Evaluating the Performance and Portability of Contemporary SYCL Implementations

Beau Johnston
Oak Ridge National Laboratory, and
Australian National University
johnstonbe@ornl.gov

Jeffrey S. Vetter
Oak Ridge National Laboratory
vetter@computer.org

Josh Milthorpe
Australian National University
josh.milthorpe@anu.edu.au

Abstract*—SYCL is a single-source programming model for heterogeneous systems; it promises improved maintainability, productivity, and opportunity for compiler optimization, when compared to accelerator specific programming models. Several implementations of the SYCL standard have been developed over the past few years, including several backends using contemporary accelerator languages, like OpenCL, CUDA, and HIP. These implementations vary widely in their support for specific features of the standard and in their performance. As SYCL grows in popularity, developers need to know how features are implemented across popular implementations in order to make proper design choices. In this paper, we evaluate the existing SYCL implementations for important SYCL features across a range of hardware in order to understand SYCL's performance and portability. This work uses the newest SYCL benchmark suite (SYCL-Bench, 38 kernels) to evaluate these four existing implementations, comparing support of language features across backends and highlighting feature completeness and performance. For features, we focus on the five major SYCL parallel constructs, using a motivating example of the matrix multiplication benchmark. Our results show that the *basic data parallelism* construct is the best choice for performance on current SYCL implementations, and we identify opportunities for improvement in several of the SYCL implementations.

I. Introduction

SYCL [2] is a single-source programming model for heterogeneous systems; it is managed as an open standard by the Khronos Group. The benefit of SYCL, as compared to programming models like OpenCL [1] is that it offers a single-source approach to application development, which can improve maintainability, productivity, and overall opportunity for downstream compiler optimizations. In fact, SYCL sits as a higher level of abstraction, offering backend implementations that map to contemporary accelerator languages, like OpenCL, CUDA [11], and HIP. Many of these lower-level approaches required the kernel and host code to be separate, which can make application development and optimization more complex.

Over the past few years, several implementations of this SYCL standard have emerged: DPC++ [8], ComputeCpp [5], triSYCL [9], and hipSYCL [3]. While the official SYCL standard requires functional equivalence of features across compliant implementations, it does not dictate how the underlying features should be implemented; this approach is similar to standards like MPI [13] or OpenCL. Moreover, each implementation supports multiple specific accelerator devices and may therefore employ a distinct backend for each device. These backends can be viewed as other accelerator programming languages and their associated framework—both compiler and runtime. For instance, ComputeCpp targets an OpenCL backend while DPC++ also offers a CUDA and PTX backend. Figure 1 shows a comprehensive view of the implementation-to-device relationships.

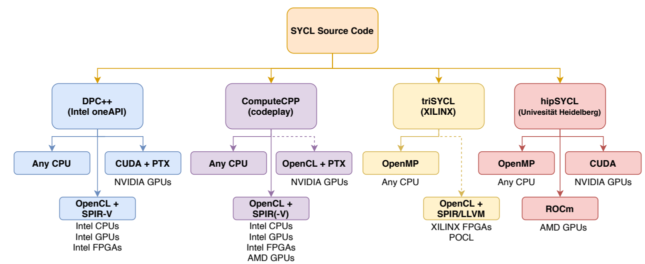


Fig. 1: Current SYCL implementations and their corresponding API backends. (A dashed line indicates experimental support.)

Separately, SYCL-Bench [10] is a recent benchmark suite, developed to characterize hardware and present optimization opportunities for SYCL implementations. As such, it is well positioned to be the primary vehicle for our evaluation.

Simply put, our goal in this effort is to understand the *performance portability* [6], [12] of SYCL, and, in particular, the performance portability of individual SYCL features across these implementations. This characterization will be valuable to developers of both applications and SYCL implementations. First, as users create their applications, it is important for them to understand the performance implications of different features and design patterns in SYCL. As is the case with other programming systems, new language abstractions often obfuscate how application code is mapped to the heterogeneous system. SYCL is no different. Second, as SYCL implementation developers create and optimize their SYCL implementations, they need to understand how applications will use its features so that they can balance the appropriate tradeoffs. More importantly, the applications can reveal how individual SYCL features are combined and how they are used with data structures and other non-SYCL language features that may promote or inhibit good performance.

978-0-7381-1088-2/20/$31.00 ©2020 IEEE
DOI 10.1109/P3HPC51967.2020.00010

TABLE I: Status of SYCL-Bench benchmarks on modern SYCL implementations.

Application	DPC++ (CUDA)	DPC++ (CPU)	DPC++ (OpenCL)	ComputeCpp (CPU)	ComputeCpp (OpenCL)	triSYCL (OpenMP)	hipSYCL (CPU)	hipSYCL (CUDA)	hipSYCL (ROCm)
2DConvolution	❍	❍		❍		❍	❍	❍	❍
2mm	❍	❍		❍		❍	❍	❍	❍
3DConvolution	❍	❍		❍		❍	❍	❍	❍
3mm	❍	❍		❍		❍	❍	❍	❍
DRAM	!	❍		❍			❍	❍	❍
arith				❍	❍	❍	❍	❍	❍
atax	❍	❍		❍	❍	❍	❍	❍	❍
bicg	❍	❍		❍	❍	❍	❍	❍	❍
blocked_transform	❍	❍		❍	❍	✗	❍	❍	❍
correlation	❍	❍		❍	❍	❍	❍	❍	❍
covariance	❍	❍		❍	❍	❍	❍	❍	❍
dag_task_throughput_independent	❍	❍		❍	❍	❍	❍	❍	❍
dag_task_throughput_sequential	❍	❍		❍	❍	❍	❍	❍	❍
fdtd2d	❍	❍		❍	❍	❍	❍	❍	❍
gemm	❍	❍		❍		❍	❍	❍	❍
gesummv	❍	❍		❍	❍	❍	❍	❍	❍
gramschmidt	❍	❍		❍	❍	❍	❍	❍	❍
host_device_bandwidth	!	❍			❍	✗	❍	❍	❍
kmeans	❍	❍		❍	❍	❍	❍	❍	❍
lin_reg_coeff	❍			❍	❍	❍	❍	❍	❍
lin_reg_error	❍	❍		❍	❍	❍	❍	❍	❍
local_mem	❍			❍	❍	❍	❍	❍	❍
matmulchain	❍	❍		❍		❍	❍	❍	❍
median	✗	❍		❍	❍	✗	✗	✗	❍
mol_dyn	❍	❍		❍	❍	❍	❍	❍	❍
mvt	❍	❍		❍	❍	❍	❍	❍	❍
nbody	✗			❍	❍	✗	❍	❍	❍
pattern_L2	❍	❍		❍	❍	❍	❍	❍	❍
reduction	❍			❍	❍	✗	❍	❍	❍
scalar_prod	❍			❍	❍	❍	❍	❍	❍
segmentedreduction	❍			❍	❍	❍	❍	❍	❍
sf	✗	❍		❍	❍	❍	❍	❍	❍
sobel	✗	✗	✗	❍	❍	✗	❍	❍	❍
sobel5	✗	✗	✗	❍	❍	✗	❍	❍	❍
sobel7	✗	✗	✗	❍		✗	❍	❍	❍
syr2k	❍	❍		❍		❍	❍	❍	❍
syrk	❍	❍		❍		❍	❍	❍	❍
vec_add	❍	❍		❍	❍	❍	❍	❍	❍

✗ indicates a failed compilation, blank entries aborted or returned a non-zero return-code, ! did not abort but PI CUDA threw an error, ❍ success on a trial run

We make the following contributions:

1) We use the SYCL-Bench suite to evaluate the functionality of four SYCL implementations (i.e., DPC++, ComputeCpp, triSYCL, hipSYCL) that target multiple backends/devices (i.e., CUDA, CPU, OpenCL, OpenMP, ROCm).
2) Our results reveal the performance portability of specific features of SYCL: basic data-parallel kernels, work-group data-parallel kernels, hierarchical data-parallel kernels, single-task kernels, and synchronization.
3) We perform a detailed evaluation[1] of the major SYCL parallel constructs in the context of a matrix multiplication benchmark, and show that the *basic data parallelism* construct is the best choice for performance on current SYCL implementations.

II. Methodology

We followed a systematic methodology in order to understand the performance portability of SYCL across a range of hardware and SYCL implementations. At the highest level, we used basic timing information to measure raw application performance. However, to understand the resulting application performance, we had to delve into the next level of detail. So, first, we identified the main features of SYCL available to users for developing applications, and, ultimately, decided to focus on the performance of particular *parallel constructs* defined in the SYCL standard. These parallel constructs are: basic data-parallel kernels, work-group data-parallel kernels, hierarchical data-parallel kernels, single-task kernels, and synchronization. Ideally, users would see these features performing similarly across SYCL implementations and similar hardware. Practically, however, we expect SYCL implementations to be optimized for specific workloads and hardware that may result in dramatic performance variability. Second, we surveyed the complete set of 38 kernels from SYCL-Bench (see Table I) to tally which features of SYCL they used. SYCL-Bench divides the benchmarks into 5 categories—micro, pattern, polybench, runtime and single-kernel—but since our investigation is from the parallel construct lens we do not comment about this division or the diversity of the benchmarks in the suite. Third, we installed and evaluated all 38 kernels across the implementations of SYCL and supported hardware (see Figure 1). Fourth, we identified and investigated differences in the kernels and SYCL features across these implementations and devices. In most cases, we had to dive into the underlying SYCL implementation using appropriate tools to understand the realized performance differences. Finally, we tried to extract some common lessons from a) the use of SYCL constructs for performance portability, and b) the impact of SYCL implementation designs on performance portability.

III. SYCL Implementations

As mentioned, several teams have developed SYCL implementations for multiple target architectures and underlying programming systems (e.g., CUDA, OpenCL, OpenMP) (see Figure 1). SYCL offers proprietary extensions which may improve performance on individual backends, but the use of

[1]We offer a Dockerhub image and Dockerfile with Jupyter artifact for interpretable results—see the digital artifact in Appendix A.

which may limit portability. In this work, we focus strictly on the standard, portable SYCL features of these implementations and avoid the use of proprietary extensions. Furthermore, we only examine non-experimental backends, resulting in a comparison across 9 different SYCL runtimes.

This section identifies which of the benchmarks successfully compile and run on each of the SYCL implementations over all available backends. Table I shows the compilation and running status of each SYCL-Bench application evaluated on these SYCL implementations.

Overall, we found that most of the SYCL-bench benchmarks built and ran successfully. We were unable to successfully build against the DPC++ OpenCL backend; thus, we only evaluated 8 of the 9 contemporary/non-experimental SYCL backends. ComputeCpp offer two backends, namely, pthreads, and OpenCL. hipSYCL has three separate backends: OpenMP for CPU devices, CUDA for Nvidia devices, and ROCm for AMD devices. TriSYCL is a header-only SYCL framework, which requires no compilation of dependencies but can only be evaluated on CPUs or architectures, which support OpenMP or Thread Building Blocks as programming models. Our evaluation focuses on SYCL v1.2.1, and uses the implementations from the GitHub hosted versions of DPC++ (git commit: 24726df), hipSYCL (5352add), TriSYCL (b97c97a), and the ComputeCpp CE [5] binary known as “ubuntu-16.04-64bit”. We used the most recent version (7f38669) of SYCL-Bench.

IV. SYCL Parallelism Constructs

Since one of the major features of SYCL is parallelism, we focus on the parallelism constructs used in SYCL-Bench benchmarks. In particular, we examine the diversity of parallelism expressed in SYCL kernels from the perspective of how they are queued and interact in the SYCL-Bench(mark) suite. We refer to this as the “parallel construct”. SYCL v1.2.1 [2] (3.6.1-3.6.5) offers five different parallel constructs, namely,

1) Basic data parallel kernels (BDP),
2) Work-group data parallel kernels (WDP),
3) Hierarchical data parallel kernels (HDP),
4) Single Task kernels (Tasks), and
5) Synchronization (sync).

We examine which of these SYCL parallel constructs are used in each application and use this in Section V to decompose application performance into pertinent parallel constructs. Given the complexity of C++ code transformations across SYCL implementations, we simply counted the number of times a construct was used in each application’s source code (independent of the implementation).

Here, we use the notation app (n) to indicate that a keyword appears n times in the source code of application app. For instance, “*nbody* (1)” means the *nbody* source code contains the feature of interest—`parallel_for`—once. Table II lists the SYCL-Bench benchmarks sorted by type of parallel construct with a count of the number of times each construct occurs in the given application.

TABLE II: Parallel constructs of SYCL-Bench benchmarks.

Application	BDP	WDP	HDP	Task	Sync
2DConvolution	1				
2mm	2				
3DConvolution	1				
3mm	3				
DRAM	1				
arith	1				
atax	2				
bicg	2				
blocked_transform	1				
correlation	5				
covariance	3				
dag_task_throughput_independent	1	1	2	1	
dag_task_throughput_sequential	1	1	2	1	
fdtd2d	3				
gemm	1				
gesummv	1				
gramschmidt	3				
host_device_bandwidth	2			1	
kmeans	1				
lin_reg_coeff		2			2
lin_reg_error	1				
local_mem		1			2
matmulchain	1				
median	1				
mol_dyn	1				
mvt	2				
nbody		1	5		2
pattern_L2	1				
reduction		2	4		1
scalar_prod		2	5		2
segmentedreduction		2	4		1
sf	1				
sobel	1				
sobel5	1				
sobel7	1				
syr2k	1				
syrk	1				
vec_add	1				

An empty entry shows that the application does not use the construct. ‘HDP’ is the sum of both `parallel_for_work_item` and `parallel_for_work_group` function occurrences within each application.

The following sections briefly describe these parallel constructs. Figure 2 illustrates how matrix multiply can be implemented with several of these constructs.

A. BDP – Basic Data-Parallel Kernels

Kernels are executed as multiple work-items and are enqueued with `parallel_for` where a range argument is provided to specify the global size of work to be done. The partitioning into work-groups is determined by the SYCL implementation/runtime. No synchronization/barrier events are supported when using this construct.

In the SYCL-Bench suite, we identify 28 benchmarks that use only basic data-parallel kernels and do not use other optimized/advanced parallel constructs outlined in the remainder of this section.

B. WDP – Work-Group Data-Parallel Kernels

Kernels are executed in user-defined dimensions: the global work-times are divided and executed in pre-defined groups. Again, the `parallel_for` function is used, with a global range argument to indicate the global size of work-items to be executed but a second argument to specify the range of each work-group size is also required. Synchronization is allowed.

Benchmarks that have been written to use work-groups use a combination of `get_local_id`, `get_local_size` and `get_local_linear_id` functions called from within the kernel for local indexing. When searching for explicit use of these functions, we see that six of the 38 benchmarks use this

TABLE III: Hardware used in the evaluation.

Alias	Name	Type	Vendor	Core Count*	Compute Units†	Memory	Processor Clock‡	Memory Clock§	TFLOPS‖	Release Date
Gold	Xeon Gold 6134	CPU	Intel	32	32	25MB (L3)	3.2–3.7GHz	10.4	3.2–3.8	2017
P100	Tesla P100	GPU	Nvidia	3584	56	12GB	1.1–1.3GHz	1.4	8.1—9.3	2016
gfx906	Radeon VII / Vega 20 (66af)	GPU	AMD	3840	60	16GB	1.4–1.7GHz	2	11.1–13.8	2019

* number of hyper-threaded cores on the Gold, CUDA cores on the Nvidia P100, and Unified Shaders on the AMD gfx906
† as reported by OpenCL device
‡ base clock frequency–maximum boost frequency
§ gigatransfers per second (GT/s)
|| single-precision floating point operations

construct: *nbody* (1), *local_mem* (1), *lin_reg_coeff* (1), *scalar_prod* (2), *reduction* (2), *segmentedreduction* (2).

C. HDP – Hierarchical Data-Parallel Kernels

SYCL provides compiler support for expressing hierarchical data parallelism that maps to the same basic execution model as work-group data-parallel kernels. A range is provided to the following enqueuing functions to specify the number of work-groups to launch and an optional size of each work-group:

`parallel_for_work_item` : Use of this function in the suite indicates there has been an attempt made to optimize the application to use private memory. This corresponds to the lowest level cache / smallest-fastest memory on the accelerator. Six benchmarks use `parallel_for_work_item`, namely, *dag_task_throughput_independent* (1), *dag_task_throughput_sequential* (1), *nbody* (4), *scalar_prod* (4), *segmentedreduction* (3) and *reduction* (3).

`parallel_for_work_group` : Presents a degree of optimization around the use of local memory, because all variables declared in this scope are allocated in work-group local memory. The same benchmarks that use `parallel_for_work_item` also use `parallel_for_work_group`. The number of times they are used differ; *dag_task_throughput_independent* (1), *dag_task_throughput_sequential* (1), *n-body* (1), *scalar_prod* (2), *segmentedreduction* (1) and *reduction* (1).

D. Task – Single-Task Kernels

A kernel is executed once, on a single compute-unit, in one work-group, as one work-item; these kernels can be executed on multiple devices and queues and encompass task-based parallelism. It is used with the `single_task` function. In the suite, three benchmarks use this construct; *dag_task_throughput_sequential* (1), *dag_task_throughput_independent* (1) and *host_device_bandwidth* (1). However, *host_device_bandwidth* submits a no-op single_task to force a read-only buffer to be copied in the microbenchmark, since this kernel does no work, it is omitted from the evaluation.

E. Sync – Synchronization

In general, operations between the host and the device require synchronization; these include buffer destruction, host accessors, command group enqueue, and queue operations. We focus on user-controlled synchronization events: those that occur within kernel execution, either globally or locally within a work-group. The `barrier` function is used inside kernels to synchronize between work-items in a work-group. It appears in six of the 38 kernels: *reduction* (1), *segmentedreduction* (1), *lin_reg_coeff* (2), *scalar_prod* (2), *nbody* (2), *local_mem* (2). *lin_reg_coeff*, *scalar_prod* and *local_mem* use a `local_space` fence to synchronize within a work-group, whereas *reduction*, *segmented_reduction* and *nbody* use the default global barrier. The *nbody* kernel contains two barriers in the same invocation, as does *local_mem*. *reduction* contains one barrier in the innermost loop of the WDP implementation.

V. Evaluation Results

Having determined the pairing of benchmarks supported by current SYCL implementations (§III) and the available parallelism expressed in each of the SYCL-Bench benchmarks (§IV), we now present a comparison of execution times of SYCL-Bench benchmarks on a range of accelerator devices. The set of accelerators used in this study is presented in Table III. Note, the architectures span four years and thus encourage readers not to read too much into relative performance across platforms. The compiler used was Clang (LLVM-9.0.1) and the Docker image is based on Ubuntu-18.04. Runtime backends used include CUDA v10.1, ROCm v3.7, Intel's OpenCL (l_opencl_p_18.1.0.015) CPU driver and OpenMP v5.0.1.

The results are presented from three different views:

1) *implementations*: presents a comprehensive list of SYCL implementations with backends, and evaluates the level of support for SYCL constructs and features using SYCL-Bench benchmarks;
2) *application-level parallelism*: examines the SYCL execution context including key features of kernel execution and methods for expressing parallelism. We scanned the source code of each SYCL-Bench application for SYCL abstractions that are recognizable to users and to the compiler/runtime implementation; and
3) *performance*: shows execution times of SYCL-Bench benchmarks to serve as a basis for comparison across accelerators.

A. Example - Matrix Multiplication

We illustrate our methodology on a well-known *matmul* benchmark, which multiplies two 1024^2 matrices. We started with the SYCL *matmulchain* benchmark and modified it by removing the chaining of matrices to form the initial basic data-parallel (BDP) version and two additional versions using work-group data-parallelism (WDP) and hierarchical data-parallelism (HDP); task-based parallelism was not assessed. We also created a serial C++ version to serve as a baseline. Figure 2 gives source code for the four different versions of

```
// serial
template <typename T>
void multiply(std::vector<T>& a, std::vector<T>& b, std::vector<T>& c, const size_t
    ↪ mat_size) {
  for(size_t i = 0; i < mat_size; ++i){
    for(size_t j = 0; j < mat_size; ++j){
      auto sum = 0;
      for(size_t k = 0; k < mat_size; ++k) {
        const auto a_ik = a[i * mat_size + k];
        const auto b_kj = b[k * mat_size + j];
        sum += a_ik * b_kj;
      }
      c[i * mat_size + j] = sum;
    }
  }
}

// basic data parallelism
template <typename T>
void multiply(cl::sycl::queue& queue, cl::sycl::buffer<T, 2>& mat_a, cl::sycl::
    ↪ buffer<T, 2>& mat_b, cl::sycl::buffer<T, 2>& mat_c, const size_t mat_size
    ↪ ) {
  queue.submit([&](cl::sycl::handler& cgh) {
    auto a = mat_a.template get_access<cl::sycl::access::mode::read>(cgh);
    auto b = mat_b.template get_access<cl::sycl::access::mode::read>(cgh);
    auto c = mat_c.template get_access<cl::sycl::access::mode::discard_write>(cgh);

    cgh.parallel_for<class MatmulBDP<T>>(
      cl::sycl::range<2>(mat_size, mat_size),
        [=](cl::sycl::item<2> item) {
        auto sum = 0;
        for(size_t k = 0; k < mat_size; ++k) {
          const auto a_ik = a[{item[0], k}];
          const auto b_kj = b[{k, item[1]}];
          sum += a_ik * b_kj;
        }
        c[item] = sum;
    });
  });
}

// work-group data parallelism
template <typename T>
void multiply(cl::sycl::queue& queue, cl::sycl::buffer<T, 2>& mat_a, cl::sycl::
    ↪ buffer<T, 2>& mat_b, cl::sycl::buffer<T, 2>& mat_c, const size_t mat_size
    ↪ , const size_t local_size) {
  queue.submit([&](cl::sycl::handler& cgh) {
    auto a = mat_a.template get_access<cl::sycl::access::mode::read>(cgh);
    auto b = mat_b.template get_access<cl::sycl::access::mode::read>(cgh);
    auto c = mat_c.template get_access<cl::sycl::access::mode::discard_write>(cgh);

    cgh.parallel_for<class MatmulWDP<T>>(cl::sycl::nd_range<2>(
      cl::sycl::range<2>(mat_size, mat_size),
      cl::sycl::range<2>(local_size, local_size)),
      [=](cl::sycl::nd_item<2> item) {
      auto sum = 0;
      for(size_t k = 0; k < mat_size; ++k) {
        const auto a_ik = a[{item.get_global_id(0), k}];
        const auto b_kj = b[{k, item.get_global_id(1)}];
        sum += a_ik * b_kj;
      }
      c[{item.get_global_id(0), item.get_global_id(1)}] = sum;
    });
  });
}

// hierarchical data parallelism
template <typename T>
void multiply(cl::sycl::queue& queue, cl::sycl::buffer<T, 2>& mat_a, cl::sycl::
    ↪ buffer<T, 2>& mat_b, cl::sycl::buffer<T, 2>& mat_c, const size_t mat_size
    ↪ , const size_t local_size) {
  queue.submit([&](cl::sycl::handler& cgh) {
    auto a = mat_a.template get_access<cl::sycl::access::mode::read>(cgh);
    auto b = mat_b.template get_access<cl::sycl::access::mode::read>(cgh);
    auto c = mat_c.template get_access<cl::sycl::access::mode::discard_write>(cgh);

    size_t num_workgroups = (mat_size + local_size - 1) / local_size;
    cgh.parallel_for_work_group<class MatmulHDP<T>>(
      cl::sycl::range<2>(num_workgroups, num_workgroups),
      cl::sycl::range<2>(local_size, local_size),
      [=](cl::sycl::group<2> group) {
        group.parallel_for_work_item([&](cl::sycl::h_item<2> item) {
          auto sum = 0;
          for(size_t k = 0; k < mat_size; ++k) {
            const auto a_ik = a[{item.get_global_id(0), k}];
            const auto b_kj = b[{k, item.get_global_id(1)}];
            sum += a_ik * b_kj;
          }
          c[{item.get_global_id(0), item.get_global_id(1)}] = sum;
        });
      });
  });
}
```

Fig. 2: Source code for matrix multiplication with different SYCL execution constructs

matmul. The serial code is a simple triply-nested loop over the indices [i,j,k] of the input matrices. In the basic data-parallel version, each work item computes a single element [i,j] of the output matrix by iterating over an entire row of matrix A and column of matrix B. The work-group data-parallel version is similar, with the addition of a specified local work-group size as a `cl::sycl::range`. Finally, the hierarchical data-parallel version explicitly specifies the number of work-groups. Note that WDP and HDP versions do not take advantage of local memory to share elements of the input matrix between work-items in the same work-group, as the purpose of this benchmark is to compare the overhead of implementations of the different execution constructs.

This experiment takes a fixed problem size of 1024^2 floating point (32-bit) elements and adjusts the local work-group size in increments from 2^0 to 2^{10}. All SYCL implementations were evaluated using the *matmul* benchmark to compare how the three major parallel constructs affect performance on the same Gold CPU, Nvidia P100, and AMD gfx906 GPU. We compared the implementations in terms of raw performance, their support for these different SYCL constructs, and to assess whether doing optimization in SYCL transforms into a performance improvement. In particular, this test was devised to assess core utilization on the system based on the way parallelism is both expressed and supported over these SYCL implementations.

Each test was executed 100 times to give a large statistical sample size. Execution times and cache-misses were collected via `std::chrono::duration` and `perf` respectively to help explain the results. We also used `top` to validate CPU core usage. The results presented in Figure 3 are separated according to the SYCL runtime—the pairing of SYCL implementation and backend—for instance, hipSYCL-OpenMP and hipSYCL-CUDA are separate SYCL runtimes. Each plot presents the time to perform matrix-multiplication on two 1024^2 matrices over an increasing x-axis corresponding to local work-group sizes; this offers a comparison among the 3 different SYCL parallel constructs and demonstrates how absolute performance varies. Only the HDP and WDP modes support explicitly setting the work-group size and so they are the only results with multiple data-points per parallel construct. As the work-group size for BDP kernels is determined by the implementation, not the developer, it is shown as 2^0th in all plots. Also, since the serial implementation conceptually has one large work-group, both Serial and BDP are placed side-by-side at work-group size 1, as points of comparison against WDP and HDP versions. Some data-points are missing due to those SYCL runtimes not supporting larger work-group sizes.

The median Serial execution time is included as a turquoise dashed line at 4.6 seconds and was computed over all SYCL runtimes since the same Xeon Gold 6134 CPU was used on all systems. The only variation from this median runtime were the DPC++ SYCL implementations. Specifically, Serial *matmul* versions on both the DPC++ pthreads and DPC++ CUDA SYCL runtimes are ≈0.6 seconds faster than other SYCL versions despite being executed on the same Gold

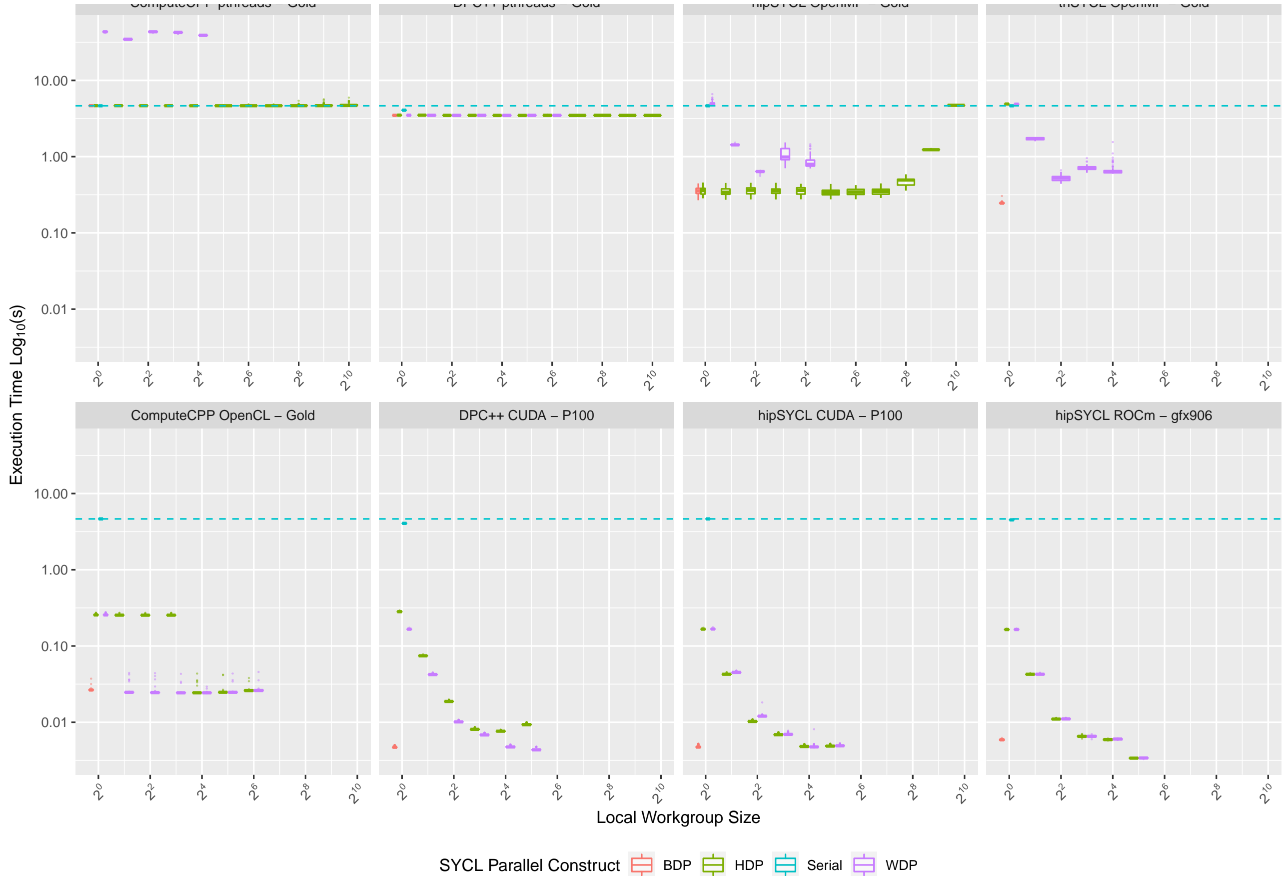


Fig. 3: Execution times of the SYCL runtimes to perform multiplication on two 1024^2 matrices presented in a log-scale; highlighting the difference in performance of parallel constructs against a Serial baseline.

CPU, because the timing loop queries the SYCL framework to synchronize with a call to `wait_and_throw()` and this function performs faster on DPC++ than the other three SYCL implementations when querying an empty execution queue.

While both the ComputeCpp and DPC++ implementations offer SYCL support on the host device, they only offer single-core execution, and thus all execution times are unchanged by increasing the work-group sizes. In general, using the host device on the ComputeCpp SYCL implementation offers no better performance than running on a single core. It is interesting that the WDP construct on the ComputeCpp-pthreads runtime—which a programmer might expect to provide opportunity for performance optimization—performs equal or worse than the other constructs for all work-group sizes. This is explained by the high cache-miss percentage inherent to the WDP construct where work-groups are scheduled for execution in a random/strided order, rather than in sequence which would lead to better cache reuse.

The DPC++ implementation, while only offloading to one-core/p-thread to perform the kernel execution on the host platform, uses an additional host thread to monitor completion. There is little difference between these parallel constructs in either execution time or cache miss percentage.

Both hipSYCL and triSYCL allow parallel execution on the host device via OpenMP backends. Almost all parallel constructs on hipSYCL—excluding the WDP at 1 Local Work-Group Size—perform better than the baseline Serial code. The WDP code performs significantly worse than the HDP variant, which is the only example of a local work-group size offering better performance than BDP—this is only slight with the HDP at a local work-group size of 32 being $\approx$18 ms faster. Some GPU runtimes may show a marginal improvement at the largest work-group size, however this trade-off comes at larger sizes crashing entirely due to hitting the hardware limit support for those sizes.

The potential benefit of HDP must be weighed against the risk of incorrectly setting the size, which can affect the performance by an order of magnitude—corresponding to insufficient parallelism to use all cores. Regarding scaling, the Intel Xeon Gold 6134 Skylake processor sports 32 hyper-threaded/16 physical cores, and when we compare against the baseline Serial code, both BDP and HDP show roughly a

13-14x speedup, while WDP has at best a 7x speedup (for work-group size of 4). Work-Group Parallelism in SYCL is the worst on all implementations. In general, BDP offers the best performance over these four implementations, which is perhaps surprising as it is the simplest and most abstract SYCL version of this kernel.

The ComputeCpp OpenCL backend performed best on the Gold, an order of magnitude faster than the OpenMP versions for all parallel constructs—excluding some smaller HDP work-group sizes, which were comparable. The execution time of HDP is similar to that on hipSYCL OpenMP for work-group sizes 1,2,4, and 8 taking around 250ms, improving to ≈25ms for larger work-group sizes. WDP follows a similar trend starting off at ≈250ms for a work-group size of 1, and all other data-points flat-line at ≈25 ms over the work-group sizes 2-64. BDP experiences similar execution times (≈26ms) to the larger—and best configured—work-group sizes of WDP and HDP constructs.

All GPU runtimes offer shorter execution times than the Gold CPU, and the parallel constructs exhibit similar trends, with execution times for both WDP and HDP decreasing as work-group size increases. Comparing the two CUDA backends, DPC++ shows better performance for WDP than for HDP with work-groups of the same size; whereas hipSYCL shows similar performance between HDP and WDP. BDP performs similarly on both DPC++-CUDA and hipSYCL-CUDA runtimes on the P100 and seems to be equivalent to the best work-group size setting of any other parallel construct. The ROCm backend on the AMD gfx906 has similarly good performance in using hipSYCL to CUDA, the same trend is shown between parallel constructs. Performance is slightly worse at a work-group size of 16 when compared to the P100, however it offers the shortest execution time for this matrix multiplication test at a 32 sized work-group. BDP performance on the gfx906 lies between these two best-sized work-groups of the HDP and WDP constructs.

We propose that the good performance of BDP comes from the partitioning of tasks between cores by the underlying framework/backend, whereas both WDP and HDP force a particular work partitioning that may be non-optimal for the hardware. To test this, we used `perf` to record the cache misses generated by each of the SYCL parallel constructs for the different implementations on the Gold CPU architecture, as shown in Figure 4. We see cache misses increasing with work-group size on all the CPU backends, up to 10% with WDP on the ComputeCPP-pthreads runtime. The serial implementation has the lowest miss-rate of all the versions and shows the baseline. The added management of threads on the multi-threaded backends—OpenMP and OpenCL—comes at the cost of poorer cache utilization.

Figure 5 highlights how these memory accesses are mapped when specified by the developer using SYCL parallel constructs. This shows a 9^2 matrix, in which each element is mapped to a work-item and each colour corresponds to a different work-group. Figure 5a) shows how work-groups set in 2-dimensions make algorithmic sense, as they are explicitly assigned sizes with WDP and HDP constructs, however, when we consider caching on CPU architectures it may be suboptimal.

As a contrast, Figure 5b) shows how the BDP may queue work-items consecutively in one dimension, and simply partition the 81 elements by the number of cores available on the hardware—configurable by the underlying device/language backend. The same number of work-groups is used, but the 1D pattern makes better use of cache due to its sequential memory access pattern. As this toy example is scaled up in size, we can see this access pattern is much more likely to ensure the same cache line is used. Thus, specifying work-group blocking can hinder accesses patterns; rather than the developer attempting to make this optimization, we see the underlying runtimes give good performance when left to perform work partitioning on their devices. We see this in the performance results with BDP kernels having the best execution times and lower cache misses of the parallel constructs on all SYCL implementations.

In summary, for the matrix multiplication example, BDP is the best choice of parallel construct, performing best on most implementations while retaining a high level of abstraction. SYCL WDP parallelism is expressed differently to HDP; in WDP local-size indicates the number of heavyweight threads to use, or the amount of parallelism to employ, whereas HDP expresses the work-group size to determine the global amount of work to do, inherent to the algorithm. Both constructs may be used as a mechanism to add an algorithmic restriction on parallelism, for instance, in pressure-system weather modelling where the same task is to occur over different resolution grids. We believe they should not be used for device specific optimization for two reasons:

1) It goes against the general purpose of SYCL as providing hardware-agnostic language abstractions; targeting the expression of parallelism to a particular device hinders performance portability.
2) The naive BDP generally achieves better performance since it ties to the strengths of the SYCL backends, which have a longer legacy of achieving good performance on the selected platform and device; this is usually heavily optimized on account of it being tied to a particular vendor, i.e. CUDA solely targets Nvidia GPUs and has been built upon for over a decade, ROCm for AMD devices, TBB for Intel CPUs etc.

Where possible, memory access patterns should be determined by the SYCL backend rather than by the developer since this may affect generality and portability of the code. In other words, the intent of the algorithm should be expressed independent of the underlying architecture; there should be tools put in place to facilitate this. A take-away message from the deep-dive is to use the more advanced parallel constructs only where it clarifies the programmer's intent based on the requirements/expression of the algorithm, not as an attempt at optimization.

We also see that selection of problem size can significantly impact performance according to backend. However, while work-group size selection is often the first step for optimiza-

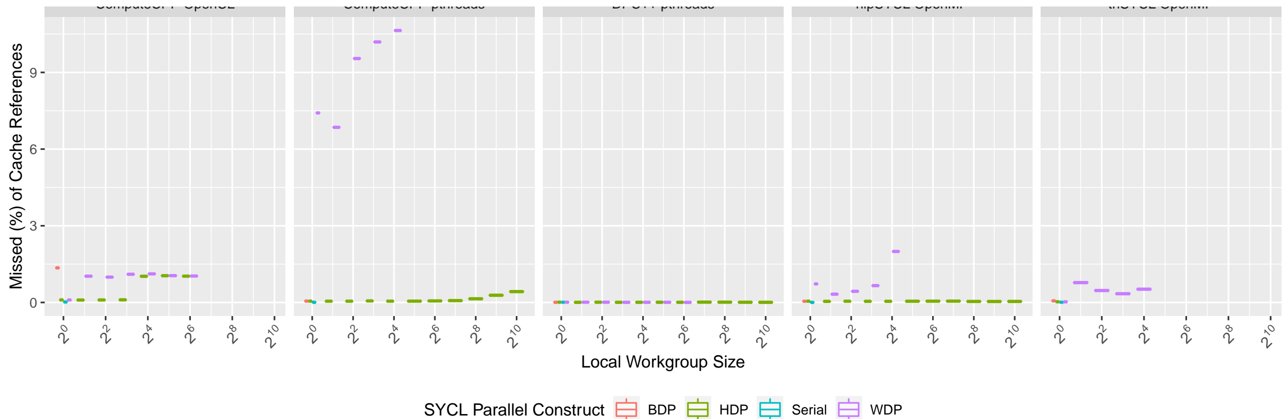
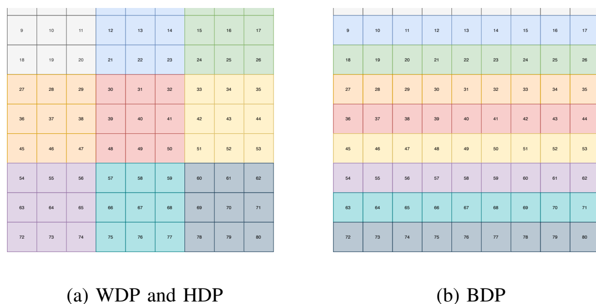


Fig. 4: Percentage of cache-misses occurring during 100 runs of the matrix-multiplication over the SYCL runtimes which target the Gold CPU, showing the penalties of using different parallel constructs.



(a) WDP and HDP

(b) BDP

Fig. 5: Work-group partitioning and the corresponding memory access patterns when using different SYCL parallel constructs to perform matrix multiplication.

tion with most accelerator languages, this may not be a good strategy with SYCL due to the additional layer of language abstraction.

B. SYCL-Bench

The *BlockedTransform* benchmark, where concurrent mandelbrot kernel execution overlaps compute with data transfers, is shown in Figure 6a) and highlights the scaling of dedicated accelerator performance rather than all computation occurring on the host. The number of iterations was selected to be 512 and the blocksize increases over the x-axis. It appears that GPU devices perform better over larger blocksizes whereas CPU devices are more affected by increasing the blocksize. Single threaded CPU backends on achieve no benefit, multi-threading on the Gold—with OpenMP and OpenCL—can handle and scale with increasing blocksizes however are still an order of magnitude worse than CUDA and ROCm GPU backends which continue to scale with the greater load.

Figure 6b) presents the counter-point to using SYCL for dedicated accelerators by highlighting the overhead of memory movements over PCI-e to these devices. It shows the Gold's memory bus is generally four orders of magnitude faster than PCI-e under different benchmark configurations.

We now examine a selection of SYCL-Bench benchmarks grouped by parallel construct. Because there are 38 unique benchmarks in the suite—and many support multiple data-primitives and sizes—there are far too many results to show in this paper. Accordingly, for each of the parallel construct benchmarks, we present results only for 32-bit float data; the results for other data-types are qualitatively similar across the different implementations, with some minor exceptions. The performance of BDP benchmarks that support 32-bit floats is shown in both Figure 7a) and WDP in Figure 7b) highlights the performance of SYCL runtimes and devices using the same data-type but between different constructs. The broad trend is that OpenMP, OpenCL, CUDA and ROCm backends perform well on both BDP and WDP. All figures—including those with other data-types—are presented in the associated Jupyter artifact.

Figure 7c) presents execution time for all SYCL-Bench HDP benchmarks across all supported data types. One unexpected result is that the *SegmentedReduction_Hierarchical* benchmark runs faster for 64-bit floats than for 32-bit floats on both hipSYCL-OpenMP and hipSYCL-ROCm, whereas for all other implementations 32-bit floats are faster (as would be expected).

Both task parallel benchmarks are presented in Figure 7d) to identify which devices and SYCL runtimes experience less variability with task parallelism. OpenCL, CUDA and OpenMP backends perform with equal variation in execution time to run both benchmarks. The exception being triSYCL which boasts both the best and less variable of the OpenMP performance, with only pthreads on DPC++ beating it on the *dag_task_throughput_sequential* test, and being the best performer on the *independent* test.

The results of synchronization kernels are presented in

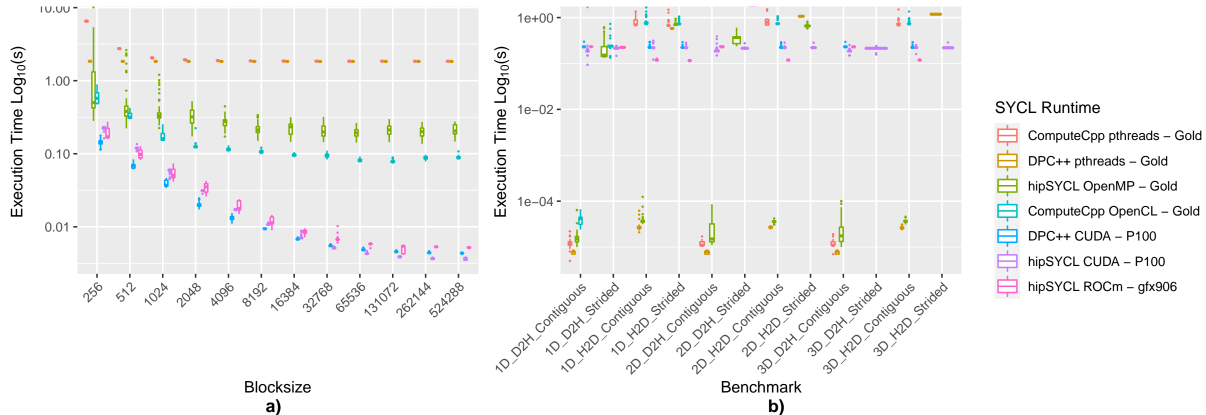


Fig. 6: Microbenchmarks **a)** *BlockedTransform* and **b)** *Bandwidth* to compare the performance of the SYCL Runtimes.

Figure 7e) to show the effect of barriers on runtimes. Both GPU devices—on all CUDA and ROCm backends—perform best for all benchmarks. ComputeCpp's OpenCL responds the best to barriers on the Gold CPU, being roughly two orders of magnitude faster than OpenMP and pthreads backends, and could serve as a baseline for the developers of these backends when looking to improve their performance. Finally, there is very little difference in execution time when considering the data-types used in each of the benchmarks.

VI. Conclusions and Future Work

SYCL is a promising language for portable HPC. It offers a single-source implementation of algorithms and increases the portability of programs by mapping back to most existing HPC languages. Ultimately, it now supports most vendors hardware and thus allows freedom from single vendor languages and implementations. Unfortunately, our results show that while many of the SYCL implementations compile and run on many different backends, the selection of optimal runtime and device is essential to guarantee good performance; this is highlighted by our results where the same code targeted to the same device experiences up to two orders of magnitude of absolute performance difference between the best and worst SYCL runtimes.

While performance-portability is a well-known problem for high-level programming models that target heterogeneous hardware through multiple backends—for example, Kokkos [7] and RAJA [4]—the problem is made more complicated under SYCL as there may be multiple implementations and backends targeting the same device. We believe many of these problems can be addressed by ensuring consistency in functionality between implementations and the addition of an intelligent run-time scheduler within SYCL, where tasks span implementations. Moving complexity from the developer to an automated system, primarily concerned with scheduling for portability is needed for SYCL to meet its full potential, and is an interesting topic to pursue in future work.

The deep-dive into Matrix Multiplication highlights BDP to be a useful parallel construct. We show that carefully selecting larger work-group sizes (HDP and WDP constructs) can still benefit GPU devices; however, this is less portable. The BDP construct is both simpler to use—division of work need not be considered—and the most performance-portable. A take-away message from the deep-dive is to use the more advanced parallel constructs only where it clarifies the programmer's intent based on the requirements/expression of the algorithm.

We also present an evaluation of the state-of-the-art in SYCL support by compiling the complete SYCL-Bench benchmark suite, and offer a breakdown of these codes into the corresponding SYCL parallel construct. We show that characterizing SYCL codes by parallel construct is a useful way to analyse expected performance. In the future, we would like to examine the OpenCL backend for DPC++, which was the one missing backend in our work.

VII. Acknowledgments

This research was supported in part by the following sources: Defense Advanced Research Projects Agency (DARPA) Microsystems Technology Office (MTO) Domain-Specific System-on-Chip Program and the US Department of Energy (DOE) Advanced Scientific Computing Research (ASCR) program. This manuscript has been co-authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan. The authors thank the ORNL Experimental Computing Laboratory (ExCL) team for its support with the compute resources.

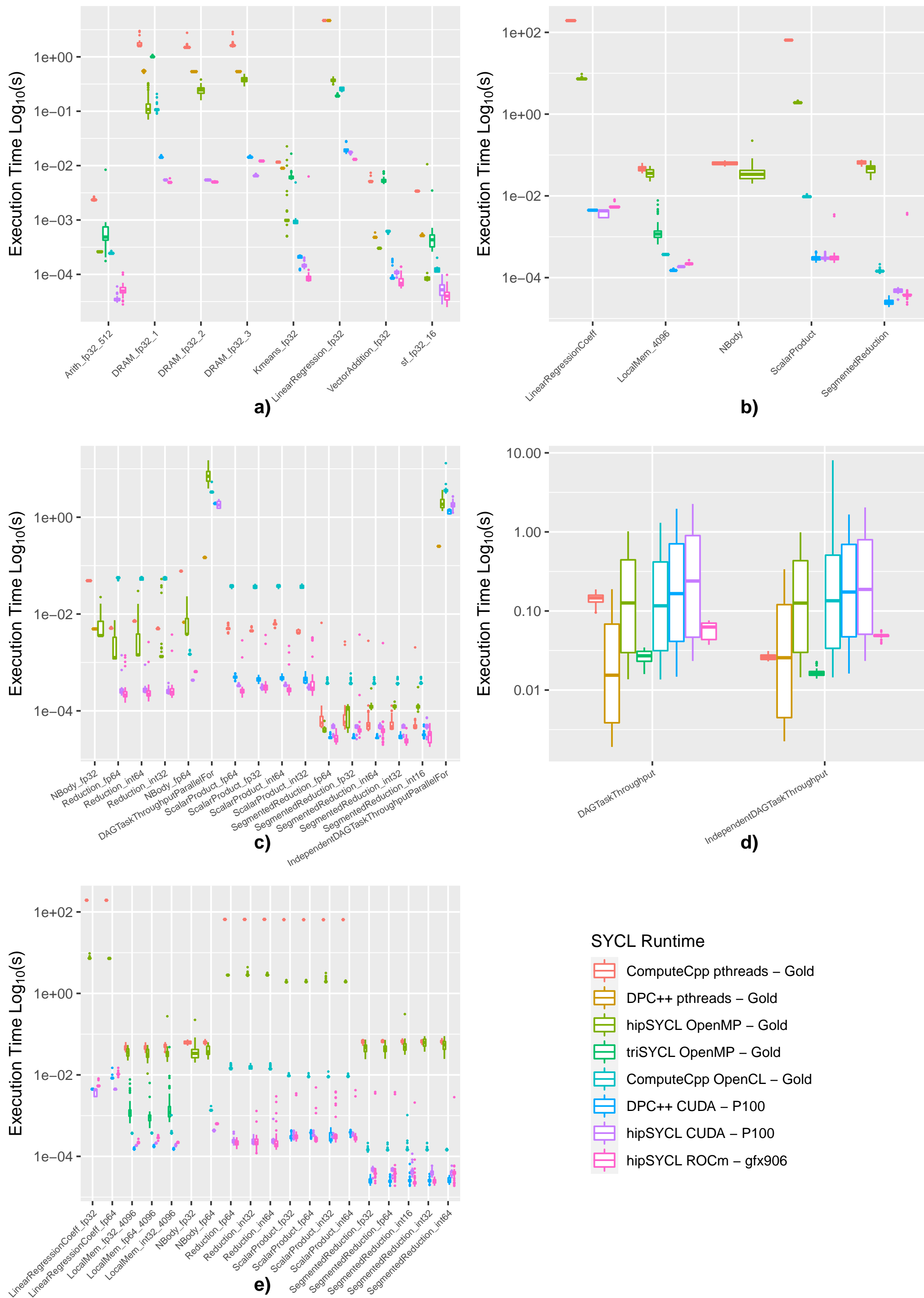


Fig. 7: Performance of a selection of SYCL-Bench benchmarks separated by parallel construct: **a)** Basic Data Parallelism (BDP), **b)** Work-Group Data Parallelism (WDP), **c)** Hierarchical Data Parallelism (HDP), **d)** Single-Task Parallelism (Task), and **e)** Synchronization (Sync).

REFERENCES

[1] "The OpenCL 1.0 specification," Khronos Group, Tech. Rep., 2008.
[2] "SYCL specification 1.2.1 revision 7," Khronos Group, Tech. Rep., 2020. [Online]. Available: https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf
[3] A. Alpay and V. Heuveline, "SYCL beyond OpenCL: The architecture, current state and future direction of hipSYCL," in *Proceedings of the International Workshop on OpenCL (IWOCL)*, 2020.
[4] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland, "RAJA: Portable performance for large-scale scientific applications," in *International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 2019.
[5] Codeplay Software Ltd, "ComputeCpp." 2020. [Online]. Available: https://www.codeplay.com/products/computecpp
[6] A. Dubey, P. H. Kelly, B. Mohr, and J. S. Vetter, "Performance portability in extreme scale computing (Dagstuhl seminar 17431)," in *Dagstuhl Reports*, vol. 7. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.
[7] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.
[8] Intel Corporation, "Intel Data Parallel C++ Compiler," 2020. [Online]. Available: https://github.com/intel/llvm
[9] R. Keryell and L.-Y. Yu, "Early experiments using SYCL single-source modern C++ on Xilinx FPGA," in *Proceedings of the International Workshop on OpenCL (IWOCL)*, 2018. [Online]. Available: https://doi.org/10.1145/3204919.3204937
[10] S. Lal, A. Alpay, P. Salzmann, B. Cosenza, A. Hirsch, N. Stawinoga, P. Thoman, T. Fahringer, and V. Heuveline, "SYCL-Bench: A Versatile Cross-Platform Benchmark Suite for Heterogeneous Computing," in *International European Conference on Parallel and Distributed Computing (Euro-Par)*, 2020.
[11] J. Nickolls and I. Buck, "NVIDIA CUDA software and GPU parallel computing architecture," in *Microprocessor Forum*, 2007.
[12] S. J. Pennycook, J. D. Sewall, and V. W. Lee, "Implications of a metric for performance portability," *Future Generation Computer Systems*, vol. 92, pp. 947–958, 2019.
[13] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference (Vol. 1): Volume 1-The MPI Core*. MIT press, 1998, vol. 1.

APPENDIX A
ARTIFACT DESCRIPTION APPENDIX: EVALUATING THE PERFORMANCE AND PORTABILITY OF CONTEMPORARY SYCL IMPLEMENTATIONS

A. Abstract

We provide the source-code to perform matrix multiplication in the three different parallel constructs plus a baseline C++ implementation. We also offer a Docker binary and Dockerfile with pre-built SYCL implementations, for others to perform a quick evaluation between them. Finally, we share the Jupyter artefact for transparency, interpretive and reproducible results presented in the paper. This contains scripts and snippets of outputs to show both how the data were collected and how they are plotted.

B. Description

1) Check-list (artifact meta information):

- **Algorithm:** Matrix Multiplication.
- **Program:** SYCL-Bench(mark) suite.
- **Compilation:** LLVM-9.0.1 with Clang, GCC 7.5.0, CUDA 10.1 and ROCm 3.8.0.
- **Binary:** -
- **Data set:** -
- **Run-time environment:** Docker, Ubuntu 18.04.
- **Hardware:** Various x86 CPUs, Nvidia and AMD GPUs.
- **Output:** Execution times of SYCL-Bench benchmarks and Matrix Multiplication; plots and analysis in the Jupyter artefact.
- **Experiment workflow:** See below.
- **Publicly available?:** Yes.

2) How software can be obtained (if available): The entire source tree is available from GitHub https://github.com/BeauJoh/sycl-bench. The repository will be maintained—the version in the archive has been tagged as "sc20"—and can be used to submit issues via the issue tracking system.

3) Hardware dependencies: The three systems used in our paper were an Intel Xeon Gold 6134 CPU, an Nvidia Tesla P100 GPU and an AMD Radeon VII/Vega 20 GPU. It is assumed the SYCL runtimes built as part of this evaluation will be forward compatible to newer architectures from these vendors and you should lodge an issue if this is not the case.

4) Software dependencies: This artefact optionally uses binder—automatic cloud hosting of Jupyter workbooks with support for docker. Thus, for initial questions the following link may be useful https://mybinder.org/v2/gh/BeauJoh/sycl-bench/sc20?filepath=sycl-performance.ipynb. This project uses Docker to facilitate reproducibility. As such, it has a dependency on Docker, and optionally if an Nvidia device is going to be used a CUDA 10.1 Runtime with nvidia-docker2. Additionally, if an AMD GPU device is intended a ROCm-enabled Linux kernel and the ROCk driver is needed.

5) Datasets: This work is based on version 7f38669 of SYCL-Bench. We have made some changes to the build system, supporting two additional SYCL implementations (hipSYCL and triSYCL). We also offer the source-code used in our deep-dive of the matrix multiplication example, implemented in 3 different SYCL parallel execution constructs and a serial baseline, basic data-parallelism (BDP), work-group data-parallelism (WDP), and hierarchical data-parallelism (HDP), see `matmul_*.cpp` in the single-kernel directory.

The dynamic Jupyter notebook, found in sycl-performance.ipynb, shows how sycl-bench was run and the results plotted. A static webpage of the analysis presented in the paper is also available https://mybinder.org/v2/gh/BeauJoh/sycl-bench/sc20?filepath=sycl-performance.ipynb.

C. Installation

All SYCL implementations have been installed in the Docker image available on Dockerhub https://hub.docker.com/r/beaujoh/syclbench. As such, the Docker session is launched with:

```
docker run -it --mount
src=`pwd`,target=/workspace,type=bind
-p 8888:8888 --net=host
--security-opt seccomp=unconfined
beaujoh/syclbench:latest bash
```

For use with an Nvidia GPU add the following flags to the run command: `--runtime=nvidia -e NVIDIA_VISIBLE_DEVICES=1`. For AMD GPU support add: `--device=/dev/kfd --device=/dev/dri --group-add video`.

D. Experiment workflow

Beakerx is already installed in the Docker image and is run with: `beakerx --allow-root`. From the `/workspace` directory—the default from the docker environment—one will be prompted with a URL to open with a token, both are to be entered into a web-browser. Open the Jupyter notebook `sycl-performance.ipynb` from the web-browser and interactive Jupyter session and one will be shown the plotted results and full experimental methodology. It may be helpful to see how different SYCL implementations are built against different backends, this is shown in the subsection on ”Generating Results”.

E. Evaluation and expected result

Also shown within the Jupyter artefact is how the `./run-suite` shell script can be used from within each SYCL implementation. For instance, `mkdir ./computecpp-benchmarks && cd ./computecpp-benchmarks && cmake ../.. -DSYCL_IMPL=ComputeCpp -DCMAKE_PREFIX_PATH=/tmp/ComputeCpp-latest && make -j16 && rm -r Makefile CMakeCache.txt CMakeFiles && cd ..` will build SYCL-Bench against the ComputeCpp SYCL implementation against the OpenCL backend. The command `./run-suite cpu` will run the full SYCL-Bench suite. At the end of the log it is expected with this version of ComputeCPP that `3DConvolution 2mm matmulchain gemm 3mm syrk 2DConvolution sobel7 syr2k DRAM` will fail to run with the OpenCL backend, in line with the results reported in Table I of the paper. To disable the OpenCL backend and default to the CPU-pthreads backend run the command `mv /etc/OpenCL/vendors/amd.icd /etc/OpenCL/vendors/amd.icdX`. Now if the same command is run `./run-suite cpu`, only the host_device_bandwidth test will return a non-zero (failure) error code. All other combinations of building the different implementations with backends is shown in the artefact alongside expected results and . Much of the notebook highlights how the data is loaded and the figures generated.