Evaluating the Performance 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, which promises improved maintainability, productivity, and opportunity for compiler optimization, when compared to programming models like OpenCL and CUDA. Several implementations of the SYCL standard have been developed over the past few years, including several backends into contemporary accelerator languages, like OpenCL, CUDA, and HIP. These implementations vary wildly 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 across a range of hardware and prominent SYCL features to understand SYCL's performance portability. This work uses the newest SYCL benchmark suite (SYCL-Bench) to evaluate all four existing implementations, comparing support of language features between backends, and highlighting those that are missing or performing poorly. We offer a detailed evaluation of the major SYCL parallel constructs in the context of a matrix multiplication benchmark. Our results show that basic kernel parallelism is the best choice for performance on current SYCL implementations, and identify opportunities for improvement in several of the target SYCL runtimes.

I. Introduction

SYCL [2] is a single-source programming model for heterogeneous systems, managed as an open standard by the Khronos Group. The benefit of SYCL, as compared to programming models like OpenCL [1] and CUDA [10], is that it offers a single-source approach 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 to most contemporary accelerator languages, like OpenCL, CUDA, and HIP. Many of these earlier approaches required kernel and host code to be separate. This SYCL standard guarantees functional equivalence of features across compliant implementations but it does not prescribe how the underlying features should be implemented (unlike standards like MPI [12] or OpenCL).

Over the past few years, several implementations of this SYCL standard have emerged: DPC++ [6], ComputeCpp [4], triSYCL [9], and hipSYCL [3]. 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.

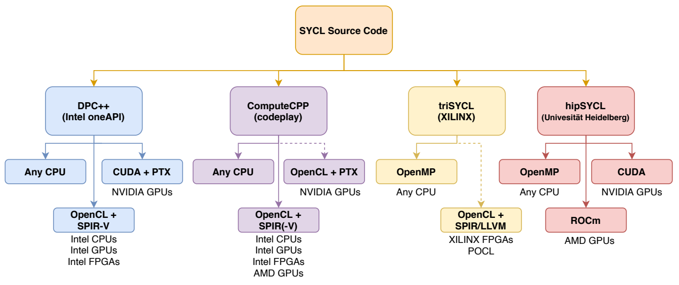


Fig. 1: Current SYCL implementations and their corresponding API backends.

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. A dashed line indicates experimental support.

Our goal in this effort is to understand the *performance portability* [5], [11] of SYCL, and, in particular, the performance portability of individual SYCL features across these implementations. This information 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 oftentimes 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 these features. 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.

We make the following contributions:

1) We use 38 benchmarks from 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 of the major SYCL parallel constructs in the context of a matrix multiplication benchmark, and show that basic kernel parallelism is the best choice for performance on current SYCL implementations.

We offer a Dockerhub image [7] and Dockerfile with Jupyter artifact [8] for interpretable results.

II. Methodology

We followed a number of steps in order to understand the performance portability of SYCL across a range of hardware and SYCL implementations.

First, we identified the main features of SYCL available to users for developing applications. At the highest level, we can use basic timing information to characterize application performance, however, this will be determined by the performance of particular *parallel constructs* defined in the SYCL standard. These parallel constructs are: basic data-parallel kernels (BPK), work-group data-parallel kernels (WGP), hierarchical data-parallel kernels (HDP), single-task kernels (task), and synchronization (sync). 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 show dramatically different performance variability. Second, we gathered and surveyed 38 SYCL kernels (see Table I) to tally which features of SYCL they were built on. Third, we surveyed, installed, and evaluated our 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

In response to the SYCL specification, several teams have developed SYCL implementations for multiple target architectures and underlying programming systems (e.g., CUDA, OpenCL, OpenMP). In addition, several implementations have added proprietary extensions to SYCL to exploit specific architectural features, which may not be efficiently exploitable via the specification. In this work, we focus strictly on the standard SYCL features of these implementations. Furthermore, we only examine non-experimental backends; resulting in a comparison between 9 different SYCL runtimes.

This section identifies which of the applications functionally 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 contemporary SYCL implementations.

Overall, we found that most of our applications built and ran successfully; however we were unable to successfully build against the DPC++ OpenCL backend, thus we only evaluate 8 out 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 [4] binary known as "ubuntu-16.04-64bit". For reproducibility, we provide a Docker image[7] with our build of SYCL-Bench with the four implementations of SYCL.

IV. SYCL Parallelism Constructs

Since one of the major features of SYCL is parallelism, we focus on the parallelism constructs used in SYCL-Bench applications. 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 [2] (3.6.1-3.6.5) offers five different parallel constructs, namely,

1) Basic data parallel kernels,
2) Work-group data parallel kernels,
3) Hierarchical data parallel kernels,
4) Single Task kernels, and
5) Synchronization.

We examine which of these SYCL parallel constructs are used in each application and later –Section V– use this 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.

A. BPK – 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 applications which use only basic data-parallel kernels and do not use the other optimized/advanced parallel constructs outlined in the remainder of this Section.

TABLE I: Status of SYCL-Bench applications 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

B. WGP – 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, and the global range argument is used for the global size of work-items to executed but a second argument specifying the range of each work-group size is also required. Synchronization is allowed.

Applications which 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 37 applications use this 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, which 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 applications 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 applications 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 applications 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 such as buffer destruction, host accessors, command group enqueue, queue operations, etc. 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 37 kernels: *reduction* (1), *segmentedreduction* (1),

TABLE II: Parallel constructs of SYCL-Bench applications.

Application	BKP	WGP	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				

lin_reg_coeff (2), *scalar_prod* (2), *nbody* (2), *local_mem* (2). *lin_reg_coeff*, *scalar_prod* and *local_mem* request a `local_space` fence –synchronization within a work-group– whereas *reduction*, *segmented_reduction* and *nbody* use the default global barrier. NDRange versions of these kernels are the ones which contain barriers – the hierarchical variations do not. The *nbody* kernel contains two barriers in the same invocation, as does *local_mem*. *reduction* contains one barrier in the innermost loop of the NDRange implementation.

F. *Overview*

Table II shows an overview of SYCL-Bench applications sorted by type of parallel constructs with corresponding counts to denote the number of times each construct occurs in the given application. An empty entry shows that the application does not use the construct. Note, the HDP value is the sum of both `parallel_for_work_item` and `parallel_for_work_group` function occurrences within each application.

V. Evaluation Results

Having determined the pairing of applications supported by current SYCL implementations (Section III) and the available parallelism expressed in each of the SYCL-bench applications (Section IV), we now present a comparison of execution times of SYCL-Bench applications on a range of accelerator devices. The set of accelerators used in this study is presented in Table III. 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 applications;
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, which presents execution times of SYCL-Bench applications to serve as a basis for comparison between accelerators.

A. *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 kernel parallelism (BKP) version and two additional versions using work-group parallelism (WGP) and hierarchical data-parallelism (HDP); task-based parallelism was not assessed. We also created a serial version to serve as a baseline. Figure 2 gives source code for the four different versions of *matmul*. The serial code is a simple triply-nested loop over the indices [i,j,k] of the input matrices. In the basic kernel 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 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 WGP 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}. Four different SYCL CPU runtimes 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 compare 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 contemporary 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

TABLE III: Hardware used in the evaluation.

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

Core Count indicates the number of hyper-threaded cores on the Gold, CUDA cores on the NVIDIA P100, and Unified Shaders on the AMD gfx906.
Compute Units offers a comparison between significantly different architectures by showing the available Compute Units available in the OpenCL setting.
Processor Clock indicates the base clock frequency and the maximum boost frequency.
Memory Clock speed is reported in terms of gigatransfers per second (GT/s).
Single-precision floating point is used for the theoretical TFLOPS.

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 WGP 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 BKP 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 BKP are placed side-by-side at work-group size 1, as points of comparison against WGP and HDP versions. Some data-points are missing due to those SYCL runtimes not supporting larger work-group sizes.

The median Serial runtime is included as a turquoise dashed line at 4.6 seconds and was computed over all SYCL runtimes since the same host-CPU –the Xeon Gold 6134– was used on all systems.The only significant variations from this median runtime were the DPC++ SYCL implementations. Specifically, Serial *matmul* versions on both the DPC++ pthreads and DPC++ CUDA SYCL runtimes are $\approx$0.6 seconds faster than other SYCL versions despite being executed on the same Gold 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 WGP 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 WGP 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 WGP @1 Local Work-Group Size– perform better than the baseline Serial code. The WGP code performs significantly worse than the HDP variant, which is the only example of a local work-group size offering better performance than BKP – this is only slight with the HDP @ 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. However, the potential benefit of using HDP must be weighed against the risk of incorrectly setting this size, which can affect the performance by an order of magnitude –this corresponds to offering an insufficient level of parallelism to occupy 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 BKP and HDP show roughly a 13-14x speedup, while WGP has at best a 7x speedup (for work-group size 4). Work-Group Parallelism in SYCL is the worst on all implementations. In general, BKP 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.

WGP achieves a speedup at a work-group size of 4 of almost 9x compared to the Serial code. However, BKP is again by far the best performer at almost 19x faster than the serial baseline, it offers good core saturation with sufficient memory/cache-line reuse and utilizes the 32 HT cores.

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 $\approx$25ms for larger work-group sizes. WGP follows a similar trend starting off at $\approx$250ms for a work-group size of 1, and all other data-points flat-line at $\approx$25 ms over the work-group sizes 2-64. BKP experiences similar execution times ($\approx$26ms) to the larger –and best configured– work-group sizes of WGP and HDP constructs.

All GPU runtimes offer shorter execution times than the Gold CPU, and the parallel constructs exhibit similar trends,

```
// 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 kernel 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 MatmulBKP<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 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 MatmulWGP<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

with execution times for both WGP and HDP decreasing as work-group size increases. Comparing the two CUDA backends, DPC++ shows better performance for WGP than for HDP with work-groups of the same size; whereas hipSYCL shows similar performance between HDP and WGP. BKP 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. BKP performance on the gfx906 lies between these two best-sized work-groups of the HDP and WGP constructs.

We propose that the good performance of BKP comes from the partitioning of tasks between cores by the underlying framework/backend, whereas both WGP 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 WGP 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 WGP and HDP constructs, however, when we consider caching on CPU architectures it may be suboptimal. As a contrast, Figure 5b) shows how the BKP 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 BKP kernels having approximately the best execution times of all the parallel constructs over all the SYCL implementations, and shows a lower cache miss.

In summary, for the matrix multiplication example, BKP is the best choice of parallel construct, performing best on most

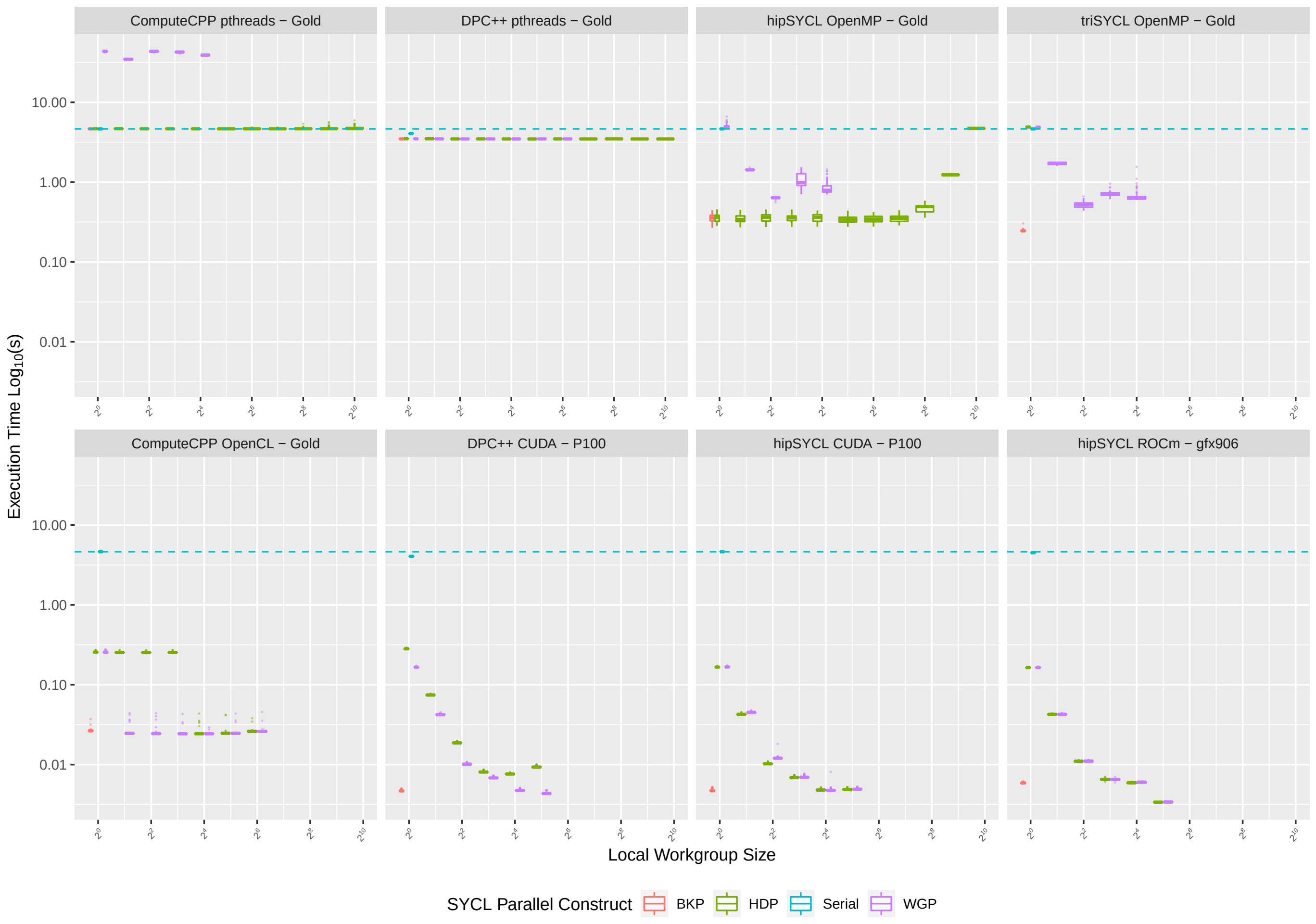


Fig. 3: Execution times of the SYCL runtimes to perform multiplication on two 1024^2 matrices presented in a log-scale; highlighting the difference in performance of parallel constructs against a Serial baseline.

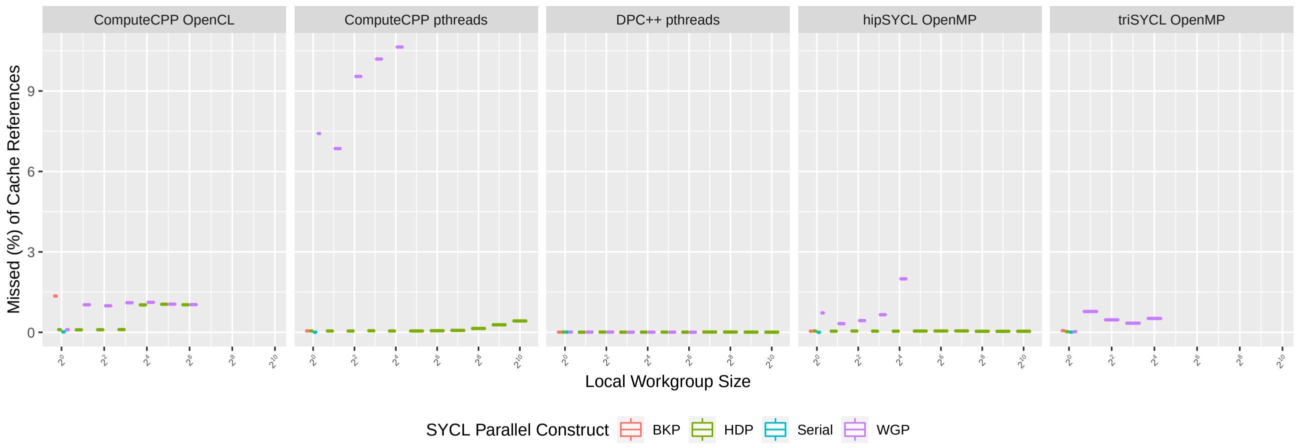
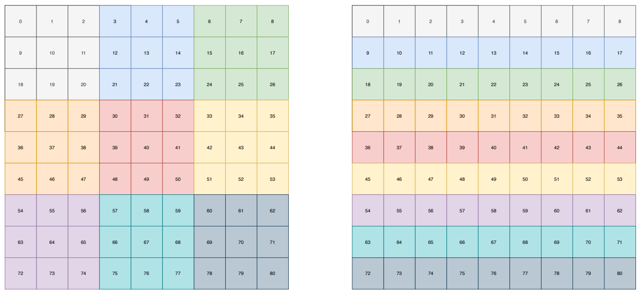


Fig. 4: Percentage of cache-misses occurring during 100 runs of the matrix-multiplication over the SYCL runtimes which target the Gold CPU – showing the penalties of using different parallel constructs.



(a) WGP and HDP (b) BKP

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

implementations while retaining a high level of abstraction. SYCL WGP parallelism is expressed differently to HDP –in WGP 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 should be used as a mechanism to add an algorithmic restriction on the parallelism to use, for instance, in pressure-system weather modelling where the same task is to occur over different resolution grids. 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 BKP generally achieves better performance since it ties to the strengths of the SYCL backends – which have a longer legacy in 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 in 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 optimization with most accelerator languages, this may not be a good strategy with SYCL due to the additional layer of language abstraction.

B. *SYCLBench*

The *BlockedTransform* benchmark highlights the scaling of dedicated accelerator performance rather than all computation occurring on the host, and is shown in Figure 6a) – where concurrent mandelbrot kernel execution overlaps compute with data transfers. The number of iterations was selected to be 512 and the blocksize increases over the x-axis. It appears 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.

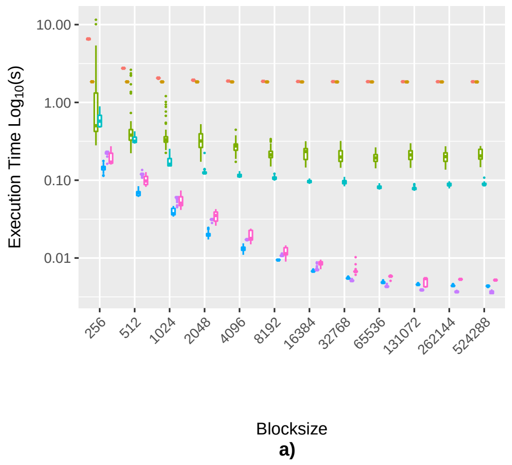
Figure 6b) presents the counter-point to using SYCL for dedicated accelerators by highlighting the overhead of memory movements over PCI-e to these devices. It shows the Gold's memory bus is generally four orders of magnitude faster than PCI-e under different benchmark configurations.

We now examine a selection of SYCL-Bench applications grouped by parallel construct. Because there are 32 unique applications –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 BKP applications that support 32-bit floats is shown in both Figure 7a) and WGP in Figure 7b) but 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. All figures –including those with other data-types– are presented in the associated Jupyter artifact [8].

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 construct applications 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 Figure 7e) to show the effect of barriers on runtimes. Both GPU devices –on all CUDA and ROCm backends– perform best on all applications. ComputeCpp's OpenCL responds the best to barriers on the Gold CPU, being roughly two orders



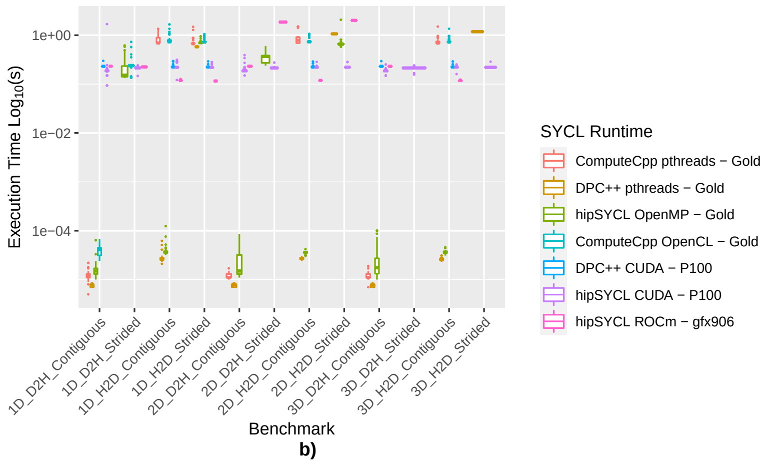


Fig. 6: Microbenchmarks **a)** *BlockedTransform* and **b)** *Bandwidth* to compare the performance of the SYCL Runtimes.

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 applications.

VI. Conclusions and Future Work

SYCL is a promising language for portable HPC. It offers a single-source implementation of algorithms, 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 of the many 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 magnitudes of absolute performance difference between the best and worst SYCL runtimes. Thus, the performance-portability problem –mapping a kernel to a device with optimal configuration– is ongoing and is made more complicated under SYCL; it gives another degree-of-freedom to the problem-space, where scheduling must now also consider selection of the same code to optimal backend. 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 BKP to be a useful parallel construct. We show tuning– carefully selecting larger work-group sizes (HDP and WGP constructs)– can still benefit GPU devices; however, this is less portable. The BKP 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 a swath of applications –from the SYCL-bench 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 last OpenCL backend for DPC++ which was the one missing backend in our work.

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*, 2020, pp. 1–1.

[4] Codeplay Software Ltd, "ComputeCpp." 2020. [Online]. Available: https://www.codeplay.com/products/computecpp

[5] 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.

[6] Intel Corporation, "Intel Data Parallel C++ Compiler," 2020. [Online]. Available: https://github.com/intel/llvm

[7] B. Johnston, "SYCL-Bench Docker image." 2020. [Online]. Available: https://hub.docker.com/r/beaujoh/syclbench

[8] ——, "SYCL-Bench Jupyter artefact." 2020. [Online]. Available: https://github.com/BeauJoh/sycl-bench

[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*, ser. IWOCL '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3204919.3204937

[10] J. Nickolls and I. Buck, "NVIDIA CUDA software and GPU parallel computing architecture," in *Microprocessor Forum*, 2007.

[11] 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.

[12] 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.

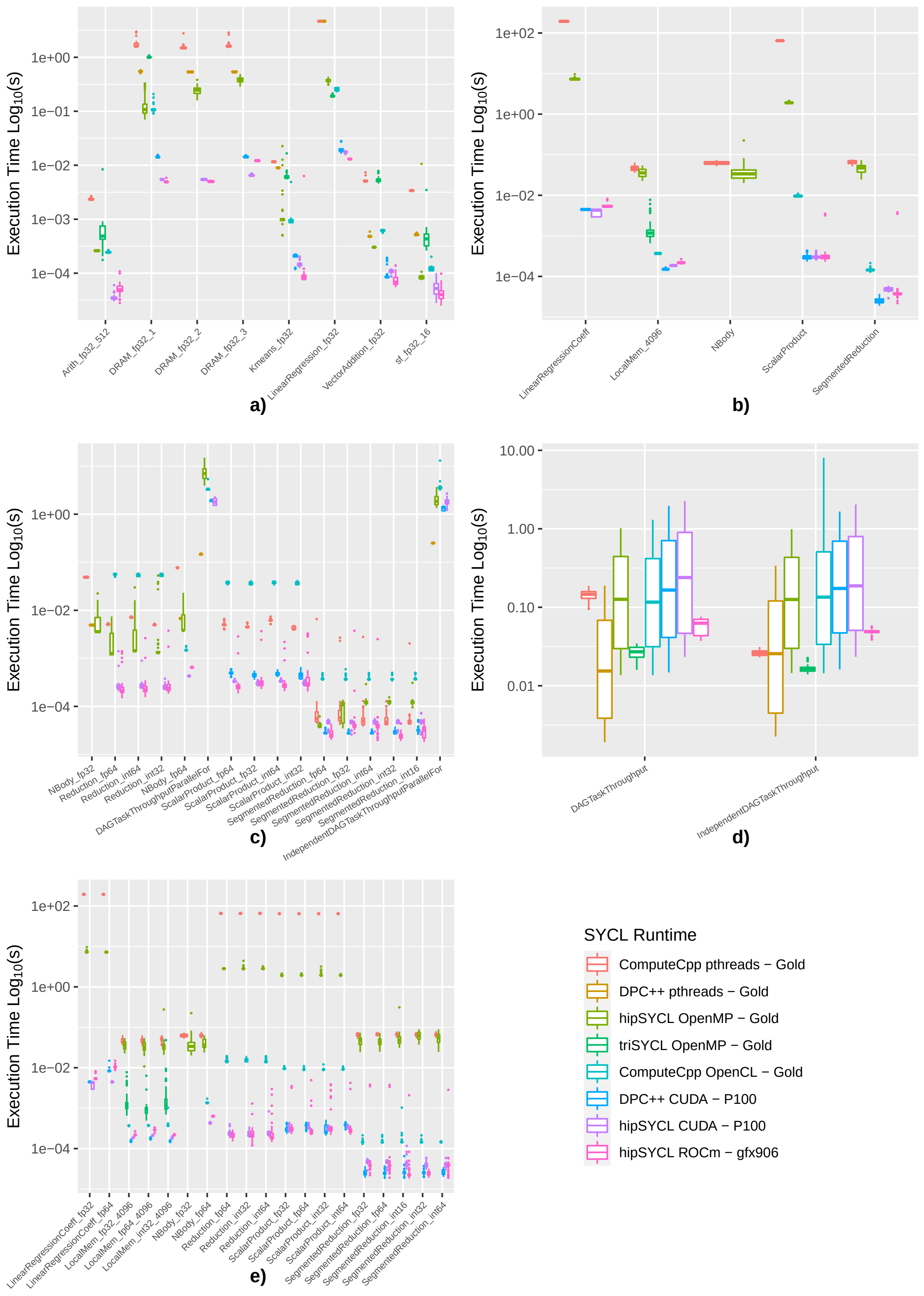


Fig. 7: Performance of a selection of SYCL-Bench benchmarks separated by parallel construct: **a)** Basic Kernel Parallelism (BKP), **b)** Work-Group Parallelism (WGP), **c)** Hierarchical Data-Parallelism (HDP), **d)** Single-Task Parallelism (Task), and **e)** Synchronization (Sync).