some codes both programming models can leverage close to the peak hardware performance. This is a fantastic position for SYCL as more HPC-style codes begin to be written in it. The open source applications in this study go some way to increasing the breadth of SYCL applications in this arena.

However, the results also show there is a clear need for widespread vendor commitment to SYCL. Targeting GPU architectures today requires using open source projects which do not get commercial support from the hardware vendors which the project supports. On CPUs, the reliance on the OpenCL driver leaves exposure to existing well known performance issues in this space. Additionally, not all CPU vendors currently offer OpenCL support and therefore by extension the possibility of SYCL 1.2.1 support. We have had to omit those CPUs from this study as a result. Although some open source solutions exist to support additional architectures, one of which we used in this study, it is important that vendors provide well tested packaged compilers and runtimes to ease the complexity of building and running SYCL codes.

The recent announcement of an extension to the LLVM/SYCL implementation used in this study to provide support for NVIDIA GPUs is a very much welcomed step in improving the widespread support for SYCL [5]. The success of any programming model lies primarily in the adoption of both users and vendors in order to establish a well supported ecosystem across many problem domains and the increased support is a positive step.

Going forward we hope to compare these results to other programming models and assess their performance portability to augment our previous study [2].

ACKNOWLEDGMENTS

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/.

CloverLeaf was originally ported to SYCL by Tom Lin during their internship within the HPC Research Group at the University of Bristol. This work was supported through ASiMoV under EPSRC grant number EP/S005072/1.

[5] https://www.codeplay.com/portal/12-16-19-bringing-nvidia-gpu-support-to-sycl-developers

REFERENCES

[1] Tom Deakin and Tom Lin. 2020. Experiences with SYCL for the hydrodynamics mini-app, CloverLeaf. https://uob-hpc.github.io/2020/01/06/cloverleaf-sycl.html. Accessed: 2020-01-06.

[2] Tom Deakin, Simon Mcintosh-Smith, James Price, Andrei Poenaru, Patrick Atkinson, Codrin Popa, and Justin Salmon. 2019. Performance Portability Across Diverse Computer Architectures. In *IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. Denver, CO, 1–13. https://doi.org/10.1109/P3HPC49587.2019.00006

[3] Tom Deakin, James Price, Matt Martineau, and Simon McIntosh-Smith. 2016. GPU-STREAM: Now in 2D! (poster). In *Supercomputing*. Salt Lake City, Utah.

[4] Tom Deakin, James Price, Matt Martineau, and Simon McIntosh Smith. 2018. Evaluating attainable memory bandwidth of parallel programming models via BabelStream. *International Journal of Computational Science and Engineering* 17, 3 (2018), 247–262. https://doi.org/10.1504/IJCSE.2017.10011352

[5] Jeff R. Hammond, Michael Kinsner, and James Brodman. 2019. A comparative analysis of Kokkos and SYCL as heterogeneous, parallel programming models for C++ applications. *ACM International Conference Proceeding Series* (2019). https://doi.org/10.1145/3318170.3318193

[6] Michael a. Heroux, Douglas W. Doerfler, Paul S. Crozier, James M. Willenbring, H. Carter Edwards, Alan Williams, Mahesh Rajan, Eric R. Keiter, Heidi K. Thornquist, and Robert W. Numrich. 2009. Improving performance via mini-applications. *Sandia National …* September (2009), 1–38. https://doi.org/10.1108/13620439610118528

[7] Balint Joo, Thorsten Kurth, M A Clark, Jeongnim Kim, Christian Robert Trott, Dan Ibanez, Daniel Sunderland, and Jack Deslippe. 2019. Performance Portability of a Wilson Dslash Stencil Operator Mini-App Using Kokkos and SYCL. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 14–25. https://doi.org/10.1109/P3HPC49587.2019.00007

[8] Simon McIntosh-Smith, Michael Boulton, Dan Curran, and James Price. 2014. On the Performance Portability of Structured Grid Codes on Many-Core Computer Architectures. In *Supercomputing*, Julian Martin Kunkel, Thomas Ludwig, and Hans Werner Meuer (Eds.). Lecture Notes in Computer Science, Vol. 8488. Springer International Publishing, Cham, 53–75. https://doi.org/10.1007/978-3-319-07518-1_4

[9] Karthik Raman, Tom Deakin, James Price, and Simon McIntosh-Smith. 2017. Improving achieved memory bandwidth from C++ codes on Intel{\circledR} Xeon Phi™ Processor (Knights Landing).

[10] Istvan Z. Reguly. 2019. Performance Portability of Multi-Material Kernels. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 26–35. https://doi.org/10.1109/P3HPC49587.2019.00008

[11] Hercules Cardoso Da Silva, Flavia Pisani, and Edson Borin. 2017. A comparative study of SYCL, OpenCL, and OpenMP. *Proceedings - 28th IEEE International Symposium on Computer Architecture and High Performance Computing Workshops, SBAC-PADW 2016* (2017), 61–66. https://doi.org/10.1109/SBAC-PADW.2016.19

[12] Angelos Trigkas. 2014. *Investigation of the OpenCL SYCL Programming Model*. Master's thesis. Edinburgh Parallel Computing Centre, University of Edinburgh.

A REPRODUCIBILITY

The list of devices and compiler and drivers has already been detailed in the main body of the paper. This appendix will detail how to obtain and build the codes on each platform. The platforms were all installed into the University of Bristol's HPC Zoo and the Environment Module commands which detail the software choices will be shown.

It is important to make sure that the Intel NUC GPU driver has the hangcheck disabled as follows:

```
echo N > /sys/module/i915/parameters/enable_hangcheck
```

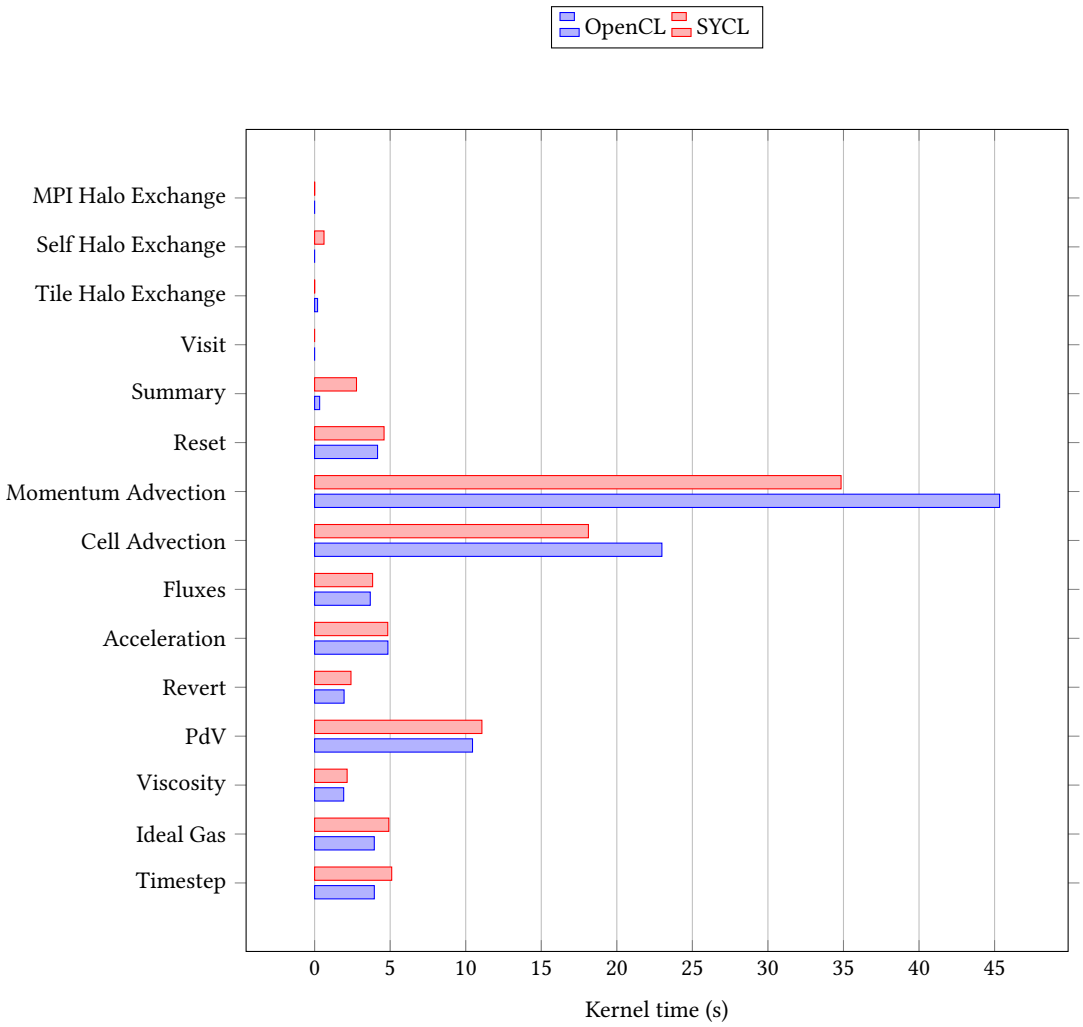

Figure 6: Kernel runtimes for CloverLeaf running on Intel NUC

A.1 BabelStream

The code can be downloaded from https://github.com/uob-hpc/babelstream.

A.1.1 AMD Radeon VII.

```
# SYCL
module load hipsycl/0.8.1-prerelease
syclcc-clang -DSYCL -O3 main.cpp SYCLStream.cpp \
  --hipsycl-gpu-arch=gfx906 \
  --hipsycl-platform=rocm -o sycl-stream_hipsycl_rocm

# OpenCL
module load cuda/10.1
make -f OpenCL.make -B

# HIP
make -f HIP.make -B EXTRA_FLAGS="-amdgpu-target=gfx906"
```

A.1.2 NVIDIA RTX 2080 Ti.

```
# SYCL
module load hipsycl/0.8.1-prerelease
syclcc-clang -std=c++14 -DSYCL -O3 main.cpp SYCLStream.cpp \
  --hipsycl-gpu-arch=sm_75 --hipsycl-platform=cuda \
  -o sycl-stream_hipsycl_cuda

# OpenCL
module load cuda/10.1
make -f OpenCL.make -B

# CUDA
module load cuda/10.1
make -f CUDA.make -B EXTRA_FLAGS="-arch=sm_75"
```

A.1.3 Intel NUC.

```
# SYCL
# LLVM/SYCL
module load llvm/sycl
```

```
clang++ -O3 -std=c++11 -lOpenCL -fsycl -DSYCL \
  main.cpp SYCLStream.cpp -o sycl-stream_llvm
# ComputeCpp
module load computecpp/1.2.0 cuda/10.1
make -f SYCL.make -B

# OpenCL
module load cuda/10.1
make -f OpenCL.make -B
```

A.1.4 Intel Xeon.

```
# SYCL
# ComputeCpp
module load computecpp/1.2.0 intel/opencl/18.1 cuda/10.1
make -f SYCL.make -B
# NB: Need to preload the Intel OpenCL runtime on running

# OpenCL
module load cuda/10.1 intel/opencl/18.1
make -f OpenCL.make -B

# OpenMP
make -f OpenMP.make COMPILER=INTEL -B
OMP_NUM_THREADS=12 OMP_PROC_BIND=true ./omp-stream
```

A.2 Heat

The code can be downloaded from https://github.com/uob-hpc/heat_sycl.

A.2.1 AMD Radeon VII.

```
# SYCL
module load hipsycl/0.8.1-prerelease
syclcc-clang -DSYCL -O3 heat_sycl.cpp \
  --hipsycl-gpu-arch=gfx906 \
  --hipsycl-platform=rocm -o heat_rocm
syclcc-clang -DSYCL -O3 heat_sycl_1drange.cpp \
  --hipsycl-gpu-arch=gfx906 \
  --hipsycl-platform=rocm -o heat_1D_rocm

# OpenCL
module load cuda/10.1
make -f Makefile.ocl -B

# HIP
make -f Makefile.hip -B
```

A.2.2 NVIDIA RTX 2080 Ti.

```
# SYCL
module load hipsycl/0.8.1-prerelease
syclcc-clang -std=c++14 -DSYCL -O3 heat_sycl.cpp \
  --hipsycl-gpu-arch=sm_75 --hipsycl-platform=cuda \
  -o heat_cuda
syclcc-clang -std=c++14 -DSYCL -O3 heat_sycl_1drange.cpp \
  --hipsycl-gpu-arch=sm_75 --hipsycl-platform=cuda \
  -o heat_1D_cuda

# OpenCL
module load cuda/10.1
make -f Makefile.ocl -B

# CUDA
module load cuda/10.1
make -f Makefile.cuda -B
```

A.2.3 Intel NUC.

```
# SYCL
# LLVM/SYCL
module load llvm/sycl
clang++ -O3 -std=c++11 -lOpenCL -fsycl \
  heat_sycl.cpp -o heat_llvm
clang++ -O3 -std=c++11 -lOpenCL -fsycl \
  heat_sycl_1drange.cpp -o heat_1D_llvm

# ComputeCpp
module load computecpp/1.2.0 cuda/10.1
module load gcc/9.1.0
module load cmake/3.14.5
mkdir -p build
cd build
cmake .. -DCMAKE_CXX_COMPILER=g++ \
  -DCMAKE_C_COMPILER=gcc \
  -DComputeCpp_DIR=<path to ComputeCpp>
make

# OpenCL
module load cuda/10.1
make -f Makefile.ocl -B
```

A.2.4 Intel Xeon.

```
# SYCL
# ComputeCpp
module load computecpp/1.2.0
module load intel/opencl/18.1
module load cuda/10.1
module load gcc/9.1.0
module load cmake/3.14.5
mkdir -p build
cd build
cmake .. -DCMAKE_CXX_COMPILER=g++ -DCMAKE_C_COMPILER=gcc \
  -DComputeCpp_DIR=<path to ComputeCpp> \
  -DOpenCL_LIBRARY=<Intel OpenCL lib>
make

# OpenCL
module load intel/opencl/18.1 cuda/10.1
make -f Makefile.ocl -B
# NB: Need to preload the Intel OpenCL runtime on running

# OpenMP
module load intel/parallel_studio/2019.4
make -f Makefile.omp -B
```

A.3 CloverLeaf

The SYCL code can be downloaded from https://github.com/uob-hpc/cloverleaf_sycl. The OpenCL can be downloaded from https://github.com/uk-mac/cloverleaf_opencl. The CUDA can be downloaded from https://github.com/uk-mac/cloverleaf_cuda. The OpenMP can be downloaded from https://github.com/uk-mac/cloverleaf_ref.

For the SYCL version we ensured that the default data access pattern was used by removing the define of `SYCL_FLIP_2D` in the `sycl_utils.hpp` file.

A.3.1 Intel NUC.

```
# SYCL
# ComputeCpp
module load computecpp/1.2.0
module load cmake/3.14.5
module load openmpi/4.0.1/gcc-8.3
cmake3 -Bbuild -H. -DCMAKE_BUILD_TYPE=Release \
  -DComputeCpp_DIR=<path_to_computecpp> \
  -DOpenCL_INCLUDE_DIR=include/
cmake3 --build build --target clover_leaf --config Release

./build/clover_leaf InputDecks/clover_bm16_short.in

# LLVM/SYCL
module load openmpi/4.0.1/gcc-8.3 llvm/sycl
OMPI_CC=clang OMPI_CXX=clang++ mpic++ -O3 -std=c++11 \
  -fsycl -lOpenCL *.cpp \
  --gcc-toolchain=/nfs/software/x86_64/gcc/7.4.0 \
  -o clover_leaf_llvm

./clover_leaf_llvm ../InputDecks/clover_bm16_short.in

# OpenCL
module load openmpi/4.0.1/gcc-8.3 intel/opencl/18.1
module load cuda/10.1
OMPI_CXX=g++ OMPI_CC=gcc OMPI_F90=gfortran make \
  COMPILER=GNU OCL_VENDOR=INTEL -B -j clover_leaf \
  COPTIONS="-std=c++98" OPTIONS="-lstdc++"
sed -i 's/CPU/GPU/' clover_bm16_short.in
cp clover_bm16_short.in clover.in
```

A.3.2 Intel Xeon.

```
# SYCL
# ComputeCpp
module load computecpp/1.2.0
module load cmake/3.14.5
module load openmpi/4.0.1/gcc-8.3
module load intel/opencl/18.1
module load khronos/opencl/icd-loader

cmake3 -Bbuild_skl -H. -DCMAKE_BUILD_TYPE=Release  \
  -DComputeCpp_DIR=<path_to_computecpp> \
  -DOpenCL_INCLUDE_DIR=include/ \
  -DOpenCL_LIBRARY=<path_to_intel_ocl>/libintelocl.so

cmake3 --build build_skl --target clover_leaf --config Release -j 4

# OpenCL
module load openmpi/4.0.1/gcc-4.8 intel/opencl/18.1
module load cuda/10.1
OMPI_CXX=g++ OMPI_CC=gcc OMPI_F90=gfortran make \
  COMPILER=GNU OCL_VENDOR=INTEL -B -j \
  clover_leaf COPTIONS="-std=c++98" OPTIONS="-lstdc++"
sed -i 's/GPU/CPU/' clover_bm16_short.in
cp clover_bm16_short.in clover.in
module rm cuda/10.1
./clover_leaf

# OpenMP
module load intel/parallel_studio/2019.4
make COMPILER=INTEL MPI_COMPILER=mpiifort \
  C_MPI_COMPILER=mpiicc OMP_INTEL=-qopenmp
cp InputDecks/clover_bm16_short.in clover.in
OMP_NUM_THREADS=12 OMP_PROC_BIND=true mpirun -np 1 ./clover_leaf
```

A.3.3 NVIDIA RTX 2080 Ti.

```
# SYCL
module load openmpi/4.0.1/gcc-8.3
module load hipsycl-master
MPI_CXX=syclcc-clang mpic++ -std=c++17 -O3 *.cpp \
  --hipsycl-gpu-arch=sm_75 --hipsycl-platform=cuda \
  -o clover_leaf_hipsycl_cuda
./clover_leaf_hipsycl_cuda ../InputDecks/clover_bm16_short.in

# OpenCL
module load cuda/10.1
module load openmpi/4.0.1/gcc-4.8
OMPI_CXX=g++ OMPI_CC=gcc OMPI_F90=gfortran make \
  COMPILER=GNU OCL_VENDOR=NVIDIA -B -j \
  clover_leaf COPTIONS="-std=c++98" OPTIONS="-lstdc++"
sed -i 's/CPU/GPU/' clover_bm16_short.in
sed -i 's/Intel/NVIDIA/' clover_bm16_short.in
cp clover_bm16_short.in clover.in
./clover_leaf

# CUDA
module load cuda/10.1
module load openmpi/4.0.1/gcc-4.8
make -B NV_ARCH=TURING \
 CODE_GEN_TURING="-gencode arch=compute_75,code=sm_75"
cp InputDecks/clover_bm16_short.in clover.in
./clover_leaf
```