SYCL-Bench 2020: Benchmarking SYCL 2020 on AMD, Intel, and NVIDIA GPUs

Luigi Crisci*
Lorenzo Carpentieri
University of Salerno, Italy

Peter Thoman
University of Innsbruck, Austria

Aksel Alpay
Vincent Heuveline
Heidelberg University, Germany

Biagio Cosenza
University of Salerno, Italy

ABSTRACT

Today, the SYCL standard represents the most advanced programming model for heterogeneous computing, delivering both productivity, portability, and performance in pure C++17. SYCL 2020, in particular, represents a major enhancement that pushes the boundaries of heterogeneous programming by introducing a number of new features. As the new features are implemented by existing compilers, it becomes critical to assess the maturity of the implementation through accurate and specific benchmarking. This paper presents SYCL-Bench 2020, an extended benchmark suite specifically designed to evaluate six key features of SYCL 2020: unified shared memory, reduction kernel, specialization constants, group algorithms, in-order queue, and atomics. We experimentally evaluate SYCL-Bench 2020 on GPUs from the three major vendors, i.e., AMD, Intel, and NVIDIA, and on two different SYCL implementations AdaptiveCPP and oneAPI DPC++.

CCS CONCEPTS

• **Computer systems organization → Heterogeneous (hybrid) systems**; • **General and reference → Performance**; • **Software and its engineering → Parallel programming languages**; **Compilers**.

KEYWORDS

SYCL, benchmark, GPU, HPC, heterogeneous computing, portability

ACM Reference Format:
Luigi Crisci, Lorenzo Carpentieri, Peter Thoman, Aksel Alpay, Vincent Heuveline, and Biagio Cosenza. 2024. SYCL-Bench 2020: Benchmarking SYCL 2020 on AMD, Intel, and NVIDIA GPUs. In *International Workshop on OpenCL and SYCL (IWOCL '24), April 8–11, 2024, Chicago, IL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3648115.3648120

*Corresponding author: lcrisci@unisa.it

IWOCL '24, April 8–11, 2024, Chicago, IL, USA

1 INTRODUCTION

Current developments in computer architecture predict that almost all aspects of future computer systems will have many more different components than in the past, leading to an era of extreme heterogeneity [36]. In particular, the heterogeneity of processors has led not only to the near widespread adoption of GPUs, but also to a proliferation of custom accelerators for various domains, such as artificial intelligence [26], molecular dynamics [33], and cosmology [29]. This ongoing trend spans all computing systems, from tiny embedded systems to large-scale high-performance computing. In fact, nine out of ten of the world's most powerful supercomputers have a heterogeneous node equipped with GPUs [3]. Unfortunately, extreme heterogeneity comes with extreme software fragmentation, emphasizing the necessity of open standards for heterogeneous computing.

The SYCL standard [2] fills this gap by providing a high-level C++ abstraction layer for writing heterogeneous, performance-portable applications. SYCL 2020, the latest major revision of the standard, opened the doors to backends other than OpenCL and introduced several new features that greatly enhance SYCL's functionality and extend its applicability to a wide range of hardware. SYCL 2020's new features, such as unified shared memory and group algorithms, provide powerful tools for achieving high performance in a portable way, but implementation efforts are now focused on an even larger number of targets, considering all possible combinations of compilers and backends. To achieve a stable software ecosystem, it is essential that SYCL 2020 implementations are mature enough so that all features perform well on all supported targets.

This paper proposes a set of extensions to SYCL-Bench[1] [25] specifically designed to evaluate the main features of SYCL 2020. To the best of our knowledge, this is the first benchmark suite specifically tailored for SYCL 2020. The contributions of this paper are:

- The first benchmark suite specifically designed for SYCL 2020, which extends SYCL-Bench with 9 templated benchmarks covering 44 configurations and six patterns.
- Benchmark analysis of six major SYCL 2020 features: unified shared memory, reduction kernel, specialization constant, group algorithms, in-order queue, and atomic;
- Experimental evaluation on NVIDIA, Intel, and AMD GPUs, as well as two different SYCL implementations AdaptiveCpp and oneAPI DPC++.

[1] SYCL-Bench is publicly available on GitHub: https://github.com/unisa-hpc/sycl-bench

The rest of the paper is organized as follows: Section 2 gives background on the evolution of SYCL 1.2.1 to SYCL 2020 and current SYCL implementations. Section 3 describes the principal SYCL 2020 features and the related benchmark methodology. Section 4 presents the benchmark results on three GPUs, while Section 5 highlights some discussion points arising from the experimental sections. Section 6 and 7 conclude the paper with related work and conclusions.

2 BACKGROUND

2.1 From SYCL 1.2.1 to SYCL 2020

SYCL was first proposed in March 2014 by the Khronos Group as a high-level programming model for OpenCL. SYCL shared much of its definitions with OpenCL, inheriting the execution model, runtime feature set, and device capabilities of the underlying model, while providing C++ usability and flexibility alongside an easier, single-source programming style. A novel working group, the SYCL Working Group, was created within the OpenCL Working Group as a sub-project. However, developers found that the bond with OpenCL was too strict, as the SYCL specification was shown to be well-suited for third-party custom backends [6]. SYCL 2020 was ratified in February 2021 and constitutes a major milestone for the SYCL ecosystem. With the novel specification, the binding with OpenCL drops, allowing for novel third-party acceleration API backends, e.g. CUDA, ROCm, LevelZero, etc. As a result, the SYCL working group was split from the OpenCL one, becoming a standalone entity in the Khronos group. Moreover, the SYCL 2020 release incorporates over 40 new features to enhance flexibility, performance, and productivity. Those additions encompass:

- Unified Shared Memory (USM), a low-level, pointer-based memory API
- Built-in parallel reduction support
- Support for native API interoperability
- Work group and subgroup common algorithm library
- SYCL atomic alignment to the C++ standard
- Runtime queries for fine-grained device selection

2.2 SYCL 2020 Current Implementations

At the time of writing, SYCL comprises two major implementations: OneAPI DPC++ and AdaptiveCpp.

OneAPI Data-Parallel C++ [8] is an LLVM-based, open-source implementation developed by Intel. It includes an OpenCL backend to target e.g. the host CPU, as well as CUDA, HIP, and Level Zero backends for NVIDIA, AMD, and Intel GPUs respectively. Furthermore, the OpenCL backend can also target FPGAs, with DPC++ including an emulator to facilitate development.

AdaptiveCpp is an open-source heterogeneous compiler and runtime implementation project for SYCL and C++ standard parallelism offload led by Heidelberg University[6]. It can run on CPUs with OpenMP or OpenCL and can target supported OpenCL devices, as well as NVIDIA, AMD, and Intel GPUs with native backends. In addition, it includes a novel single-source, single compiler pass (SSCP)[7] which executes a single compiler invocation during compilation instead of the usual separate host-device passes, substantially lowering compilation times. In the AdaptiveCpp SSCP compiler, code is JIT-compiled at runtime from a unified code representation based on LLVM IR.

Implementation	Backends	Target Hardware
OneAPI DPC++	CUDA	NVIDIA GPUs
	HIP	AMD GPUs
	OpenCL	OpenCL devices
	LevelZero	Intel GPUs
AdaptiveCpp	OpenMP	Any CPU
	CUDA (via nvc++, clang CUDA or SSCP JIT)	NVIDIA GPUs
	HIP (via clang HIP or SSCP JIT)	AMD GPUs
	OpenCL (via SSCP JIT)	OpenCL devices
	LevelZero (via SSCP JIT)	Intel GPUs

Table 1: Summary of the major SYCL implementations with the relative backends and supported hardware

While both implementations are not officially SYCL 2020 compliant, they implement the majority of the core features. Alongside them, there are other minor implementations in the SYCL ecosystem: NeoSYCL [23] is an implementation designed for the NEC SX-Aurora TSUBASA supercomputer. TriSYCL [16], developed by AMD, has been one of the first available SYCL implementations and, while it exposes both an OpenCL and TBB backend, it's mostly used for experimenting on Xilinx FPGAs. Sylkan [34] is an experimental implementation targeting Vulkan APIs, which can potentially target any GPUs supporting the Khronos standard. Alongside SYCL implementations, multiple projects aim to bring SYCL to domain-specific application fields, such as distributed computing [32], real-time energy optimizations [15], and approximate computing [11].

3 SYCL 2020 BENCHMARKED FEATURES

In this section, we describe the principal SYCL 2020 features and the relative designed benchmarks.

3.1 Unified Shared Memory

Unified Shared Memory (USM) is a pointer-based, low-level memory API for handling memory allocations in a SYCL environment. When using USM, the devices and the host share the same address space, guaranteeing that pointers will be consistent across different memory spaces. Unified Shared Memory also allows the allocation of memory accessible from both host and devices, transparently migrating data between memories without requiring explicit memory operations.

USM provides three kinds of allocations:

- *malloc_host*: allocates pinned memory on the host. Pinned memory, or page-locked memory, is stored in the DRAM and cannot generate a page fault on access. It's also accessible on the device, but it cannot be migrated to the device memory and instead is queried from host memory at each access.
- *malloc_device*: allocates memory on the dedicated device memory. Such allocation is not accessible from the host or any other devices and requires explicit copy operations to be migrated[2].
- *malloc_shared*: allocated memory on shared memory, i.e. the memory is accessible from both host and device. It is usually

[2]Optionally available within the same SYCL context if the backend has P2P support

implemented with a page fault-based system to transparently migrate memory from the host and devices, enabling fine-grained memory transfers.

Compared to the buffer-accessor model, USM allows to have more fine-grained control over memory allocations. Furthermore, USM pointers do not carry the Accessors abstraction overhead, potentially requiring less space on devices, e.g., registers on GPUs. On the other hand, the malloc/free paradigm employed by USM is more bug-prone, e.g. memory leaks by missing frees. Furthermore, USM does not implicitly create a kernel task graph as done by the buffer-accessors, and thus the user must explicitly declare dependencies between tasks, shifting the task scheduling optimization duty to the end user. For Unified Shared Memory, we include three new specialized benchmarks:

(1) Task scheduling latency: With USM, SYCL 2020 users have two ways to handle data allocations. While with buffer-accessors the SYCL runtime needs to check for data dependencies and task scheduling optimizations, with USM no such thing occurs, and the task graph must be manually created. Thus, the scheduling time of the two memory models could be significantly different. In this benchmark, we want to measure the task scheduling overhead of the two SYCL memory models. We launch a series of small kernels in a for loop, with linear dependencies using both buffer-accessors and USM. For the latter, we employ device allocations to minimize run-time overhead during kernel execution. We take the *system time*, i.e., the time spent by the SYCL runtime to prepare the kernel execution. In this way, we expect to effectively measure the SYCL runtime scheduling overhead.

(2) Host-Device transfers: USM provides three different kinds of allocation, each of which exposes different behaviors when accessed on the host and/or the device. However, it's unclear when a certain allocation is more suitable for a specific use case. For example, *host* allocations do not require to be manually migrated to the device, but each access is more expensive as it generates a slow host-device memory transfer. Shared allocations migrate on-demand between devices but require specialized hardware support. Therefore, choosing the right allocation is essential. To explore the performance of USM allocations, SYCL-Bench 2020 includes a novel *Host-Device transfers* benchmark. The purpose of the benchmark is to simulate different offloading scenarios and measure which USM allocation is more suited. It is organized as follows: for each allocation kind, we first issue a host-device copy operation if data are not device-accessible. Then we submit a vector addition kernel on the device, followed by a device-host copy (if the current allocation is not host accessible). Finally, we perform a host kernel on the resulting data. This process is included in a loop, whose iteration number can be tweaked from the command line. This way, we simulate a common offloading scenario where data are moved back and forth from host to device. The amount of memory access executed by the device and host kernels is controlled by an *instruction mix* parameter: it controls the proportion between host and device memory access, e.g. an instruction mix of 10 means that the device kernel performs 10x memory access compared to the host kernel. In this way, by varying the amount of work performed on the device we can measure the performance of automatic memory migration and host access overhead.

(3) Pinned vs Non-Pinned allocation: Pinned memory allocation can greatly improve bandwidth on some devices. For instance, NVIDIA GPUs cannot write pageable memory on the host when performing a device-host copy operation. Instead, they allocate a temporary, page-locked memory buffer and copy the data to it before moving the data to the original destination [1]. However, allocating pinned memory is an expensive operation, which can inhibit the bandwidth advantage. In this benchmark, we measure the overhead of using pinned memory in SYCL kernels. We allocate both pinned and non-pinned memory, and then submit *n* Host-Device and Device-Host copy operations. In this way, we measure how many copies (e.g. buffer reuse) are necessary to justify the initial allocation overhead on GPUs from different hardware vendors.

3.2 In-order queue

SYCL 2020 introduced a novel *in-order* attribute for the `sycl::queue` object: when a queue is defined as *in-order*, all commands submitted to the queue are executed in a FIFO order, meaning that each task has an implicit dependency on the previously submitted tasks. While it could appear as a minor inclusion in the SYCL standard, *in order* queues come with a non-negligible series of advantages. Because tasks are executed in the submission order, a SYCL implementation could optimize the SYCL task graph creation or even avoid building it, potentially reducing task scheduling latency. For example, when using buffer-accessors the SYCL runtime can avoid checking for read-write dependencies with previously submitted kernels. This could be crucial in latency-bound applications, where the SYCL runtime overhead could be detrimental. On the other hand, the SYCL runtime cannot exploit parallelism opportunities by overlapping kernel executions.
For this feature, we extended the *Task scheduling latency* benchmark by adding a variant with *in-order* and *out-of-order* queues. To allow for a fair comparison, we manually build the task graph using the *depends_on* member function of the SYCL *handler* with USM and out-of-order queues. By comparing the kernel submission latency we can evaluate if a SYCL implementation takes advantage of the optimization opportunities, e.g., optimizing away graph generation.

3.3 Reduction Kernel

Reductions are a common pattern widely adopted in parallel applications to combine elements into a single output. This aggregation is achieved through the application of a specified associative and commutative operation. Relying on optimized implementation of the reduction pattern is crucial for enhancing overall application performance. SYCL 2020 introduced the support for built-in reduction operations that allow writing reduction kernels by leveraging the novel *sycl::reducer* class and the *sycl::reduction* function. The *sycl::reducer* is an implementation-defined object, encapsulating a reduction variable that exposes an interface that defines the operations allowed on that variable. Although the precise definition of the reducer class depends on the SYCL implementation, the functions and operators facilitating the modification of the reduction variable are well-defined by the SYCL standard and assured to be

supported by every SYCL implementation. The reducer exposes a *combine()* function which joins the element of a single work item with the value of the reduction variable using a pre-defined operator, e.g., addition, multiplication. etc. The SYCL built-in kernel reduction comes with several advantages in performance, portability, code readability, and maintenance. Manually implementing kernel reductions on each target architecture can be difficult and time-consuming since it necessitates a thorough understanding of each target architecture together with a delicate tuning phase. In this way, SYCL hides low-level implementation details, allowing the user to focus on the core application. Furthermore, SYCL 2020 reduction kernels usually require fewer lines of code compared to manual implementations, reducing codebase sizes and improving readability. We included a novel benchmark in SYCL-Bench 2020 for SYCL kernel reductions. The primary objective is to assess the performance enhancements brought about by this new feature compared with manually optimized reductions [17]. We employ `sycl::range parallel_for` instead of `sycl::nd_range parallel_for` for writing the kernel reduction: as it enforces less strict requirements on thread scheduling, SYCL implementations are free to apply additional optimizations compared to `nd_range` ones. In this way, we plan to better evaluate the kernel reduction implementation quality. Additionally, to explore possible optimization the benchmark is parameterized by a coarsening factor, determining the number of elements combined by each thread.

3.4 Group Algorithms

SYCL 2020 introduced group algorithms a set of functions that provide support for operations involving groups of work items, such as group barriers, and collective operations (shift, permute, reduction, etc...). All group algorithm functions take a group as the first argument that defines which group of work items executes the specified function (sub-group or work group). Group algorithm functions abstract away low-level hardware details by encapsulating complex operations into a single, well-defined function optimized for the target architecture. In contrast, a manually implemented function may not be as well tuned to the underlying hardware, potentially resulting in suboptimal performance. Group algorithm functions are designed to be device-agnostic, allowing them to adapt to different devices. Furthermore, group algorithm functions are often more concise and readable than manually implemented alternatives. To evaluate the performance achieved by the group algorithms function we used the *sycl::reduce_over_group* collective function as a case study. The reason behind this choice is straightforward: as reduction performance strongly depends on the target device, they represent a perfect study case to evaluate SYCL implementation quality. The *sycl::reduce_over_group* takes three parameters: the group of work items involved in the operation, the variable that has to be combined with the other values passed by the work items in the group, and the operator. In this benchmark, we performed a partial reduction over the elements specified by each work item in the same group through the *sycl::reduce_over_group*. Then the result produced by each work group is combined into a single output value using a tree reduction.

3.5 Specialization Constants

Specialization constants are a new feature introduced in SYCL 2020 which is intended to allow implementations to optimize kernels for a specific set of input values. Crucially, these values need to be known ahead of the actual kernel execution and remain constant throughout it, but they need not be known at static compilation time, i.e. when the host application itself is compiled.

This feature potentially enables significant speedups in case the constant, or propagation of it, enables the compiler to eliminate or simplify a non-negligible amount of kernel code. However, it also requires a large amount of implementation work – and, even in high-quality implementations, may induce additional overhead for the initial kernel launch after a specialization constant was changed. As such, the efficacy of this feature may vary significantly across different implementations and hardware backends, which makes it a very interesting target for analysis in a benchmark suite targeted at SYCL 2020.

To conclusively evaluate this feature in a sufficiently realistic use case, SYCL-Bench 2020 includes a dedicated *SpecConst* benchmark. It is based on an implementation of a generic 2D 9-point stencil code which allows the user to set the stencil values as specialization constants. In the benchmark, this generic implementation is then used to implement a 5-point stencil, with the outer corners set to 0. To provide context for interpreting the results achieved using specialization constants, the benchmark also features two alternative implementations, expanded via templates at compile time: one which uses standard dynamic variables for the stencil weights, and one which sets them fully statically at (host) compile time. Finally, to be able to flexibly adjust the relative arithmetic and memory intensity of the benchmark, the number of inner iterations of the kernel code can be adjusted, potentially performing more operations on the same amount of global data.

3.6 Atomics

SYCL 2020 deprecates the *sycl::atomic* class in favor of the *sycl::atomic_ref* to be aligned with the C++ atomic model. The SYCL *atomic_ref* class adds three template parameters to the standard C++ atomic_ref: *sycl::memory_order*, specifying the memory synchronization order of the atomic operation; *sycl::memory_scope*, defining the work items and devices to which the memory ordering constraints of the atomic operation are applied; and *sycl::access::address_space*, indicating the address space of the object referenced by the *atomic_ref* class.

Since the previous release of SYCL-Bench does not include a use case for atomic operations, we developed a new benchmark to evaluate the performance of atomic operations for the different SYCL implementations and hardware. The main purpose of the benchmark is to investigate how different SYCL implementations map atomic operations for each target architecture, as well as atomic hardware support on different vendors' GPUs. The benchmark implements a sum reduction using only the *atomic_ref* class and the *atomic_fetch_add functions*. For all the benchmarks we used *sycl::memory_order::relaxed* as it is the only one that SYCL specification guarantees to be supported on all devices [2]. As *sycl::address_space* and *sycl::memory_scope* we consider the general case where the object referenced by the atomic_ref is in the

sycl::global_space and the ordering constraint applies only to work-items executing on the same device as the calling work item (*sycl::memory_scope::device*). The benchmark is templated on the datatype on which the atomic operation is performed.

4 EXPERIMENTAL EVALUATION

We present the results obtained on three GPUs from three principal GPU vendors, i.e. NVIDIA Tesla V100S, AMD MI100, and Intel Max 1100.

The NVIDIA node is equipped with an Intel Xeon Gold 5218 CPU, operating at 2.30GHz and featuring 64 cores. Additionally, it incorporates an NVIDIA Tesla V100S connected via PCI Express. The GPU comprises 80 Streaming Multiprocessors, totaling 5120 cores running at a frequency of 1.245GHz, supplemented by 620 Tensor Cores. The Tesla V100S achieves 8.2 TFLOP/s FP64, 16.4 TFLOP/s with FP32, 32.2 TFLOP/s with FP16, and 130 TFLOP/s utilizing FP16 Tensor cores. The GPU has 32GB HBM2 memory, attaining a high bandwidth of up to 1132 GB/s. It includes 128KB L1 Cache per Streaming Multiprocessor (SM), 6MB L2 Cache, and 256KB registers per SM.

The AMD node is configured with an AMD EPYC 7313 CPU clocked at 3.7GHz, featuring 16 x 2 cores. Additionally, it incorporates an AMD MI100 GPU connected via PCI Express. The GPU is comprised of 120 compute units, for a total of 7680 cores, each running at a frequency of 1.0 GHz. The MI100 GPU delivers up to 11.5 TFLOP/s FP64, 23.1 TFLOP/s FP32, and an 184.6 TFLOP/s using FP16. With 32 GB of HBM2 memory, the GPU achieves a high bandwidth of up to 1129 GB/s. Further specifications include a 16KB L1 Cache per Compute Unit, an 8MB L2 Cache, and 256KB registers.

The Intel node was provided by the CINECA consortium [4], it is equipped with an Intel(R) Xeon(R) Platinum 8480+ and an Intel Max 1100, connected via PCI Express. It has 56 Xe cores, for a total of 7168 cores. The max clock is 1550 MHz, achieving 22.3 TFLOP/s with FP32 and FP64 [3] It is equipped with 48 GB of HBM2E memory, with a maximum bandwidth of 1228.8 GB/s, alongside 28MB L1 cache and 108 MB L2 cache.

For the SYCL implementations, we chose Intel DPC++ (git commit sha f43cd7b) and AdaptiveCpp (git commit sha eeebfd4) which are the two main SYCL implementations at the time of writing. . Additional software stack is Clang 17.0.1 (for building AdaptiveCpp), CUDA 12.1 (driver 535.129.03), ROCm 5.5.0 (driver 505.302.01), and LevelZero driver version 170.007.42. We run each experiment 10 times and take the median as the reference value.

4.1 Unified Shared Memory

4.1.1 Task scheduling latency. In Figure 1 we show the benchmark results when running 50000 kernels on every hardware and SYCL implementation. From the picture, it's clear that the DPC++ AMD backend suffers from high scheduling overhead compared to any other hardware and implementations, with a 7x slowdown compared to the same benchmark running with AdaptiveCpp and 14x slowdown compared to the one running on the Tesla V100S with the same implementation. To allow for a better evaluation of the results, we also include in Figure 2 the results without the results of the

[3]Unofficial data as Intel has not released official performance statistics yet.

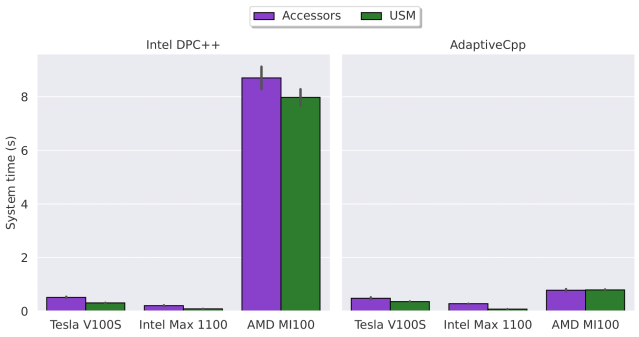


Figure 1: Task scheduling latency with 50000 kernels scheduled

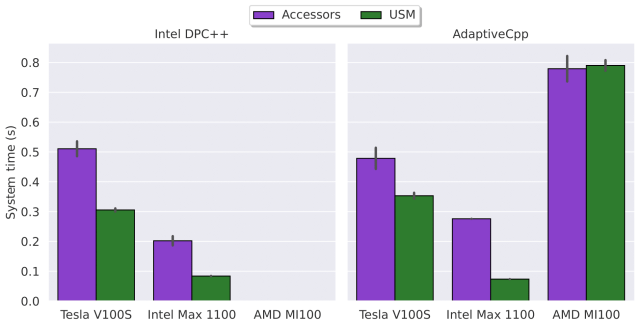
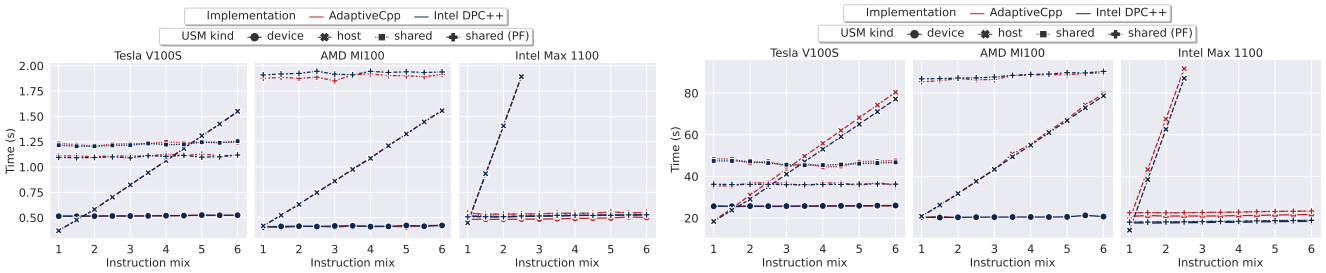


Figure 2: Task scheduling latency with 50000 kernels scheduled (no AMD HIP DPC++ on AMD MI100)

HIP AMD DPC++ backend. With DPC++, scheduling kernels with USM is faster than accessors on every hardware, achieving 1.64x, 2.2x, and 1.07x respectively on NVIDIA, Intel, and AMD GPUs. In particular, the Intel Max shows remarkably short scheduling time, with the accessor benchmark being 2.5x times faster than the corresponding one on the Tesla V100S.

AdaptiveCpp exposes two main tuning parameters that influence the runtime scheduling behavior: the `HIPSYCL_ALLOW_INSTANT_SUBMISSION` macro and the `ACPP_RT_MAX_CACHED_NODES` environment variable. The former allows the submission of USM operations to in-order queues via a low-latency instant submission mechanism, while the latter determines the maximum number of nodes the runtime is allowed to cache to perform a batched submission. If it is set to 0, the runtime is forced into an eager submission mode. We experimented with all combinations of those macros: for NVIDIA and Intel hardware, setting the `HIPSYCL_ALLOW_INSTANT_SUBMISSION` gives a 8% speedup on USM. On the other hand, on AMD the best setup is without any optimization macros. The USM benchmark performs better than accessors on both NVIDIA and Intel GPUs, achieving 1.4x and 3.1x speedup respectively. However, AMD shows a consistent slowdown compared to the other backends also with AdaptiveCpp. We investigated the reason for this slowdown by looking at the SYCL implementations source code. Both implementations use hip streams to build SYCL queues on AMD, allocating multiple streams to exploit parallelism. This performance gap is



(a) 2 GB, 1 iteration **(b) 1 GB, 100 iterations**

Figure 3: Host-Device transfers benchmark results

likely to derive from a ROCm internal issue when handling multiple streams concurrently instead of scheduling all operations on a single one, which has been shown to increase submission latency in some environments[4]. We believe this issue is connected to the DPC++ AMD backend, and we plan to further investigate the origin of the problem.

4.1.2 Host-Device transfers. Figure 3 illustrates the results for *Host-Device transfers* benchmark. On the left, Figure 3a shows the results with a 2GB data size and one iteration. On the right, Figure 3b shows the results with a 1GB data size and 100 loop iterations. For the *shared* allocation, we also include the results when using the SYCL *prefetch* function for the V100S and Max 1100, which hints to the runtime that the data are going to be accessed on the device, potentially migrating the data before executing the kernel. On AMD GPUs, on-demand host-device page migration requires XNACK, an AMD feature disabled by default. Without XNACK enabled, *shared* allocations behave like *host* ones, with data being allocated on the host. However, XNACK is known to be experimental and unstable. We experienced random kernel failure as well as GPU hanging when enabled, therefore we disabled it for this analysis. However, *shared + prefetch* allocations can be used even without XNACK, as no page faults are issued, therefore we include just that one in our benchmark. From the figures, it is clear that the results are more hardware-dependent than implementation-dependent, as both AdaptiveCpp and DPC++ perform similarly. As expected, the *host* allocation time scales linearly with the amount of device memory access, while the other allocations are less influenced. However, there are two interesting cases: (1) on low instruction mix, the Tesla V100S and Intel Max 1100 *host* allocations perform better than both *device* and *shared*. On the Tesla V100S, *device* and *shared* need an *instruction_mix* > 1.5 *and* > 4.5 respectively to outperform the *host* performance. On the other hand, on Intel Max 1100 the advantage quickly disappears above a 1:1 device-host memory access proportion. (2) On the MI100, *host* allocation always performs worse than *device*, while achieving a great speedup compared to *shared* allocations for `instruction_mix <=6`. However, increasing the instruction mix would inevitably lead the *host* line to cross the *shared* one. *Shared* allocations perform worse than *device* on every hardware, while being particularly slower on the AMD MI100, getting up to 450% slowdown on the benchmark (b) due to the poor AMD managed memory system support. In Figure 4 we show the impact of the *prefetch* operation on *shared* allocations. On the

[4]https://github.com/AdaptiveCpp/AdaptiveCpp/issues/1246

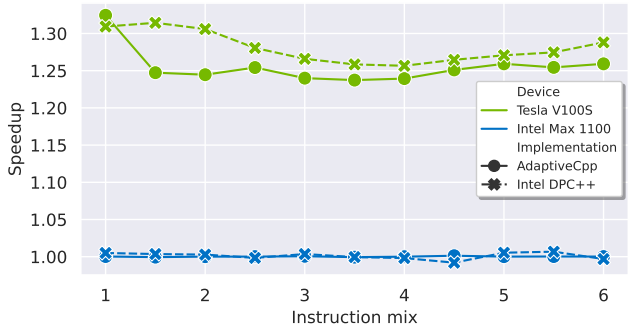
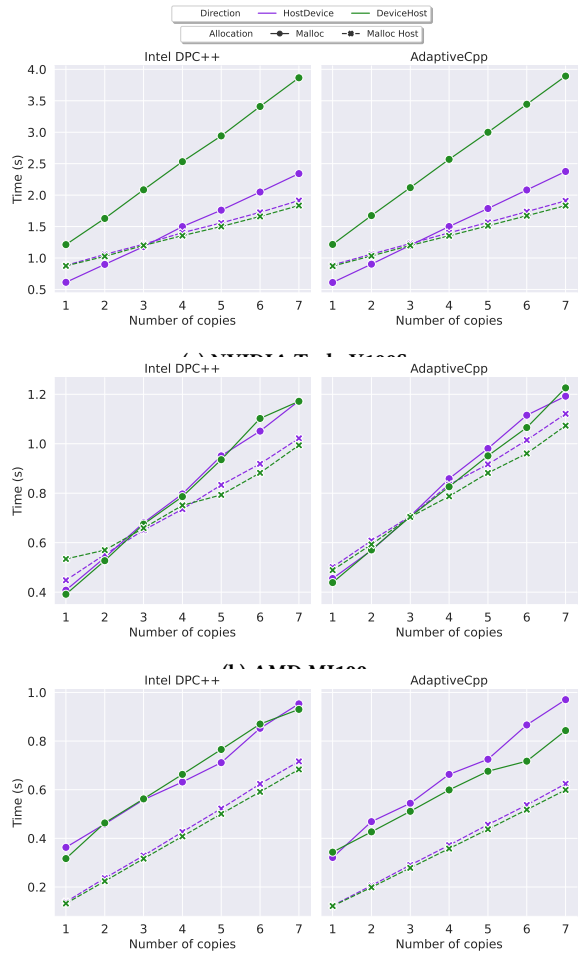


Figure 4: Prefetched speedup in respect to non-prefetched shared allocation on the Instruction mix benchmark (Figure 3b)

Tesla V100S prefetching memory gives up to 31% improvement with DPC++ and AdaptiveCpp. On the other hand, prefetching on Intel hardware seems to give no advantage at all with both implementations, showing that the Intel driver can efficiently migrate the shared memory without hints. While in benchmark (a) both implementations perform similarly when increasing the iterations in benchmark (b) AdaptiveCpp performs worse on *shared* and *host* compared to DPC++ on Intel hardware, which is unexpected as both implementations map shared allocation onto native function call. In particular, DPC++ is on average 23% faster on *shared* and *shared + prefetch* allocations on the Intel Max 1100, while being on par with the MI100 and V100S.

4.1.3 Pinned vs Non-Pinned allocation. In Figure 5 we show the results for the *Pinned vs non-Pinned* benchmark. `Malloc` refers to the pageable memory, while `malloc host` to the pinned one. We use a data size of 2GB and we measure the execution time by varying the number of copies performed using the same allocation. Both implementations performed similarly, showing how results are more hardware-dependant than implementation-dependant. On the Tesla V100S, the pinned memory allocation overhead is more evident with *Host-Device* copies, where using pageable memory allocations is faster for < 3 copies, with a maximum speedup of 1.4x with 1 issued copy operation. This means that at least 3 buffer reuses are required to match the initial overhead. Conversely, *Device-Host* copies are always faster on pinned memory. This is probably due to the CUDA driver, which is unable to write on pageable



(c) Intel Max 1100

Figure 5: Pinned vs Non-Pinned benchmark with 2GB memory transfers

memory, and thus performs a two-phase copy, allocating a page-locked memory location as a temporary buffer. On the AMD MI100, pageable memory performs better on both *Host-Device* and *Device-Host* copy operations. DPC++ requires at least 3 copies to match the overhead, while AdaptiveCpp requires 4 copies. However, the speedup is less evident, with a maximum speedup of 1.1x on both implementations. On the other hand, on the Intel Max 1100, we observe a completely different behavior, with pinned allocation performing always better than pageable memory. This shows that, while `sycl::malloc_host` is an essential addition to the SYCL standard, the user should be aware of the connected overhead and should take care in choosing the right allocation depending on the user application scenario.

4.2 In-order Queue

Figure 6 shows the results for the *task scheduling latency* benchmark. We issued 50000 kernel launches using *in-order* and *out-of-order* queues. As we did for the original benchmark, we include in Figure 7 the results without the ROCm DPC++ backend.

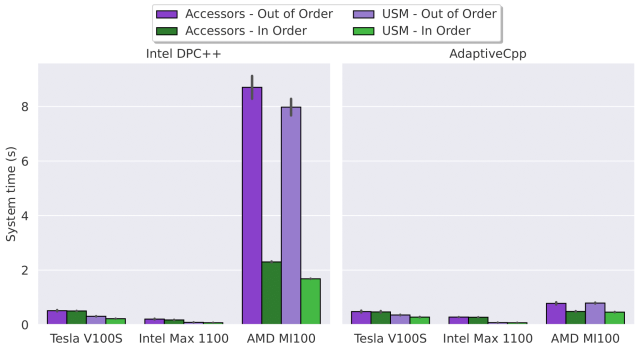


Figure 6: Task scheduling latency when launching 50000 kernels with *in-order* and *out-of-order* queues

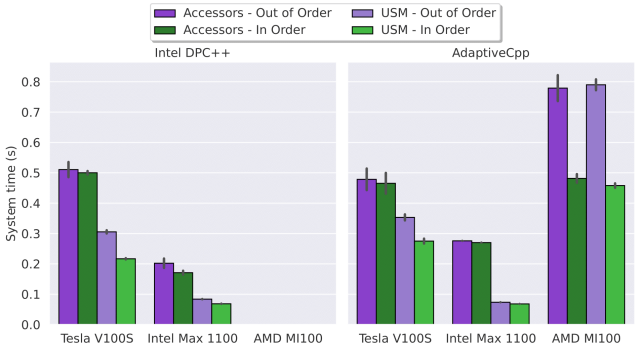


Figure 7: Task scheduling latency when launching 50000 kernels with *in-order* and *out-of-order* queues (no ROCm DPC++)

With DPC++, we see a considerable speedup with the ROCm backend, achieving a 4x and 4.4x speedup respectively with accessors and USM. However, scheduling times are still considerably higher than every other benchmark which suggests there could be a bug in the kernel submission logic within the DPC++ compiler on AMD. Surprisingly, on the other platforms *in-order* queues provide the best speedups when targeting USM rather than accessors. Moreover, the latter shows comparable results both on the Intel Max and the NVIDIA Tesla, highlighting that probably the SYCL task graph is still created under the hood. For AdaptiveCpp, the `HIPSYCL_ALLOW_INSTANT_SUBMISSION` macro, while having no impact on the Accessor benchmarks, showed an even greater impact on performance for USM when targeting *in-order* queues, achieving 28% speedup on the Tesla V100, 21% on the AMD MI100, and 13% on the Intel Max 1100. For the accessors, the only benchmark where *in-order* queues seem to have an effect is on the AMD MI100, where it gets a 63% speedup compared to *out-of-order* queues. This is probably due to the multiple hip stream issue mentioned in section 4.1.1, as *in order* queues are mapped onto a single stream instead. Therefore, the ROCm latency issue is not present. On the other hand, the minimal speedup on the other platforms reveals that currently little is done by the SYCL implementation to exploit the optimization opportunities led by the *in order* queues. For USM, AdaptiveCpp gets a considerable speedup of 65% and 107% on NVIDIA and AMD platforms respectively, while on the Intel Max 1100, the impact is

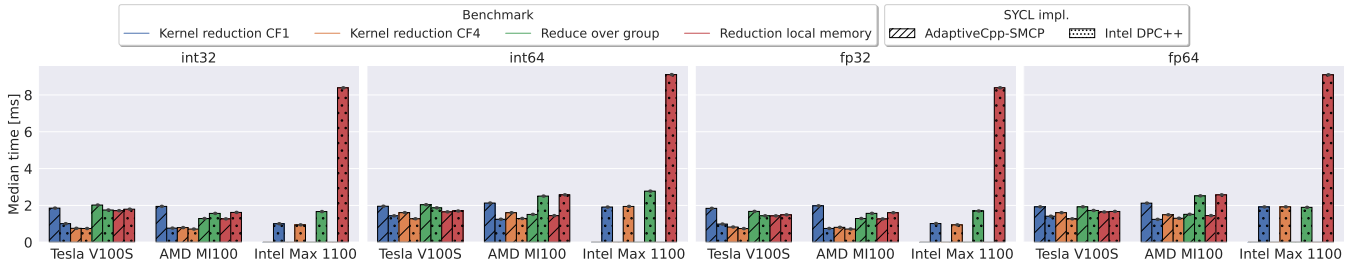


Figure 8: SYCL 2020 kernel reductions with coarsening factor 1 and 4 compared to *reduce_over_group* and local memory reductions

negligible. Overall, *in-order* queues consistently provide a speedup compared to *out-of-order* queue when using Unified Shared Memory, showing that manually building the task graph can be an expensive operation for the SYCL runtime. On the other hand, the impact of *in-order* queues on accessors is negligible, showing that implementations do not exploit all the possible optimizations yet.

4.3 Reduction Kernel

Figure 8 shows the results for the *reduction kernel* benchmark compared with the *local memory reduction* benchmark on four data types (int32, int64, fp32, fp64) and 150,000,000 elements. The AdaptiveCpp results for the Intel Max 1100 are excluded as the feature is not currently supported.

In evaluating the *kernel reduction* benchmark with a coarsening factor 1, the Intel DPC++ implementation demonstrates approximately a 2x speedup compared to the AdaptiveCpp implementation across all hardware platforms. The speedup of DPC++ over AdaptiveCpp resides in thread coarsening. AdaptiveCpp implements the kernel reduction with a tree reduction approach, launching multiple kernels until a single element is produced. Furthermore, thread coarsening optimization is left to the user since selecting the optimal coarsening factor requires parameter-tuning strategies. Differently, the DPC++ kernel reduction leverages fast atomic and the reduce_over_group function. In more detail, when using the `sycl::range parallel_for`, DPC++ adjusts the kernel grid size to spread the computation on all the available device compute units. If the number of threads specified by the user cannot be scheduled concurrently on the device resources, DPC++ applies thread coarsening in order to perform the reduction in a single-step kernel execution. The use of thread coarsening under the hood clarifies the speedup of DPC++ over AdaptiveCpp. In fact, by increasing the coarsening factor to 4, AdaptiveCpp achieves a 2x speedup compared to the version without thread coarsening showing the same performance as DPC++. In contrast, for the DPC++ implementation adjusting the coarsening factor does not notably impact performance.

We compare the built-in reduction with a manually implemented one using local memory, inspired by [17]. On the AMD and NVIDIA GPUs the *kernel reduction* benchmark with Intel DPC++ achieves ~ 2x times speedup compared to the manually implemented local memory reduction. In contrast, for AdaptiveCpp without considering the thread coarsening optimization, the two benchmarks have similar performance. The major performance improvement of kernel reduction over reduction with local memory is registered on Intel Max 1100 where we have ~ 8 times speedup for all data types.

Overall the built-in SYCL reduction for both implementations can be considered a performing and easy-to-write alternative to the manually implemented reduction. However, the use of SYCL kernel reductions should not exclude user-driven optimizations such as thread coarsening. Instead, users can focus on application-specific optimizations while ignoring the reduction implementation details.

4.4 Group Algorithms

Figure 8 shows the performance comparison between the *reduce_over_group*, *local memory reduction* and *kernel reduction* benchmarks on four data types (int32, int64, fp32, fp64) and 150.000.000 elements. The AdaptiveCpp results for the Intel Max 1100 are excluded due to the absence of support for the *reduce_over_group* feature in its implementation. On the Tesla V100S, both implementations achieve similar performance for all the data types. On AMD MI100 both implementations have the same behavior for 32-bit data, while AdaptiveCpp with 64-bit data achieves 2 times speedup compared to DCP++. The two implementations have different ways to implement *sycl::reduce_over_group*. AdaptiveCpp implements the *reduce_over_group* in two steps. First, it applies reduction on each sub-group leveraging shuffle operation mapped to sub-group primitives depending on the target architectures. Then, the results of the sub-group reductions are loaded into local memory and the final output is computed using a local memory tree reduction. Differently, the DPC++ implementation maps reduce_over_group over SPIRV intrinsics, which are lowered to a sub-group shuffle implementation on every platform.

On Tesla V100S and AMD MI100, the *reduce_over_group* achieves similar performance to the manually implemented reduction, while on Intel Max 1100 the DPC++ shows 4 times speedup compared to the manually implemented reduction. Comparing the two possibilities of implementing a reduction using SYCL features we can notice that for AdaptiveCpp the kernel reduction with coarsening factor 1 and the *reduce_over_group* benchmark show the same performance for all the hardware and data types. Differently, in DPC++, even if kernel reduction is implemented with the *sycl::reduce_over_group* there is a 2x speedup compared to *reduce_over_group* benchmark. Differently, in DPC++ the *reduce_over_group* benchmark is 2x slower compared to the kernel reduction, even if it is implemented using *sycl::reduce_over_group* under the hood. The reason behind this

Figure 9: Specialization constants performance on NVIDIA V100S, AMD MI100, and Intel Max 1100 with DPC++

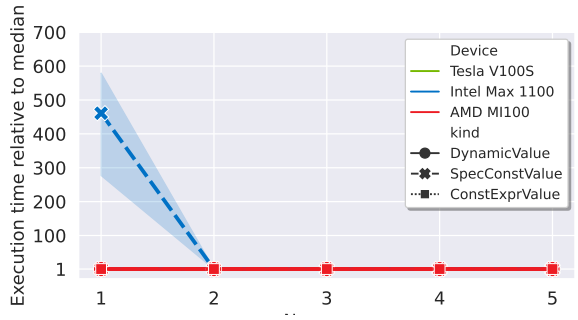


Figure 10: Specialization constant overhead of the *nth* run. The value shown is the aggregate mean of all datatypes with the 95% confidence interval

lies in thread coarsening, which is automatically applied in the case of kernel reduction with `sycl::range`. Group algorithms are available only `nd_range parallel_for`, therefore no automatic thread coarsening can be applied.

4.5 Specialization Constants

Figure 9 illustrates the performance achieved by the *SpecConst* benchmark as execution time relative to a baseline of using fully dynamic variables (*DynamicValue*). *SpecConstValue* corresponds to the kernel version using the SYCL 2020 specialization constants feature, and *ConstExprValue* sets the weights at static compile time. The benchmark was performed using 32-bit and 64-bit integer and floating point data, with a range spanning 1 (IL1), 16 (IL16), and 64 (IL64) inner loop repetitions which represents a progression from a more heavily memory-bandwidth-limited to a compute-limited scenario.

In this setup, the expectation for a full, high-quality implementation of specialization constants would be to achieve performance between the dynamic baseline and the fully constant weights, ideally as close as possible to the latter. This is what we observe on the Intel Max 1100 platform. As expected, the potential speedup scales with the complexity of the individual arithmetic operations on the target platform, as well as the arithmetic intensity of the kernel as governed by the `IL` parameter. For Intel Max 1100, `int64` operations are the most expensive in this benchmark, and a speedup of more than 4x can be observed by using specialization constants in the `int64 / IL64` case.

For both the Tesla V100S and AMD MI100 backends, no kernel specialization occurs, and the benchmark results instead show the potential slowdown incurred by a fallback implementation of the specialization constants API. The maximum relative overhead in this case is significantly higher on V100S at 20%, but note that this depends on the quality of the baseline implementation. Three related curious cases are the performance of `int32 / IL16` on V100S and `int32 / IL1+IL16` on MI100 respectively, where the *SpecConstValue* implementation outperforms *DynamicValue* despite no compile-time specialization being performed. We have verified that these results are repeatable and consistent, and believe that they are caused by the kernel compiler heuristics making more advantageous optimization decisions for these particular cases due to the code structure produced in the specialization constant version.

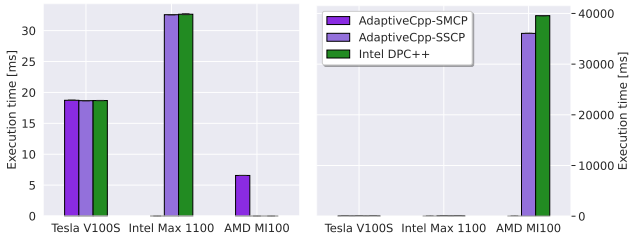
In addition to the potential speedup in kernel execution times, a second relevant aspect influencing the real-world utility of specialization constants is the overhead incurred for kernel compilation after adjusting a specialization constant. Figure 10 illustrates the relative execution time (compared to each median) across 5 runs of the benchmark. We observe that the execution times are very consistent, except for the first run after setting a new specialization constant value on Intel Max 1100. This result confirms the earlier observation, based on kernel execution times, that only the Intel backend currently fully implements specialization constants with compiler optimizations.

In terms of real-world utility, the absolute cost of these kernel re-compilation steps is quite relevant. In our benchmarks, compiling the kernel can take between 200 and 600 times as long as executing it once, indicating that specialization constants should currently only be used on this platform if the specialized kernels are either executed very frequently without further changes to the specialization constant values, or if individual kernel execution times are very high.

4.6 Atomics

The *Atomic* benchmark has been executed on 4 data types (int32, int64, fp32, fp64) with 9.400.000 elements. For AdaptiveCpp, we used both the SMCP (single-source, multiple compiler pass) and the generic SSCP (single-source, single compiler pass) [7] compilation flows as they diverge in the atomic implementation strategy.

For the AMD MI100, floating point atomic operations are simulated using a CAS loop. Support for built-in floating-point is rather incomplete: the functions have no return value, e.g., `fetch_add` return `void` instead of the previous value, and it's limited to 32-bit



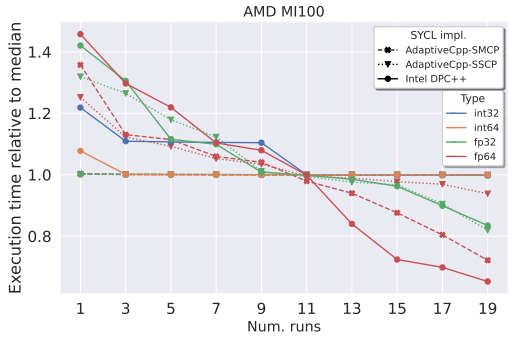
(a) No CAS loop AMD MI100 (b) With CAS loop AMD MI100

Figure 11: Atomic operations performance with 32-bit floating point

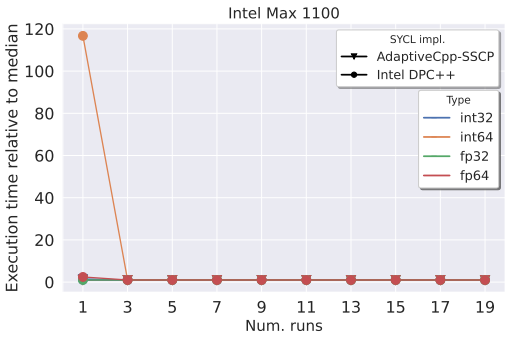
floating-point. The HIP toolchain exposes the *-munsafe-fp-atomics* parameter to enable fast atomic operations: it includes a compiler pass that checks if the return value of the atomic operations is used[5]. If not, issue the fast built-in, otherwise, it falls back to the CAS loop. Figures 11a and 11b show the execution times achieved by the *atomic* benchmark on the Tesla V100S, Intel Max 1100, and AMD MI100 GPUs using 32-bit floating-point operations. Both SYCL implementations exhibit similar performance on NVIDIA Tesla V100S and Intel Max 1100. Differently, atomic operations for AdaptiveCpp-SSCP and DPC++ are ~6000 times slower compared to AdaptiveCpp-SMCP (Figure 11a and 11b). Upon inspecting the AMD intermediate representation, we observed that for DPC++ and AdaptiveCpp-SSCP the *-munsafe-fp-atomics* option is disregarded leading the compiler to fallback on the CAS loop implementation, which drastically affects the overall performance. By analyzing the source code, we found that DPC++ skips the flag and directly handles the fast atomic built-in generation by manually checking the target architecture. On the AMD MI100, no builtin is generated as it lacks the required return value. Therefore, fast atomics are not available on the MI100 with DPC++. On the other hand, AdaptiveCpp-SSCP does not use the default clang HIP toolchain, therefore the flag cannot be interpreted. Conversely, using the unsafe option with AdaptiveCpp-SMCP the compiler correctly generates the *global_atomic_add_f32* builtin, leading to a ~ 6000 times speedup compared to DPC++ and AdaptiveCpp-SSCP. The results for the 64-bit floating-point, int 32, and int 64 are omitted since they show trends similar to the one observed for fp32 atomics. Both SYCL implementations demonstrate comparable performance on all the hardware. However, the AMD MI100 achieves a 6000x slowdown on fp64 compared to other hardware due to the lack of fast atomic support.

On AMD and Intel platforms, we observed some variability in the execution time depending on the run number. Figure 12 illustrates the execution time of each run normalized to the median runtime across 20 runs of the benchmark on AMD MI100 and Intel Max 1100. Looking at the AMD MI100 results for DPC++ and AdaptiveCpp-SSCP, we notice that the execution times on fp32 and fp64 curiously decrease for each run. Instead, AdaptiveCpp-SMCP time decreases only with fp64. Differently, the execution times on int32 and int64 are consistent for AdaptiveCpp-SMCP and DPC++. We guess that this behavior is related to the CAS loop, which is generated in all the results that show this variability. On Intel Max 1100, DPC++ exhibits a high overhead for the first run on int64 and fp64, due to the JIT compilation process. However, while also AdaptiveCpp-SSCP relies on JIT compilation, it does not show the same overhead, probably due to some additional optimizations happening in the DPC++ toolchain.

[5]https://github.com/ROCm-Developer-Tools/hipamd/issues/19



(a) AMD MI100



(b) Intel Max 1100

Figure 12: Atomic operations overhead

5 IMPLEMENTATION MATURITY

DPC++ is the only compiler with a functionally correct implementation of the benchmarked SYCL 2020 features. However, the implementation quality is not uniform across backends. For example, while specialization constants work as expected on Intel hardware, on AMD and NVIDIA they might not impact performance at all, or even be detrimental. The AMD HIP backend in particular appears to not yet be as mature, e.g., showing excessively high scheduling overheads.

AdaptiveCpp has a uniform implementation support across backends but there are differences between the older SMCP compilation flow and the new SSCP compiler. This reflects its implementation history. For these features that are available in the SMCP compiler but not in SSCP such as group algorithms or SYCL 2020 reductions, there are no inherent obstacles for implementing them to our knowledge. Instead, implementation complexity, especially for SYCL 2020 reductions, is the driving factor for the delayed implementation: SYCL 2020 reductions need to work for any data type (which may or may not be supported in atomic operations), and any binary operator (whether that operator has a known identity or not) and they need to work both if the device has sufficient or

insufficient local memory for a work group reduction in local memory. They also need to work with both range-based `parallel_for`, and `nd_range parallel_for` which might be implemented using different execution models under the hood. Providing optimized reduction implementations for all combinations of these parameters can be challenging. In this work, we have focused on investigating simple reductions over simple scalar types. It is however likely that even more performance and functionality variability can be found if more cases of reductions are investigated. It is unclear whether the current, broadly general reduction design in SYCL is beneficial, or whether it ties up implementation resources to cover niche applications that are rarely needed in practice.

Unfortunately, there are features not covered by AdaptiveCpp: for example, specialization constants are entirely unimplemented because of compilation flow issues[6]. Similar arguments are being made for other new features not investigated in this work, such as kernel bundles and host tasks. Using these features is therefore currently a problem for codes that aim to be portable between the two implementations.

Both implementations provide a rich set of extensions, cover roughly the same hardware. AdaptiveCpp's single-pass compiler offers a mechanism to build universal binaries that can offload to all supported devices. Minor portability differences include DPC++ also supporting FPGAs, and AdaptiveCpp currently having more robust support for non-x86 CPU targets as it can leverage the host toolchain for kernel compilation. All in all, this implies that the average user may in practice not need to switch between implementations at all. This is convenient for users but also reduces the incentive to avoid implementation-specific idioms or performance assumptions, or call out the lack of support for specific features and fix underlying problems in the specification.

6 RELATED WORK

Benchmark suites offer a systematic approach for evaluating the efficiency, speed, and resource utilization of different coding paradigms. Several benchmarks have been defined to analyze specific aspects of computing architectures, compilers, and applications. Bienia et al. [10] proposed PARSEC a benchmark suite that implements a set of C/C++ applications from different areas to provide support in the characterization and development of Chip-Multiprocessors (CMPs). Kulkarni et al. [24] defined a set of five applications to study and understand the patterns of parallelism and locality in sparse graph computations. Moreover, with the rise of heterogeneous systems and programming models such as OpenCL and SYCL, several studies proposed benchmarks to analyze their performance [12, 18, 20]. Jin et al. [18] developed a SYCL version of Rodinia [12] a benchmark suite to study CPU and GPU platforms. Danalis et al. [13] developed SHOC a benchmark suite composed of CUDA and OpenCL programs specifically designed to analyze the performance of OpenCL over CUDA on different architectures. Jin et al. [20] proposed HeCbench a collection of benchmarks, and mini-applications from many open-source projects written with different programming models (CUDA, HIP, and SYCL) to facilitate the performance portability evaluation of SYCL. Deakin et al. [14] enabled memory bandwidth analysis on a wide range of devices other than CPUs by implementing the four main kernels of the STREAM benchmark and a dot product, with several programming models: Kokkos, RAJA, OpenMP, OpenACC, SYCL, OpenCL, and CUDA.

In recent years, an increasing set of SYCL-specific benchmarks have been proposed. In particular, some works focused on evaluating SYCL implementations in terms of compile-time [35], performance portability [21, 28, 31], and implementation maturity over the years [27]. Moreover, several studies focused on optimizing SYCL for specific target architectures [19, 30]. SYCL 2020 provides several features to improve the performance and portability of applications on heterogeneous platforms. Alpay et.al. [5] measured the usage of interoperability options in SYCL applications using SYCL *host tasks*. Ashbaug et.al. [9] analyzed the SYCL 2020 memory model after the addition of *sycl::atomic_ref*, as well as novel features to be added to the SYCL memory model. Joube et.al. [22] used a PCI-bound microbenchmark to compare USM and Accessors performance and programmability. While these works provide an overview of SYCL functionalities, we present the first set of benchmarks explicitly designed to measure SYCL 2020 functionalities and performance on a wide range of hardware.

7 CONCLUSIONS

We presented SYCL-Bench 2020, an extension of the SYCL-Bench benchmark suite to SYCL 2020. We added nine new benchmarks covering 44 configurations, analyzing six SYCL 2020 features including unified shared memory, reduction kernel, specialization constants, in-order queues, group algorithms, and atomic_ref. Experimental results on three GPUs and two SYCL implementations provide interesting results about the potential of each feature as well as the maturity of the compilers. We show how unified shared memory can greatly reduce scheduling latency, with up to 3.1x speedup on Intel GPUs with AdaptiveCpp. Shared allocations perform poorly on AMD due to missing hardware support. We also demonstrated how some feature support depends on the hardware: specialization constants are not beneficial on platforms such as NVIDIA and AMD as they lack of toolchain support, leading to detrimental performance. Both standard implementations showed similar performance, with only the DPC++ AMD backend appearing to be not as mature as the other backends. In terms of feature coverage, AdaptiveCpp lacks of specialization constants and reductions on some devices. Overall, the SYCL 2020 standard in its current implementations and back-end support already provides excellent performance portability for most features. In future work, we plan to extend the analysis to other SYCL 2020 features such as host tasks, in addition to including missing features from AdaptiveCpp.

ACKNOWLEDGMENTS

This project has received funding from the European Union's HE research and innovation programme under grant agreement No. 101092877 (SYCLops) and from the European High-Performance Computing Joint Undertaking under grant agreement No. 956137 (LIGATE project). Additionally, it has received funding from the Austrian Research Promotion Agency (FFG) via the UMUGUC project

[6] https://github.com/AdaptiveCpp/AdaptiveCpp/issues/1296#issuecomment-1867849239

(FFG #4814683) and from the Italian Ministry of University and Research under PRIN 2022 grant No. 2022CC57PY (LibreRT project).

REFERENCES

[1] 2012. How to optimize data transfers in CUDA. https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/.

[2] 2023. SYCL Specification. https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html.

[3] 2023. Top500 chart. https://www.top500.org/.

[4] 2024. Cineca official website. https://www.cineca.it/.

[5] Aksel Alpay, Thomas Applencourt, Gordon Brown, Ronan Keryell, and Gregory Lueck. 2022. Using Interoperability Mode in SYCL 2020. In *International Workshop on OpenCL* (Bristol, United Kingdom, United Kingdom) *(IWOCL'22)*. Association for Computing Machinery, New York, NY, USA, Article 21, 1 pages. https://doi.org/10.1145/3529538.3529997

[6] Aksel Alpay and Vincent Heuveline. 2020. SYCL beyond OpenCL: The architecture, current state and future direction of hipSYCL. In *Proceedings of the International Workshop on OpenCL*. 1–1.

[7] Aksel Alpay and Vincent Heuveline. 2023. One Pass to Bind Them: The First Single-Pass SYCL Compiler with Unified Code Representation Across Backends. In *Proceedings of the 2023 International Workshop on OpenCL* (, Cambridge, United Kingdom,) *(IWOCL '23)*. Association for Computing Machinery, New York, NY, USA, Article 7, 12 pages. https://doi.org/10.1145/3585341.3585351

[8] Ben Ashbaugh, Alexey Bader, James Brodman, Jeff Hammond, Michael Kinsner, John Pennycook, Roland Schulz, and Jason Sewall. 2020. Data parallel c++ enhancing sycl through extensions for productivity and performance. In *Proceedings of the International Workshop on OpenCL*. 1–2.

[9] Ben Ashbaugh, James C Brodman, Michael Kinsner, Gregory Lueck, John Pennycook, and Roland Schulz. 2021. Toward a Better Defined SYCL Memory Consistency Model. In *International Workshop on OpenCL* (Munich, Germany) *(IWOCL'21)*. Association for Computing Machinery, New York, NY, USA, Article 20, 3 pages. https://doi.org/10.1145/3456669.3456696

[10] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. 72–81.

[11] Lorenzo Carpentieri and Biagio Cosenza. 2023. Towards a SYCL API for Approximate Computing. In *Proceedings of the 2023 International Workshop on OpenCL*. 1–2.

[12] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 44–54.

[13] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. 2010. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd workshop on general-purpose computation on graphics processing units*. 63–74.

[14] 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.

[15] Kaijie Fan, Marco D'Antonio, Lorenzo Carpentieri, Biagio Cosenza, Federico Ficarelli, and Daniele Cesarini. 2023. SYnergy: Fine-grained Energy-Efficient Heterogeneous Computing for Scalable Energy Saving. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[16] Andrew Gozillon, Ronan Keryell, Lin-Ya Yu, Gauthier Harnisch, and Paul Keir. 2020. triSYCL for Xilinx FPGA. In *The 2020 International Conference on High Performance Computing and Simulation. IEEE*.

[17] Mark Harris et al. 2007. Optimizing parallel reduction in CUDA. *NVIDIA developer technology* 2, 4 (2007), 1–39.

[18] Zheming Jin. 2020. *The rodinia benchmark suite in SYCL*. Technical Report. Argonne National Lab.(ANL), Argonne, IL (United States). Argonne Leadership ....

[19] Zheming Jin and Jeffrey S Vetter. 2022. Understanding performance portability of bioinformatics applications in sycl on an nvidia gpu. In *2022 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE, 2190–2195.

[20] Zheming Jin and Jeffrey S Vetter. 2023. A Benchmark Suite for Improving Performance Portability of the SYCL Programming Model. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 325–327.

[21] Beau Johnston, Jeffrey S. Vetter, and Josh Milthorpe. 2020. Evaluating the Performance and Portability of Contemporary SYCL Implementations. In *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 45–56. https://doi.org/10.1109/P3HPC51967.2020.00010

[22] S Joube, H Grasland, D Chamont, and E Brunet. 2023. Comparing SYCL data transfer strategies for tracking use cases. *Journal of Physics: Conference Series* 2438, 1 (feb 2023), 012018. https://doi.org/10.1088/1742-6596/2438/1/012018

[23] Yinan Ke, Mulya Agung, and Hiroyuki Takizawa. 2021. NeoSYCL: A SYCL Implementation for SX-Aurora TSUBASA. In *The International Conference on High Performance Computing in Asia-Pacific Region* (Virtual Event, Republic of Korea) *(HPC Asia 2021)*. Association for Computing Machinery, New York, NY, USA, 50–57. https://doi.org/10.1145/3432261.3432268

[24] Milind Kulkarni, Martin Burtscher, Calin Casçaval, and Keshav Pingali. 2009. Lonestar: A suite of parallel irregular programs. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 65–76.

[25] Sohan Lal, Aksel Alpay, Philip Salzmann, Biagio Cosenza, Alexander Hirsch, Nicolai Stawinoga, Peter Thoman, Thomas Fahringer, and Vincent Heuveline. 2020. SYCL-Bench: A Versatile Cross-Platform Benchmark Suite for Heterogeneous Computing. In *Euro-Par 2020: Parallel Processing*, Maciej Malawski and Krzysztof Rzadca (Eds.). Springer International Publishing, Cham, 629–644.

[26] Andrew Lavin. 2015. Fast Algorithms for Convolutional Neural Networks. *CoRR* abs/1509.09308 (2015). arXiv:1509.09308 http://arxiv.org/abs/1509.09308

[27] 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* (Munich, Germany) *(IWOCL'21)*. Association for Computing Machinery, New York, NY, USA, Article 8, 13 pages. https://doi.org/10.1145/3456669.3456701

[28] Nenad Mijić and Davor Davidović. 2023. Benchmark DPC++ code and performance portability on heterogeneous architectures. In *2023 46th MIPRO ICT and Electronics Convention (MIPRO)*. 331–337. https://doi.org/10.23919/MIPRO57284.2023.10159832

[29] Tetsu Narumi, Yousuke Ohno, Noriaki Okimoto, Takahiro Koishi, Atsushi Suenaga, Noriyuki Futatsugi, Ryoko Yanai, Ryutaro Himeno, Shigenori Fujikawa, Makoto Taiji, and Mitsuru Ikei. 2006. Gordon Bell finalists II - A 55 TFLOPS simulation of amyloid-forming peptides from yeast prion Sup35 with the special-purpose computer system MDGRAPE-3. In *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing, November 11-17, 2006, Tampa, FL, USA*. ACM Press, 49. https://doi.org/10.1145/1188455.1188506

[30] Goutham Kalikrishna Reddy Kuncham, Rahul Vaidya, and Mahesh Barve. 2021. Performance Study of GPU applications using SYCL and CUDA on Tesla V100 GPU. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC49654.2021.9622813

[31] Istvan Z Reguly. 2023. Evaluating the performance portability of SYCL across CPUs and GPUs on bandwidth-bound applications. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. 1038–1047.

[32] Philip Salzmann, Fabian Knorr, Peter Thoman, Philipp Gschwandtner, Biagio Cosenza, and Thomas Fahringer. 2023. An Asynchronous Dataflow-Driven Execution Model For Distributed Accelerator Computing. In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 82–93. https://doi.org/10.1109/CCGrid57682.2023.00018

[33] David E. Shaw, J.P. Grossman, Joseph A. Bank, Brannon Batson, J. Adam Butts, Jack C. Chao, Martin M. Deneroff, Ron O. Dror, Amos Even, Christopher H. Fenton, Anthony Forte, Joseph Gagliardo, Gennette Gill, Brian Greskamp, C. Richard Ho, Douglas J. Ierardi, Lev Iserovich, Jeffrey S. Kuskin, Richard H. Larson, Timothy Layman, Li-Siang Lee, Adam K. Lerer, Chester Li, Daniel Killebrew, Kenneth M. Mackenzie, Shark Yeuk-Hai Mok, Mark A. Moraes, Rolf Mueller, Lawrence J. Nociolo, Jon L. Peticolas, Terry Quan, Daniel Ramot, John K. Salmon, Daniele P. Scarpazza, U. Ben Schafer, Naseer Siddique, Christopher W. Snyder, Jochen Spengler, Ping Tak Peter Tang, Michael Theobald, Horia Toma, Brian Towles, Benjamin Vitale, Stanley C. Wang, and Cliff Young. 2014. Anton 2: Raising the Bar for Performance and Programmability in a Special-Purpose Molecular Dynamics Supercomputer. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 41–53. https://doi.org/10.1109/SC.2014.9

[34] Peter Thoman, Daniel Gogl, and Thomas Fahringer. 2021. Sylkan: Towards a Vulkan Compute Target Platform for SYCL *(IWOCL'21)*. Association for Computing Machinery, New York, NY, USA, Article 3, 12 pages. https://doi.org/10.1145/3456669.3456683

[35] Peter Thoman, Facundo Molina Heredia, and Thomas Fahringer. 2022. On the Compilation Performance of Current SYCL Implementations. In *International Workshop on OpenCL* (Bristol, United Kingdom, United Kingdom) *(IWOCL'22)*. Association for Computing Machinery, New York, NY, USA, Article 6, 12 pages. https://doi.org/10.1145/3529538.3529548

[36] Jeffrey S. Vetter, Ron Brightwell, Maya Gokhale, Pat McCormick, Rob Ross, John Shalf, Katie Antypas, David Donofrio, Travis Humble, Catherine Schuman, Brian Van Essen, Shinjae Yoo, Alex Aiken, David Bernholdt, Suren Byna, Kirk Cameron, Frank Cappello, Barbara Chapman, Andrew Chien, Mary Hall, Rebecca Hartman-Baker, Zhiling Lan, Michael Lang, John Leidel, Sherry Li, Robert Lucas, John Mellor-Crummey, Paul Peltz Jr., Thomas Peterka, Michelle Strout, and Jeremiah Wilke. 2018. Extreme Heterogeneity 2018 - Productive Computational Science in the Era of Extreme Heterogeneity: Report for DOE ASCR Workshop on Extreme Heterogeneity. (12 2018). https://doi.org/10.2172/1473756