SimSYCL: A SYCL Implementation Targeting Development, Debugging, Simulation and Conformance

Peter Thoman
peter.thoman@uibk.ac.at
Department of Computer Science
University of Innsbruck
Innsbruck, Austria

Fabian Knorr
fabian.knorr@uibk.ac.at
Department of Computer Science
University of Innsbruck
Innsbruck, Austria

Luigi Crisci
lcrisci@unisa.it
Department of Computer Science
University of Salerno
Fisciano, Italy

ABSTRACT

The open SYCL standard has established itself as a cross-vendor, cross-platform means to develop software which benefits from GPU and accelerator parallelism. Inherent difficulties in portability between and debuggability of programs for these targets remain. However, as we demonstrate, the SYCL specification lends itself to be implemented purely in software in a manner that is accessible to debuggers and which can be employed to simulate the characteristics of different hardware targets.

We introduce SimSYCL, a novel library-only SYCL implementation with extensive simulation and verification capabilities. By executing all SYCL commands synchronously on the host CPU, it is able to diagnose various manifestations of undefined behavior within kernels, and grants developers the ability to step into kernels with an ordinary debugger to discover other logic errors.

We demonstrate that the reduced complexity of this approach, combined with an implementation focus on fast compilation, considerably speeds up the edit-compile-debug cycle compared to other SYCL implementations while maintaining reasonable runtime performance. Furthermore, we show how SimSYCL's simulation capabilities allow unit-testing user code for cross-platform portability, and that its comprehensive validation detects and reports several classes of user errors which remain undiagnosed by performance-focused implementations.

CCS CONCEPTS

• **Computing methodologies** → *Parallel programming languages*; *Massively parallel algorithms*; *Graphics processors*; • **General and reference** → *Performance.*

KEYWORDS

SYCL, simulation, development, debugging, GPU, HPC

ACM Reference Format:
Peter Thoman, Fabian Knorr, and Luigi Crisci. 2024. SimSYCL: A SYCL Implementation Targeting Development, Debugging, Simulation and Conformance. In *International Workshop on OpenCL and SYCL (IWOCL '24), April 08–11, 2024, Chicago, IL, USA*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3648115.3648136

IWOCL '24, April 08–11, 2024, Chicago, IL, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1790-1/24/04
https://doi.org/10.1145/3648115.3648136

1 INTRODUCTION

In the recent history of computing, high-performance hardware across various computational domains has increasingly tended towards heterogeneity and accelerators as a means of extracting the highest possible performance at a given power budget. This proliferation of hardware architectures lead to rising difficulties in programmability, and as a result an equally large proliferation of programming models [7, 30] were proposed in the research community.

Compared to high-performance hardware, large-scale software is long-lived, and application developers and institutions expect long-term support as well as a wealth of readily available and reliable development tools [34]. In accelerator computing, the only platform which has truly managed to reach mainstream adoption is Nvidia CUDA. Initially released in 2006, and with comprehensive resources widely available by 2011 [8], CUDA had a first-mover advantage, but also benefits from a broad and growing range of development tools [5, 26].

However, CUDA is limited to one hardware vendor, and focused on a particular type of accelerator hardware. The Khronos SYCL standard [29] provides a royalty-free, vendor-agnostic single-source C++ API for targeting various parallel hardware architectures. Originally envisioned as a layer which builds on the prior Khronos OpenCL standard [15], the SYCL 2020 specification enables SYCL implementations to directly target an even larger set of hardware [3].

As we will detail in section 1.1, the current SYCL ecosystem offers a large number of independent implementations, with at least two broadly supported production-quality options. However, the vast majority of engineering efforts within this ecosystem are currently directed at improving performance and broadening support for different target hardware. This is a natural direction given the overall goals of SYCL, however, other aspects of the software development life cycle also need to be addressed to further improve the utility, usability and appeal of SYCL.

To this end, we present *SimSYCL*, a SYCL implementation targeting development, debugging, simulation, verification and conformance use cases. SimSYCL favors strict compliance with the SYCL specification, comprehensive checking of prerequisites, simplicity in regards to debugging, compatibility with existing development tools, and fast compilation over runtime performance. It implements synchronous, sequential execution wherever possible, minimizes external dependencies, and enables the simulation of various hardware architectures in terms of their SYCL device specifications. Figure 1 provides an overview of how SimSYCL fits into SYCL application development as a debugging, testing and verification tool,

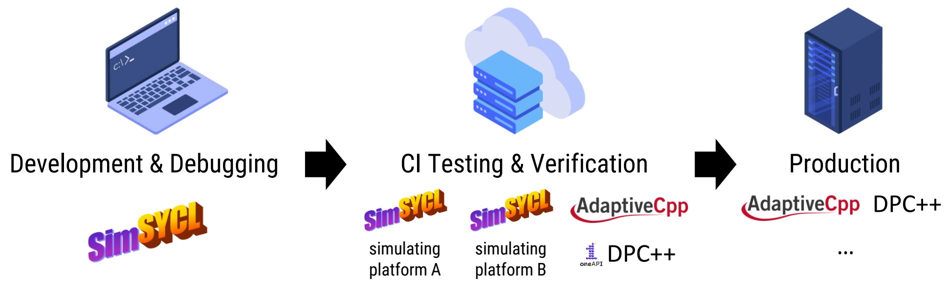


Figure 1: Overview of the intended use cases of SimSYCL in SYCL application development

while leaving actual production work to other implementations more suitable for this purpose.

The core contributions of this work include:

- A new SYCL implementation, SimSYCL, optimized for development aspects that are harder to accomodate in implementations designed to maximize execution speed.
- A detailed discussion of the use cases SimSYCL targets, how they informed the goals of its design, and how the current implementation achieves these goals.
- An evaluation of the debugging and verification capabilities, the compile-time performance and the execution speed of SimSYCL compared to existing SYCL implementations.

The remainder of this paper is structured as follows. Sections 1.1 and 1.2 provide more detailed background information on the current SYCL ecosystem and how SimSYCL complements it, while section 1.3 embeds this research into the broader field of related work. Section 2 outlines the design and intended use cases for SimSYCL, distilling a set of goals from this analysis. The most relevant core implementation aspects are discussed in section 3. Section 4 provides an evaluation of various aspects of SimSYCL: its ability to aid debugging and verification, its simulation capabilities, and compile-time as well as runtime performance. The current state of and future directions for SimSYCL are discussed in section 5.

1.1 The Current SYCL Ecosystem

As of the end of 2023, there are two major, widely-used SYCL implementations which both support a variety of hardware: Data-Parallel C++ and AdaptiveCpp.

Data-Parallel C++ (DPC++) [4] is an open source implementation included in Intel's OneAPI initiative which is built on top of the LLVM/Clang compilation infrastructure [18], with an intention to upstream SYCL support to mainline Clang in the future. It fully supports CPUs and Intel GPUs, with more experimental backends for Nvidia CUDA and AMD ROCm.

AdaptiveCpp [3] (ACpp, previously known as hipSYCL and Open SYCL) is an implementation with a background in academia. It features multiple compilation flows targeting arbitrary CPUs with OpenMP, Nvidia and AMD GPUs each with respective vendor-specific backends, and a generic single-pass LLVM-based compilation flow with a JIT compilation phase for device code.

In addition to these two implementations with broad hardware support, Codeplay's ComputeCpp played a significant role over the history of SYCL, however, its development was discontinued in 2023 [22], with Codeplay focusing on contributing to the open-source DPC++ implementation.

Beyond these mainstream options, there is a large variety of experimental SYCL implementations with either an academic research or vendor-specific background. NeoSYCL [14] is an implementation designed for the NEC SX-Aurora TSUBASA supercomputer. TriSYCL [10] provides an experimental implementation of SYCL for FPGA hardware. The Sylkan [31] proof-of-concept prototype demonstrated that SYCL programs can be compiled for hardware supporting Vulkan [27] compute shaders. In a similar vein, the SYCLops [25] converter translates SYCL-specific LLVM IR to MLIR.

Some related projects are not directly SYCL implementations, but aim to extend the ecosystem in similar ways. The Celerity [23] runtime system provides an API very close to SYCL, but extends its applicability to clusters of accelerators with automatic work and data distribution. Other projects seek to expand the capabilities of existing implementations to support additional hardware, such as the Huawei Ascend AI chipset [9].

1.2 The Case for SimSYCL as a Dedicated SYCL Implementation

As we outlined above, there is a healthy and growing ecosystem of SYCL implementations. However, they are largely targeted at achieving high performance, supporting specific classes of hardware, or enabling experimentation towards these goals. While these are of primary importance for a technology aimed squarely at effectively leveraging modern parallel high-performance hardware, these goals also frequently conflict with other use cases, and specifically with some of the goals of SimSYCL.

For example, supporting a variety of hardware will generally introduce a larger set of dependencies, reducing portability and increasing the barrier to entry. Similarly, aiming to achieve optimal performance enforces design decisions which increase the complexity of the code base, increasing compile times and potentially making debugging more difficult. As such, while it is tempting to believe that all the features of SimSYCL could be integrated into one of the existing implementations with less overall effort, the outcome would be very unlikely to match a dedicated implementation in terms of ease-of-access, portability, and overall utility for development use cases.

By dropping all requirements associated with supporting exotic hardware or achieving maximum production performance, SimSYCL can not only enable portable and low-barrier-of-entry development and debugging with quick iteration cycles, but the implementation can also be focused towards other use cases that are critical when building, maintaining and testing SYCL software. These include rigorous verification, as well as the ability to simulate various target hardware platforms and their properties – such as local memory limits or sub-group sizes – without physically accessing them. This ability is particularly useful for enabling the continuous integration testing of SYCL applications and frameworks built on top of SYCL without setting up complex dedicated testing infrastructure featuring various target hardware devices.

Finally, due to the relative simplicity of its library-only implementation, and its very strict adherence to the specification, we hope that SimSYCL can serve as an "executable specification": if future proposed additions to the SYCL API are first implemented in SimSYCL – potentially in addition to some implementation proposed by e.g. an independent hardware vendor – then this ensures that they can be implemented in a library-only framework, and gives a baseline indication of the effort involved at the interface level. In this way, the standardization process could also move closer towards the model adopted by ISO C++, preferring proposals for which implementation experience has been demonstrated within multiple independent frameworks and platforms.

1.3 Further Related Work

While the current SYCL ecosystem forms the primary backdrop of SimSYCL, some additional, relevant related work exists in the broader spectrum of development, simulation and debugging tools for accelerator hardware.

There is a rich history of powerful GPU simulation frameworks which can simulate various hardware configurations [20, 33]. These types of simulators are generally designed to provide very accurate, low-level simulation in order to study performance and energy characteristics [6]. Conversely, SimSYCL is an API-level implementation of SYCL designed to aid development, testing and functional verification of SYCL programs, and has no ambitions to simulate the *performance* characteristics of specific hardware. This allows for minimal compile-time and execution overhead compared to more heavy-weight low-level hardware simulation.

Focusing specifically on debugging, there has been substantial work in allowing more convenient access and insight into device-side code in various programming frameworks [11], including for the OpenCL execution model [21] and, specifically, SYCL [2]. Knobloch and Mohr provide a comprehensive overview on debugging and analysis tools in their survey [16].

On the hardware architecture level, research has looked into reducing or eliminating non-determinism in GPUs in order to improve debuggability and correctness testing [13]. Unlike these approaches, which seek to improve the debugging experience directly on hardware devices, SimSYCL aims to replicate the device environment as much as possible within a standard host program, allowing developers to leverage a wealth of existing debugging tools.

Recently, Patel et al. presented a virtual GPU device designed specifically for OpenMP offloading [19]. In terms of goals and use cases, this is quite similar to SimSYCL, and the authors point out several advantages of this approach which match our own motivations. However, due to the adherence of the SYCL spec to the C++ standard, SimSYCL can be provided as a library-only solution, and as such the practical design and implementation is fundamentally different from an OpenMP compilation environment.

2 DESIGN AND USE CASES

This section outlines the primary intended use cases for SimSYCL, and explains how they influenced its design and feature set. From each use case, we distill a small set of core design aspects or features which the implementation of SimSYCL needs to accommodate.

2.1 Use Case 1: Development

Improving the overall development experience for programmers using SYCL in their applications is of the utmost importance in order to ensure its sustained growth and viability as a platform.

A core aspect of any platform or programming language which influences developer productivity and satisfaction is *build latency*, i.e. the amount of time it takes to go from implementing a change to running a test or debugging the resulting application. Software engineering research indicates that any incremental improvement to build latency increases the likelihood for developers to be able to stay focused on a given task [12], in turn affecting overall productivity and satisfaction with their development environment.

In this context, the heavy reliance of SYCL on C++ templates presents some difficulty, as common strategies for reducing build latency such as moving the majority of the implementation to separate translation units can not be applied universally. Additionally, the requirement of potentially compiling for multiple target architectures leads to some implementations adopting expensive multi-pass compilation schemes. These factors compound, and previous work has show that various SYCL compilation infrastructures suffer from relatively long build latencies [32].

For SimSYCL, this motivates the decision to accept the necessity for small increases in complexity or layering of the design if they allow the implementation to remove larger dependencies from the public header surface. Constraining the use of such dependencies to smaller, separate translation units means they will not affect the compile times of projects using SimSYCL.

Another highly important feature for ease of development is *portability*. When evaluating and designing for this aspect, we need not only consider the baseline fact that a given implementation can in principle work on a given target platform, but also the ease with which it can be accessed. For SimSYCL, this aspect provides a motivation to keep external dependencies as limited as possible, and maintain portability to all major operating systems used in development, and all widely-used C++ compilers: Clang and GCC on Linux and macOS, as well as MSVC on Windows. Existing SYCL implementations often lack full support across all of these operating systems and compilers, as they are rarely used in production when targeting actual accelerators.

In summary, to maximize the utility of SimSYCL as a development platform, the design needs to focus on minimizing *build latency* and enabling broad and barrier-free *portability*. We will evaluate the success of the current SimSYCL implementation in regards to these goals in section 4.4.

2.2 Use Case 2: Debugging

The *debugging* of parallel applications and frameworks is widely recognized as highly challenging, and successful platforms provide a wealth of tools to support this task. As SimSYCL does not focus on achieving high performance, it already has some inherent advantages for the debugging use case: the implementation can omit any complexity which is not necessary to provide a conforming execution. The resulting relative *simplicity* makes it easier for application developers to understand the mechanics of the API where necessary.

At the same time, a fundamental goal of SimSYCL is maintaining compatibility with existing debugging and analysis tools. This is accomplished by providing a *library-only* implementation, which enables compatibility with common debugging infrastructure across platforms. As one example, SimSYCL allows developers to easily step into and out of kernel functions using standard debuggers. Furthermore, SimSYCL directly enables low-friction access to address sanitizers [24] as a build-time option, which is always enabled in its own unit tests. This generally allows identifying any memory-related errors much more rapidly.

Finally, the verification features outlined in section 2.4 can also aid in debugging, in case errors are introduced in an application by a non-conforming use of the SYCL API which may not be checked for or sufficiently reported by other SYCL implementations.

We will evaluate how these SimSYCL features support debugging SYCL applications in section 4.2.

2.3 Use Case 3: Simulation

SYCL is explicitly designed to support a wide variety of target hardware with varying capabilities, features, and execution behavior. In fact, the *info::device* structure prescribed by the SYCL 2020 specification [29] enumerates more than 70 individual aspects and capabilities of a SYCL device which can vary across target hardware. As such, in order to test the compatibility of a given application across a large set of potential target platforms, a complex testing setup and access to all targeted hardware is necessary.

As SimSYCL focuses on *simulation* rather than high-performance execution, the relationship between the platform and the application developer or their testing infrastructure can be inverted: rather than the SYCL implementation informing the application of a given hardware device's features and capabilities, SimSYCL can be provided with a hardware profile which instructs it to perform the program execution *as-if* a given target platform was used.

This enables testing various aspects of platform compatibility without access to any specialized hardware. Combined with a design focused on broad operating system and compiler support and few external dependencies as outlined in section 2.1, this also enables much more straightforward integration of SimSYCL within the pipelines of a standard e.g. cloud-based CI infrastructure.

We explore some of the hardware and platform aspects which SimSYCL can simulate, and how this supports cross-platform compatibility testing, in section 4.3.

2.4 Use Case 4: Verification

For many of the operations possible in SYCL, the SYCL 2020 standard specifies an exhaustive set of constraints and preconditions. Some of these are relatively easy to verify and check, while others require tracking of additional information which might incur a compile-time and/or runtime execution time and memory usage penalty. Therefore, in most cases, SYCL implementations are not required to report if any of these preconditions are violated, and may instead fail silently.

SimSYCL, on the other hand, aims to provide full *verification* of all of these constraints and preconditions. Wherever possible, this verification occurs at compile time, as detailed in section 3.2. When compile-time verification is not viable, either due to the standard requiring specific template type definitions, or due to runtime behavior needing to be checked, SimSYCL will track all the information necessary to perform a complete validation at runtime.

In addition to checking the validity of API calls in terms of their constraints and preconditions, SimSYCL also provides a different kind of verification. Its design and implementation takes special care *not* to offer functionality in the SYCL namespace or within the public interface of any SYCL class which is not specified by the SYCL standard. Common SYCL implementations frequently offer additional features that are not currently standardized. While larger and more involved features are generally marked as experimental or non-standard, there can also be smaller additions such as some extra member functions which are not. These are often useful and can therefore easily slip into a codebase, but they reduce portability of code which makes use of them.

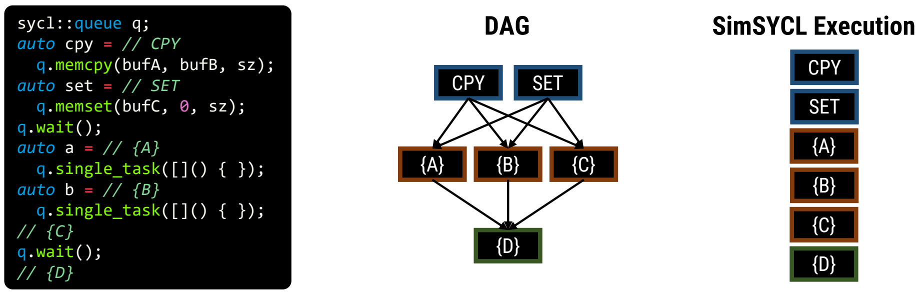
As the goal of SimSYCL is for any program that works in it to also work in any other standards-compliant SYCL implementation, it aims to only include specifically the functionality required by the standard. As such, compiling with SimSYCL can serve as a simple test to verify that no implementation-specific functionality was inadvertently used.

2.5 Use Case 5: Executable Specification

This close adherence to the standard also enables the fifth and final use case we envision for SimSYCL: serving as an "executable specification" of the SYCL standard. While the standard includes the declaration of the SYCL interface and some example code, it is not actually fully formalized and frequently relies on prose to describe the intended behavior.

On the other hand, existing implementations are of course fully specified as working code, but, due to their different and complex goals regarding performance and hardware support (as outlined in section 1.2), they introduce a lot of complexity that is not inherent to the specification. As such, they are less suitable for serving as a relatively easily understood baseline code version of the standard.

Conversely, SimSYCL aims to be the simplest possible conformant SYCL implementation. It is designed to only introduce complexity when it is in service of debugging, verification or simulation, and can therefore serve as a vehicle for exploration and prototyping of API changes, at a level more involved and formalized than a prose document, but less complex than an implementation in one of the full SYCL platforms aimed at production-quality performance and accelerator hardware support.



```
sycl::queue q;
auto cpy = // CPY
  q.memcpy(bufA, bufB, sz);
auto set = // SET
  q.memset(bufC, 0, sz);
q.wait();
auto a = // {A}
  q.single_task([]() { });
auto b = // {B}
  q.single_task([]() { });
// {C}
q.wait();
// {D}
```

Figure 2: SimSYCL enforces synchronous, in-order execution for queue operations

3 IMPLEMENTATION

In this section we will briefly outline some of the more fundamental implementation choices behind SimSYCL, and how they serve to further the important goals of the design which result from the use cases identified in section 2.

SimSYCL is an open source project available on GitHub[1] under the MIT license. Its continuous integration testing verifies compatibility across three compilers (GCC, Clang, and MSVC) and operating systems (Linux, Windows, and macOS).

3.1 Synchronous Execution

A core principle of SimSYCL which is fundamentally distinct from all SYCL implementations which aim at runtime performance is *synchronous execution* of commands. SYCL queues, by default, provide an inherently asynchronous and potentially parallel API, generally requiring implementations to build and maintain an execution graph (DAG) at runtime which encodes and enforces the dependency relationship between commands. However, asynchronicity is not required to conform with the vast majority of the SYCL specification, outside of some edge cases which we outline in section 5.1.

In SimSYCL, all commands execute in program order. The SYCL API does not provide the means to construct backward dependency edges between commands, so this is guaranteed to implicitly fulfill all buffer- and event dependencies between command groups. The resulting simplicity aids in debugging and development, as all errors and exceptions can be reported synchronously, and makes reasoning over the execution more feasible. This approach also enables more seamless integration with development tools that are ill-equipped to deal with asynchronous parallel code.

However, the choice of synchronous execution is not without its drawbacks: as a result, while SimSYCL offers simulation tools to verify kernel execution across distinct parallel configurations and execution orders, the same is not possible between commands that would be concurrent in a DAG-based implementation.

Figure 2 illustrates the effect of this implementation decision. The SYCL code excerpt depicted on the left implies the DAG shown in the center of the image. As the `in_order` property is not set on q, any execution order of the set of boxes on each line is correct, e.g. SET → CPY → B → C → A → D would be a valid execution strategy. However, in SimSYCL the execution will always be fixed and synchronous, as per the sequence shown to the right of fig. 2.

[1]https://github.com/celerity/SimSYCL

3.2 Compile-time Validation

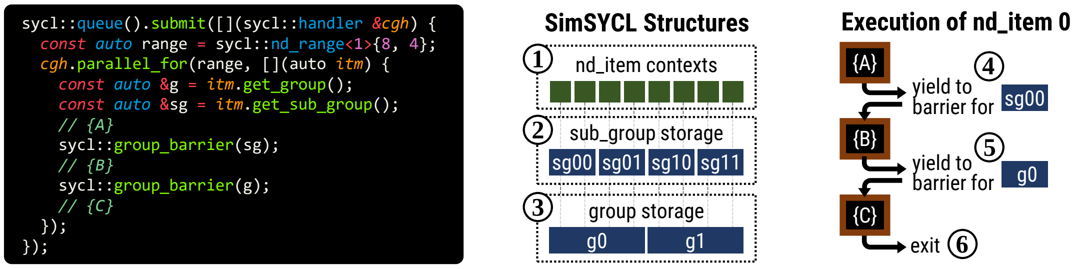
As outlined in section 2.4, SimSYCL aims to verify prerequisites and conditions specified by the standard at compile time, wherever this is possible in a library-only setting without any changes to the types and their public interface. At the implementation level, a concern with this is how it interacts with the goal of low build latency. C++17-style compile-time verification relies on template metaprogramming [1], which often necessitates that the compiler constructs and instantiates a large number of structures unrelated to the actual execution.

As such, we chose to leverage C++20 *concepts* wherever possible to formalize and implement constraints which are expressed as prose in the current version of the SYCL standard.

```
1  template<typename T>
2  concept SyclFloat =
3         std::is_same_v<T, float>
4      || std::is_same_v<T, double>
5      || std::is_same_v<T, sycl::half>;
6
7  template<typename T>
8  concept GenFloat =
9      SyclFloat<T> ||
10     (    (Swizzle<T> || Vec<T> || MArray<T>)
11       && SyclFloat<typename T::element_type> );
12
13 template<GenFloat T1, GenFloat T2>
14     requires( std::same_as<T1,T2>
15           || MatchingVec<T1,T2> )
16 auto max(T1 x, T2 y) { ... }
```

Listing 1: Basic SimSYCL concepts for math operations

Listing 1 provides an example of this approach. It contains the SimSYCL specification for `GenFloat` on lines 7–11, describing the *generic floating point* type as used in the SYCL standard section 4.17.8. It depends on some other previously-defined concepts which are omitted for brevity, but illustrates how concepts allow formulating these constraints in a succinct, readable and precise fashion. Based on such concepts, the constraints on mathematical operations can be easily expressed; an example for the `max` function is provided on lines 13–16.



```
sycl::queue().submit([](sycl::handler &cgh) {
  const auto range = sycl::nd_range<1>{8, 4};
  cgh.parallel_for(range, [](auto itm) {
    const auto &g = itm.get_group();
    const auto &sg = itm.get_sub_group();
    // {A}
    sycl::group_barrier(sg);
    // {B}
    sycl::group_barrier(g);
    // {C}
  });
});
```

Figure 3: Interleaved execution of nd_items in SimSYCL work groups

3.3 Interleaved Work Item Execution in Groups

While most potential queuing-related asynchronicity can be serialized without losing core SYCL features or functionality, the same does not apply to group- and sub-group-level parallelism. Using any *group operation* defined in the SYCL standard outside of trivial cases will lead to a kernel that can only complete successfully if the execution of individual work items is interleaved.

Consider `sycl::group_barrier` as arguably the simplest group operation: An implementation must guarantee that all work items of the participating group execute until, but not beyond, the barrier. Once all work items have done so, each must be resumed right after the barrier with their local variables and stack intact.

This is a source of complexity which cannot be avoided by SimSYCL whenever work groups are involved, in particular in ND-range kernels. To keep all execution confined to a single operating system thread, SimSYCL will create a separate execution *context* for each work item, which maintains a copy of the processor's register set together with a separate stack to provide cooperative task switching in user-space.

The *scheduler loop*, which is triggered as part of an ND-range parallel-for invocation, will repeatedly yield control to the context of each work item, either in round-robin or randomized order. Any work item that arrives at a barrier or other construct will yield control back to the context of the scheduler loop in order to then resume the next eligible item.

The following operations must yield control:

- The group functions: `group_broadcast` and `group_barrier`
- All group algorithms, including all variations of `any`, `all`, `none`, `shift`, `permute`, `select`, `reduce` and `scan`.
- Atomic operations on data stored in global or local memory, as it is feasible for applications to rely on the independent forward progress of work items to build their own synchronization mechanisms within a work group.

Note that while C++20 does provide stackless coroutines as a core language feature, these cannot be used to implement context switching in SimSYCL, as this would require the user to `co_await` every call to a SYCL function that potentially yields.

Figure 3 illustrates the internal data structure and execution principle for work items in a `parallel_for` over an `nd_range` on a simulated device configured with a subgroup size of 2. As per SimSYCL's synchronous execution principle, the command group is executed immediately. Before launching any kernel code, it constructs both the surface-level SYCL API structures such as `sycl::nd_item`s and `sycl::group`s as well as a set of required internal data structures:

① a context associated with each `nd_item` that allows the scheduler loop to resume any individual work item after it has yielded control
② one storage object for each active sub-group, maintaining state required to implement sub-group operations; this object is internally referenced by each `sycl::sub_group` instance received by the user
③ an equivalent storage object associated with each active `sycl::group`

The execution of individual work items is illustrated in fig. 3 on the right. For each item, the code in `// {A}` is executed, after which the item will yield control in order to allow all other items in the sub-group to arrive at the barrier ④. After the sub-group barrier has completed, the scheduler loop will at some point in time return control to the same work item, which will execute any code in `// {B}` before yielding once again in the group barrier ⑤. After being resumed once more, `// {C}` runs and the work item exits by yielding control one final time ⑥. At this point, the internal context information of this work item will indicate that it has completed, and it will no longer be scheduled.

3.3.1 *Group Operations.*

Group and sub-group operations and algorithms are all implemented in this interleaved execution framework based on a uniform structure designed to allow rigorously verifying all aspects of the user program's group interactions. For each work item, the associated context tracks the number of group and sub-group operations this item has encountered. Other than referring to the correct one of these two counters, and using their associated storage object, group and sub-group operations act equivalently, so for the remainder of this description we will simply refer to both as *groups*.

Each group storage object maintains a list of all group operations and algorithms which were performed on it, with each entry tracking the operation type, its participants, and optionally any per-operation information required for verifying its input and performing it. When a work item encounters a group operation, SimSYCL queries the associated counter in the work item context, and the list in the group's storage. If the operation is newly encountered, all required information is initialized and it is added to the list before the work item yields.

On the other hand, if an operation with this consecutive ID already exists, SimSYCL verifies that the newly encountered operation on the current work item matches the existing group record. If not, a detailed error is reported. Otherwise, any necessary per-item information is added to the group structure before the item yields. When execution returns to a given item, it checks whether all participants have reached the current operation based on information in the group storage. If not, it yields, otherwise it performs any required operation and returns control to the user code.

By meticulously tracking the sequence of group operations and all associated information provided by the user program, SimSYCL is capable of detecting and reporting several classes of related errors, as evaluated in section 4.2.

3.4 External Dependencies

As discussed in section 2.1, barrier-free and low-friction portability and deployment are crucial for SimSYCL. As such, the implementation seeks to avoid external dependencies as much as possible. Currently, SimSYCL only requires a standards-compliant C++20 compiler and the *boost.context* [2] library to be available on the target system. Libraries for dealing with JSON [3] and environment variables [4] in the context of simulated system definition are automatically fetched on build. *Catch2* [5] is used in SimSYCLs own unit tests, but not required for deployment.

It is tempting to try to eliminate the boost dependency, but this goal has to be weighed against the substantial complexity and maintenance overhead of adding a portable and fully-featured cross-platform cooperative multitasking implementation to SimSYCL. As outlined in section 3.3, interleaved code execution is necessary in order for SimSYCL to be able to provide meaningful `nd_range` parallelism, simulation and debugging support.

4 EVALUATION

4.1 Experimental Setup

While evaluation of the debugging, verification and simulation capabilities of SimSYCL is largely hardware-independent, the results for compilation and execution times in the latter parts of this section do depend on the system configuration. Table 1 lists the most crucial hardware information and versions of the relevant software components of our testing platform. Note that no GPUs or accelerators are used.

For all timing results (including compilation and benchmark times) each experiment was repeated 5 times and the median value is reported.

4.2 Debugging and Verification

In this section, we evaluate the debugging and verification capabilities of SimSYCL, and in particular compare the handling of user code that is in violation of prerequisites set forth in the SYCL specification to that of other implementations.

[2] https://www.boost.org/doc/libs/1_84_0/libs/context
[3] https://github.com/nlohmann/json
[4] https://github.com/ph3at/libenvpp
[5] https://github.com/catchorg/Catch2

Table 1: Hardware / Software stack for evaluation

Operating System	Ubuntu 22.04.2 LTS
C++ Compiler	Clang version 17.0.6
SYCL implementations	SimSYCL git revision aa0762efcb
	AdaptiveCpp git revision 3952b468c9
	DPC++ git revision 25c3666dff
CPU	2x AMD EPYC 7763 64-Core
System Memory	16x Micron 3200 MT/s DDR4 64 GB
Mainboard	Supermicro H12DSG-O-CPU

4.2.1 Group Operations. Group functions and algorithms as defined by the SYCL specification feature a large set of constraints and prerequisites for their correct use, violation of any of which generally results in undefined behavior. This is necessary to allow high-performance implementations of these functions which are often crucial to kernel performance, but also introduces significant potential for undetected user errors.

A general requirement across all of these operations is that they must be encountered by all work items in a given group, in the same order.

```
sycl::queue q;
q.submit([](sycl::handler& cgh) {
    cgh.parallel_for(sycl::nd_range<1>(2,2),
                     [=](sycl::nd_item<1> item) {
        auto id = item.get_global_id(0);
        if(id == 0) {
            sycl::group_barrier(item.get_group());
        }
    });
});
```

Listing 2: Minimal group divergence code sample

Listing 2 contains a minimal code sample of divergence in terms of group operations encountered by items within a work group. In a group of two work items, only item 0 executes the `group_barrier`, while item 1 exits the kernel.

For this input program, SimSYCL produces the following output:

```
SimSYCL check failed: id_equivalent
  at simsycl/group_operation_impl.cc:37:5
group operation id mismatch:
  group recorded operation "barrier",
  but work item 1 is trying to perform "exit"
```

The exact behavior depends on how the SimSYCL debugging and verification capabilities are configured. The default is to output error reporting information as above and immediately abort. Alternatively, SimSYCL can be configured (via `SIMSYCL_CHECK_MODE`) to throw an exception, or simply log the error and try to continue executing regardless.

For the simple test case in listing 2, the behavior of other SYCL implementation varies as expected in case of undefined behavior.

The library-only ACpp backend stalls indefinitely, while on the DPC++ CPU backend the program simply exits with error code 0.

```
sycl::queue q;
q.submit([](sycl::handler& cgh) {
    cgh.parallel_for(sycl::nd_range<1>(2,2),
                     [=](sycl::nd_item<1> item) {
        auto sg = item.get_sub_group();
        auto delta = item.get_local_id(0) + 1;
        auto r = sycl::shift_group_left(sg, 0, delta);
    });
});
```

Listing 3: Code sample performing an invalid group shift

In addition to general aspects required for the validity of all group operations, individual operations also specifiy additional prerequisites, all of which SimSYCL validates. We have chosen the `shift_group_left` operation as a representative in listing 3. For this code sample, both ACpp and DPC++ execute the program without complaint, while SimSYCL produces the following output:

```
SimSYCL check failed: per_op.delta == delta
  at simsycl/sycl/group_algorithms.hh:158:69
group shift delta mismatch:
  other group items specified delta 1,
  but work item 1 is trying to specify 2
```

We believe that this type of verification can be very helpful during development, but it also requires a level of bookkeeping which can and should not be expected from performance-oriented implementations.

4.2.2 *Compile-time Validation.*

SimSYCL leverages C++20 concepts to perform extensive compile-time validation, as outlined in section 3.2. As a consequence, there are several errors which are reported during the compilation process with SimSYCL which other implementations may not catch. In terms of API surface, vector swizzles are one of the more complex aspects of the SYCL specification to fully support.

```
sycl::vec<double, 4> v;
v.xyww() = sycl::vec<double, 4>(42.0);
```

Listing 4: Invalid swizzle assignment

Listing 4 serves as an example of this behavior. In this code snippet, an assignment is performed on the swizzle `xyww` of a 4-component vector. This is invalid, as the specification states "that a `__swizzled_vec__` that is used in an l-value expression may not contain any repeated element indexes".

Attempting to compile this code snippet with SimSYCL results in the following (abbreviated) compiler output:

```
test.cpp:6:12: error: no viable overloaded '='
    6 |   v.xyww() = vec<double, 4>(42.0);
      |   ~~~~~~~~ ^ ~~~~~~~~~~~~~~~~~~~~
...
simsycl/sycl/vec.hh:232:19: note:
   candidate function not viable: constraints not satisfied
  232 |     swizzled_vec &operator=(const value_type &rhs)
      |                   ^
simsycl/sycl/vec.hh:233:18: note:
      because 'allow_assign' evaluated to false
  233 |         requires(allow_assign)
      |                  ^
```

Both DPC++ and ACpp compile this code without any errors or warnings being issued.

4.2.3 *Additional Automatic Validation.*

Due to space constraints, we cannot detail all types of automatic validation SimSYCL performs in this paper. Some further examples include verifying the set of allowed operations in hierarchical parallelism, requirement validation on memory operations, and accessor boundary checks.

4.2.4 *Interactive Debugging.*

Since execution in SimSYCL happens synchronously on the CPU, all user code can be inspected using an interactive debugger such as GDB. This is true even for kernel code, although execution flow within ND-range kernels can be hard to follow as SimSYCL will perform a context switch between work items at every yield point (see section 3.3) and the stack outside the kernel invocation will be invisible to the debugger.

4.3 Simulation

SimSYCL allows application developers and users to describe the simulated system either through the `simsycl::configure_system()` API or by passing the system configuration as a JSON file via the `SIMSYCL_SYSTEM` environment variable at runtime. A system configuration consists of a set of platforms and set of devices, both of which are defined in terms of all their `sycl::info::platform` and `sycl::info::device` properties. An excerpt on such a configuration in JSON format is shown in listing 5.

4.3.1 *Configurable Properties.*

All properties specified are visible when querying them through `sycl::platform::get_info` and `sycl::device::get_info`, and thus naturally influence SYCL device selectors. Some have additional effects on the runtime behavior of SimSYCL:

- USM allocation functions will verify that the total allocated size does not exhaust the global memory of the device.
- A kernel launch will check that the device is capable of launching with the specified global and local ranges and that the total amount of required local memory is within bounds.
- The number of *compute units* per device decides how many work groups will be interleaved when executing ND-range kernels.
- The size of sub-groups is chosen in accordance with the first sub-group size configured in the target device.

The last point in particular allows developers to verify the correctness of their application across vendors, as e.g. Nvidia GPUs have a fixed sub-group size of 32, while AMD cards have 32 or 64 work-items per sub-group depending on the hardware generation.

4.3.2 *Detecting non-portability.*

Listing 6 shows a non-portable kernel that uses sub-group cooperative functions to efficiently compact

```json
{
    "devices": {
        "SimSYCL RTX 2070 Super": {
            "device_type": "gpu",
            "max_work_item_sizes<1>": [1024],
            "max_work_item_sizes<2>": [1024, 1024],
            "max_work_item_sizes<3>": [64, 1024, 1024],
            "local_mem_size": 65536,
            "name": "SimSYCL virtual GPU",
            "global_mem_size": 8589934592,
            "sub_group_sizes": [32],
            "vendor": "SimSYCL CUDA",
            "version": "0.1"
        }
    },
    "platforms": {
        "SimSYCL CUDA": {
            "name": "CUDA",
            "vendor": "SimSYCL"
        }
    }
}
```

Listing 5: JSON specification for simulating a system with a single Nvidia RTX 2070 Super GPU (excerpt)

a vector of 32-bit integers by stripping zero-words and indicating their position in a bit map. The kernel implicitly assumes that the sub-group size matches the bit-width of the words, which is only true for a sub-group size of 32.

With SimSYCL, the application developer is able to detect the non-portability of this code by unit-testing its output against the known-correct result while switching between simulated device configurations of different sub-group size, and *without* requiring access to physical hardware with the evaluated properties.

4.4 Compile-time Performance & Build Latency

As outlined in section 2.1, fast compile times and thus low build latency are an important factor in developer productivity, and SimSYCL is specifically designed to reduce compilation times as much as possible within the framework required by the SYCL specification. In this section, we evaluate the effectiveness of this approach by comparing the build latency of SYCL projects across implementations and configurations.

For each project, the evaluation is performed by configuring CMake[6] to use the Ninja build system[7], and either a "Release" or "Debug" configuration. Additionally, the *mold* linker[8] is used to minimize linking times. The reported results are the median of 5 overall wall-clock time measurements of `ninja`, with a `ninja clean` between each. Despite this representing quite a complex internal

```cpp
sycl::queue().parallel_for(
    sycl::nd_range<1>(32 * num_blocks, 32 * num_blocks),
    [in, out](sycl::nd_item<1> i)
{
    sycl::sub_group sg = item.get_sub_group();
    uint32_t word = in[i.get_global_linear_id()];
    uint32_t non_zero_mask = uint32_t(word != 0)
        << (31 - sg.get_local_linear_id());
    uint32_t header = sycl::reduce_over_group(
        sg, non_zero_mask, sycl::bit_or<>());

    uint32_t out_words = sg.leader() + (word != 0);
    uint32_t out_pos = sycl::exclusive_scan_over_group(
        i.get_group(), out_words, sycl::plus<>());
    if(sg.leader()) {
        out[out_pos] = header;
        out[out_pos + 1] = word;
    } else {
        out[out_pos] = word;
    }
});
```

Listing 6: Non-portable kernel with implicit assumptions about sub-group size

workflow with significant I/O and process-level parallelism, the overall times are highly consistent.

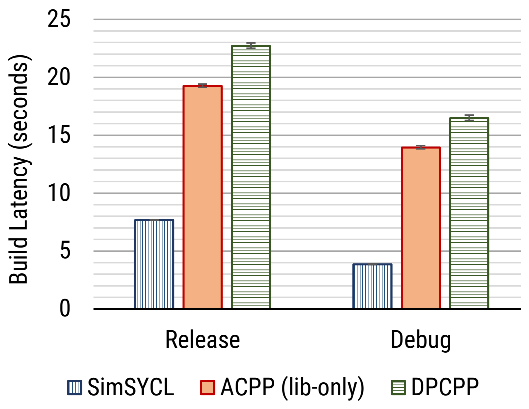


Figure 4: Build latency for SYCL-Bench

Figure 4 shows these total build times for the SYCL-Bench benchmark suite [17]. As generally expected, all release-mode builds take longer to complete than the respective debug-mode builds as the compiler performs more involved analysis and optimization. From the SimSYCL productivity-focused perspective, we consider debug-mode results more significant, as they represent the typical build cycle during development and debugging.

On our test system, in debug mode, the entirety of SYCL-Bench builds in under 4 seconds with SimSYCL. ACpp in library-only mode takes roughly 14 seconds, and DPC++ is slightly slower still. Note that for both alternative implementations we chose the fastest

[6]https://cmake.org
[7]https://ninja-build.org
[8]https://github.com/rui314/mold

available compilation mode: for ACpp, this is "OMP library-only" among the available compilation flows, and for DPC++ this means that only a single target platform is set via `-fsycl-targets`.

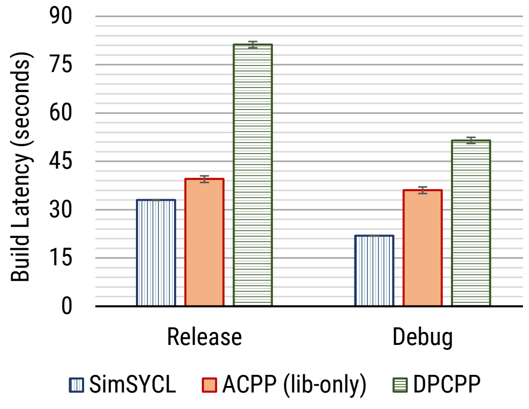


Figure 5: Build latency for Celerity

The Celerity runtime system [23] is a significantly larger code base than SYCL-Bench, and also features a different overall code composition, with a larger portion of code that does not interact with either the SYCL host API or device code at all. As such, it is likely to be more representative of larger-scale projects using SYCL only in part of their implementation. The build latency results for this project are presented in fig. 5.

Likely as a result of this shift in code composition, the relative advantage of SimSYCL over ACpp in debug build latency shrinks, but is still very significant. In SYCL-Bench, the SimSYCL debug build time is 27% of ACpp's time, while for Celerity the SimSYCL build takes 61% in the same comparison. DPC++ build times in release mode are particularly protracted for this project on our test system. Unlike the library-only ACpp version, it is not possible to use the exact same compiler version for DPC++, so there could be several discrepancies leading to this outcome. We tested all available DPC++ targets, and the overall range of results is consistent across all of them, as long as only a single backend is selected.

4.5 Runtime Performance

Achieving competitive runtime performance is explicitly *not* a goal of SimSYCL. However, several of its use cases, such as development and CI testing across a variety of simulated hardware configurations, benefit from being able to run programs at an acceptable performance level.

To evaluate whether this is the case, fig. 6 depicts the results of running the basic Sycl-Bench [17] *VectorAddition_fp64* benchmark across a range of vector data sizes from 1kB to 10GB. As a point of comparison for SimSYCL, we chose the AdaptiveCpp pure library-only OpenMP compilation flow, as it is the most similar in its requirements and goals, as well as the DPC++ CPU backend. All implementations are compiled in their respective Release configurations, with the same compiler version (refer to table 1 for details). Note that the scales on both axes in the chart are logarithmic.

When interpreting these results, we were initially taken aback by the significant performance difference between AdaptiveCpp

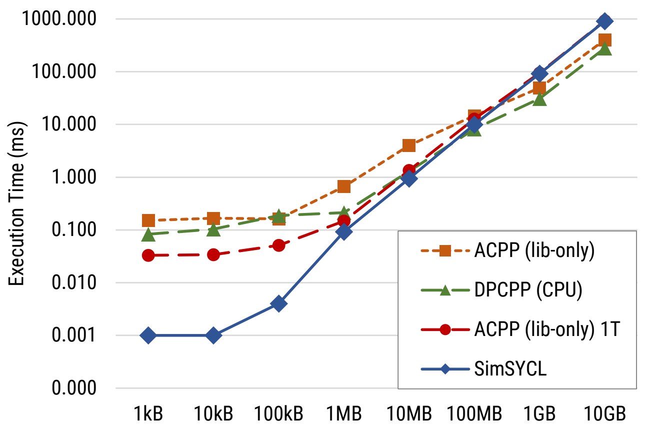


Figure 6: Runtime performance in the SYCL-Bench "VectorAddition_fp64" benchmark

and SimSYCL at not only very small but also moderate problem sizes, with SimSYCL outperforming the library-only configuration of AdaptiveCpp by a factor of 152 at the smallest size of 1kB and still by 10x at 1MB. At these sizes, the relative variance across runs of the AdaptiveCpp implementation was also significant, while all other results were very consistent.

We determined that this enormous discrepancy is due to the large number of hardware threads on our test system (256), which reduces the efficiency of a naive OpenMP-based implementation for small problem sizes, with thread creation and management dominating the execution time. To mitigate the impact of this factor, the results marked as *ACpp (lib-only) 1T* were measured while setting the environment variable `OMP_NUM_THREADS=1`. In this case, the difference between SimSYCL and AdaptiveCpp in its OMP-library-only configuration shrinks to 44% at 10MB, and performance at even larger sizes is almost identical.

However, even with one thread, for the very small sizes below 100kB of data, SimSYCL is still 34x faster than AdaptiveCpp in its library-only configuration. We surmise that this is related to the overhead required for queuing, scheduling etc. which is necessary for a performance-focused SYCL implementation, and which SimSYCL can omit due to its synchronous execution principle. At large problem sizes, the N-thread OpenMP-based library-only ACpp version outperforms SimSYCL (and 1T ACpp) by up to 3x in this benchmark, and the more heavyweight DPC++ CPU backend is slightly faster still at 3.25x.

To summarize, SimSYCL performs well at small problem sizes due to its simple low-overhead design. This is particularly relevant for development and functional testing, which is usually done on reduced data sets and represents SimSYCL's primary target use case. At larger problem sizes performance drops off in a perfectly linear fashion, as expected from a sequential implementation. The same overall behavior can be observed across other benchmarks, with an increasing gap in large problem sizes for more compute-intensive workloads.

4.6 Conformance to the SYCL Specification

The Khronos SYCL Conformance Test Suite (CTS) makes completeness and correctness of SYCL implementations quantifiable [28].

Using a fork[9] of the official code that adds build instructions for SimSYCL and fixes minor bugs in the CTS itself and the SYCL implementations listed in table 1, we evaluate what percentage of CTS test cases successfully compile and how many of the compiled tests pass at runtime. After disabling optional checks for the incompletely implemented `sycl::half` type as well as the checks for OpenCL interoperability (which we cannot possibly support) and without activating the expensive *full conformance mode*, we arrive at the coverage metrics summarized in table 2.

Table 2: Conformance with the SYCL CTS

Implementation	TCs passed / compiled / total
SimSYCL	37 / 44 / 64
AdaptiveCpp (OMP backend)	12 / 29 / 64
DPC++ (OpenCL CPU backend)	60 / 63 / 64

Despite its early state of development, SimSYCL is able to pass more than 50% of CTS cases, the remaining failures being primarily caused by missing implementations for rarely-used features such as images, sub-devices, and certain math functions, as well as deprecated SYCL APIs.

The observation that the mature AdaptiveCpp performs much worse than DPC++ in this metric is owed to the fact that CTS and DPC++ are developed in tandem, while ACpp is currently not regularly tested against the CTS to correct bugs in either codebase.

5 OUTLOOK AND FUTURE WORK

5.1 Limitations

While SimSYCL currently already implements a considerable share of the SYCL standard and is able to diagnose many user errors that other implementations might not, there are still principal limitations that are either inherent to library-only CPU implementations or a result of the trade-off between conformance and debuggability.

There are niche scenarios in which a SYCL program requires true DAG-based asynchronicity between the host application and code inside command groups. For instance, a command group with an accessor that overlaps with the lifetime of a host accessor defined before it in application scope would need to wait for the host accessor to go out of scope before commencing execution. Furthermore, a kernel may explicitly synchronize with subsequent host code through the use of atomics on shared memory, or a host task might do so through standard C++ threading primitives. SimSYCL detects and reports overlapping accessors as unsupported but is currently unable to spot explicit host-to-device communication patterns.

Furthermore, since all memory is allocated on the host, the user will not observe access violations when accessing device-only memory from the host or vice versa, which can hinder the debugging of USM-based applications to some extent. A future version of SimSYCL might toggle virtual-memory permissions at the kernel boundary through system APIs like `mprotect` to force a segmentation fault on an illegal access.

[9]https://github.com/fknorr/SYCL-CTS/releases/tag/simsycl-submission

The specification includes SYCL-specific C++ `[[attributes]]` to supply compile-time arguments and requirements to the backend device compiler. No library-only SYCL implementation can support these features, and neither can SimSYCL in its current form.

Finally, due to SimSYCL's runtime-configurable simulation feature, compile-time queries such as `sycl::is_compatible` cannot provide exact answers.

5.2 Future Work

An accompanying, optional Clang compiler plugin could provide SimSYCL with support for SYCL C++ attributes and compile-time queries on kernels. It might further enforce requirements on types captured within kernels (to deny value-capturing a `sycl::buffer` for instance) and operations permitted, such as denying the throwing of exceptions or calling of virtual functions. More advanced compiler support might annotate local- and global-memory accesses and monitor their coincidence with synchronizing atomic operations to detect data races at runtime.

An alternative, asynchronous DAG-based runtime for SimSYCL might support programs that require true asynchronicity as laid out in section 5.1. Such a feature should remain optional as it trades the debuggability of synchronous execution for the support of rather uncommon SYCL patterns.

6 CONCLUSIONS

In this paper we introduced SimSYCL, a novel, synchronous and host-only implementation of the SYCL standard. It aids in quick application development, portability testing, and debugging. The abandonment of true asynchronicity paired with a focus on implementation simplicity over runtime performance lead to a small API footprint, speeding up the compilation of SYCL applications. In debug mode, which is highly relevant during development, the total build latency of SimSYCL for the SYCL-Bench suite is less than one third of other mainstream SYCL implementations in their fastest-building configurations.

SimSYCL aids in cross-platform CI and portability testing by allowing both developers and users of an application to freely define the characteristics of the executing platform and devices, enabling applications built with SimSYCL to be tested for compatibility without requiring access to the simulated physical hardware.

To facilitate debugging, SimSYCL extensively tracks the state of various kernel operations in order to detect user errors that constitute undefined behavior according to the standard, but cannot be diagnosed explicitly by other implementations. For application bugs that do not result from direct mis-use of SYCL APIs, SimSYCL aids the developer by allowing them to step into kernel functions with their ordinary host debugger.

Finally, its minimalism and strict adherence to the standard uniquely position SimSYCL to potentially serve as an executable specification of the SYCL standard in the future, enabling the prototyping of new features ahead of standardization.

Despite this large supply of development and testing features, SimSYCL is able to achieve sequential performance competitive with other library-only, host-only implementations of the standard, and computing mid-sized problems for the purpose of automatic verification and continuous integration testing remains viable.

ACKNOWLEDGMENTS

This research is supported by the Austrian Research Promotion Agency (FFG) via the UMUGUC project (FFG #4814683), as well as the European High-Performance Computing Joint Undertaking (JU) project LIGATE under grant agreement #956137.

REFERENCES

[1] David Abrahams and Aleksey Gurtovoy. 2004. *C++ template metaprogramming: concepts, tools, and techniques from Boost and beyond.* Pearson Education.

[2] Barış Aktemur, Markus Metzger, Natalia Saiapova, and Mihails Strasuns. 2020. Debugging SYCL Programs on Heterogeneous Intel® Architectures. In *Proceedings of the International Workshop on OpenCL (IWOCL '20).* Association for Computing Machinery, New York, NY, USA, Article 13, 10 pages. https://doi.org/10.1145/3388333.3388646

[3] Aksel Alpay and Vincent Heuveline. 2020. SYCL beyond OpenCL: The Architecture, Current State and Future Direction of hipSYCL. In *Proceedings of the International Workshop on OpenCL (IWOCL '20).* Association for Computing Machinery, New York, NY, USA, Article 8, 1 pages. https://doi.org/10.1145/3388333.3388658

[4] Ben Ashbaugh, Alexey Bader, James Brodman, Jeff Hammond, Michael Kinsner, John Pennycook, Roland Schulz, and Jason Sewall. 2020. Data Parallel C++: Enhancing SYCL Through Extensions for Productivity and Performance. In *Proceedings of the International Workshop on OpenCL (IWOCL '20).* Association for Computing Machinery, New York, NY, USA, Article 7, 2 pages. https://doi.org/10.1145/3388333.3388653

[5] Lorenz Braun and Holger Fröning. 2019. Cuda flux: A lightweight instruction profiler for cuda applications. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS).* IEEE, 73–81.

[6] Robert A. Bridges, Neena Imam, and Tiffany M. Mintz. 2016. Understanding GPU Power: A Survey of Profiling, Modeling, and Simulation Methods. *ACM Comput. Surv.* 49, 3, Article 41 (2016), 27 pages. https://doi.org/10.1145/2962131

[7] Jianbin Fang, Chun Huang, Tao Tang, and Zheng Wang. 2020. Parallel programming models for heterogeneous many-cores: a comprehensive survey. *CCF Transactions on High Performance Computing* 2, 4 (2020), 382–400.

[8] Rob Farber. 2011. *CUDA application design and development.* Elsevier.

[9] Wilson Feng, Rasool Maghareh, and Kai-Ting Amy Wang. 2021. Extending DPC++ with Support for Huawei Ascend AI Chipset. In *International Workshop on OpenCL (IWOCL'21).* Association for Computing Machinery, New York, NY, USA, Article 13, 4 pages. https://doi.org/10.1145/3456669.3456684

[10] Gauthier Harnisch, Andrew Gozillon, Ronan Keryell, Lin-Ya Yu, Ralph Wittig, and Luc Forget. 2021. SYCL for Vitis 2020.2: SYCL & C++ 20 on Xilinx FPGA. In *IWOCL SYCLCon 2021: 9th International Workshop on OpenCL and SYCL.*

[11] Qiming Hou, Kun Zhou, and Baining Guo. 2009. Debugging GPU Stream Programs through Automatic Dataflow Recording and Visualization. In *ACM SIGGRAPH Asia 2009 Papers (SIGGRAPH Asia '09).* Association for Computing Machinery, New York, NY, USA, Article 153, 11 pages. https://doi.org/10.1145/1661412.1618499

[12] Ciera Jaspan and Collin Green. 2023. Developer productivity for humans, part 4: Build latency, predictability, and developer productivity. *IEEE Software* 40, 4 (2023), 25–29.

[13] Hadi Jooybar, Wilson W.L. Fung, Mike O'Connor, Joseph Devietti, and Tor M. Aamodt. 2013. GPUDet: A Deterministic GPU Architecture. *SIGARCH Comput. Archit. News* 41, 1 (2013), 1–12. https://doi.org/10.1145/2490301.2451118

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

[15] Ronan Keryell, Ruyman Reyes, and Lee Howes. 2015. Khronos SYCL for OpenCL: A Tutorial. In *Proceedings of the 3rd International Workshop on OpenCL (IWOCL '15).* Association for Computing Machinery, New York, NY, USA, Article 24, 1 pages. https://doi.org/10.1145/2791321.2791345

[16] Michael Knobloch and Bernd Mohr. 2020. Tools for GPU Computing – Debugging and Performance Analysis of Heterogenous HPC Applications. *Supercomputing Frontiers and Innovations* 7, 1 (2020), 91–111. https://doi.org/10.14529/jsfi200105

[17] Sohan Lal, Aksel Alpay, Philip Salzmann, Biagio Cosenza, Alexander Hirsch, Nicolai Stawinoga, Peter Thoman, Thomas Fahringer, and Vincent Heuveline. 2020. SYCL-bench: a versatile cross-platform benchmark suite for heterogeneous computing. In *Euro-Par 2020: Parallel Processing: 26th International Conference on Parallel and Distributed Computing, Warsaw, Poland, August 24–28, 2020, Proceedings 26.* Springer, 629–644.

[18] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD conference*, Vol. 5. 1–20.

[19] Atmn Patel, Shilei Tian, Johannes Doerfert, and Barbara Chapman. 2021. A Virtual GPU as Developer-Friendly OpenMP Offload Target. In *50th International Conference on Parallel Processing Workshop (ICPP Workshops '21).* Association for Computing Machinery, New York, NY, USA, Article 24, 7 pages. https://doi.org/10.1145/3458744.3473356

[20] Jason Power, Joel Hestness, Marc S. Orr, Mark D. Hill, and David A. Wood. 2015. gem5-gpu: A Heterogeneous CPU-GPU Simulator. *IEEE Computer Architecture Letters* 14, 1 (2015), 34–36. https://doi.org/10.1109/LCA.2014.2299539

[21] James Price and Simon McIntosh-Smith. 2015. Oclgrind: an extensible OpenCL device simulator. In *Proceedings of the 3rd International Workshop on OpenCL (IWOCL '15).* Association for Computing Machinery, New York, NY, USA, Article 12, 7 pages. https://doi.org/10.1145/2791321.2791333

[22] Ruyman Reyes, Codeplay. 2023. The Future of ComputeCpp. https://codeplay.com/portal/news/2023/07/07/the-future-of-computecpp

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

[24] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12).* 309–318.

[25] Alexandre Singer, Frank Gao, and Kai-Ting Amy Wang. 2022. SYCLops: A SYCL Specific LLVM to MLIR Converter. In *International Workshop on OpenCL (IWOCL'22).* Association for Computing Machinery, New York, NY, USA, Article 13, 8 pages. https://doi.org/10.1145/3529538.3529992

[26] Mohamed Tarek Ibn Ziad, Sana Damani, Aamer Jaleel, Stephen W. Keckler, and Mark Stephenson. 2023. CuCatch: A Debugging Tool for Efficiently Catching Memory Safety Violations in CUDA Applications. *Proc. ACM Program. Lang.* 7, PLDI, Article 111 (2023), 24 pages. https://doi.org/10.1145/3591225

[27] The Khronos Group. 2020. Vulkan 1.2.166 Specification. https://www.khronos.org/registry/vulkan/specs/1.2/pdf/vkspec.pdf

[28] The Khronos Group. 2023. Khronos Launches SYCL 2020 Adopters Program and Open Source Conformance Test Suite. https://www.khronos.org/news/press/khronos-launches-sycl-2020-adopters-program-and-open-source-conformance-test-suite

[29] The Khronos Group. 2023. SYCL Specification, Version 2020 Revision 8. https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html

[30] Peter Thoman, Kiril Dichev, Thomas Heller, Roman Iakymchuk, Xavier Aguilar, Khalid Hasanov, Philipp Gschwandtner, Pierre Lemarinier, Stefano Markidis, Herbert Jordan, Thomas Fahringer, Kostas Katrinis, Erwin Laure, and Dimitrios S. Nikolopoulos. 2018. A Taxonomy of Task-Based Parallel Programming Technologies for High-Performance Computing. *J. Supercomput.* 74, 4 (2018), 1422–1434. https://doi.org/10.1007/s11227-018-2238-4

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

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

[33] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. 2012. Multi2Sim: A Simulation Framework for CPU-GPU Computing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12).* Association for Computing Machinery, New York, NY, USA, 335–344. https://doi.org/10.1145/2370816.2370865

[34] Michael L. Van De Vanter, D. E. Post, and Mary E. Zosel. 2005. HPC Needs a Tool Strategy. In *Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications (SE-HPCS '05).* Association for Computing Machinery, New York, NY, USA, 55–59. https://doi.org/10.1145/1145319.1145335