On measuring the maturity of SYCL implementations by tracking historical performance improvements

Wei-Chen Lin
Tom Deakin
Simon McIntosh-Smith
wl14928@bristol.ac.uk
Tom.Deakin@bristol.ac.uk
S.McIntosh-Smith@bristol.ac.uk
University of Bristol
Bristol, UK

ABSTRACT

SYCL is a platform agnostic, single-source, C++ based, parallel programming framework for developing platform independent software for heterogeneous systems. As an emerging framework, SYCL has been under active development for several years, with multiple implementations available from hardware vendors and others. A crucial metric for potential adopters is how mature these implementations are; are they still improving rapidly, indicating that the space is still quite immature, or has performance improvement plateaued, potentially indicating a mature market? This study presents a historical study of the performance delivered by all major SYCL implementations on a range of supported platforms. We use existing HPC-style mini-apps written in SYCL, and benchmark these on current and historical revisions of each SYCL implementation, revealing the rate of change of performance improvements over time. The data indicates that most SYCL implementations are now quite mature, showing rapid performance improvements in the past, slowing to more modest performance improvements more recently. We also compare the most recent SYCL performance to existing well established frameworks, such as OpenCL and OpenMP.

CCS CONCEPTS

• **Software and its engineering** → **Parallel programming languages**; • **Computing methodologies** → *Massively parallel and high-performance simulations*; • **Social and professional topics** → **History of software**.

KEYWORDS

SYCL, GPGPUs, performance portability, performance history, benchmarking

ACM Reference Format:
Wei-Chen Lin, Tom Deakin, and Simon McIntosh-Smith. 2021. On measuring the maturity of SYCL implementations by tracking historical performance improvements. In *International Workshop on OpenCL (IWOCL'21), April 27–29, 2021, Munich, Germany.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3456669.3456701

IWOCL'21, April 27–29, 2021, Munich, Germany

ACM ISBN 978-1-4503-9033-0/21/04…$15.00
https://doi.org/10.1145/3456669.3456701

1 INTRODUCTION

SYCL is a modern C++ based parallel programming specification by the Khronos Group. Unlike other well established independent parallel programming frameworks such as OpenMP, SYCL has deep lineage in OpenCL, an existing platform agnostic parallel programming framework based on the C99 standard.

SYCL may be seen as the spiritual successor to OpenCL, with heavy emphasis on productivity and ease of use via some higher level abstractions. As with OpenCL, the Khronos Group publishes SYCL specifications for interested parties, typically device vendors, to provide optimised implementations for running SYCL programs.

The first revision of the SYCL specification was published in March 2014, and since then the specification has received regular updates. Since the release of version 1.2 in December 2015, we saw a growing interest in fully functional SYCL implementations, with multiple implementations starting to appear. Software vendor Codeplay was the first to show promising progress, beginning with their OpenCL and SPIR based implementation named *ComputeCpp*. This was followed by a major open-source effort called hipSYCL, led by Aksel et al. from the University of Heidelberg beginning work in July 2018. Finally, Intel announced development of their unified oneAPI software package which includes a SYCL compiler named *DPC++* in December 2018. An experimental implementation called triSYCL has also been developed by Xilinx, but given its focus on FPGAs, we will not include triSYCL in our study.

Since each SYCL implementation's inception, each of them has made significant progress in terms of performance and standard adherence. At the time of writing, both Codeplay's ComputeCpp and Intel's oneAPI have declared their implementation to be production ready, although platform support has been limited to SPIR-capable OpenCL runtimes. Currently, this means only x86 CPUs and Intel GPUs are officially supported. On the other hand, hipSYCL makes use of existing compute backends such as OpenMP for CPU and LLVM's support for GPUs, and thus platform support is now quite broad.

One barrier to wider SYCL adoption is the maturity of these SYCL implementations. There are many ways this metric could be measured; as our interest is High Performance Computing (HPC),

we choose to use performance of the application at runtime as our key metric of interest. We thus are very interested in how the performance of the three main SYCL implementations has developed over time. If performance is still changing rapidly, this would indicate that the implementations are still relatively immature, and a potential adopter may decide to wait until things have settled down. If instead we see evidence of significant performance improvement in the past, but now performance changes have slowed in number and shrunk in size, this might be evidence that the implementations are reaching (performance) maturity, giving greater confidence to a potential adopter.

To help answer this question about the relative performance maturity of the main SYCL implementations, this study measures the progress they have made with respect to performance throughout their development. We select a range of representative and idiomatic HPC applications written in SYCL 1.2.1 and alternative implementations in other established frameworks and programming models.

This study makes the following contributions:

- We present a suite of performance benchmark mini-apps, implemented in SYCL and other parallel programming models, representing both compute-bound and memory-bound HPC workloads.
- We perform a historical study of performance improvements or regressions over time for all three major SYCL implementations on their supported platforms.
- We benchmark the most recent SYCL performance against other open standard parallel programming frameworks, such as OpenCL and OpenMP, on the same hardware platform..

1.1 Related work

It is often common practice for compiler implementers to maintain a collection of performance regression tests for them to measure the performance of their compilers. These are not regularly published however, and so tracking the performance of compilers externally is still incredibly useful as shown in this study. As a motivating example, the Scala compiler publishes performance data[1], however this is primarily focused on compile times rather than performance of the resulting generated code. The study of De Oliveira et al. provides a useful summary of the approaches and tools used for these regression tests [4].

SYCL by its nature is portable to a wide variety of hardware, and many of the implementations we look at target a range of hardware from multiple vendors. Therefore, the study tracks the performance of different SYCL implementations over a range of hardware from different vendors, for different applications, which greatly expands on the task at hand of tracking the performance of one compiler (or implementation) on their targets.

Our earlier work conducted a performance portability study for all major parallel programming frameworks, including SYCL, across a range of HPC hardware [6]. The study began to track the changes in performance of different programming models over a number of years, using previously published results. SYCL was not included in this historical analysis as we did not have sufficient prior results. Here we consider a more fine grained approach and track multiple versions of the compilers and runtimes.

There have been previous studies conducted on SYCL's overall performance through different mini-apps and micro-benchmarks. Lal et al. implemented SYCL-Bench, an attempt to form a standardised SYCL performance and specification conformance benchmark [11]. A more direct study focusing on comparing hipSYCL versus plain CUDA implementations using the same compute kernel was undertaken by Brian et al. [9]. These studies like many others snapshot the performance of an application and implementation toolchain at a specific point in time. However, as we show here, performance often improves (or even regresses) over time, and so such prior comparisons are limited to showcasing the performance available at their time of publication.

Tracking the historical changes of benchmarks in a systematic way is an important endeavour. One such attempt, the Top 500[2], shows the value in the wealth of historical data, especially in how the performance changes over time, across a huge variety of systems and architectures.

2 MINI-APPS IN THE STUDY

We have chosen a selection of idiomatic HPC mini-apps that are representative of typical real-world HPC workloads. There are two key considerations in our benchmark mini-app set. First, each mini-app must have existing implementations in other established parallel programming frameworks of interest, such as OpenCL and OpenMP. This is important as it helps us establish a control group for what the state of the art can attain. Second, the range of mini-apps should cover different compute characteristics, from compute-bound to memory bandwidth-bound. This allows us to characterise how well a SYCL implementation behaves under different bottleneck scenarios.

2.1 BabelSteam

BabelStream is a port of the standard STREAM memory benchmark implemented in a wide range of parallel programming frameworks, including SYCL [7]. The benchmark implements five memory bandwidth-bound kernels: Copy, Mul, Add, Triad and Dot. These kernels have very light compute requirements, and it is expected that most parallel programming frameworks should be able to attain high percentages of the theoretical peak memory bandwidth of the underlying hardware.

BabelStream has a relatively small codebase, with approximately 300 lines of SYCL code and another 600 lines of driver code (the driver is common between all the programming models). For SYCL and OpenCL, the benchmark makes no attempt at choosing the optimal work-group size; this is left to the runtime to select an optimal configuration.

BabelStream's benchmark procedure involves setting the array size, data type, and device as flags and then finally the execution of the program. The program output will indicate the *peak* memory bandwidth achieved for all five kernels. For this study, we have included results only for the Triad kernel as shown in algorithm 1, this is widely regarded as the most representative of real-world

[1] https://github.com/scala/compiler-benchmark

[2] https://www.top500.org

behaviour, requiring two loads and one store per data item that it processes.

We have selected 2^{29} elements for the array size with double precision types to maintain consistency with previous studies [6]. For Triad, this represents a total of 13GBs worth of data transfer.

Algorithm 1 BabelStream Triad kernel

```
1: procedure TRIAD(a[], b[], c[], scalar, n)   ▷ a, b, c are arrays of size n
2:     for i ← 0, n do
3:         a[i] ← b[i] + scalar ∗ c[i]
```

2.2 BUDE

The Bristol University Docking Engine (BUDE) is a molecular dynamics application that simulates the docking of different molecules to predict the structure of the resulting complex [15]. The BUDE mini-app is the latest addition to our collection of performance-portability mini-apps. The mini-app is a proxy application that implements the computationally intensive virtual screening process of the full BUDE application.

BUDE is a compute-bound mini-app. The main (only) kernel precomputes a set of transformations from a list of molecular poses which are then applied to the Cartesian product of all possible ligand (drug molecule) and protein combinations. To complete the screening process, we compute the energy for each ligand and protein combination, which involves heavy use of single-precision square root, absolute value, and general arithmetic operations in a tight loop.

Similar to CloverLeaf, BUDE's benchmark procedure requires existing input decks. We have selected the `bm1` deck which contains 26 ligands, 938 proteins, with 65,536 pose trials. For this study, we are interested in the normalised arithmetic intensity for each platform. BUDE conveniently tallies the total runtime and then derives the peak kernel FLOP/s at the end of kernel execution.

The use of hierarchical parallelism has been avoided for BUDE, and all kernel invocations use plain `cl::sycl::parallel_for` calls. This was done to avoid scheduling issues as frequently observed in OpenCL programs when run on CPUs.

2.3 CloverLeaf

CloverLeaf is a performance proxy application for computing 2D hydrodynamics via the compressible Euler equations [8]. The mini-app implements the equation via a second-order accurate method which makes use of a structured grid on the Cartesian plane along with a set of compute kernels that traverses this grid.

Like most structured-grid codes, CloverLeaf is a memory-bound mini-app. Most of the kernels are fairly simple, involving few built-in math function calls on large slices of the 2D grid. There are two kernels that require non-trivial reduction of multiple values. As the codebase predates SYCL 2020, these were implemented as a generic reduction through C++ templates and `cl::sycl::parallel_for`; the mechanisms available in SYCL 1.2.1. A particularly important property of CloverLeaf is the overall kernel count and execution flow as shown in algorithm 3 where many short compute kernels

Algorithm 2 BUDE kernel

```
1: procedure FASTEN( in xform_{3×3}[], proteins[], ligands[], i
                     out energy[])
   ▷ Values radius, distdslv, rDistdslv, phphbNz, elcdst1, elcdst,
   HARDNESS, CNSTNT are part of the simulation parameter
   specified as constant values in protein and ligand atoms

2:     il ← 0
3:     do
4:         lpos_{1×3} ← xform · ligands[il].pos_{1×3}
5:         ip ← 0
6:         do
   ▷ Calculate distance between atoms
7:             dist ← distance(lpos, proteins[ip].pos_{1×3})

   ▷ Calculate the sum of the sphere radii
8:             distbb ← dist − radius

   ▷ Calculate steric energy
9:             energy[i] ← energy[i]+
                 (1 − dist ∗ (1/radius))∗
                 (distbb < 0?2 ∗ HARDNESS : 0)

   ▷ Calculate formal and dipole charge interactions
10:            chrgE ← chrgInit∗
                 (distbb < 0.f?1 : (1 − distbb ∗ elcdst1))∗
                 (distbb < elcdst?1 : 0)
11:            energy[i] ← energy[i]+
                 (typeE? − |chrgE| : chrgE) ∗ CNSTNT

   ▷ Calculate the two cases for Nonpolar-Polar repulsive interactions
12:            dslvE = dslvInit∗
                 ((distbb < distdslv&&phphbNz)?1 : 0.f)∗
                 (distbb < 0?1 : (1 − distbb ∗ rDistdslv))

13:            energy[i] ← energy[i] + dslvE ∗ 0.5

14:            ip ← ip + 1
15:        while ip ≤ natpro                ▷ iterate over natpro
16:        il ← il + 1
17:    while il ≤ natlig                    ▷ iterate over natlig
```

are executed in a tight loop. This is representative of typical HPC applications and allows us to compare kernel scheduling overheads.

CloverLeaf's benchmark procedure requires selection of an input deck. The input deck specifies the size of the grid, initial parameters for the simulation, and the iteration count. We have selected the `clover_bm16.in` deck which sets up a 3840×3840 grid for 2955 iterations, to be consistent with other benchmark studies using this mini-app [5, 12–14]. We measure the total runtime for the simulation as output.

Algorithm 3 High-level CloverLeaf kernel overview

▷ Each procedure may invoke multiple kernels in succession
procedure IDEAL_GAS
▷ Compute the pressure and sound speed via ideal gas equation of state with a fixed gamma
procedure VISCOSITY
▷ Compute artificial viscosity via the Wilkin's method to smooth out shock front and prevent oscillations
procedure PDV
▷ Compute delta in energy and density in a cell via velocity gradients
procedure CALC_DT
▷ Compute the minimum timestep based on CFL conditions, velocity gradient, and velocity divergence.
procedure ACCELERATE
▷ Update velocity field via cell pressure and viscosity gradients
procedure FLUX_CALC
▷ Compute edge volume fluxes based on the velocity fields
procedure ADVECTION
▷ Setup fields for the next iteration
procedure RESET_FIELD
▷ Compute edge volume fluxes based on the velocity fields
procedure FIELD_SUMMARY
▷ Compute the total mass, internal energy, kinetic energy and volume weighted pressure
procedure HYDRO
    while $step < maxStep$ **do**
        ideal_gas() ▷ timestep
        viscosity()
        calc_dt()
        PdV()
        accelerate()
        PdV()
        flux_calc()
        advection()
        reset_field()
        field_summary()
        $step \leftarrow step + 1$

3 SYCL IMPLEMENTATIONS

We adopted the main three actively developed SYCL implementations for this study. We wanted to undertake a historical study, evaluating the performance of our benchmarks through time as each SYCL implementation was developing. This is in order to gain some insight as to whether the implementations have started to stabilise at a performance plateau, or whether they were still rapidly changing in terms of performance. To gather these multiple data points through time, we need access to multiple versions of each SYCL implementation.

For proprietary SYCL implementations, we attempt to collect all public releases directly from the vendor. For the open-source SYCL implementations, we have more availability in terms of available versions of the implementations, so we chose to select revisions at specific intervals, as described below.

3.1 ComputeCpp

ComputeCpp[3] is Codeplay's proprietary, LLVM derived, SYCL conformant implementation; it is one of the earliest SYCL implementations. Codeplay started development of a prototype SYCL implementation, *SYCLONE*, as early as 2014. SYCLONE left prototype stage around Q4 2015 and was renamed to ComputeCpp with commercial support.

The implementation supports any hardware platform that has a SPIR capable OpenCL runtime. Currently, only Intel's OpenCL runtime is officially supported, meaning ComputeCpp will only run on x86 CPUs and Intel GPUs at the time of writing as shown in table 1. Intel's OpenCL runtime implementation makes use of Intel Threading Building Block (TBB) to support parallelism on the CPU. For GPUs, the runtime simply ingests a SPIR binary and executes the program.

For a time, ComputeCpp was the only production-ready SYCL implementation; many SYCL mini-apps, including the ones used for this study, were first tested using ComputeCpp due to a lack of alternatives. More recently, Codeplay has been focusing effort on implementation of the SYCL 2020 specification.

As ComputeCpp is proprietary software, revisions are selected based on their publicly released versions. Codeplay has made all past versions of ComputeCpp available for download on their website. We have selected all SYCL 1.2.1 compliant releases, starting from 0.9.1 (released July 2018) to their latest 2.3.0 (released November 2020) release, capturing 20 versions of ComputeCpp through 2 years and 3 months worth of development.

3.2 DPC++

DPC++[10] is Intel's fork of LLVM which adds language-level SYCL support. Similar to ComputeCpp, DPC++ requires Intel's own CPU and GPU OpenCL runtime and follows the same execution model. Alternatively, it can also run on Intel's Level Zero API as shown in table 1.

DPC++ has been under heavy development since Q1 2019. Beginning from December 2020, the DPC++ compiler has reached the production-ready stage and has been shipped as part of oneAPI. Similar to ComputeCpp, recent revisions are focused on aligning Intel's SYCL extensions with the SYCL 2020 specification.

Intel conveniently provides nightly binary builds on GitHub. For revision selection, we select nightly snapshots that are two weeks apart. As Intel only started providing usable nightly builds since March 2020, we have tracked the short term improvements up until January 2021, totalling 16 snapshots across 10 months of development. Intel has also made regular releases of the OpenCL CPU runtime, to be paired with their TBB library. We have tested each revision of the compiler with all releases of the OpenCL CPU runtime to track improvements on both fronts.

3.3 hipSYCL

hipSYCL[1] is an open-source implementation of SYCL based on LLVM that supports multiple backends. The effort is led by Aksel et al. from the University of Heidelberg.

The breadth of platform support has been an ongoing weakness for most SYCL implementations; hipSYCL's multi-backend design allows it to support the widest range of hardware platforms. For CPUs, hipSYCL directly translates to OpenMP, thus being able to take advantage of all existing LLVM optimisations on this front. For GPUs, hipSYCL acts as a wrapper for SYCL function calls which abstract on top of vendor-specific frameworks such as CUDA or HIP. Finally, hipSYCL is continuously adding new backends. The latest experimental backend being another SYCL implementation: DPC++, which expands platform support to all platforms supported by DPC++ as shown in table 1.

The project started in July 2018 and is currently under heavy development, with three stable milestone releases over the years. To track performance improvements of hipSYCL, we select revisions by git commits roughly 2 months apart, starting with the first stable release 0.8.0 (released September 2019), ending with the latest commit at January 2021, capturing 10 snapshots across 1 year and 4 months of development. Due to hipSYCL's development process, if a chosen commit happens to be part of an ongoing pull request, any future commits selected will be part of that branch when possible. For each commit, we build and sanity check the resulting compiler by compiling sample SYCL programs.

Table 1: Platform support for selected SYCL implementations

Platform \ SYCL	DPC++	ComputeCpp	hipSYCL
OpenMP (CPU)	●	○	●
OpenCL SPIR (CPU/GPU)	●	●	◐[a]
Intel Level Zero (CPU/GPU)	●	○	◐[a]
Nvidia (GPU)	◐[b]	◐[c]	●[d]
AMD ROCm (GPU)	○	○	●

[a] Built on DPC++, experimental and work in progress
[b] PTX, experimental
[c] PTX, experimental, incomplete and discontinued
[d] CUDA

4 HARDWARE AND SOFTWARE CONFIGURATION

All mini-apps are compiled with the highest possible optimisation level, typically `-Ofast` or `-O3`. Unless otherwise mentioned, compiler flags used for each compiler and platform are constant throughout all tested versions.

4.1 Hardware platforms

We select devices that are directly supported by our SYCL implementations as shown in Table 2. All figures below reference the short name of each platform for brevity.

For CPU devices, we have selected commonly used HPC CPUs from both Intel the AMD. Both the Intel Xeon CPU and the AMD Zen2 CPU are installed in a dual socket configuration. We utilise all available processors during our benchmark with the most performant scheduling and affinity options. For OpenMP, these are set via the `OMP_PLACES` and `OMP_PROC_BIND` variables. For frameworks that are not NUMA aware, we used `numactl` with the appropriate flags.

For GPU devices, we are constrained what the SYCL implementations have support for. NVIDIA's V100 GPU was selected for its support in hipSYCL. Both DPC++ and ComputeCpp have experimental support for NVIDIA devices, albeit only on very recent releases for both compilers and with varying levels of completeness as shown in table 1. Because of this, we have decided to only include NVIDIA GPUs for hipSYCL at this time, hoping to add the others to a future version of this study. To allow comparison between DPC++ and ComputeCpp on the GPU front, we have selected Intel's P630 iGPU. This particular iGPU is part of a Xeon CPU which is a representative configuration of HPC and data centre use cases.

For AMD Zen2 CPU, Intel Xeon CPU, and NVIDIA V100 GPU, we have used the Isambard Multi-Architecture Comparison System (MACS)[3] which features a wide range of HPC hardware nodes for testing. For Intel GPUs, we have used Intel's oneAPI DevCloud service. DevCloud provides a selection of Intel CPU and GPU nodes that can be accessed remotely.

4.2 Software platforms

For CPU platforms, we have installed all required versions of SYCL compilers and runtime locally, either as binary from the vendors or compiled directly from source. For cases where LLVM is a build dependency — as is the case with hipSYCL on non-CPU platforms— we have selected the release version of LLVM 10.0 across all benchmarks. The compiler used for compiling alternative frameworks is GCC 10.2.0 and GCC 8.1.0 for CUDA based projects. LLVM does not ship with libstdc++ and so it needs an existing GCC installation to provide this. We used GCC 8.2.0 as the toolchain for LLVM (specified via the `--gcc-toolchain` flag), to avoid the problem where HPC environments such as Isambard provide older versions of GCC which can cause compile errors due to old `libstdc++` headers.

On NVIDIA nodes, we use CUDA toolkit version 10.1.243 with driver version 440.64. For compiling OpenMP targeting GPUs, we used Cray CCE 10.0.0. For Intel GPUs, the GPU drivers preinstalled on the DevCloud reports itself as OpenCL 3.0 NEO 20.46.18421.

5 RESULTS

In the following section we will present the historical performance data we have collected for our SYCL-based benchmarks running on versions of ComputeCpp, DPC++ and hipSYCL stretching back almost two years.

In the results, where presented as normalised runtimes, these are normalised for each hardware platform individually, not just once per chart. Thus for the three lines typically shown, one per SYCL implementation, each line should have at least one data point normalised to 1.0.

We have used different array sizes for the BabelStream benchmark on Intel's iGPU as it was unable to support sizes larger than 2^{29} due to software limitations.

[3] https://gw4-isambard.github.io/docs/user-guide/MACS.html

Table 2: Platform details

Name	Architecture	Short name	Device Type	Peak Mem. BW (GB/s)	Peak FP32 FLOP/s (GFLOP/s)
NVIDIA Tesla V100	Volta	v100-isambard	Discrete GPU	900	14000
Intel UHD P630 (Intel Xeon E2176G)	Gen9.5	uhdp630-devcloud	Integrated GPU + CPU	42.6	460
Intel Xeon Gold 6230 (20-cores)	Cascade Lake	cxl-isambard	HPC CPU (2-socket)	281.6	4096
AMD EPYC 7742 (64-cores)	Zen2 (Rome)	rome-isambard	HPC CPU (2-socket)	409.6	9216

5.1 ComputeCpp

Early on in our study, we discovered that only the most recent version of Intel's OpenCL runtime for CPU (2020.11.11.0.04) was able to produce results; older versions experienced segmentation faults or deadlocks upon kernel submission. The root cause of this problem seems to stem from incompatibility with older system libraries.

BabelStream results, as presented in Figure 1 showed a relatively consistent runtime throughout all ComputeCpp versions, but with a significant jump in performance in July 2019 from version 1.1.4 onwards. The release notes for that particular version only mentioned general bug fixes and build improvements, so it is not clear what was behind this performance increase.

What is interesting here is the comparatively lower bandwidth achieved on both Intel CPU and Intel GPUs compared to AMD CPUs prior to the step-change in July 2019, given the reliance on Intel's own OpenCL runtime. The difference mostly disappeared at the same time as the performance improvements arrived in version 1.1.4. The overall bandwidth achieved recently on each platform is around 70% of what alternative frameworks has achieved, which while not exceptional (we've achieved up to 85% of peak memory bandwidth in other scenarios), shows the ComputeCpp implementation is able to achieve a relatively high fraction of the theoretical peak memory bandwidth on all tested platforms.

Turning to results for CloverLeaf, as shown in Figure 2, these are only partially complete as several versions of ComputeCpp that we tested has crashed at runtime for the Intel iGPU. Over time, ComputeCpp showed a significant improvement in runtime for CloverLeaf on CPUs: up to an 80% improvement for Cascade Lake and 34% for Rome. CloverLeaf's built-in kernel profiler showed uniform improvements across all compute kernels, the improvements are proportionate to the overall reduction in runtime. As CloverLeaf's kernels are all unique and covers both iterations and reductions, the improvements observed here should be universally applicable.

Finally, results for BUDE on ComputeCpp are shown in Figure 3. We see a series of performance regressions, with a performance impact of up 20%, spanning nearly a year, starting in January 2020 with version 1.2.0 and only fixed after the release of 2.2.1 in October 2020. One potential cause of this is the transition of ComputeCpp to LLVM 7.0 in 1.2.0, and then on to 8.0 in 1.3.0, as described by ComputeCpp's release notes. The two versions that then carried the regression, 2.0.0 and 2.1.0, mainly focused on SYCL 2020 implementations such as USM, shedding light on why the performance regression persisted for so long.

As ComputeCpp is using Intel's OpenCL runtime, there is a possibility that Intel would have made certain architecture-based optimisations or assumptions for their own hardware that might result in inefficiencies when used on other x86 processors. This is however not obviously the case, as we see a near constant 15% higher fraction of peak FLOP/s achieved on AMD's platform. In part this may reflect the challenge of efficiently exploiting the wider vectors on the latest Intel platforms compared to the latest AMD platforms (512-bit vs 256-bit).

Overall for the ComputeCpp SYCL implementation, we see generational improvements, and occasional regressions on the CPU side. The performance on (Intel) GPUs remained mostly stable with one moderate improvement in memory access benefiting BabelStream in version 1.1.4. One can conclude that ComputeCpp is now relatively mature, and its performance is not now changing significantly or often.

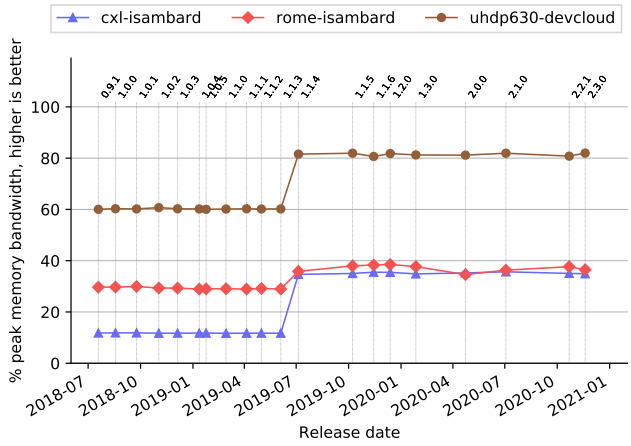

Figure 1: BabelStream on ComputeCpp results

5.2 DPC++

DPC++ follows the same execution model of ComputeCpp (dependency on Intel's OpenCL runtime), which means it also suffered from the same problems we encountered while testing with older CPU runtimes. This section only discusses results produced by the latest version (2020.11.11.0.04) of the CPU runtime.

Results for BabelStream on DPC++ are presented in Figure 4. Results for all platforms have remained remarkably consistent during DPC++'s lifetime, with the Intel GPU runs consistently achieving nearly 80% of the peak memory bandwidth, while both the Intel

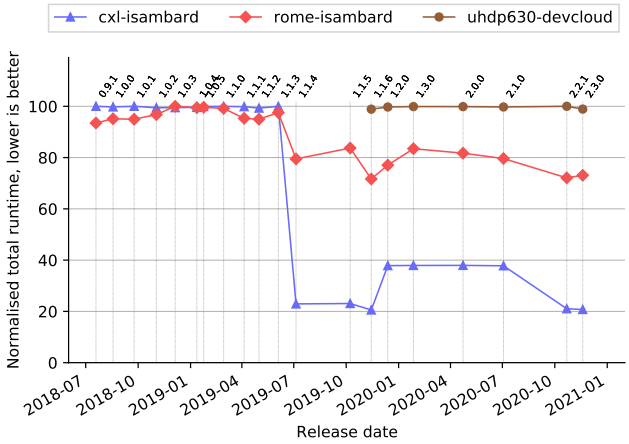

Figure 2: CloverLeaf on ComputeCpp results

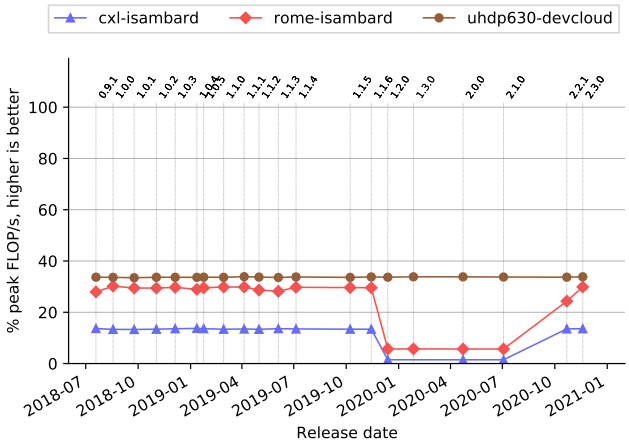

Figure 3: BUDE on ComputeCpp results

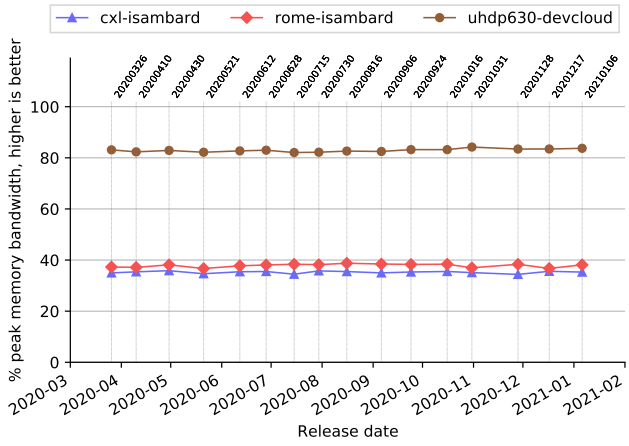

Figure 4: BabelStream on DPC++ results

and AMD CPU results are closer to 40%. The lower bandwidth observed here are consistent with larger problem sizes used in previous studies[6]. These results suggest high levels of maturity for DPC++ on purely memory bandwidth-bound kernels.

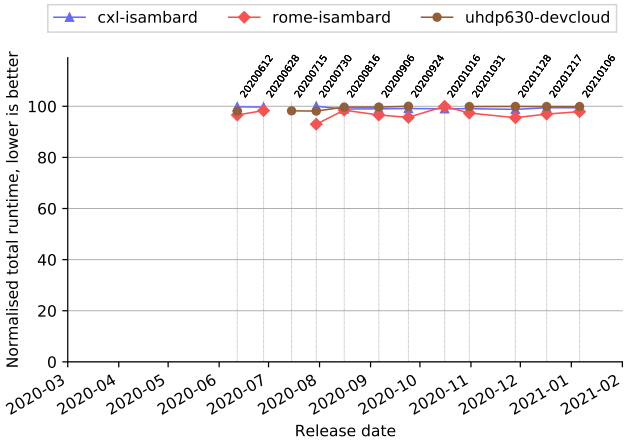

Figure 5: CloverLeaf on DPC++ results

Similar to results for BabelStream, DPC++ CloverLeaf results are shown in Figure 5. We again see a consistent runtime across all versions of DPC++ for all supported platforms. This is within expectation as both BabelStream and CloverLeaf are memory bandwidth-bound.

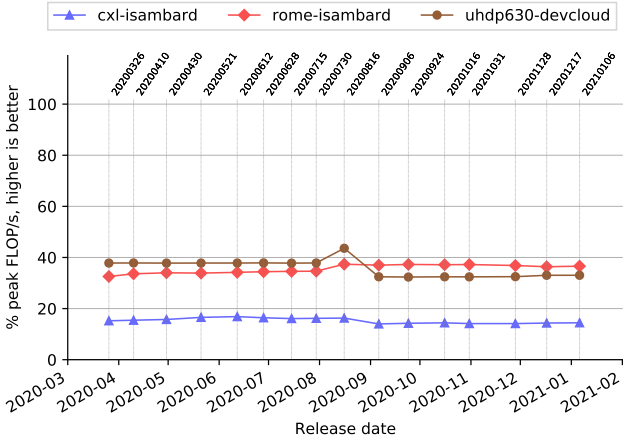

Figure 6: BUDE on DPC++ results

Results for BUDE are given in Figure 6. We observe a small improvement on the AMD platform but minor regressions on Intel platforms from the nightly build in mid August 2020 (`20200816`) onwards. The release notes point to the addition of *Dead Argument Elimination* optimisations (commit `b0d98dc` and `f53ede9`) which were added during that time. Impressively, the SYCL version of BUDE achieves nearly 40% of peak single-precision floating-point

on AMD's Rome CPU when using the Intel DPC++ SYCL implementation.

Overall, DPC++ has shown to be a highly consistent throughout the short development span across 10 months with no major performance regressions or improvements for our benchmarks during this time. This may suggest the implementation has reached a potential equilibrium in terms of performance on the compiler side. The commit history showed active development on the following fronts:

- Cross-platform capabilities as DPC++ is part of oneAPI which also supports Windows and macOS.
- Continued work on adherence to SYCL 2020.
- Support for alternative backends such as CUDA and FPGA.

With this in mind, we expect further performance improvements to come from both the compiler and runtime once work on platform support and spec adherence reaches a milestone.

5.3 hipSYCL

Results for BabelStream using hipSYCL are shown in Figure 7. The commit (`81b750e`) leading up to the 0.9 release (`cff515c`) saw the addition of the new *boost-fiber*-based scheduler for hipSYCL's OpenMP backend. We see an encouraging performance uplift following those commits on CPU platforms. Performance for the CUDA backend seems to have reached maturity, with the results showing higher than 90% of the peak bandwidth.

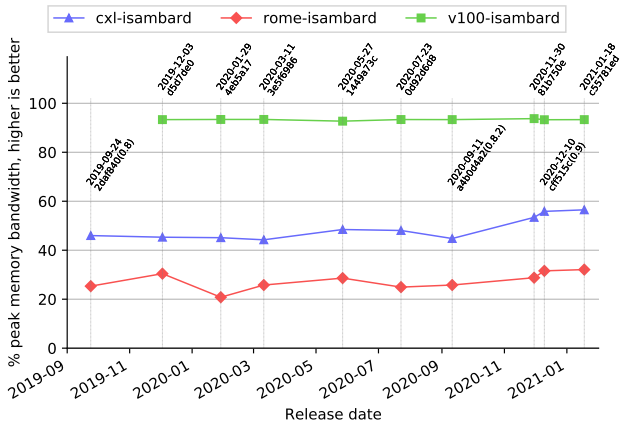

Figure 7: BabelStream on hipSYCL results

Similar to ComputeCpp, we see frequent build failures and segfaults on certain hipSYCL versions and platforms for CloverLeaf, hence the missing results further back in the timeline. Revisions after 0.8.2 showed promising performance improvements of between 20% and 40% on NVIDIA GPU and Intel/AMD CPUs, respectively. However, the results shown were for runs which did not all pass CloverLeaf's built-in verification; CloverLeaf's verification passed for all versions of ComputeCpp and DPC++.

Further investigation found that the failure was caused by the two reduction kernels responsible for computing values directly required for the verification. We have contacted the repository owner which promptly resolved the issue. The root cause of the

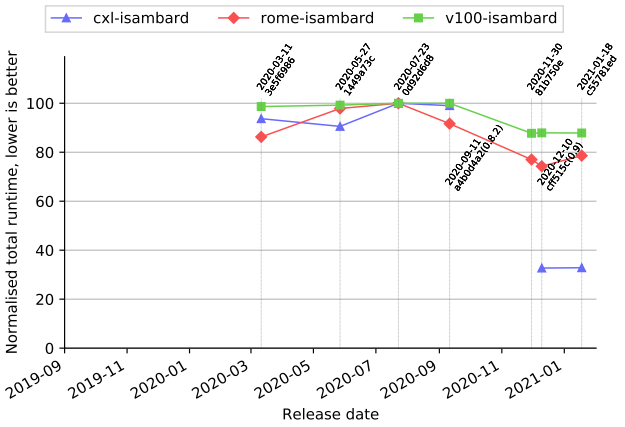

Figure 8: CloverLeaf on hipSYCL results

issue stems from an optimisation bug where hipSYCL eliminates dependency edges between kernels to achieve parallel execution. The bug was fairly obscure due to how the reduction kernel was written: CloverLeaf's reduction functions are implemented in a generic way to abstracts away implementation details of the tree reduction commonly seen in OpenCL. The implementation makes heavy use of C++ templates for capturing accessors, resulting in the same buffer being captured multiple times with different access modes. This has caused hipSYCL to derive dependency edges incorrectly based on the order of accessor constructions, thus eliminating real data dependencies in the reduction step. At the time of writing, this issue[2] has been resolved in upstream and will be included in the upcoming 0.9.1 release.

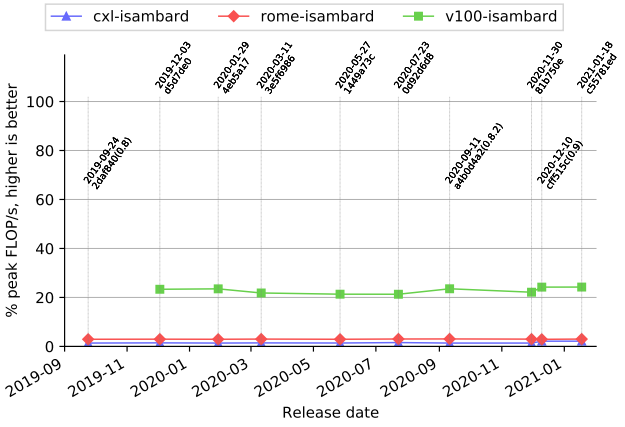

Figure 9: BUDE on hipSYCL results

Performance for BUDE on hipSYCL has remained low throughout all snapshots as shown in Figure 9. This result suggests poor interaction between LLVM's OpenMP backend and hipSYCL. The NVIDIA GPU performance is much better, however, achieving 20% of the V100's peak single-precision floating-point capability.

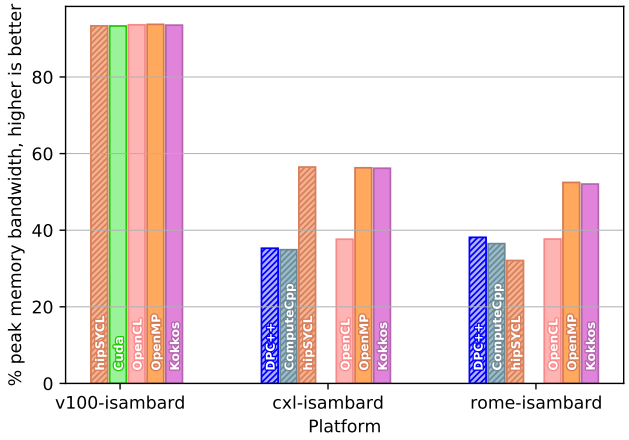

Figure 10: BabelStream: SYCL vs alternative framework results

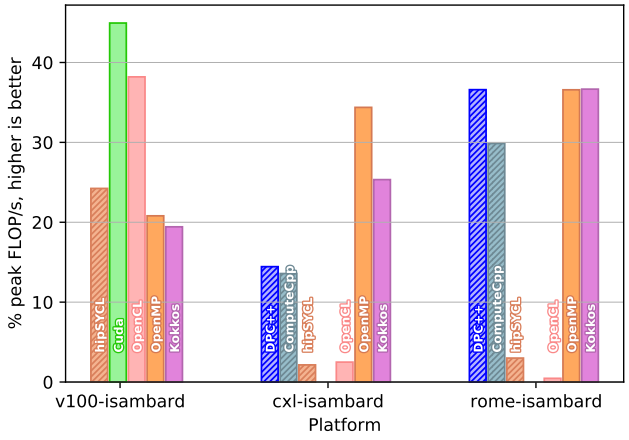

Figure 11: BUDE: SYCL vs alternative framework results

5.4 Alternative SYCL implementations

Finally, we have compared the latest results for all three of our studied SYCL implementations versus contemporary parallel programming frameworks. We compare SYCL performance to the following frameworks: OpenCL, OpenMP, CUDA, and Kokkos. OpenCL, OpenMP, and CUDA are all C99 derived frameworks that require direct compiler support. Kokkos is a C++ derived framework that is implemented as a library with no specific compiler requirements (although runtime requirements such as CUDA libraries are still required).

Results for the memory bandwidth-bound BabelStream benchmark are shown in Figure 10. The bandwidth achieved on each platform are in line with existing literature[6]. As expected, the performance profile of SYCL implementations that depended on Intel's OpenCL runtime is very similar to OpenCL's, which share the same runtime environment. Given the higher abstraction level of SYCL, it is encouraging to see performance parity with OpenCL which focuses more on exposing low-level controls.

For independent implementations such as hipSYCL, we see highly competitive performance compared to vendor specific APIs like CUDA. Because hipSYCL on the CPU uses LLVM's OpenMP backend, we also observe performance parity with the OpenMP implementation. We attribute hipSYCL's lower performance on AMD Rome to LLVM 10's immaturity on the Zen2 architecture. As later revision of hipSYCL supports more up-to-date LLVM releases, future versions of this study will attempt to accommodate for this.

The results for the compute-bound BUDE benchmark are shown in Figure 11. Performance disparity on Intel Cascade Lake and AMD Rome suggests room for improvement for compute-bound operations. In particular, hipSYCL performed very poorly compared to the OpenMP implementation. This is surprising and requires deeper analysis on the emitted code. Results from OpenCL is also surprising as the implementation shares the same runtime with DPC++ and ComputeCpp. Considering BUDE's OpenCL kernel versus the SYCL kernel, which is nearly identical, this disparity likely originates from the code emitted by Intel's OpenCL runtime online compiler. This isn't a problem for DPC++ or ComputeCpp because the SPIR instructions are generated ahead of time.

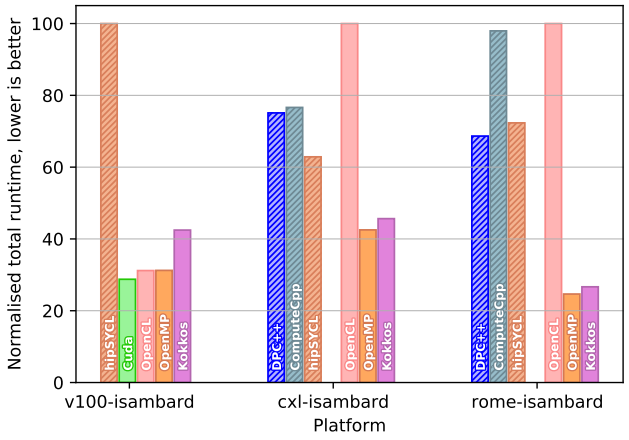

Figure 12: CloverLeaf: SYCL vs alternative framework results

The results for the memory-bound CloverLeaf benchmark are shown in Figure 12. Initial analysis suggests hipSYCL has a relatively high kernel invocation overhead on CUDA platforms. This is less obvious in both BabelStream and BUDE as both only invoke a single kernel with no complex kernel dependency requirements and at a much lower frequency. On the other hand, CloverLeaf contains more than 170 unique kernels with complex dependencies which is called in a tight loop for up to 2955 iterations as shown previously in algorithm 3.

OpenCL once again performed poorly, we observe a ~25% regression from SYCL implementations hosted on the same runtime. SYCL results are closer to what we previously observe in BabelStream although with a further 20~40% regression. Based on CloverLeaf's built-in profiling data, we suspect the regression again stems from

kernel overhead which is amplified by the amount of complex buffer dependencies between kernels.

6 CONCLUSION

This study has found that all major SYCL implementations are approaching production-readiness from incremental performance improvements made throughout the last few years. The two vendor-backed SYCL implementations, DPC++ and ComputeCpp, both showed competitive performance compared to alternative frameworks such as OpenMP and OpenCL.

In many cases, we see regressions happen at various points in time and persist for several versions. This can be partially mitigated by switching to a different SYCL implementation, further highlighting the advantage of adopting an open-standard programming model supported by multiple implementations.

Performance for compute-bound applications has largely remained consistently low for all implementations, as demonstrated with results from the BUDE mini-app. DPC++ was the exception here.

The CloverLeaf mini-app has presented a challenge for all SYCL implementations. Being the largest codebase, totalling 8,000 LOC with the highest kernel count, both ComputeCpp and hipSYCL had difficulty compiling the code, or produced binaries that crash at runtime. DPC++ only suffered a single crash when running on CPUs, further asserting its robustness. CloverLeaf also helped revealed a potential bottleneck in terms of kernel overhead; of all the frameworks benchmarked, SYCL provides a relatively straightforward method of declaring data dependencies through accessors, this may have caused extra burden on the runtime in an effort to extract more parallelism.

To this end, we hope this study contributes to the improved stability and robustness of all SYCL implementations. The mini-apps presented in this paper could serve as a viable baseline for SYCL regression detection, both in performance and general usability. Based on the findings with BUDE, we also expect further optimisations on compute-bound applications.

On the productivity front, we are excited to see all SYCL implementations actively adopting SYCL 2020 features. All implementations are already supporting major SYCL 2020 features such as reductions, in-order queues, and USM.

Finally, the narrow platform support for SYCL has been a major factor in slow adoption. We have seen strong evidence in the improved platform support as shown in DPC++'s commit history. With open source implementations such as hipSYCL also approaching a production ready state, we expect the situation to further improve. Overall we conclude that, given the relative maturity of the three main SYCL implementations, potential adopters of SYCL can now start to seriously consider it as an option.

ACKNOWLEDGMENTS

This work makes use of Intel's DevCloud online cluster service available for developers at https://intelsoftwaresites.secure.force.com/devcloud/oneapi. This work used the Isambard UK National Tier-2 HPC Service (http://gw4.ac.uk/isambard/) operated by GW4 and the UK Met Office, and funded by EPSRC (EP/P020224/1).

REFERENCES

[1] Aksel Alpay. 2021. hipSYCL. https://github.com/illuhad/hipSYCL.
[2] Aksel Alpay. 2021. Regression with hipSYCL >= 0.8.2 for hand-rolled tree reduction. https://github.com/illuhad/hipSYCL/issues/464.
[3] Codeplay. 2021. ComputeCpp™Community Edition. https://developer.codeplay.com/products/computecpp/ce/home/.
[4] A. B. De Oliveira, S. Fischmeister, A. Diwan, M. Hauswirth, and P. F. Sweeney. 2017. Perphecy: Performance Regression Test Selection Made Simple but Effective. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 103–113. https://doi.org/10.1109/ICST.2017.17
[5] Tom Deakin, Simon McIntosh-Smith, James Price, Andrei Poenaru, Patrick Atkinson, Codrin Popa, and Justin Salmon. 2019. Performance portability across diverse computer architectures. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 1–13.
[6] Tom J Deakin, Andrei Poenaru, Tom Lin, and Simon N Mcintosh-Smith. 020. Tracking Performance Portability on the Yellow Brick Road to Exascale. In *Proceedings of the Performance Portability and Productivity Workshop P3HPC*. nstitute of Electrical and Electronics Engineers (IEEE), United States.
[7] 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.
[8] J. A. Herdman, W. P. Gaudin, S. McIntosh-Smith, M. Boulton, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis. 2012. Accelerating Hydrocodes with OpenACC, OpenCL and CUDA. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. 465–471. https://doi.org/10.1109/SC.Companion.2012.66
[9] Brian Homerding and John Tramm. 2020. Evaluating the Performance of the HipSYCL Toolchain for HPC Kernels on NVIDIA V100 GPUs. In *Proceedings of the International Workshop on OpenCL (IWOCL '20)*. Association for Computing Machinery, New York, NY, USA, Article 16, 7 pages. https://doi.org/10.1145/3388333.3388660
[10] Intel. 2021. LLVM. https://github.com/intel/llvm.
[11] Sohan Lal, Aksel Alpay, Philip Salzmann, Biagio Cosenza, Nicolai Stawinoga, Peter Thoman, Thomas Fahringer, and Vincent Heuveline. 2020. SYCL-Bench: A Versatile Single-Source Benchmark Suite for Heterogeneous Computing. In *Proceedings of the International Workshop on OpenCL (IWOCL '20)*. Association for Computing Machinery, New York, NY, USA, Article 10, 1 pages. https://doi.org/10.1145/3388333.3388669
[12] Andrew Mallinson, David A Beckingsale, WP Gaudin, JA Herdman, JM Levesque, and Stephen A Jarvis. 2013. Cloverleaf: Preparing hydrodynamics codes for exascale. *The Cray User Group* 2013 (2013).
[13] Matt Martineau, Simon McIntosh-Smith, Mike Boulton, and Wayne Gaudin. 2016. An evaluation of emerging many-core parallel programming models. In *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores*. 1–10.
[14] 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 *International Supercomputing Conference*. Springer, 53–75.
[15] Simon McIntosh-Smith, James Price, Richard B Sessions, and Amaurys A Ibarra. 2015. High performance in silico virtual drug screening on many-core processors. *The international journal of high performance computing applications* 29, 2 (2015), 119–134.

A REPRODUCIBILITY

This section details the source repository and build procedure of each mini-app used to obtain the results in Section 5.

A small Scala based benchmarking application available on GitHub at https://github.com/UoB-HPC/sycl_performance_history. The application automates source code retrieval, compilation, and job submission to the job management systems or each platform. The repository contains build instructions and examples of use. The mini-apps used in this paper are available in source form from public GitHub repositories:

- BabelStream - https://github.com/UoB-HPC/BabelStream
- Bude - https://github.com/UoB-HPC/bude-portability-benchmark
- CloverLeaf (OpenMP Target) - https://github.com/UoB-HPC/cloverleaf_openmp_target

- CloverLeaf (Kokkos) - https://github.com/UoB-HPC/cloverleaf_kokkos
- CloverLeaf (SYCL) - https://github.com/UoB-HPC/cloverleaf_sycl
- CloverLeaf (CUDA) - https://github.com/UK-MAC/CloverLeaf_CUDA
- CloverLeaf (OpenCL) - https://github.com/UK-MAC/CloverLeaf_OpenCL
- CloverLeaf (OpenMP) - https://github.com/UK-MAC/CloverLeaf_ref

The following section summarises the compilation and execution commands used on each platform.

A.1 Offline compiler preparation

Create the `<root>` directory for source code and also the following directories for compiler packages: `<oclcpuexp>`, `<dpcpp>`, `<computecpp>`. For CloverLeaf, we will be using oneAPI's MPI library, install the oneAPI Base Toolkit from https://software.intel.com/content/www/us/en/develop/tools/oneapi/base-toolkit.html and set the installation directory to `<oneapi_root>`

For ComputeCpp, registration is required to obtain compiler binaries. Download the required versions manually from the drop-downs on https://developer.codeplay.com/products/computecpp/ce/download and extract them to the `computecpp` directory. For DPC++ and the CPU runtimes, nightliy binaries are available on GitHub at https://github.com/intel/llvm/releases, manually download and extract the required versions. Place CPU runtimes in `oclcpuexp` and DPC++ compilers in the `dpcpp` directory.

A.1.1 hipSYCL. The following script illustrates how to build select revisions of hipSYCL for Isambard MACS. Make adjustment for system paths and installation paths as required.

```
module load cmake/3.12.3 gcc/10.2.0 git
module load boost/1.73.0/gcc-10.2.0
module load llvm/10.0

BRANCH="stable"
INSTALL_DIR="<hipsycl_root>"
git clone --recurse-submodules \
  -b $BRANCH \
  https://github.com/illuhad/hipSYCL.git

# checkout required commit here (e.g git checkout <commit_hash>)

cmake -Bbuild -H. \
    -DCMAKE_C_COMPILER="$(which␣gcc)" \
    -DCMAKE_CXX_COMPILER="$(which␣g++)" \
    -DCMAKE_INSTALL_PREFIX="$INSTALL_DIR" \
    -DWITH_CUDA_BACKEND=ON \
    -DWITH_ROCM_BACKEND=OFF \
    -DWITH_CPU_BACKEND=ON
cd build
make -j $(nproc)
make install
```

A.2 BabelSteam

```
# Load modules on Isambard MACS only:
module load cmake/3.18.3
module load gcc/8.2.0

export LD_LIBRARY_PATH=\
  "<oclcpu>/oclcpuexp.<ver>/x64:${LD_LIBRARY_PATH:-}"
export LD_LIBRARY_PATH=\
  "<oclcpu>/tbb.<ver>/lib/intel64/gcc4.8:${LD_LIBRARY_PATH:-}"
export OCL_ICD_FILENAMES=\
  "<oclcpu>/oclcpuexp.<ver>/x64/libintelocl.so"

git clone -b "computecpp_fix" \
  "https://github.com/UoB-HPC/BabelStream.git" \
  <builddir>

cd <builddir>

# ComputeCpp
make -f SYCL.make \
  COMPILER="COMPUTECPP" \
  TARGET="CPU" \
  SYCL_SDK_DIR="<computecpp>/computecpp-.<ver>" \
  EXTRA_FLAGS="␣\
-DCL_TARGET_OPENCL_VERSION=220␣\
-D_GLIBCXX_USE_CXX11_ABI=0␣\
-I"<ocl_headers>/include"␣\
-L"<oclcpu>/oclcpuexp.<ver>/x64"␣\
--gcc-toolchain=$(realpath␣"$(dirname "$(which␣gcc)")"/..)␣\
"

# DPC++
make -f SYCL.make \
  COMPILER="DPCPP" \
  TARGET="CPU" \
  SYCL_DPCPP_CXX="<dpcpp>/dpcpp-compiler.<ver>/bin/clang++" \
  SYCL_DPCPP_INCLUDE="-I<dpcpp>/dpcpp-compiler.<ver>/include/sycl" \
  EXTRA_FLAGS="␣\
-DCL_TARGET_OPENCL_VERSION=220␣\
-fsycl␣\
-march=<skylake-avx512|skylake>␣\
--gcc-toolchain=$(realpath␣"$(dirname "$(which␣gcc)")"/..)␣\
"

# hipSYCL
# bring hipSYCL installation to path (i.e make syclcc available)
make -f SYCL.make \
  COMPILER="HIPSYCL" \
  SYCL_SDK_DIR="$(dirname␣"$(which syclcc)")/.." \
  EXTRA_FLAGS="␣\
-march=<skylake-avx512|skylake>␣\
--gcc-toolchain=$(realpath␣"$(dirname "$(which␣gcc)")"/..)␣\
"

SUBSTRING="Xeon" # select platform name
export DEVICE=\
  $("./sycl-stream" --list | \
   grep -a SUBSTRING | \
```

```
    cut -d ':' -f1 | head -n 1)

  ./sycl-stream --device $DEVICE --arraysize 536870912
```

A.3 CloverLeaf

```
# Load modules on Isambard MACS only:
module load cmake/3.18.3
module load gcc/8.2.0

export LD_LIBRARY_PATH=\
  "<oclcpu>/oclcpuexp.<ver>/x64:${LD_LIBRARY_PATH:-}"
export LD_LIBRARY_PATH=\
  "<oclcpu>/tbb.<ver>/lib/intel64/gcc4.8:${LD_LIBRARY_PATH:-}"
export OCL_ICD_FILENAMES=\
  "<oclcpu>/oclcpuexp.<ver>/x64/libintelocl.so"

git clone -b "sycl_history" \
  "https://github.com/UoB-HPC/cloverleaf_sycl.git" \
  <builddir>

cd <builddir>

# DPC++
cmake -B"<root>/build" -H. \
  -DCMAKE_BUILD_TYPE=Release \
  -DSYCL_RUNTIME="DPCPP" \
  -DDPCPP_BIN="<dpcpp>/dpcpp-compiler.20200816/bin/clang++" \
  -DDPCPP_INCLUDE="<dpcpp>/dpcpp-compiler.20200816/include/sycl" \
  -DCMAKE_C_COMPILER="gcc" \
  -DCMAKE_CXX_COMPILER="g++" \
  -DMPI_AS_LIBRARY="ON" \
  -DMPI_C_LIB_DIR="<oneapi_root>/mpi/2021.1.1/lib" \
  -DMPI_C_INCLUDE_DIR="<oneapi_root>/mpi/2021.1.1/include" \
  -DMPI_C_LIB="<oneapi_root>/mpi/2021.1.1/lib/release/libmpi.so" \
  -DCXX_EXTRA_FLAGS="
-fsycl
-march=skylake-avx512
--gcc-toolchain=$(realpath␣"$(dirname "$(which␣gcc)")"/..)
"

# ComputeCpp
cmake -B"<root>/build" -H. \
  -DCMAKE_BUILD_TYPE=Release \
  -DSYCL_RUNTIME="COMPUTECPP" \
  -DComputeCpp_DIR="<computecpp>/computecpp-<ver>-centos-64bit" \
  -DOpenCL_LIBRARY="<oclcpu>/oclcpuexp.<ver>_rel/x64/libOpenCL.so"
  -DCMAKE_C_COMPILER="gcc" \
  -DCMAKE_CXX_COMPILER="g++" \
  -DMPI_AS_LIBRARY="ON" \
  -DMPI_C_LIB_DIR="<oneapi_root>/mpi/2021.1.1/lib" \
  -DMPI_C_INCLUDE_DIR="<oneapi_root>/mpi/2021.1.1/include" \
  -DMPI_C_LIB="<oneapi_root>/mpi/2021.1.1/lib/release/libmpi.so"

# hipSYCL
cmake -B"<root>/build" -H. \
  -DCMAKE_BUILD_TYPE=Release \
  -DSYCL_RUNTIME="HIPSYCL" \
  -DCMAKE_C_COMPILER="gcc" \
  -DCMAKE_CXX_COMPILER="g++" \
  -DHIPSYCL_INSTALL_DIR="$(dirname␣"$(which syclcc)")/.." \
  -DHIPSYCL_PLATFORM="cpu" \
  -DDISABLE_ND_RANGE="ON" \
  -DMPI_AS_LIBRARY="ON" \
  -DMPI_C_LIB_DIR="<oneapi_root>/mpi/2021.1.1/lib" \
  -DMPI_C_INCLUDE_DIR="<oneapi_root>/mpi/2021.1.1/include" \
  -DMPI_C_LIB="<oneapi_root>/mpi/2021.1.1/lib/release/libmpi.so"
  -DCXX_EXTRA_FLAGS="
-fsycl
-march=skylake-avx512
--gcc-toolchain=$(realpath␣"$(dirname "$(which␣gcc)")"/..)
"

cmake \
  --build "<root>/build" \
  --target "cloverleaf" \
  --config Release -j $(nproc)

./cloverleaf --file "InputDecks/clover_bm16.in" --device "Xeon"
```

A.4 BUDE

```
# Load modules on Isambard MACS only:
module load cmake/3.18.3
module load gcc/8.2.0

export LD_LIBRARY_PATH=\
  "<oclcpu>/oclcpuexp.<ver>/x64:${LD_LIBRARY_PATH:-}"
export LD_LIBRARY_PATH=\
  "<oclcpu>/tbb.<ver>/lib/intel64/gcc4.8:${LD_LIBRARY_PATH:-}"
export OCL_ICD_FILENAMES=\
  "<oclcpu>/oclcpuexp.<ver>/x64/libintelocl.so"

git clone -b "master" \
  "https://github.com/UoB-HPC/bude-portability-benchmark.git" \
  <builddir>

cd <builddir>

# DPC++
cmake -B"<root>/build" -H. \
  -DCMAKE_BUILD_TYPE=Release \
  -DSYCL_RUNTIME="DPCPP" \
  -DDPCPP_BIN="<dpcpp>/dpcpp-compiler.20200816/bin/clang++" \
\ -DDPCPP_INCLUDE="<dpcpp>/dpcpp-compiler.20200816/include/sycl" \
  -DCMAKE_C_COMPILER="gcc" \
  -DCMAKE_CXX_COMPILER="g++"
  -DNUM_TD_PER_THREAD="2" \
  -DCXX_EXTRA_FLAGS="
-fsycl
-march=skylake-avx512
--gcc-toolchain=$(realpath␣"$(dirname "$(which␣gcc)")"/..)
"

# ComputeCpp
cmake -B"<root>/build" -H. \
```

```
  -DCMAKE_BUILD_TYPE=Release \
  -DSYCL_RUNTIME="COMPUTECPP" \
  -DComputeCpp_DIR="<computecpp>/computecpp-<ver>-centos-64bit" \
  -DOpenCL_LIBRARY="<oclcpu>/oclcpuexp.<ver>_rel/x64/libOpenCL.so"
  -DCMAKE_C_COMPILER="gcc" \
  -DCMAKE_CXX_COMPILER="g++" \
  -DNUM_TD_PER_THREAD="2"

# hipSYCL
cmake -B"<root>/build" -H. \
  -DCMAKE_BUILD_TYPE=Release \
  -DSYCL_RUNTIME="HIPSYCL" \
  -DCMAKE_C_COMPILER="gcc" \
  -DCMAKE_CXX_COMPILER="g++" \
  -DNUM_TD_PER_THREAD="2" \
  -DHIPSYCL_INSTALL_DIR="$(dirname␣"$(which syclcc)")/.." \
  -DHIPSYCL_PLATFORM="cpu" \
  -DDISABLE_ND_RANGE="ON" \
 \-DCXX_EXTRA_FLAGS="
-fsycl
-march=skylake-avx512
--gcc-toolchain=$(realpath␣"$(dirname "$(which␣gcc)")"/..)
"

cmake \
  --build "<root>/build" \
  --target "bude" \
  --config Release -j $(nproc)

./bude -n 65536 -i 8 --deck "data/bm1" --wgsize 0 --device "Xeon"
```