SYCL-Bench: A Versatile Cross-Platform Benchmark Suite for Heterogeneous Computing

Sohan Lal[1], Aksel Alpay[2], Philip Salzmann[3], Biagio Cosenza[4], Alexander Hirsch[3], Nicolai Stawinoga[1], Peter Thoman[3], Thomas Fahringer[3], and Vincent Heuveline[2]

[1] Technical University of Berlin, Germany
[2] Heidelberg University, Germany
[3] University of Innsbruck, Austria
[4] University of Salerno, Italy

Abstract. The SYCL standard promises to enable high productivity in heterogeneous programming of a broad range of parallel devices, including multicore CPUs, GPUs, and FPGAs. Its modern and expressive C++ API design, as well as flexible task graph execution model give rise to ample optimization opportunities at run-time, such as the overlapping of data transfers and kernel execution. However, it is not clear which of the existing SYCL implementations perform such scheduling optimizations, and to what extent. Furthermore, SYCL's high level of abstraction may raise concerns about sacrificing performance for ease of use. Benchmarks are required to accurately assess the performance behavior of high-level programming models such as SYCL. To this end, we present SYCL-Bench, a versatile benchmark suite for device characterization and runtime benchmarking, written in SYCL. We experimentally demonstrate the effectiveness of SYCL-Bench by performing device characterization of the NVIDIA TITAN X GPU, and by evaluating the efficiency of the hipSYCL and ComputeCpp SYCL implementations.

Keywords: SYCL Benchmarks · Heterogeneous Computing · SYCL Runtime · Cross Platform.

1 Introduction

The pursuit of high performance and energy efficiency led to the emergence of heterogeneous computing, where different parts of an application benefit from specialized hardware better suited for the problem. Hardware accelerators such as GPUs, FPGAs, and many-core CPUs are used as co-processors resulting in heterogeneous architectures. To achieve optimal performance, such hardware typically also requires dedicated code paths. However, existing programming models either lack industry support, are specific to certain vendors (such as NVIDIA's CUDA), or too low level and cumbersome to use (e.g. OpenCL) to find universal adoption. SYCL[12] is a recent, royalty-free open standard published by the Khronos Group intended for programming a wide range of heterogeneous architectures. Its high-level single-source programming model combines the portability of OpenCL with modern C++ constructs and idioms. Mundane tasks such as scheduling, data management, and synchronization are handled implicitly by the SYCL runtime, increasing programmer productivity. While the SYCL runtime may automatically perform optimizations such as overlapping data transfers and kernel executions, it is not apparent whether any particular implementation actually employs such optimizations for a given code pattern. As SYCL is a recent standard, to the best of our knowledge only individual benchmarks exist to evaluate the different implementations, whereas a cross-platform benchmark suite has not yet been proposed. We present *SYCL-Bench*[5], a versatile benchmark suite written in SYCL. The main goal of SYCL-Bench is to evaluate the performance of both devices and different SYCL implementations. To this end, SYCL-Bench not only contains benchmarks to characterize hardware, but also SYCL-specific benchmarks that present optimization opportunities to the SYCL runtime and test how well a particular implementation capitalizes on those opportunities. In summary, we make the following main contributions:

- We present the first benchmark suite designed specifically for SYCL: SYCL-Bench includes 62 codes suited for hardware characterization and 9 codes to evaluate SYCL-specific runtime features.
- The benchmark suite models various use cases and enables detailed evaluation of different SYCL implementations and their optimization strategies, thereby facilitating adoption and further development of SYCL.
- We experimentally demonstrate the effectiveness of SYCL-Bench by performing device characterization on an NVIDIA GTX TITAN X and by evaluating two different implementations, hipSYCL and ComputeCpp.

2 The SYCL programming model

SYCL is a programming model for heterogeneous computing that builds on pure C++. This means that SYCL does not extend the C++ language itself in any way. As a SYCL program is always a valid C++ program, a SYCL implementation for CPUs can be implemented without requiring a dedicated compiler. This property can, for example, be used to debug heterogeneous applications written in SYCL with regular CPU debugging tools. When accelerators are targeted, a SYCL implementation requires a dedicated SYCL compiler that identifies kernels, extracts them, and compiles them either into an intermediate representation (such as SPIR or PTX) or machine code for the accelerator. The resulting device binary is then typically embedded by the SYCL implementation within the host binary for execution. Unlike OpenCL, where kernel code is usually either loaded at runtime from a source file or stored in an application as a string, kernel code and host code in SYCL are stored in the same source file, similarly to e.g. CUDA. SYCL is, therefore, a *single-source* programming model, enabling modern C++ design approaches such as templates to work seamlessly and in a type-safe manner across boundaries of host and device code.

[5] https://github.com/bcosenza/sycl-bench

In SYCL, the execution of data parallel kernels is organized by a task graph. This task graph is implicitly constructed by the SYCL runtime based on data access specifications that a programmer associates with a kernel by constructing *accessor* objects. If two kernels request conflicting accesses to the same data (e.g. both require read-write access), the SYCL runtime introduces a dependency between the two kernels based on the order in which they were submitted. Note that this only guarantees correctness with respect to the execution order of multiple kernels, race conditions within kernels (e.g., on the level of individual instructions) are not covered.

SYCL follows the execution and memory model of OpenCL: *work items* constitute a unit of work that is processed in parallel. They are grouped in *work groups*. Within a work group, the execution of work items can be synchronized. There is a host memory, a global memory on the accelerator, local memory that is shared between the work items of a group and per-work-item private memory.

SYCL kernels can be submitted in four different ways:

- A single, non-parallel task is submitted using `single_task()` functionality.
- A basic `parallel_for` mechanism that, from the programmer's point of view, does not group parallel work items together in work groups.
- An hierarchical parallel for, where a first level of parallelism for the work groups is initiated using `parallel_for_work_group()`. Inside the invocation `parallel_for_work_group()`, another level of parallelism can be created using `parallel_for_work_item()`. With hierarchical parallel for kernels, the programmer can optionally control the work group size that will be used to execute the kernel on the hardware. Additionally, local memory can be used in these types of kernels.
- `ndrange parallel_for` provides a method for invoking kernels that grants explicit control over work group sizes, allowing the usage of local memory and explicit barriers in SYCL code. In principle, it is not more powerful than hierarchical parallel for, but rather provides a programming model that is more familiar to programmers who have a background in OpenCL or CUDA.

While SYCL is still a relatively new programming model with the first implementation reaching official specification conformance in August 2018[13], there is a growing SYCL ecosystem including projects such as the SYCL parallel STL, a Tensorflow port to SYCL as well as four major SYCL implementations: Codeplay's commercial ComputeCpp[5], the open-source LLVM-based SYCL[10] led by Intel, hipSYCL[1], an open-source implementation led by Heidelberg University, as well as triSYCL[20], an open-source project mainly funded by Xilinx. Together, these four implementations allow a SYCL program to target any CPU[6], GPUs from at least four vendors, and FPGAs from two vendors. Table 1 summarizes different implementations and supported platforms.

[6] Given that a suitable C++ compiler exists for the hardware.

Table 1: A summary of different SYCL implementations, backends, supported platforms, and specification conformance.

Implementation	Backends	Supported hardware	Conformance
ComputeCpp	OpenCL SPIR/SPIR-V OpenCL PTX (Experimental)	Intel CPUs, Intel GPUs, ARM Mali NVIDIA GPUs	SYCL 1.2.1
hipSYCL	CPU (OpenMP) CUDA ROCm	any CPU NVIDIA GPUs AMD GPUs	pre-conformance
LLVM SYCL	OpenCL SPIR-V CUDA (Experimental)	Intel CPUs Intel GPUs, Intel FPGAs NVIDIA GPUs	pre-conformance
triSYCL	CPU (OpenMP, TBB) OpenCL SPIR (Experimental)	any CPU Xilinx FPGAs	pre-conformance

3 Benchmarks Design Methodology

SYCL-Bench has been designed to accomplish multiple goals. First, like traditional benchmark suites, it contains benchmarks designed to characterize the performance of existing and future hardware that can be programmed using SYCL. The range of potential target architectures is very broad: it includes all OpenCL-conformant devices, addressed with the approach defined by SYCL 1.2.1 of interpreting SYCL as a higher-level model for OpenCL[7]; alternatively, SYCL implementations may support additional ways to target specific hardware without using OpenCL (e.g., hipSYCL targets NVIDIA and AMD devices by extending Clang's CUDA frontend with support for SYCL constructs). For device characterization, particular attention is given to GPU architectures, addressed with a specific set of microbenchmarks. This set of architectural microbenchmarks is complemented by a set of applications and single kernels.

The SYCL programming model and its peculiar aspects are also central to the design of the benchmarks. For example, the benchmark codes are written in modern C++, using template types to broaden the evaluation set.

Additionally, SYCL-Bench includes a number of codes that explicitly create complex inter-task dependencies, thus implicitly stressing the efficiency of the SYCL runtime implementations. Since SYCL implementations may implement the various mechanisms to submit SYCL kernels (see Section 2) differently and with varying performance characteristics, many benchmarks include variants for several of those mechanisms. Lastly, we also present a set of synthetic patterns

[7] This approach assumes the existence of one or more OpenCL implementations available on the host machine. If no OpenCL implementation is available, then the SYCL implementation provides only the SYCL host device to run kernels on [12].

to benchmark the SYCL runtime overhead and task throughput. To summarize, SYCL-Bench contains three categories of benchmarks:

Microbenchmarks A set of architectural microbenchmarks with different patterns stressing different hardware subsystems, e.g. arithmetic or the memory subsystem. They have been designed to emphasize performance characterization on GPU devices.

Applications/Kernels These are real-world applications and kernels from different domains such as linear algebra, image processing, molecular dynamics. The main goal of this category is to test the performance of different devices and SYCL implementations for real-world code patterns.

SYCL Runtime Benchmarks These benchmarks are designed to stress the SYCL runtime. This category includes multiple-kernels that can generate different task graphs and stress different aspects of the SYCL runtime. Examples include the benchmarks to measure the scheduling latency and the capabilities of the SYCL implementation to automatically overlap compute operations and data transfers.

3.1 Microbenchmarks

We present five distinct microbenchmarks designed to quantitatively evaluate various device performance characteristics through the lens of SYCL. The first, `DRAM`, measures the achievable device memory bandwidth by copying single and double precision floating-point values between two buffers. As an added twist, it can also measure the performance for two and three-dimensional buffers, thus indirectly quantifying how efficient a given SYCL implementation's mapping of higher-dimensional indices to the underlying hardware is. The `local_mem` benchmark is similar in spirit, measuring the attainable local memory bandwidth by repeatedly swapping single and double precision floating-point values inside a work group's local memory allocation.

The `arith` and `sf` benchmarks exercise the device's main arithmetic units and special function units, respectively. Both execute a tight loop, the former doing repeated multiply-add operations, and the latter applying three trigonometric functions (sin, cos, tan) in series. Finally, `host_device_bandwidth` measures the transfer bandwidth between the host and device memory, by copying large, contiguous and strided chunks of one, two, and three-dimensional buffers.

3.2 Applications/Kernels

To ensure the diversity of the benchmark suite, it is essential to include applications/kernels from different domains. Even applications from the same domain may exhibit different features. Therefore, we include applications/kernels from a wide range of domains such as image processing, linear algebra, data mining, data analytics. There are mainly two sources of applications and kernels. We ported 15 CUDA applications/kernels from PolyBench suite [8] to SYCL and developed 9 additional SYCL applications/kernels to cover image processing, data analytics, and physics simulation domains. In addition, applications and kernels are also equipped with a functional validation framework that further validates SYCL implementations on a wide range of benchmarks. Table 2 shows the list of applications and kernels along with their domain.

3.3 SYCL Runtime Benchmarking

DAG Task Throughput for Sequential Tasks In this benchmark, for a given problem size N, N kernels are launched that request read-write access to the same buffer, and the time from submission to the completion of all N kernels is measured. Because more than one kernel accesses the same memory object, a read-write conflict arises that forces the SYCL runtime to process the kernels sequentially. These memory accesses, therefore, represent an edge in the resulting DAG (directed acyclic graph). In order to verify that each kernel has completed successfully, the buffer holds a counter which is incremented by one by each kernel. Since the kernel itself is trivial, this benchmark is dominated by the scheduling latency of the SYCL implementation and the latency of the backend used by the SYCL implementation (e.g. OpenCL for ComputeCpp or HIP for hipSYCL). Because SYCL implementations may have different scheduling code paths or different amounts of execution overhead for different types of kernel invocations, this benchmark comes in variants that utilize the various mechanisms in SYCL to submit kernels (`single_task`, basic `parallel_for`, `ndrange parallel_for` and hierarchical `parallel_for`).

DAG Task Throughput for Independent Tasks The DAG task throughput benchmark for independent tasks is very similar to the benchmark described in Section 3.3. However, here given a problem size N, N independent tasks are spawned. The independence is guaranteed by creating one buffer per kernel so that each kernel only accesses its own buffer and no conflicts arise. To verify that each kernel has been executed successfully, each kernel simply sets the content of the buffer to a unique number that is different for each kernel submission. The runtime to submit and complete all kernels is measured. While this benchmark is also sensitive to the scheduling latencies and overheads in the SYCL implementation, it additionally allows the SYCL implementation to exploit hardware concurrency, such as running multiple kernels concurrently on a device to improve the overall throughput. Note that, while more complex dependencies between tasks compared to our two throughput benchmarks may be interesting to increase the load on the SYCL task synchronization mechanisms and the task dependency analysis, the throughput benchmarks provide a way of testing two easy-to-understand extreme cases: The case where no tasks can be run concurrently (sequential throughput benchmark) and the case where everything can run concurrently (independent throughput benchmark). They can therefore be used to estimate the overhead that can be expected from a SYCL implementation in ideal, well-defined scenarios.

Block Transform The blocked transform benchmark divides an input array into chunks of configurable size, and submits a kernel for each chunk that requests read/write access only to its chunk. Each kernel then performs a tunable number of Mandelbrot iterations on the input data. This only serves as a dummy workload to extend the kernel runtime. The actual focus of the benchmark is to test whether the SYCL implementation is able to automatically overlap the data transfers needed to copy the chunk data to the device and the kernels operating on each chunk. Because the kernels are independent, a SYCL implementation might even be able to execute multiple kernels concurrently, if this is supported by the hardware. Additionally, the benchmark is also sensitive to whether the SYCL implementation is capable of transferring data at sub-buffer granularity at all (i.e., individual data transfers per chunk). When running on CPU, a SYCL implementation might also be able to remove the data transfers entirely as the kernel and host would be running in the same memory space. This can also be investigated with this benchmark. The resulting DAG is illustrated in Figure 1.

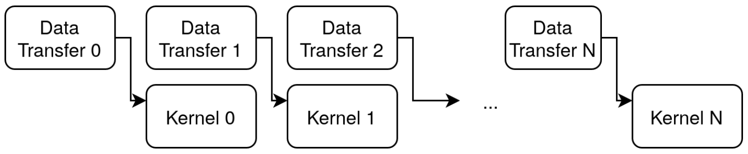

Fig. 1: The DAG for blocked transform. The arrows represent dependencies.

Table 2 shows the full list of benchmarks in the SYCL-Bench suite. The three categories **Micro**, **Application/Kernel**, and **Runtime** contain five, twenty four, and three benchmarks, respectively. We also leverage SYCL's support for C++ templates to instantiate benchmarks with different data types. As a result, the SYCL-Bench consists of 26 microbenchmarks codes, 36 applications/kernels codes (total 62 codes for hardware characterization) and 9 codes to evaluate the efficiency of the SYCL-runtime.

4 Experimental Evaluation

We present results obtained on a machine equipped with both a high-end NVIDIA GPU and Intel CPU, representing two important target architectures for SYCL. However, given the selective support of different hardware platforms in current SYCL implementations as shown in Table 1, the set of implementations to compare was limited. It was, therefore, necessary to restrict the evaluated SYCL implementations to a common denominator that supports both our CPU and GPU, namely hipSYCL and ComputeCpp. Table 3 shows details of our experimental setup.

[8] Ported from PolyBench suite [8].

Table 2: The detailed list of benchmarks included in the SYCL-Bench suite.

Category	Benchmark Name	Short	Domain
Micro	arith, DRAM, local_mem, sf	-	Microbench
	host_device_bandwidth	-	Microbench
App/Kernel	lin_reg_coeff	LRC	Data Analytics
	lin_reg_error	LRE	Data Analytics
	median	MEDIAN	Image Processing
	mol_dyn	MD	Physics Simulation
	scalar_prod	SP	Linear Algebra
	sobel3/5/7	SOBEL3/5/7	Image Processing
	vec_add	VA	Linear Algebra
	2DConvolution[8]	2DCON	Image Processing
	2mm[8]	2MM	Linear Algebra
	3DConvolution[8]	3DCON	Image Processing
	3mm[8]	3MM	Linear Algebra
	atax[8]	ATAX	Linear Algebra
	bicg[8]	BICG	Linear Algebra
	correlation[8]	CORR	Data Mining
	covariance [8]	COV	Data Mining
	fdtd2d[8]	FTD2D	Stencils
	gemm[8]	GEMM	Linear Algebra
	gesummv[8]	GESUM	Linear Algebra
	gramschmidt[8]	GRAMS	Linear Algebra
	mvt[8]	MVT	Linear Algebra
	syr2k[8]	SYR2K	Linear Algebra
	syrk[8]	SYRK	Linear Algebra
Runtime	blocked_transform	BT	Microbench
	dag_task_throughput_independent	DTI	Microbench
	dag_task_throughput_sequential	DTS	Microbench

4.1 ComputeCpp PTX Performance

In order to target NVIDIA GPUs with ComputeCpp 1.3, it is necessary to use the experimental[9] ComputeCpp PTX backend. This is because the NVIDIA OpenCL implementation does not support ingesting kernels in the SPIR format, which is normally used by ComputeCpp. Because of the experimental quality of this backend, we expect to see an overall lower performance in microbenchmarks and applications/kernels when compared to hipSYCL.

However, we found that even very short running kernels (of the order of microseconds, when executed using hipSYCL) could sometimes run for tens of *milliseconds*. In fact, with very high probability ($> 90\%$) the third consecutive run of a very short running kernel would inexplicably require approximately

[9] `https://developer.codeplay.com/products/computecpp/ce/guides/platform-support/targeting-nvidia-ptx?version=1.3.0`

Table 3: Hardware and software used for our experiments.

Hardware	Intel Xeon CPU E5-2699 v3	2.30 GHz 32 GiB	DDR4
	NVIDIA GTX TITAN X (Maxwell)	1.0 GHz / 1.215 GHz	(boost)
Software	Ubuntu 16.04	Linux 4.15	Clang 9.0.1
	NVIDIA OpenCL 1.2	Intel OpenCL 2.0	CUDA 10.1
	hipSYCL 0.8.1-master(`12406c8c`)	ComputeCpp 1.3	

100 milliseconds to complete. As a workaround for this performance anomaly, we determined that by using SYCL's built-in event profiling capabilities, we were able to obtain timings that were in line with our expectations. As these timings reflect the actual kernel execution time in hardware, relying solely on them would give ComputeCpp an unfair advantage over hipSYCL's host-side timings. We, therefore, decided to proceed as follows: For all measurements taken on NVIDIA hardware using ComputeCpp, we will provide two values. Results marked as **ComputeCpp PTX** include the full execution time as observed by the user, including the inexplicable overhead. A second value, **Kernel only**, shows the execution time that is close to what could ideally be expected without the overhead. Crucially however, unlike for hipSYCL, these results include no runtime, driver, and kernel launch overhead. Note that we cannot rely on event profiling in general, as this functionality is currently not available in some SYCL implementations, including hipSYCL.

4.2 Microbenchmarking

This section describes the results we obtained by running the benchmarks described in Section 3.1 on an NVIDIA GTX TITAN X. Figure 2 shows the microbenchmarking results. All microbenchmarks were executed 20 times, and we present the best result obtained out of these runs. Missing bars indicate failed verification of benchmark results. For the `DRAM` benchmark 3.375 GiB of memory were copied between two buffers. As can be seen in Figure 2a, ComputeCpp's real-world performance is limited considerably by the aforementioned performance bug. Considering ComputeCpp's kernel time only, both implementations achieve about 78% of the Titan X's 336.6 GiB/s theoretical maximum for one and two-dimensional single and double-precision floating-point copies. For three-dimensional copies, hipSYCL exhibits a significant drop in throughput. On first sight, this might indicate a choice of work group size that does not allow for full memory coalescing. However, closer investigation reveals that the computation of linear buffer offsets becomes too expensive in three dimensions to be completely hidden by DRAM access latencies. More specifically, the device code generated by hipSYCL performs the same linear offset computation twice, once for the reading buffer access, and another time for the write access. Explicitly computing the linear offset once within the kernel, and using raw pointer accesses for reading and writing, alleviates this inefficiency and allows hipSYCL

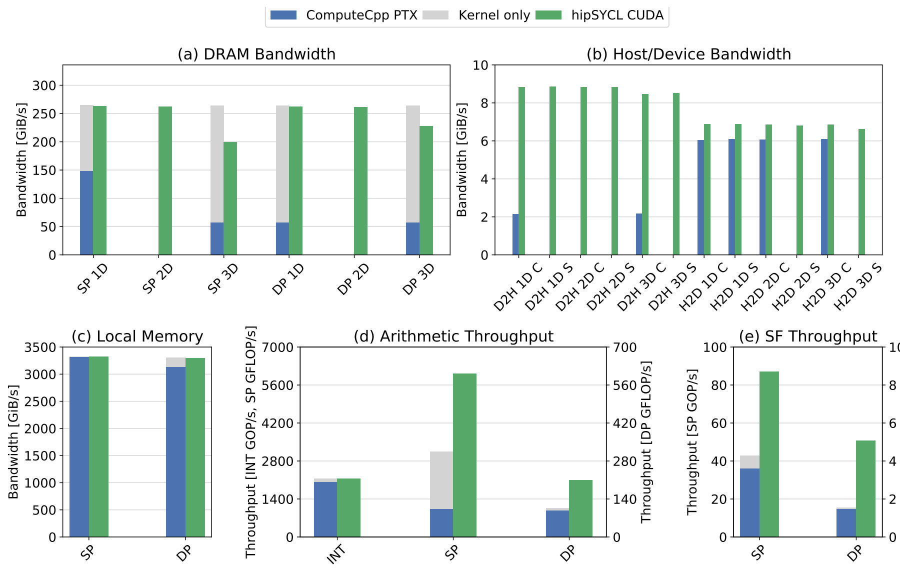

Fig. 2: Microbenchmarking results on NVIDIA GTX TITAN X.

to achieve full throughput again. Figure 2b indicates that host $\leftrightarrow$ device copy bandwidth is relatively unaffected by the type of transfer performed, with only three-dimensional contiguous and strided device-to-host copies dipping slightly for hipSYCL. ComputeCpp's performance is somewhat worse for host-to-device copies and considerably worse for device-to-host copies. Furthermore, many of the variants could not correctly be verified for ComputeCpp, which we again attribute to the experimental nature of the PTX backend. Figure 2c shows similar local memory performance for both implementations, achieving approximately 3300 GiB/s for single and double precision copies. This is in line with results for the GTX TITAN X published by Lopes et al.[16], when adjusting for the higher boost clock used in our testing setup.

Moving on to arithmetic throughput in Figure 2d, we see that integer performance is considerably lower than single-precision performance for both implementations. This is to be expected, as the IMAD instructions used in `arith` are emulated on Maxwell[11]. Curiously, with 3134 single-precision GFLOP/s, even for the idealized kernel-only measurement, ComputeCpp achieves little more than half of hipSYCL's 6016 GFLOP/s. This indicates that the device compiler might not map the benchmark kernel's multiply-add operations to corresponding FMA instructions. Examination of the PTX device code generated by ComputeCpp confirms that this is indeed the case. Both implementations approximately achieve the expected $1/32$-th in double-precision performance compared to single precision. Finally, for the `sf` benchmark's result, shown in Figure 2e, we again see ComputeCpp achieving only about half of hipSYCL's single-precision performance, and a third in double precision. However, even hipSYCL's perfor-

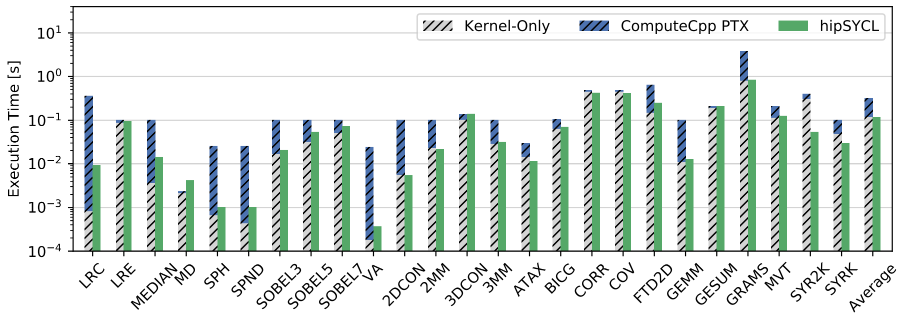

Fig. 3: hipSYCL and ComputeCpp runtime on NVIDIA TITAN X.

mance is much lower than the theoretical maximum (which, with Maxwell's 32 SFUs per SM should be approximately 1000 GOP/s, depending on clock speed). Examination of the PTX device code reveals that both implementations emulate the trigonometric functions rather than mapping to the corresponding SFU intrinstics (e.g. CUDA's `__cosf`). At the time of writing, it is therefore not possible to benchmark SFU throughput on NVIDIA using either SYCL implementation.

4.3 Applications / Kernels

Figure 3 and Figure 4 show the execution time of benchmarks using hipSYCL and ComputeCpp implementations running on NVIDIA GTX TITAN X and Intel Xeon CPU, respectively. For measuring the execution time, we run each benchmark 10 times and pick the median of the samples. In this work, we focus on 32-bit data types.

Figure 3 shows that hipSYCL outperforms the ComputeCpp implementation across most of the benchmarks. On average, hipSYCL is 2.7× faster than ComputeCpp on NVIDIA TITAN X. As mentioned earlier, this is primarily due to the experimental PTX backend support for NVIDIA GPUs which has limitations such as no support for OpenCL builtins. The suffixes ND and H are used to differentiate between `ndrange parallel_for` and `hierarchical parallel_for` implementations. The `scalar_prod` (SP) benchmark provides both variants.

Figure 4 shows the execution time of benchmarks using hipSYCL and ComputeCpp implementations running on Intel Xeon CPU. Figure 4 shows that the setup consisting of ComputeCpp with Intel's OpenCL implementation outperforms hipSYCL with the LLVM OpenMP implementation across most of the benchmarks except SOBEL3, SOBEL5, SOBEL7, and 2DCON benchmarks. On average, ComputeCpp is 25.2× faster than hipSYCL on CPU. The main reason for the higher execution time for hipSYCL is that some benchmarks use `ndrange parallel_for`. `ndrange parallel_for` cannot be implemented efficiently by library-based SYCL implementations without dedicated compiler support (such as the CPU backends in hipSYCL and triSYCL) because it allows

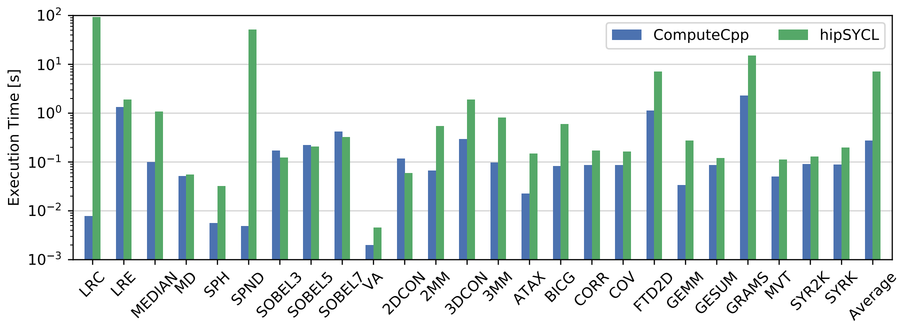

Fig. 4: Comparison of hipSYCL and ComputeCpp runtime on Intel Xeon CPU.

for explicit barriers. To implement correct barrier semantics, a SYCL implementation is forced to launch one thread per SYCL work item. For the usual fine-grained parallelism exposed in typical SYCL applications at the work item level, this forces the SYCL implementation to spawn a very large number of threads (often much more than numbers of cores available in typical CPUs), each of which is only assigned a small amount of work. This parallelization scheme is not a good fit for CPU architectures. If we take out benchmarks which implement `ndrange parallel_for` (LRC, SPND), ComputeCpp is 4.5× faster than hipSYCL.

Of the evaluated benchmarks in this category, for instance, SPND and LRC use `ndrange parallel_for`. The execution time of LRC and SPND are 94.07 (s) and 51.33 (s) using hipSYCL compared to 0.008 (s) and 0.005 (s) using ComputeCpp, respectively. Therefore, for applications that are expected to show performance portability, it is highly recommended to prefer hierarchical parallel for over `ndrange parallel_for`. The figure shows that the `hierarchical parallel_for` implementation (SPH) is significantly faster for `scalar_prod`. It will be interesting for future work to test other C++ compilers and OpenMP implementations with hipSYCL (e.g., Intel C++ Compiler with Intel OpenMP implementation).

4.4 SYCL Runtime

We measured the runtime of hipSYCL and ComputeCpp implementations using `dag_task_throughput_sequential` and `dag_task_throughput_independent` benchmarks on NVIDIA TITAN X. We varied the problem size that corresponds to the number of submitted kernels. For the `dag_task_throughput_sequential`, we observed that not only is the SYCL runtime overhead almost the same for hipSYCL and ComputeCpp, but also for the four different kernel invocations. This is likely because for sequential GPU tasks, runtimes are dominated by latencies below the level of the SYCL implementation (driver, PCIe, GPU).

Figure 5 shows the SYCL runtime overhead of hipSYCL and ComputeCpp implementations when executing `dag_task_throughput_independent` on NVIDIA

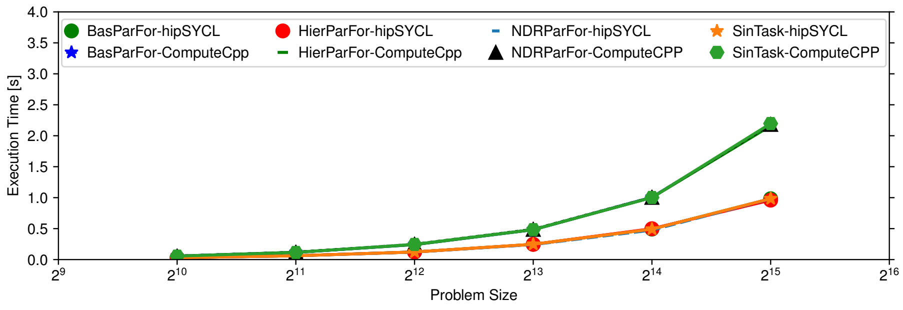

Fig. 5: SYCL runtime overhead of hipSYCL and ComputeCpp implementations on NVIDIA GTX TITAN X. The DAG consists of independent tasks.

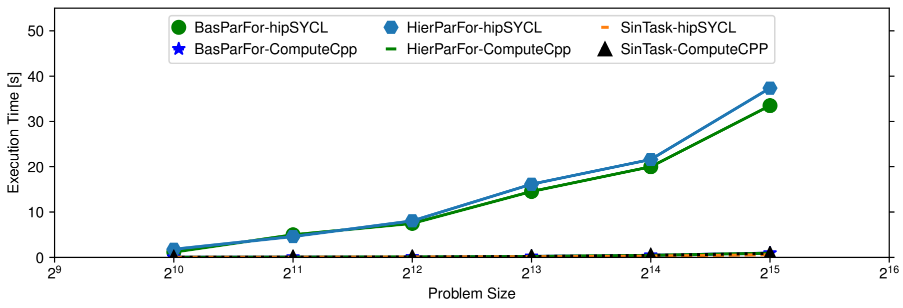

Fig. 6: SYCL runtime overhead of hipSYCL and ComputeCpp implementations on Intel Xeon CPU. The DAG consists of independent tasks.

TITAN X. In contrast to the sequential tasks, we see that the ComputeCpp stack exhibits a higher runtime overhead compared to the hipSYCL implementation. Moreover, the gap is proportional to the number of submitted tasks, which means that ComputeCpp has a higher average latency per submitted kernel. There are two possible explanations for this behavior. An implementation could be faster for this test because it executes the kernels concurrently on the hardware, or because it has a lower scheduling overhead. Since hipSYCL does not launch kernels concurrently for this benchmark, the performance performance gap can be explained by a lower scheduling overhead in hipSYCL.

Figure 6 shows the execution time of the `dag_task_throughput_independent` benchmark on Intel Xeon CPU. As shown in the figure, submitting independent kernels is significantly slower for hipSYCL basic parallel for and hierarchical parallel for kernels compared to hipSYCL single task kernels and ComputeCpp. Since basic and hierarchical parallel for kernel invocations in hipSYCL require spawning OpenMP threads, OpenMP overheads are likely an explanation for this behavior.

5 Related Work

Benchmarking has been used to characterize heterogeneous architectures and different programming models [3,4,2,9,7,6,14]. Che et al. [3] proposed Rodinia benchmark suite to study emerging platforms such as GPUs. Che et al. [4] later extended their work and characterized Rodinia benchmark suite and also compared to contemporary CMP workloads. Burtscher et al. [2] did a quantitative study of irregular programs on GPUs and presented two metrics called control-flow irregularity and memory-access irregularity and investigated how irregular GPU kernels differ from regular kernels with respect to these metrics. Kulkarni et al. [14] presented a benchmark suite called Lonestar that is targeted for graph-based irregular programs and characterized the first five programs from this suite. The results show that even irregular applications can be accelerated using modern multi-core machines. Fang et al. [7] designed and implemented Mars, a runtime system for distributed data processing. They also ported six representative applications on Mars. Danalis et al. [6] designed a benchmark suite called Scalable HeterOgeneous Computing benchmark suite (SHOC) and used it to compare OpenCL and CUDA programming models. Grauer-Gray et al. [8] implemented PolyBench codes for processing on GPU using CUDA, OpenCL, and HMPP, a pragma-based compiler.

Some researchers have used microbenchmarks as well as benchmarks to understand the performance as well as power characteristics of GPUs [23,17,21,15]. Thoman et al. [21] proposed microbenchmarks suite called uCLbench to characterize and compare OpenCL performance of existing and future devices. Zhang et al. [23] designed a set of microbenchmarks to study the power consumption of different functional units of a GPU. Mei and Chu [17] studied the characteristics of the memory hierarchy using microbenchmarks. Specifically, they investigated GPU cache systems and investigated the throughput/latency of GPU global and shared memory. Lal et al. [15] studied bottlenecks that cause low performance and low energy efficiency in GPU workloads.

There are a few works on SYCL benchmarking as it a relatively new programming model [18,19]. Potter and Keir [18] described a methodology for creating efficient domain specific embedded languages on top of the SYCL for the OpenCL standard. There are also some works which compare different programming models. For example, Silva et al. [19] analyzed the performance and characteristics of SYCL, OpenMP, and OpenCL using two benchmarks. The results indicated that benchmarks that rely on SYCL runtimes are not on par with OpenMP and OpenCL. However, the gap is shrinking compared to previous studies. Thoman et al. [22] developed the Celerity programming environment based on SYCL, enabling developers to scale C++ applications to distributed memory clusters with relative ease, and included some benchmark results comparing against an MPI+OpenCL software stack. While these works provide some limited insight into SYCL performance compared to other programming models, we present the first SYCL benchmark suite that contains a complete and diverse set of benchmarks to characterize both hardware devices and runtime performance aspects of different SYCL implementations.

6 Conclusions

We presented SYCL-Bench, the first benchmark suite specifically written in and for SYCL, featuring three categories of benchmarks and a total of 71 code patterns. We experimentally demonstrated the effectiveness of SYCL-Bench by performing device characterization of the NVIDIA TITAN X GPU, showing that near-peak performance can be achieved for metrics such as arithmetic throughput and DRAM bandwidth. We also evaluated the efficiency of two SYCL implementations: hipSYCL outperformed ComputeCpp on average by 2.7× in real-world performance on TITAN X, and ComputeCpp was 4.5× faster than hipSYCL on Intel Xeon without ndrange benchmarks. While ComputeCpp's performance on TITAN X is primarily hampered by the experimental PTX backend, hipSYCL's CPU performance is much lower because of API constructs that cannot be implemented efficiently without a dedicated compiler. In the future work, we plan to evaluate other SYCL implementations such as triSYCL and Intel SYCL. SYCL-Bench is publicly available along with the testing framework.

Acknowledgement

This research has been partially funded by the FWF (I 3388) and DFG (CO 1544/1-1, project number 360291326) as part of the DACH project CELERITY.

References

1. Alpay, A.: hipSYCL. `https://github.com/illuhad/hipSYCL`
2. Burtscher, M., Nasre, R., Pingali, K.: A Quantitative Study of Irregular Programs on GPUs. In: Proc. IEEE, IISWC (2012)
3. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J., Lee, S.H., Skadron, K.: Rodinia: A Benchmark Suite for Heterogeneous Computing. In: Proceedings of the IEEE International Symposium on Workload Characterization, IISWC (2009)
4. Che, S., Sheaffer, J.W., Boyer, M., Szafaryn, L.G., Wang, L., Skadron, K.: A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads. In: Proceedings of the IEEE International Symposium on Workload Characterization, IISWC (2010)
5. Codeplay Software: ComputeCpp. `https://codeplay.com/products/computesuite/computecpp`
6. Danalis, A., Marin, G., McCurdy, C., Meredith, J.S., Roth, P.C., Spafford, K., Tipparaju, V., Vetter, J.S.: The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In: Proceedings of the 3rd Workshop on GPGPU (2010)
7. Fang, W., He, B., Luo, Q., Govindaraju, N.K.: Mars: Accelerating MapReduce with Graphics Processors. IEEE Transactions on Parallel and Distributed Systems
8. Grauer-Gray, S., Xu, L., Searles, R., Ayalasomayajula, S., Cavazos, J.: Auto-tuning a High-level language Targeted to GPU Codes. In: Proceedings InPar (2012)
9. Group, I.R.: Parboil Benchmarks Suite (2007), http://impact.crhc.illinois.edu/parboil.php
10. Intel: SYCL Compiler. `https://github.com/intel/llvm`

11. Jia, Z., Maggioni, M., Smith, J., Scarpazza, D.P.: Dissecting the nvidia turing t4 gpu via microbenchmarking. arXiv preprint arXiv:1903.07486 (2019)
12. Khronos: SYCL 1.2.1. Tech. rep., Khronos Group, Inc. (2020), `https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf`
13. Khronos Group: Codeplay announces world's first fully-conformant sycl 1.2.1 solution. https://www.khronos.org/news/permalink/codeplay-announces-worlds-first-fully-conformant-sycl-1.2.1-solution (2018)
14. Kulkarni, M., Burtscher, M., Cascaval, C., Pingali, K.: Lonestar: A Suite of Parallel Irregular Programs. In: IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS (2009)
15. Lal, S., Lucas, J., Andersch, M., Alvarez-Mesa, M., Elhossini, A., Juurlink, B.: GPGPU Workload Characteristics and Performance Analysis. In: International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS (2014)
16. Lopes, A., Pratas, F., Sousa, L., Ilic, A.: Exploring gpu performance, power and energy-efficiency bounds with cache-aware roofline modeling. In: 2017 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS. pp. 259–268 (2017)
17. Mei, X., Chu, X.: Dissecting GPU Memory Hierarchy Through Microbenchmarking. IEEE Transactions on Parallel and Distributed Systems (2017)
18. Potter, R., Keir, P., Bradford, R.J., Murray, A.: Kernel composition in sycl. In: Proceedings of the 3rd Workshop on OpenCL, IWOCL (2015)
19. Silva, H.C.D., Pisani, F., Borin, E.: A Comparative Study of SYCL, OpenCL, and OpenMP. In: Proc. SBAC-PADW (2016)
20. The triSYCL Project: triSYCL. `https://github.com/trisycl/trisycl`
21. Thoman, P., Kofler, K., Studt, H., Thomson, J., Fahringer, T.: Automatic OpenCL Device Characterization: Guiding Optimized Kernel Design. In: Euro-Par 2011: Parallel Processing (2011)
22. Thoman, P., Salzmann, P., Cosenza, B., Fahringer, T.: Celerity: High-level c++ for accelerator clusters. In: Yahyapour, R. (ed.) Euro-Par 2019: Parallel Processing. Springer International Publishing (2019)
23. Zhang, Y., Hu, Y., Li, B., Peng, L.: Performance and Power Analysis of ATI GPU: A Statistical Approach. Proc. 6th NAS (2011)