



Exploring SYCL as a Portability Layer for High-Performance Computing on CPUs

Downloaded from: <https://research.chalmers.se>, 2026-10-16 14:17 UTC

Citation for the original published paper (version of record):

Abram, H., Papadopoulou, N., Pericas, M. (2025). Exploring SYCL as a Portability Layer for High-Performance Computing on CPUs. ISC High Performance 2025 Research Paper Proceedings (40th International Conference). <http://dx.doi.org/10.23919/ISC.2025.11017734>

N.B. When citing this work, cite the original published paper.

Exploring SYCL as a portability layer for high-performance computing on CPUs

Hari Abram
Chalmers University of Technology,
University of Gothenburg
Gothenburg, Sweden
hariv@chalmers.se

Nikela Papadopoulou
University of Glasgow
Glasgow, U.K.
nikela.papadopoulou@glasgow.ac.uk

Miquel Pericàs
Chalmers University of Technology,
University of Gothenburg
Gothenburg, Sweden
miquelp@chalmers.se

Abstract—As multicore vector processors improve in computational and memory performance, running SIMT (Single Instruction Multiple Threads) programs on CPUs has become increasingly appealing, potentially eliminating the need for dedicated GPU hardware. SYCL is a royalty-free cross-platform C++ programming model for heterogeneous computing that implements the SIMT model and provides a path to run GPU programs on CPU hardware. Several SYCL implementations have been developed. To understand their performance on multicore vector CPUs, this paper systematically evaluates the two major SYCL implementations using a set of micro-benchmarks to assess key features such as memory bandwidth, kernel scheduling, synchronization, and vectorization. We test SYCL implementations across Intel Icelake, AMD Zen2, Fujitsu A64FX, and AWS Graviton3 systems, comparing Ahead-of-Time (AoT) and Just-In-Time (JIT) compilation flows, with OpenCL and OpenMP backends, and compare the results with native OpenMP implementations. Based on the results from these experiments, we generate a table of recommendations for SYCL developers. In general, our results show that AoT outperforms JIT in memory bandwidth, while JIT excels in scheduling and synchronization. SYCL on A64FX achieves near-peak memory bandwidth but underperforms in parallelization and reduction tasks. Our study also explores data management strategies, including USM and buffer & accessor (B&A), and backend configurations, revealing that these factors significantly impact performance. Our findings underscore the importance of selecting the appropriate toolchain to maximize SYCL’s potential for portable and high-performance execution across diverse hardware platforms.

Index Terms—Benchmarking, CPUs, SYCL, OpenMP, Intel OneAPI, DPC++, AdaptiveCpp

I. INTRODUCTION

Recent advances in heterogeneous architectures have raised concerns about performance portability and the complexity of application porting. To address these, the Khronos Group developed the SYCL standard [1], which aims to target performance portability across different architectures. SYCL follows an SPMD (single-program, multiple-data) approach similar to OpenCL/CUDA and is primarily designed for GPU-based computing. However, current trends in CPU architectures, such as long vector widths and high memory bandwidth exemplified by processors like the Fujitsu A64FX, are being explored to achieve performance levels comparable to those of GPUs [2]–[6]. These developments underscore the need to analyze the performance of SYCL on CPUs.

The open nature of SYCL allows for various implementations, each relying on different compilation toolchains and runtimes. When targeting CPUs, Intel DPC++ [7] employs JIT compilation with an OpenCL backend. At the same time, AdaptiveCpp [8] can employ either JIT compilation, with OpenCL or OpenMP runtime, or AoT(Ahead-Of-Time) compilation with OpenMP. This differs from GPU-targeted SYCL, which typically relies on vendor-provided runtime systems, such as CUDA with NVIDIA GPUs. Achieving peak performance when executing SYCL code on CPUs is challenging due to the wide variety of available runtimes and compilation methods. This raises the critical question of which toolchain should be selected when targeting specific CPUs.

SYCL can act as a portability layer across GPUs and CPUs, but its performance on multicore vector CPUs needs careful evaluation. Inheriting its design from OpenCL, SYCL is more closely aligned with GPU-like architectures. Therefore, its effectiveness on modern CPUs requires further investigation and analysis. SYCL offers multiple ways to parallelize a kernel offloaded to a runtime, such as utilizing various iteration spaces (`sycl::range` vs `sycl::nd_range`), dimensions, and data management methods (USM (Unified Shared Memory) vs B&A (Buffer and Accessors)). Combining these diverse methods with the various toolchains available in SYCL implementations presents a challenge for developers aiming to achieve acceptable performance in their applications, considering these methods were developed to target GPU-like architectures. Motivation: Given the diverse array of toolchains available for targeting CPUs through SYCL, how can we effectively leverage these toolchains to optimize performance on multicore vector CPUs? This paper characterizes the performance of SYCL methods, focusing on the impact of these methods on vectorization, scheduling, and synchronization.

Quantifying the influence of runtimes and compilation flows on the performance of SYCL features through application-level benchmarks is challenging due to the difficulty of achieving isolation across SYCL features, given the interplay of the various SYCL features used in such benchmarks. Additionally, although similar analyses have been conducted on specific applications [9]–[17], it remains challenging to pinpoint the specific SYCL features that contribute to performance bottlenecks. To address this, we have developed a suite of micro-

benchmarks¹ that enable a bottom-up approach for assessing the performance of SYCL features across various implementations and runtimes on CPUs. These micro-benchmarks complement existing SYCL benchmark suites such as SYCL-Bench [18], HeCBench [19] and PRK [20], by enhancing the scope and depth of the benchmarks. We achieve this through the use of different methodologies and by incorporating more parametrization, allowing for a more comprehensive evaluation of SYCL performance across various configurations.

Our microbenchmarks evaluate the impact of runtime on memory bandwidth, parallelization, vectorization, and other factors on four processors with different ISAs: Intel Icelake, AMD Zen2, Fujitsu A64FX, and AWS Graviton3. These four processors are selected to showcase different vectorization and parallelization capabilities. We also compare the performance of SYCL against OpenMP by developing equivalent OpenMP benchmarks, which serve as our baseline, considering OpenMP is a standard parallel programming model for CPUs. Note that, unlike SYCL, OpenMP requires specific constructs to address GPUs, compared to the standard method when targeting CPUs, this requires an extra effort to make OpenMP single-source portable, though this effort may be minimal. Our analysis aims to identify key SYCL features and the impact of runtime and backend choices on performance, to achieve portable and acceptable performance across systems.

With our microbenchmarks, we observe the performance difference across SYCL implementation versions, compilation flows and backends while testing for performance portability across systems. Specifically, we observe that JIT with OpenCL is preferred when data movement between main memory and the processor is minimal, whereas AoT with OpenMP is preferred when there is a need for high memory bandwidth. In addition, we show that SYCL on x86 machines outperforms AArch64 systems in scheduling and synchronization tasks. When SYCL is compared to OpenMP, performance superiority varies by system and SYCL implementation, with some JIT workflows outperforming OpenMP, particularly in parallelization, where JIT compilation performs up to 10% better than OpenMP. More detailed findings are presented in Table VIII. Overall, our experiments suggest that SYCL performance on multi-core vector CPUs is influenced by factors such as compilation flows, backend choice, and system-specific hardware features.

This paper makes the following contributions:

- We systematically evaluate the various runtimes and compilation flow toolchains of SYCL implementations when targeting modern multicore CPUs with vectorization capabilities.
- We assess the potential of SYCL for portability across different CPU architectures by evaluating its utilization of available features on various hardware platforms.
- We show that the achieved memory bandwidth is strongly influenced by the compilation flow, with AoT compilation achieving the highest possible bandwidth across all

systems. In contrast, parallelization shows better performance with JIT compilation. Additionally, our findings reveal that, on certain systems, the cost model plays a significant role in the vectorization of the kernel.

The remainder of this paper is organized as follows. We discuss related work in Section II. Section III outlines our SYCL micro-benchmarks. Section IV provides details on the experimental configuration. In Section V, we present our experimental results for the developed micro-benchmarks and discuss our findings. We conclude this work in Section VI.

II. RELATED WORK

With the rising popularity of SYCL, several works have used SYCL as the programming model of their choice [9]–[16], which has led to the development of several SYCL benchmark suites [18], [20]–[22]. The SYCL-Bench suite [18], [21] offers valuable insights into the performance of SYCL. It has certain similarities to our work, particularly in trying to characterize the SYCL runtime effects, focusing on task scheduling and data management. However, the primary target of this research work is GPUs, with benchmarks specifically designed to characterize GPU runtimes and architectures. Our work aims to target multicore vector-capable CPUs by focusing on the specific aspects of SYCL performance. Still, the main reason for developing our microbenchmarks rather than using SYCL-bench is due to the degree of freedom available and the in-depth analysis we can achieve, compared to using other benchmark suites. For example, in SYCL-bench, each kernel is predominantly executed using a single method, whereas, in our micro-benchmark suite, we test multiple methods to initiate the execution of kernel for each of the micro-benchmarks.

Several research works also have benchmark suites to assess SYCL performance in the heterogeneous landscape. The PRK benchmarks [20], [22] comprise three benchmarks implemented in SYCL: stream, 2D stencil, and point-to-point synchronization. While these benchmarks are useful, they have limitations in fully characterizing SYCL implementations and their runtime behavior. The Rodinia Benchmarks [17] have also been ported to SYCL. The complexity of these benchmarks imposes constraints on characterizing the runtime effects of SYCL features. Moreover, the intricacy makes it challenging to study the effects of SYCL constructs in isolation. HeCBench [19] has been developed to test the performance portability of SYCL at the application level. These three benchmark suites are primarily designed to test the SYCL performance at the application level.

Recent work by [23] investigates the offloading of the kernel matrix assembly in PLSSVM (Parallel Least Squares Support Vector Machine) using various SYCL methods. This research shares a similar objective to our study, but differs in its approach to analyzing performance overheads. While the paper focus on the overall performance of kernels in different kernel offloading methods, our study measures the overheads incurred during the scheduling of parallel tasks using various offloading methods. This allows us to analyze task scheduling overhead independently, without the influence of kernel computation

¹<https://github.com/HariAbram/microSYCL.git>

overhead. Several research works have predominantly focused on the process of porting applications to SYCL, documenting the experiences and challenges encountered. Examples of previous research work with SYCL include the conjugate gradient [9], molecular dynamics applications [12], stellar merges simulations [13], epistasis detection [14], LU decomposition [15], cluster methods [10], bandwidth-bound applications [24], Fourier transform library [16] and AI graphs [11]. In recent work, Joubé et al. [25] implement diverse memory management strategies within SYCL to evaluate their effectiveness, this approach aims to provide a more comprehensive understanding of the intricacies surrounding the performance implications of employing these features in SYCL-based applications.

III. METHODOLOGY

This work focuses primarily on the performance characterization of various SYCL features through the utilization of microbenchmarks. To effectively understand the impact of compilation toolchains and runtimes on the various parameters through which SYCL kernels are executed on CPUs, flexibility and parametrization are essential when offloading these microkernels. This need for flexibility led to the development of our suite of microbenchmarks. The inspiration for this approach comes from the EPCC benchmark suite [26], a well-established collection of OpenMP microbenchmarks designed to evaluate the performance of parallel constructs and features in OpenMP. Within our micro-benchmark suite, we analyze a range of SYCL features, including memory bandwidth of Unified Shared Memory (USM) allocations, parallel constructs employing `range` and `nd_range` objects, various SYCL data management techniques, accumulation via atomic operations, reduction, as well as group and subgroup barriers. Additionally, our suite includes microbenchmarks to evaluate the impact of vectorization on SYCL kernels, when various SYCL features are combined. Each microbenchmark is executed 100 times, with the resulting average, median, maximum, and minimum execution times used for performance comparison. It is important to note that all variables considered in this micro-benchmark study are of type `double`. Although the experiments were also conducted using the `float` type, similar trends were observed for both types.

In the following subsections, we provide detailed descriptions of each of these features and the corresponding microbenchmarks that we have developed for their evaluation. Furthermore, we outline our definition of the baseline performance for each feature, also explaining how we obtain the baseline, to effectively measure any additional overhead introduced by the SYCL software stack.

A. Microbenchmarks for Evaluating SYCL Features

1) *Memory bandwidth*: The SYCL standard encompasses three distinct methods for USM data allocation, namely, `sycl::malloc_host`, `sycl::malloc_shared`, and `sycl::malloc_device`. The first method allocates shared data and pins it to the host memory, facilitating access by both

the host and the device, while the second method allocates a shared memory region that is similarly accessible by both the host and the device, but pins it to the device. The third method allocates device-specific memory, which can only be accessed by the device, and is pinned to the device. In our microbenchmark, we evaluate the memory bandwidth observed when employing these different memory allocation methods. Furthermore, we examine the memory bandwidth performance when using the `range` and `nd_range` iteration spaces. These different data management strategies in SYCL are used to target heterogeneous computing systems, where data has to be transferred from CPU to accelerator or GPU, it is important to note that since the experiments within our work execute their kernels on a CPU, the host and device are equivalent from the implementation perspective. We then compare this memory bandwidth with the bandwidth achieved with the BabelStream [27] benchmark, which uses OpenMP for parallelization.

2) *Parallelization*: The `parallel_for` method is a principal parallelization construct in SYCL that serves as the means to delegate computational tasks to the device. It accepts two key inputs: a kernel, represented in the form of a lambda function, and the iteration space defined using `range` and `nd_range` objects. The method proceeds to generate a distinct kernel instance for every iteration within the specified iteration space, where the kernel instance corresponds to the lambda function that is executed on individual threads created by the runtime.

Kernel instances generated with the `range` template are entirely independent of each other, and there is no communication or synchronization between these distinct instances. On the other hand, the `nd_range` class template introduces a concept of global and local range. The global range is subdivided into subgroups defined by the local range. Kernel instances associated with the local range possess shared memory, facilitating communication and synchronization of computation among these specific instances.

SYCL offers two distinct methods for offloading a kernel to a device, depending on the data management strategy. The first method involves the use of buffers and accessors, where data is initially transferred to a `sycl::buffer` and subsequently accessed through a `sycl::accessor` within the `sycl::handler`. The second approach involves Unified Shared Memory (USM); here, global variables created through the SYCL shared context can be accessed within the `sycl::handler`, unlike variables created with general memory allocations. The general approach to offloading a kernel is demonstrated in Listing 1.

The microbenchmark designed for this feature assesses the synchronization and scheduling overheads of the SYCL backend runtime. This evaluation encompasses a range of iteration spaces, each characterized by varying dimensions. Furthermore, the benchmark serves to explore the influence of both the problem size and the iteration space dimensions on performance. These parameters are determined when declaring a `range` or `nd_range` type, denoted as `range<dim>`.

```

sycl::queue Q;
sycl::buffer<TYPE , 1> buff(var,1);
Q.submit([&](sycl::handler& cgh){
    sycl::accessor acc = buff.get_access
        <sycl::access::mode::read_write>(cgh);
    cgh.parallel_for<>(sycl::range<1>
        (global), [=](sycl::item<1> it){

        ///computation
    });
});

```

Listing 1: Kernel offloading for B&A data management

The aim is to scrutinize how different problem sizes and dimension choices impact the overall performance of the kernel offloading process.

To conduct a comparison between a SYCL kernel offloaded with a two-dimensional iteration space and its OpenMP counterpart, we employ the `omp parallel for collapse(2)` construct within an equivalent OpenMP microbenchmark. This approach allows for a direct comparison between SYCL and OpenMP in scenarios involving two-dimensional iteration spaces [8].

```

for(size_t i = 0; i < 1024; i++) {
    if (sum[thread_id] < 0)
    {
        break;
    }
    sum[thread_id] += 1;
}

```

Listing 2: Kernel computation

The microbenchmark for parallelization serves a twofold purpose: firstly, to identify the most effective combination of features for achieving parallelization, and secondly, to investigate whether this optimal combination remains consistent across different vendors, SYCL implementations, and backend runtimes. The kernel computation that is offloaded in the microbenchmark is outlined in Listing 2, and the `thread_id` is obtained with `sycl::item.get_thread_id(0)`. To accurately measure the overhead associated with parallelization, the computational tasks executed by each thread are kept minimal to avoid directly influencing the benchmark results. The variable `sum` is accessed either using SYCL’s unified shared memory (USM) or via a buffer and accessor, depending on whether the kernel is offloaded in a shared context or not. This approach allows us to distinctly quantify the influence of various parallelization techniques on performance.

3) *Atomics and Reduction*: To assess the performance of atomic operations across various SYCL implementations and to compare them against OpenMP atomic constructs, we introduce a sum accumulation kernel, employing atomics to perform the accumulation of an array into a variable.

For the SYCL atomics microbenchmark, we employ different data management strategies. An array is declared and managed either through USM or using a buffer and accessors. The accumulation operation (summation) is carried out on this array within the kernel. The kernel is offloaded to the device and the array is accessed in the kernel, depending on

whether the array is managed through a buffer and accessors or USM. The SYCL kernel offloaded in this microbenchmark is presented in Listing 3.

```

auto j = it.get_id(0);
auto v = sycl::atomic_ref<TYPE,
    sycl::memory_order::seq_cst,
    sycl::memory_scope::device,
    sycl::access::address_space
        ::global_space>(
    sum[0]);
v.fetch_add(m[j]);

```

Listing 3: Atomics implementation

We additionally implement reduction using the `sycl::reduction` method. We highlight that the `sycl::reduction` syntax differs for AdaptiveCpp and DPC++ when utilizing Buffer and Accessor data management, as presented in Listing 4. This is a result of changes in syntax across different versions of the SYCL standard. Between these two SYCL implementations, DPC++ conforms to the latest standard.

```

auto m_acc = m_buff.get_access<
    sycl::access::mode
        ::read>(cgh);
#ifdef (AdaptiveCpp)
auto sum_acc = sum_buff.get_access<
    sycl::access::mode
        ::read_write>(cgh);
auto sum_red = sycl::reduction(sum_acc,
    sycl::plus<TYPE>());
#else if defined (DPC++)
auto sum_red = sycl::reduction(sum_buff, cgh,
    sycl::plus<TYPE>());
#endif
cgh.parallel_for(sycl::
    nd_range<1>(global, local),
    sum_red, [=](sycl::
    nd_item<1>it, auto &sum){

    auto j = it.get_global_id(0);
    sum += m_acc[j];
});

```

Listing 4: SYCL reduction implementation

For comparison purposes, we implement an OpenMP atomics microbenchmark that uses an atomic construct. We also introduce an OpenMP benchmark that utilizes a reduction construct. It is worth noting that the OpenMP reduction construct is known for offering high performance [28] when conducting reduction operations.

4) *Barriers*: In the SYCL programming model, barriers can be performed at the group level or the subgroup level. In our microbenchmark, we evaluate these two different barriers, both with USM and with buffer and accessors, but also with different dimensions. We compare the performance of these barriers with the OpenMP barriers.

B. Vectorization

This study extends its analysis to examine the vectorization capabilities of the various SYCL implementations across different machines. Additionally, we investigate the compiler’s auto-vectorization capabilities when using different iteration

spaces, such as `sycl::range` and `sycl::nd_range`, and assess the effectiveness of employing different data management strategies. The aim is to gain comprehensive insights into how these factors impact the overall auto-vectorization performance of SYCL implementations across different hardware and SYCL constructs.

To thoroughly assess the vectorization properties, we introduce four microbenchmarks in our set: GEMM (general matrix multiplication), GEMV (general matrix-vector multiplication), outer product, and cross-product. In these microbenchmarks, we focus on evaluating the auto-vectorization capabilities of SYCL implementations. The outer product benchmark analyzes vectorization across work items, while the GEMM kernel focuses on the vectorization of loops within work items for a 2-dimensional iteration space. Similarly, GEMV tests vectorization for a 1-dimensional iteration space. Additionally, we include a cross-product kernel that implements conditionals inside the kernel to analyze their impact on vectorization. Listings 5, 6, 7, and 8 present the kernels for outer-product, GEMM, GEMV, and cross-product, respectively. We also test the `sycl::reduction` construct, introduced in Listing 4, to analyse the capabilities of SYCL implementations to vectorize the standard SYCL synchronization constructs. We additionally compare this with the OpenMP reduction clause.

```
cgh.parallel_for<>(sycl::nd_range<2>(global1, local1),
    [=](sycl::nd_item<2>it){

    auto i = it.get_global_id(0);
    auto j = it.get_global_id(1);
    auto N = it.get_global_range(0);
    m1_acc[i*N+j] = v1_acc[i]*v2_acc[j];
});
```

Listing 5: Outer product Kernel

```
cgh.parallel_for<>(sycl::nd_range<2>(global1, local1),
    [=](sycl::nd_item<2>it){

    auto i = it.get_global_id(0);
    auto j = it.get_global_id(1);
    TYPE temp = 0.0;
    for (size_t k = 0; k < N; k++)
    {
        temp += m2_acc[i*N+k]*m1_acc[k*N+j];
    }
    m3_acc[i*N+j] = temp;
});
```

Listing 6: GEMM Kernel

```
cgh.parallel_for<>(sycl::nd_range<1>(global, local),
    [=](sycl::nd_item<1>it){

    const int i = it.get_global_id(0);
    const int N = it.get_global_range(0);
    TYPE temp = 0.0;
    for (size_t k = 0; k < N; k++)
    {
        temp += m1_acc[i*N+k] * v1_acc[k];
    }
    v2_acc[i] = temp;
});
```

Listing 7: GEMV Kernel

```
cgh.parallel_for<>(sycl::nd_range<2>(global1, local1),
    [=](sycl::nd_item<2>it){

    auto i = it.get_global_id(0);
    auto j = it.get_global_id(1);
    auto N = it.get_global_range(0);
    auto tmp = std::pow(-1, i+j);
    if (i == j)
    {
        m1_acc[i*N+j] = 0;
    }
    else
    {
        m1_acc[i*N+j] = v1_acc[i]*v2_acc[j]*tmp;
    }
});
```

Listing 8: Cross product Kernel

It is essential to highlight that all arrays and pointers are initialized with the `__restrict__` keyword. This keyword serves as an indication to the compiler that the memory locations of these arrays are independent, which is a critical consideration for the accurate evaluation of auto-vectorization properties. In the vectorization tests, we consider a benchmark to be vectorized if the vector instruction counters show that all cores have executed vector instructions and that the counts are comparable to the average number of vector instructions executed, calculated across all cores. This ensures that the observed vectorization is not a result of computations unrelated to the offloaded kernel but instead has a meaningful impact on performance. We also analyze the compiler reports with `-Rpass-analysis` flags, for a more thorough summary of auto-vectorization. While reporting vectorization results, we focus on whether the systems execute vector instructions and the type of vector instructions.

IV. EXPERIMENTAL SETUP

This work focuses primarily on assessing the performance of SYCL features when executed on CPUs. To this end, we select four CPUs for our experiments, an Intel Xeon Platinum 8358, an AMD EPYC™ 7742, a Fujitsu A64FX, and an AWS Graviton3. We present the details of the experimental platforms in Table I.

In this study, we use two distinct SYCL implementations, namely AdaptiveCpp and DPC++. Table II provides an overview of the SYCL implementations with different compilation flows and backends. Notably, the AdaptiveCpp implementation is available with two different compilation flows, JIT compilation and AoT compilation, each with distinct backends.

Table III presents the list of experiments conducted in this study, which were carefully selected from a larger exploration space. In the following paragraphs, we will briefly explain the setup and tools used for each experiment.

To analyze historical performance trends, we opt to use stable versions of each SYCL implementation from 2022, 2023, and 2024, conducting the comparison on x86 machines. For older versions (v2022) of AdaptiveCpp, we had to use LLVM v14.0.0 for compilation, as older versions of AdaptiveCpp

TABLE I
PLATFORMS

Machine	Architecture	Cores per Node	Threads per Core	NUMA nodes per CPU	Max Frequency (MHz)	L1 Cache	L2 Cache	L3 Cache	SIMD Width
Fujitsu A64FX	ARM-SVE 64-bit	48	1	4	2200	3 MB	32 MB	N/A	512 bit
AMD EPYC Zen2	x86-64	128	2	2	2250	4 MB	64 MB	512 MB	256 bit
Intel Xeon Platinum	x86-64	32	1	2	3700	32 KB	1 MB	22 MB	512 bit
AWS Graviton3	ARM-SVE 64-bit	64	1	1	2300	80 KB	8 MB	N/A	256 bit

TABLE II
SYCL IMPLEMENTATIONS COMPILE FLOWS

SYCL Implementation	Compilation	Backend
AdaptiveCpp	JIT	OpenCL; OpenMP
	AoT	OpenMP
DPC++	JIT	OpenCL

depend on obsolete syntax that is no longer used in newer versions of LLVM, while newer versions of AdaptiveCpp were compiled with LLVM v18.1.8. For DPC++, we leverage the available versions provided by the package manager, specifically for this study, we utilized v2022.4 v2023.4 and v2024.2.

TABLE III
EXPERIMENTS CONDUCTED IN THIS STUDY

Experiments conducted in this study	Machines
Historical comparison across different versions for AdaptiveCpp and DPC++	Intel Icelake, AMD Zen2
Backend comparison for OpenMP and OpenCL runtimes with AdaptiveCpp JIT and AoT compilation flows.	Intel Icelake
Portability comparison of AdaptiveCpp on the systems mentioned in Table I	All the machines
Vectorization tests across the systems and SYCL implementations, for different backend and compilation flows.	All the machines

We analyze the overheads associated with different compilation workflows and backends on a single system, specifically an Intel Xeon Platinum machine based on the Icelake microarchitecture. For this analysis, we utilize the latest versions of AdaptiveCpp, compiled with LLVM v18.1.8.

To compare SYCL performance across different machines, we utilize AdaptiveCpp, as this implementation functions seamlessly across all the systems. On Fujitsu A64FX, it has been shown in the Fujitsu compiler package documentation [29] that the `XOS_MMM_L_PAGING_POLICY` environment variable has a significant impact on C++ code performance on A64FX. However, this variable affects only the Fujitsu compilers, which are not employed by any of the SYCL implementations. We therefore do not set this environment variable in our experiments.

We use LLVM to compile OpenMP code, which uses the respective LLVM OpenMP runtime with v18.1.8. In all experiments, we set the environment variable `OMP_PROC_BIND` to true, as this results in better performance for both the AdaptiveCpp (when offloaded to the OpenMP runtime) and OpenMP code.

It is important to acknowledge that the systems examined

in this study feature different numbers of cores per node. In our work, we restrict our experiments to the physical cores and do not utilize Simultaneous Multi-Threading (SMT).

V. EXPERIMENTAL RESULTS

Our analysis systematically compares the experimental results obtained with various SYCL implementations, considering that these implementations may leverage different compilation flows and backend runtimes.

Our evaluation assesses the performance of SYCL features in relation to their dependency on the backend runtimes. We evaluate each micro-benchmark with a range of problem sizes, to determine whether the performance indicators scale with increasing problem sizes. The term "problem size" varies for each micro-benchmark and may refer to parameters such as array size, the number of instances of the kernel executed, or a combination of both. We compare the different SYCL versions and implementations in terms of execution time. To compare with OpenMP, we normalize the SYCL results against the OpenMP results. We omitted confidence intervals, as the difference between mean, min and max is insignificant. On average, we see a difference of $< 1\%$, $< 5\%$ and $< 1\%$ between min, max and median, compared to the mean, respectively, across benchmarks.

In the following subsections, we first present the historical comparison of the AdaptiveCpp and DPC++ on x86 machines. Next, we compare different backends with AdaptiveCpp on a single machine. Subsequently, we evaluate the performance of AdaptiveCpp on different machines. Finally, we examine the vectorization properties of the various SYCL implementations and backends.

A. Historical Comparison

For the historical comparison of SYCL versions, we present the compilation flows used for each benchmark and version in Table IV. For memory copy operations in the latest versions of AdaptiveCpp, we employed Ahead-Of-Time (AoT) compilation with OpenMP runtime, as this compilation flow achieves higher bandwidth.

TABLE IV
BENCHMARK COMPILE FLOWS

Microbenchmark	AdaptiveCpp			DPC++
	v22	v23	v24	all versions
Memory bandwidth	AoT	AoT	AoT	JIT
Parallelization	AoT	JIT	JIT	JIT
Barriers	AoT	JIT	JIT	JIT

Figure 1, presents the results for the various versions of SYCL implementations, specifically AdaptiveCpp (ACpp) and DPC++. For comparison, we also include the results of BabelStream, which uses OpenMP, compiled with Clang++ v18.1.8. Our observations indicate that AdaptiveCpp achieves higher bandwidth compared to DPC++, and its performance is closer to the bandwidth achieved by the BabelStream OpenMP benchmark. The difference in the maximum bandwidth achieved by the two implementations is related to runtimes, with AdaptiveCpp utilizing OpenMP runtime and DPC++ employing OpenCL runtime in this benchmark.

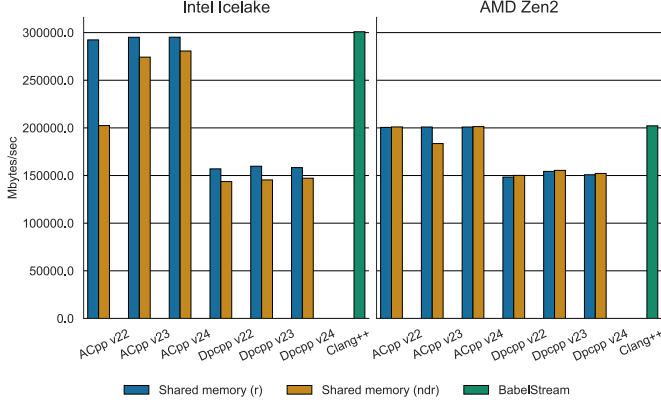


Fig. 1. Memory bandwidth across various versions, for a problem size of 2GB. ndr and r in parenthesis correspond to nd_range and range, respectively.

Figure 2, presents the results for range and nd_range, normalized against OpenMP results. For this micro-benchmark, the problem size corresponds to the size of the iteration space. We observe that DPC++ performance consistently improves year over year on both systems. Notably, on Intel systems, DPC++ demonstrates twice the performance of AdaptiveCpp for one-dimensional iteration spaces. However, our experiments indicate that, for two-dimensional iteration spaces, the performance of DPC++ and AdaptiveCpp is similarly poor. We omit these results for the sake of brevity.

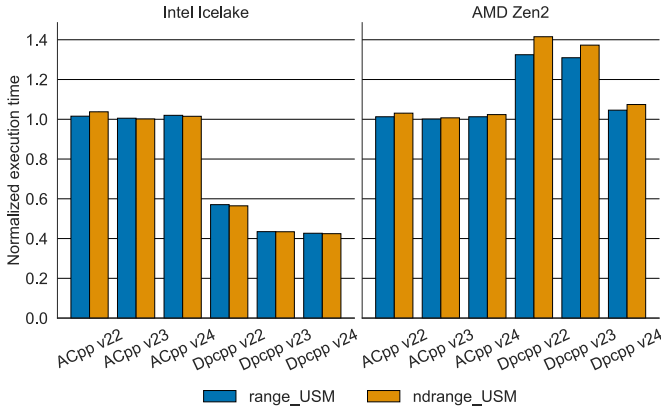


Fig. 2. Parallelization results normalized against OpenMP for a problem size of 2^{20}

Figure 3 presents the results for the barriers at group and subgroup levels, normalized against OpenMP results, compiled with Clang++ (v18.1.8). As in the parallelization benchmark, the problem size corresponds to the size of the iteration space. Overall, there is no significant difference in performance across the different SYCL versions. However, we observe that in AdaptiveCpp v23, group barrier performance is noticeably worse than subgroup barrier performance. This discrepancy coincides with the introduction of the JIT compilation in that version.

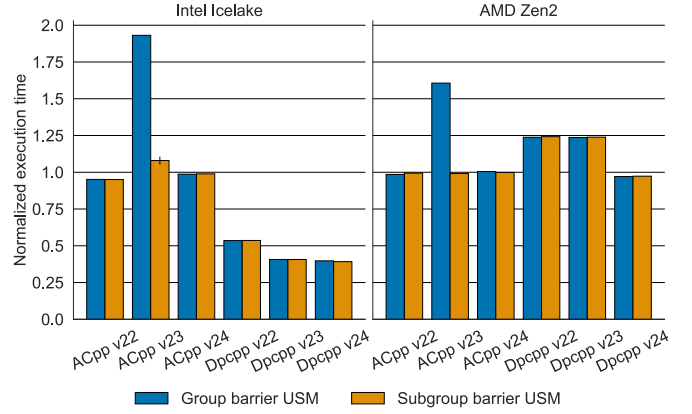


Fig. 3. Barriers results normalized against OpenMP for a problem size of 2^{20}

B. Backend Comparison

Comparing different backends used for offloading SYCL kernels provides insights into the overheads associated with each backend and compilation flow. Since AdaptiveCpp can offload to various backends via its generic JIT compilation flow, we utilized AdaptiveCpp to perform a backend comparison on the Intel Icelake machine. In Figures 4, 5, 6, 7, *JIT ocl* and *JIT omp* correspond to Just-In-Time compiled SYCL code offloaded to OpenCL and OpenMP respectively, and similarly, *AoT omp*, corresponds to Ahead-of-Time compiled SYCL code offloaded to OpenMP.

Figure 4 presents the memory bandwidth results for the various backends and compilation flows when the memory operations are performed on shared memory variables. We observe that the AoT compilation flow with the OpenMP backend achieves the highest bandwidth among the different backends compared. When comparing the bandwidth obtained by DPC++ with the OpenCL backend in Figure 1 to that of the generic OpenCL backend of AdaptiveCpp, the results are similar. This suggests that the OpenCL backend incurs significant overhead during memory operations.

Through memory counters in LIKWID, we observe that with *JIT ocl*, the TLB load and store misses were 3 and 3.7 times higher, and total loads and stores 13 and 15 times higher, respectively, compared to *AoT omp*. Similarly, cycle stalls due to memory operations during the execution of *JIT ocl* code were 18.5 times higher than those observed with *AoT omp* code. Although these findings suggest potential reasons for

the differences between *JIT ocl* and *AoT omp*, no such stark contrasts were observed between *JIT omp* and *AoT omp*. This indicates that the memory model used in OpenCL [30] may be a key factor contributing to the poorer performance observed with *JIT ocl* and *AoT omp*.

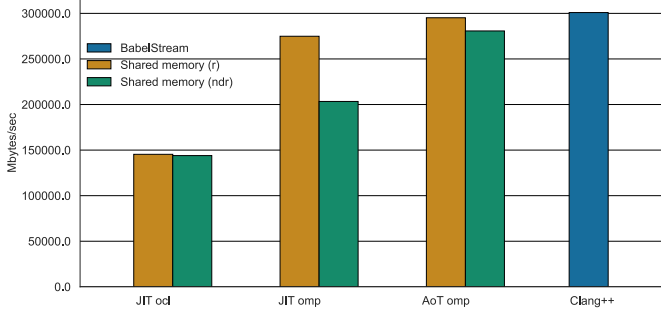


Fig. 4. Memory bandwidth across various backends, for a problem size of 2GB, on Intel Icelake machine.

We also examine the memory bandwidth when using the range and nd_range iteration spaces, where memory operations using the range iteration space achieve higher bandwidth. We have additionally compared the memory bandwidth with the memory allocated on the host or the device, observing similar bandwidth levels to those achieved with shared memory.

Figure 5 presents the results for our parallelization micro-benchmark, using the range feature with USM and Buffer & Accessor (BA) data management, normalized against OpenMP results. The results for nd_range are similar to those for range. For this micro-benchmark, the problem size corresponds to the size of the iteration space. We analyze various problem sizes, to understand if the performance scales with increasing problem sizes. We observe that the OpenCL backend is up to 10% more performant than the OpenMP backend on the Intel Icelake architecture, though, in the case of small problem size (2^{16}), this is the opposite. Furthermore, the performance of *JIT omp* and OpenMP are equivalent, and between USM and B&A we don't observe significant differences as there is very little data movement in this benchmark. We also note that *AoT omp* results are similar to *JIT omp*, though these results are not present in the figure.

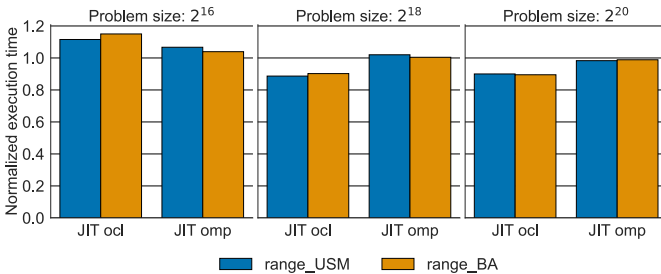


Fig. 5. Parallelization results normalized against OpenMP, with USM and BA data management, on Intel Icelake machine.

Figure 6 presents the atomics results for different backends and data management strategies normalized against OpenMP execution time. In this context, the problem size refers to the array size that is summed using atomic operations. When comparing different backends, we observe that the performance is generally similar for both OpenCL and OpenMP.

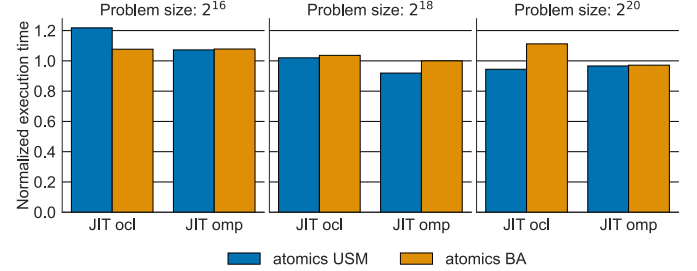


Fig. 6. Atomics results normalized with OpenMP, with USM and BA data management, on Intel Icelake machine.

Figure 7 presents the results for the reduction micro-benchmark with different backends, normalized to OpenMP execution time. We observe that the SYCL overhead compared to OpenMP is substantially higher, with USM-managed reductions performing up to 5 times worse than OpenMP reductions. More in-depth analysis of this SYCL feature through hardware counters shows that USM and B&A have a higher instruction count than OpenMP in all the counters we observed, such as cache misses, cache hits, TLB load and store misses, and cycle stalls. Analyzing the traces of these kernels, we saw that SYCL kernels spend the predominant amount of time on `__sched_yield()` function compared to OpenMP, corresponding to more time spent on resource contention. Another observation is that, for larger problem sizes, the performance difference between OpenMP and SYCL (both USM and B&A) significantly decreases.

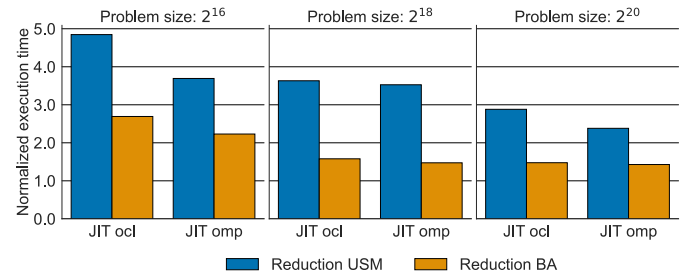


Fig. 7. Reduction results normalized against OpenMP, with USM and BA data management, on Intel Icelake machine.

C. Portability comparison

In this subsection, we present results for the micro-benchmarks on various platforms, the specifications of which are presented in Table I. These experiments focus on understanding the performance portability of various SYCL features across machines and architectures. To ensure portability across

the platforms, we utilize the AdaptiveCpp SYCL implementation.

Figure 8 presents the results for memory bandwidth on various systems. We observe that on the A64FX system, SYCL achieves bandwidth comparable to that of the OpenMP BabelStream benchmark, which we use as a yardstick for comparison. However, on other machines, there is a slight decrease in the bandwidth obtained by SYCL compared to OpenMP. We note that, on the A64FX processor, the maximum achievable bandwidth is 830 GB/s [31], but this performance can only be reached using `zfill` and `software prefetch` optimizations, which are available exclusively with the Fujitsu compiler.

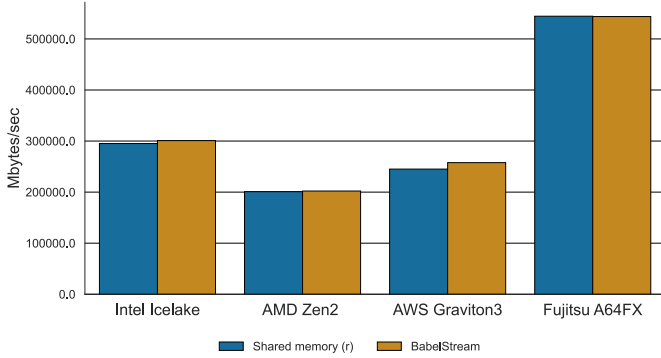


Fig. 8. Memory bandwidth across various machines, for a problem size of 2GB.

Figure 9 presents the results for the parallelization, with USM data management and `sycl::range` iteration space, atomics, and reduction benchmarks. In the parallelization micro-benchmark, AdaptiveCpp on A64FX performs worse among all the machines tested. It is well-known that A64FX performance is highly dependent on the compiler used, with the Fujitsu compiler achieving significantly better performance compared to Clang or GCC [31]. However, none of the SYCL compilers we are aware of are based on the Fujitsu compiler. AArch64 machines in general perform worse than x86 machines in the parallelization benchmark.

Similarly, we observe that the performance of atomics is 60 times worse on A64FX compared to other machines tested. This issue seems to be related to the CAS or LL-SC instructions used in A64FX, as discussed in [32]. Comparing reduction on various machines shows that Intel and Graviton3 systems perform best out of all the machines.

We summarize our observations from our micro-benchmark study across systems, backends, and compilation flows in Table VIII.

D. Vectorization

In this subsection, we present the results of the benchmarks that test for the auto-vectorization of the kernels across the work items and inside the work items. The benchmarks we consider are detailed in Section III.

Table V presents the compilation flags used for both SYCL implementations across all systems in this study. The `-Ofast`

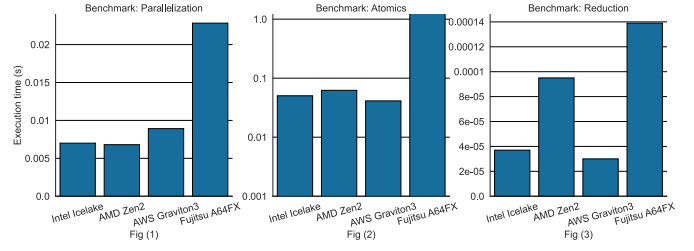


Fig. 9. Results across various systems for different benchmarks with a problem size of 2^{20}

flag is crucial for vectorizing the GEMM kernel with the AdaptiveCpp SYCL implementation. On the A64FX system, vectorization is not achieved without the `-mcpu=a64fx+sve` flag.

TABLE V
COMPILER FLAGS

Machines	Compiler flags
Fujitsu A64FX	<code>-Ofast -mcpu=a64fx+sve</code>
AMD EPYC Zen2	<code>-Ofast -march=native</code>
Intel Xeon Platinum	<code>-Ofast -march=native</code>
AWS Graviton3	<code>-Ofast -mcpu=neoverse-v1</code>

Table VII presents the vectorization results. The hardware counters used to analyze vector instructions differ across machines, and these differences will be discussed during the explanation of each machine. We used the LIKWID performance tool with `perf` events to get the instruction count from each core. The hardware counters used in this work are presented in Table VI.

TABLE VI
HARDWARE COUNTERS USED IN THIS STUDY

Machine	Hardware counters
Intel Icelake	<code>fp_arith_inst_retired_128B_packed_double</code> <code>fp_arith_inst_retired_256B_packed_double</code> <code>fp_arith_inst_retired_512B_packed_double</code>
AMD Zen2	<code>retired_sse_avx_flops_add_sub</code> <code>retired_sse_avx_flops_mult</code> <code>retired_sse_avx_flops_fma</code>
AWS Graviton3	<code>sve_inst_spec</code>
Fujitsu A64FX	<code>simd_inst_retired</code> <code>sve_inst_retired</code>

On the Intel machine, we have counters to capture 128b, 256b, and 512b vector instruction. Table VII shows that for all benchmarks using the OpenMP backend, only 128b and 256b vector instructions are retired. In contrast, when using the OpenCL backend, 512b vector instructions are also executed. This observation holds for both AdaptiveCpp and DPC++ implementations.

On the AMD machine, we have counters to capture `add-sub`, `mult` and `fma` vector instructions, which correspond to addition-subtraction, multiplication, and fused-

TABLE VII
VECTORIZATION RESULTS FOR THE BENCHMARKS, DIFFERENT MACHINES HAVE DIFFERENT COUNTERS TO MEASURE THE VECTOR INSTRUCTIONS, DETAILED IN TABLE VI. HERE `ocl` CORRESPONDS TO THE OPENCL BACKEND. `X` CORRESPONDS TO NO VECTOR INSTRUCTIONS.

Benchmarks	Intel Icelake				AMD Zen2			A64FX		Graviton3	
	ACPP			DPC++	ACPP		DPC++	ACPP		ACPP	
	JIT		AoT	JIT	JIT		AoT	JIT	AoT	JIT	AoT
	omp	ocl	omp	ocl	omp	omp	ocl	omp	omp	omp	omp
mat-vec	128b; 256b	512b	128b; 256b	128b; 512b	add-sub; fma	add-sub; mult	fma	simd; sve	simd; sve	✗	sve
mat-mul	128b; 256b	512b	128b; 256b	128b; 512b	add-sub; fma	add-sub; mult	fma	simd; sve	simd; sve	✗	✗
outer-product	256b	128b; 512b	256b	128b; 512b	mult	mult	mult	simd; sve	simd; sve	✗	✗
cross-product	256b	128b; 512b	256b	128b; 512b	add-sub; mult; fma	add-sub; mult; fma	add-sub; mult; fma	✗	✗	✗	✗
sycl::reduction	✗										

TABLE VIII
KEY FINDINGS

	SYCL		
	Across Systems	Across Backends	With different Compilation flows
Memory bandwidth	On A64FX we achieve bandwidth closer to BabelStream benchmark.	OpenMP backend achieves higher bandwidth compared to OpenCL	AoT achieves higher bandwidth compared to JIT compilation flow.
Parallelization	Performance on x86 systems is better than AArch64 systems.	OpenCL performs slightly better than OpenMP	JIT compilation is better than AoT
Atomics	We observe that A64FX has the worst performance across all systems.	OpenMP performs slightly better than OpenCL	Negligible difference across different compilation flows
Reduction	A64FX and AMD systems have inferior performance compared to Intel and Graviton3		
Barriers	There is a negligible difference across the systems, backends and compilation flows		
Vectorization	x86 systems perform vectorization better than AArch64 systems.	OpenCL and OpenMP perform comparably concerning vectorization.	JIT can perform complex vector instructions, such as <code>fma</code> compared to AoT.
	<code>sycl::reduction</code> feature is not vectorized by any of the toolchains on all the systems		

multiply-add instructions, respectively. We observe that with `AdaptiveCpp`, AoT compilation prefers `mult` instructions compared to `fma` instructions. In contrast, JIT compilation, with both `AdaptiveCpp` and `DPC++` generally favors `fma` instructions over `mult` and `add-sub`.

On the A64FX system, we have counters to capture SIMD and SVE vector instructions. Generally, for both compilation flows, we observe the use of SIMD and SVE instructions. However, for the `cross-product` benchmark, no vector instructions are generated with either JIT or AoT compilation. This particular benchmark includes an `ifelse` control flow within the kernel, suggesting that the `AdaptiveCpp` compiler on A64FX may not be sophisticated enough to effectively compile kernels containing `ifelse` conditionals.

On the Graviton3 system, we have counters to capture SVE vector instructions. Generally, we observe that with the JIT compilation, none of the benchmarks are vectorized, while with the AoT compilation, only the `GEMV` and `outer-product` benchmarks are vectorized. The poor vectorization performance of `AdaptiveCpp` on Graviton3, compared to A64FX, is likely due to the absence of Graviton3-specific architecture flags and optimizations in LLVM Clang. Analyzing compiler reports on the Graviton3 machine we find out that `GEMM` kernel is not vectorized because the cost model suggests that vectorization is unprofitable. With `cross-product` kernel we observe that `AdaptiveCpp` is not able to vectorize `ifelse` conditionals, similar to A64FX.

With `sycl::reduction`, we observed that none of the SYCL implementations were able to vectorize this construct on any of the machines tested. In contrast, a similar OpenMP benchmark was successfully vectorized with a reduction construct.

Overall, we observe that both SYCL implementations effectively vectorize across work items on all systems, utilizing various dimensions, iteration spaces, and data management methods, except for the Graviton3 system when using JIT compilation with the OpenMP backend. Similarly, loops within work items are vectorized successfully across all systems, except Graviton3. However, the ability to vectorize conditionals within work items differs between x86 and AArch64 systems, with x86 machines demonstrating the capability to vectorize conditionals effectively.

VI. CONCLUSION

In this work, we have developed a set of micro-benchmarks to study the performance of various SYCL features on available toolchains and compilation flows of SYCL implementations. Furthermore, we analyze the performance overheads of these SYCL features over standard programming models, such as OpenMP. Our benchmarks test memory bandwidth, parallelization, synchronization, and vectorization. We evaluated two different SYCL implementations, `AdaptiveCpp` and `DPC++`, on four different platforms: Intel Icelake, AMD EPYC Zen2, Fujitsu A64FX, and AWS Graviton3.

Our experimental results demonstrate that different compilation flows offer advantages in distinct aspects of the SYCL programming model. Specifically, Ahead-Of-Time (AoT) compilation is more effective for achieving higher memory bandwidth, while Just-In-Time (JIT) compilation performs better in parallelization tasks. Comparing different backends, we observe similar trends. Across systems, SYCL on the A64FX processor exhibits generally poor performance, while other systems perform comparably to each other. The vectorization benchmarks reveal that auto-vectorization depends heavily

on the sophistication of the compiler and its cost model. Specifically, we observe that on the Graviton3 system, the compiler struggles with auto-vectorization, indicating a lack of optimization for vectorized instructions on this architecture.

For future work, we plan to integrate our benchmarks with SYCL-Bench to complement the existing benchmark suite. We also intend to conduct a more detailed study on vectorization, expanding on the initial findings in future research.

ACKNOWLEDGMENT

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under Framework Partnership Agreement No. 800928 and Specific Grant Agreement No. 101036168 (EPI SGA2). The JU receives support from the European Union's Horizon 2020 research and innovation programme, and from the Swedish Research Council, among others. Additionally, this work has received funding from the project PRIDE from the Swedish Foundation for Strategic Research with reference number CHI19-0048, and the project P4PIM from the Swedish Research Council (VR) with project ID 2020-04892. The computations were enabled by resources provided by Chalmers e-Commons at Chalmers. The computations were enabled by resources provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS), partially funded by the Swedish Research Council through grant agreement no. 2022-06725. The authors thank the Barcelona Supercomputing Center, Spain for providing access to Mare Nostrum 4 ARM-CTE partition, the University of Bristol, UK for providing access to Isambard 2; and RIKEN-RCCS, Japan for access to A64FX servers. We thank Jens Domke for interesting discussions and interpretations of our experimental data.

REFERENCES

- [1] Khronos inc. (2024). [Online]. Available: <https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html>
- [2] S. Matsuoka, "Fugaku and a64fx: the first exascale supercomputer and its innovative arm cpu," in *2021 Symposium on VLSI Circuits*, 2021, pp. 1–3.
- [3] Intel, Inc. Speed-up memory bandwidth bound workloads with hbm. [Online]. Available: <https://www.intel.com/content/www/us/en/high-performance-computing/hpc-high-bandwidth-memory-video.html>
- [4] Advanced Micro Devices, Inc. AMD 3D V-Cache™ Technology. [Online]. Available: <https://www.amd.com/en/products/processors/technologies/3d-v-cache.html>
- [5] Nec. Nec vector engine models. [Online]. Available: https://www.nec.com/en/global/solutions/hpc/sx/vector_engine.html
- [6] D. Ditzel. (2024) Invited talk: Why i chose to build esperanto's ai and hpc accelerator around risc-v. [Online]. Available: https://sc24.supercomputing.org/proceedings/workshops/workshop_pages/misc257.html
- [7] Intel. (2022) OneAPI. [Online]. Available: <https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html>
- [8] Aksar Alpay. (2024) AdaptiveCPP. [Online]. Available: <https://github.com/AdaptiveCpp/AdaptiveCpp.git>
- [9] I. Baratta, C. Richardson, and G. Wells, "Performance analysis of matrix-free conjugate gradient kernels using sycl," in *International Workshop on OpenCL*, ser. IWOCL'22. New York, NY, USA: Association for Computing Machinery, 2022.
- [10] A. Bagusetty, A. Panyala, G. Brown, and J. Kirk, "Towards cross-platform portability of coupled-cluster methods with perturbative triples using sycl," in *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2022, pp. 81–88.
- [11] K. Narasimhan, O. E. Farouki, M. Goli, M. Tanvir, S. Georgiev, and I. Ault, "Towards performance portability of ai graphs using sycl," in *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2022, pp. 111–122.
- [12] A. Alekseenko and S. Páll, "Comparing the performance of sycl runtimes for molecular dynamics applications," in *Proceedings of the 2023 International Workshop on OpenCL*, ser. IWOCL'23. New York, NY, USA: Association for Computing Machinery, 2023.
- [13] G. Daiß, P. Diehl, H. Kaiser, and D. Pflüger, "Stellar mergers with hpx-kokkos and sycl: Methods of using an asynchronous many-task runtime system with sycl," in *Proceedings of the 2023 International Workshop on OpenCL*, ser. IWOCL'23. New York, NY, USA: Association for Computing Machinery, 2023.
- [14] Z. Jin and J. S. Vetter, "Performance portability study of epistasis detection using sycl on nvidia gpu," in *Proceedings of the 13th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, ser. BCB'22. New York, NY, USA: Association for Computing Machinery, 2022.
- [15] D. Nasikan and V. Yaremenko, "Performance evaluation of lu matrix decomposition using the sycl standard," *Technology audit and production reserves*, vol. 3, no. 2(71), p. 6–9, Jun. 2023.
- [16] V. R. Pascuzzi and M. Goli, "Benchmarking a proof-of-concept performance portable sycl-based fast fourier transformation library," in *Proceedings of the 10th International Workshop on OpenCL*, ser. IWOCL'22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: <https://doi.org/10.1145/3529538.3529996>
- [17] Z. Jin, "The rodinia benchmark suite in sycl," 6 2020. [Online]. Available: <https://www.osti.gov/biblio/1631460>
- [18] 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 *Euro-Par 2020: 26th International European Conference on Parallel and Distributed Computing*, ser. Euro-Par'20. Springer International Publishing, 2020.
- [19] Z. Jin and J. S. Vetter, "A benchmark suite for improving performance portability of the sycl programming model," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023, pp. 325–327.
- [20] E. Georganas, R. F. Van Der Wijngaart, and T. G. Mattson, "Design and implementation of a parallel research kernel for assessing dynamic load-balancing capabilities," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 73–82.
- [21] L. Crisci, L. Carpentieri, P. Thoman, A. Alpay, V. Heuveline, and B. Cosenza, "Sycl-bench 2020: Benchmarking sycl 2020 on amd, intel, and nvidia gpus," in *Proceedings of the 12th International Workshop on OpenCL and SYCL*, ser. IWOCL'24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: <https://doi.org/10.1145/3648115.3648120>
- [22] Lal, Sohan and Alpay, Aksel and Salzmann, Philip and Cosenza, Biagio and Hirsch, Alexander and Stawinoga, Nicolai and Thoman, Peter and Fahringer, Thomas and Heuveline, Vincent. (2024) ParRes. [Online]. Available: <https://github.com/ParRes/Kernels.git>
- [23] M. Breyer, A. Van Craen, and D. Pflüger, "Evaluation of sycl's different data parallel kernels," in *Proceedings of the 12th International Workshop on OpenCL and SYCL*, ser. IWOCL'24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: <https://doi.org/10.1145/3648115.3648130>
- [24] I. Z. Reguly, "Evaluating the performance portability of sycl across cpus and gpus on bandwidth-bound applications," in *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W'23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1038–1047. [Online]. Available: <https://doi.org/10.1145/3624062.3624180>
- [25] S. Joubé, H. Grasland, D. Chamont, and E. Brunet, "Comparing sycl data transfer strategies for tracking use cases," *Journal of Physics: Conference Series*, vol. 2438, no. 1, p. 012018, feb 2023.
- [26] J. M. Bull, F. Reid, and N. McDonnell, "A microbenchmark suite for openmp tasks," in *OpenMP in a Heterogeneous World*, B. M. Chapman, F. Massaioli, M. S. Müller, and M. Rorro, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 271–274.
- [27] "Evaluating attainable memory bandwidth of parallel programming models via babelstream," vol. 17, no. 3, p. 247–262, jan 2018.

- [28] J. Ciesko and S. L. Olivier, "Characterizing the performance of task reductions in openmp 5.x implementations," in *OpenMP in a Modern World: From Multi-device Support to Meta Programming*, M. Klemm, B. R. de Supinski, J. Klinkenberg, and B. Neth, Eds. Cham: Springer International Publishing, 2022, pp. 35–49.
- [29] Fujitsu. (2024) Fujitsu compiler package. [Online]. Available: <https://software.fujitsu.com/jp/manual/manualindex/p22000026e.html>
- [30] Khronos inc. (2024) OpenCL Standard. [Online]. Available: https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_Env.html#validation-rules
- [31] A. Poenaru, T. Deakin, S. McIntosh-Smith, S. Hammond, and A. Younge, "An evaluation of the fujitsu a64fx for hpc applications," in *Cray User Group 2021*, May 2021, cray User Group 2021 ; Conference date: 03-05-2021 Through 05-05-2021. [Online]. Available: <https://cug.org/cug-2021/>
- [32] R. Jesus and M. Weiland, "A study on the performance implications of aarch64 atomics," in *High Performance Computing*, A. Bhatele, J. Hammond, M. Baboulin, and C. Kruse, Eds. Cham: Springer Nature Switzerland, 2023, pp. 279–296.