Harnessing the full power of modern hardware accelerators using idomatic C++

STEFAN BUTZ, University of Hagen, Germany

Modern hardware accelerators such as GPUs and FPGAs provide immense computational power, yet their utilization remains challenging due to the complexity of existing programming models. While low-level solutions such as CUDA and OpenCL offer high performance, they require specialized knowledge, limiting accessibility. High-level approaches like SYCL aim to bridge this gap by providing a modern, C++-based abstraction for heterogeneous computing.

This paper explores the integration of SYCL as a backend for C++ Standard Parallelism (stdpar) through the AdaptiveCpp compiler. By leveraging compiler-driven transformations, this approach enables efficient execution of stdpar workloads on heterogeneous devices while maintaining the high-level abstraction expected from the C++ standard library. Key challenges in execution, memory management, and synchronization are identified, and solutions leveraging SYCL's task-based execution model and Unified Shared Memory (USM) are presented.

Performance optimizations, including kernel launch overhead reduction, memory pooling, and conditional offloading, are analyzed to demonstrate the effectiveness of this integration. The evaluation confirms that the AdaptiveCpp approach provides competitive performance compared to vendor-specific implementations while maintaining portability across different hardware architectures.

This study highlights the broader implications of compiler-assisted parallelism, demonstrating how tighter integration between C++ standard features and heterogeneous programming models can simplify adoption and improve efficiency. Future C++ standards, including senders/receivers and std::simd, will further enhance the expressiveness and performance of parallel programming in modern software development.

Additional Key Words and Phrases: C++, SYCL, parallelism, CUDA, GPU, stdpar

1 MOTIVATION

Modern hardware accelerators, such as GPUs and FPGAs, offer significant computational power. They are essential for improving the performance of many applications. However, using these accelerators is challenging because they require specialized knowledge of their architectures and programming models.

To address this, many libraries and frameworks have been created. Low-level libraries like CUDA for NVIDIA GPUs [10] and ROCm for AMD GPUs [3] provide detailed control over hardware. However, they also demand in-depth understanding of hardware-specific programming. This makes them difficult for developers who are not experts in high-performance computing.

Higher-level frameworks aim to simplify programming for accelerators while maintaining good performance. OpenCL, [8] for instance, provides a platform-neutral model that works across CPUs, GPUs, and other accelerators. SYCL builds on OpenCL, offering a modern C++ interface that aligns with standard C++ practices. A key strength of SYCL is its multi-platform support, allowing developers to write portable code that can run on a variety of hardware, including GPUs, CPUs, and FPGAs, without modification. This feature ensures that applications can scale across diverse hardware architectures, providing flexibility and ease of use. [11]

Despite these advancements, many frameworks still require developers to learn programming models and manage tasks like device selection and memory transfers. This adds complexity and slows adoption, especially for those outside specialized computing fields. Simplifying this process can save time and reduce the learning curve for developers.

Cost is another important factor. While accelerators improve computational performance, they also require substantial investments in hardware. Additionally, the time developers spend learning and coding for these systems contributes to

Author's address: Stefan Butz, University of Hagen, Hagen, Germany, stefan.butz@studium.feruni-hagen.de.

project costs. Simplifying the programming process helps reduce these expenses. It allows developers to focus on solving domain-specific problems rather than hardware-specific challenges. Faster project completion also brings business benefits, such as quicker time-to-market and improved competitiveness.

C++ is one of the most widely used programming languages. [7] It is known for its performance, flexibility, and large ecosystem. With the introduction of parallel algorithms in C++17, the language has made a major step towards simplifying parallel programming. These algorithms allow developers to express parallelism at a high level, avoiding the need to understand the details of the underlying hardware. [13]

In their research, Aksel Alpay and Vincent Heuveline demonstrated how the SYCL programming model can act as a backend for C++ standard parallelism. A key advantage of this approach is that developers can use idiomatic C++ without having to learn a new programming model. By simply utilizing AdaptiveCpp's compiler, developers can seamlessly integrate SYCL into their workflows. This approach ensures that parallelism is implemented in a familiar C++ environment while still achieving portability and efficiency across various hardware platforms. This combination of SYCL and C++ standard parallelism creates a unified and accessible approach to modern parallel programming. [2]

As the foundation of this work, understanding the SYCL programming model is essential. The next section introduces SYCL, its core principles, and its design, explaining how it provides a modern, high-level interface for heterogeneous computing while maintaining compatibility with standard C++ practices.

2 INTRODUCTION OF THE SYCL PROGRAMMING MODEL

The following introduction is based on SYCL 2020 specification [5] and a SYCL Programming Guide [11]. SYCL, an open standard developed by the Khronos Group, is a modern programming model designed to address the challenges of heterogeneous computing. Originating from the OpenCL ecosystem, SYCL builds on OpenCL's foundation but introduces higher-level abstractions and a more intuitive programming model. Its purpose is to simplify the development of applications that target heterogeneous systems, including CPUs, GPUs, and FPGAs, by leveraging standard C++ features. SYCL combines performance, developer productivity, and compatibility with C++ into a unified framework.

SYCL's core advantages include:

(1) **Single-Source Programming**: Unified source files contain both host and device code, eliminating the separation required by lower-level models like OpenCL.
(2) **Portability Across Devices**: A single SYCL program can target multiple devices (CPUs, GPUs, and FPGAs) without requiring changes to the code.
(3) **High-Level Abstractions**: Developers can use standard C++ constructs, such as templates, lambda expressions, and object-oriented programming, making SYCL more approachable to those familiar with C++.

In the following, the core concepts of SYCL are introduced, illustrated by a simple example program that serves as a reference for deeper explanations.

2.1 A Simple SYCL Example

Listing 1 shows a minimal SYCL program that demonstrates its essential elements, including memory management, kernel execution, and synchronization.

This program represents a fundamental SYCL workflow. The host initializes the data using a standard C++ vector. A buffer is created to encapsulate this data, enabling it to be transferred to and accessed by the device. The queue submits a kernel to the device, where the computational work is performed. Specifically, the `parallel_for` construct represents

the kernel that runs on the device. Within this kernel, each work-item processes an element of the buffer, doubling its value. Once the kernel execution completes, the updated data is automatically synchronized back to the host memory when the buffer's scope ends. The host code then prints the modified data.

The key concepts demonstrated in this example, including memory management, task submission, synchronization, and kernel execution, are now explained in greater detail.

```
#include <CL/sycl.hpp>
#include <iostream>
using namespace sycl;

int main() {
    constexpr size_t N = 16;
    std::vector<int> data(N, 1); // Host code: Initialize a vector with 16 elements, all set to 1

    // Host code: Create a SYCL queue to submit work
    queue q;

    {
        // Host code: Create a buffer that wraps the vector data
        buffer<int, 1> buf(data.data(), range<1>(N));

        // Host code: Submit a command group to the queue
        q.submit([&](handler& h) {
            // Host and device code: Create an accessor to access the buffer in write mode
            auto accessor = buf.get_access<access::mode::write>(h);

            // Device code: Define the kernel using a parallel_for construct
            h.parallel_for(range<1>(N), [=](id<1> idx) {
                accessor[idx] *= 2; // Device code: Double each element
            });
        });
    } // Host code: Buffer scope ends here, data is synchronized back to the host

    // Host code: Print the modified data
    for (int i = 0; i < N; ++i) {
        std::cout << data[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Listing 1. A Simple SYCL Program

2.2 Core Concepts of SYCL

SYCL's design revolves around several core concepts that simplify heterogeneous programming. These concepts are discussed in detail below, with references to the example program to clarify their implementation.

2.2.1 Single-Source Programming and Host vs. Device Code. SYCL's single-source programming model simplifies development by combining host and device code into a single source file written in standard C++. This unification contrasts with traditional models like CUDA and OpenCL, where host and device code are written separately. The kernel

in the example is defined using a lambda expression within the host code, highlighting SYCL's ability to seamlessly integrate host and device logic.

However, SYCL imposes certain **limitations on device code** due to hardware constraints. Device code cannot use dynamic polymorphism, exceptions, or runtime type information (RTTI). For instance, features such as `typeid` and virtual function calls are not supported. These restrictions ensure compatibility with a wide range of devices and maintain computational efficiency [11].

2.2.2 Asynchronous Task Graphs, Queues, and Synchronization. SYCL's execution model is built on the concept of asynchronous task graphs, which allow computational tasks and memory operations to execute in parallel while respecting dependencies. A task graph is represented as a **directed acyclic graph (DAG)**, where nodes correspond to tasks (e.g., kernel executions or data transfers), and edges represent dependencies between these tasks. This model is fundamental to achieving high performance in heterogeneous computing environments, as it allows overlapping computation and memory transfers, reducing idle time on both host and device.

At the core of SYCL's task execution model are **queues**, which serve as a bridge between the host and devices. Queues are responsible for submitting tasks to the device and managing their execution order. Each queue is associated with a specific device, such as a CPU, GPU, or FPGA, and tasks submitted to the queue are executed on that device.

SYCL supports two types of queues: **in-order** and **out-of-order**. In an in-order queue, tasks are executed strictly in the order they are submitted. Each task must complete before the next one begins, regardless of whether there are dependencies between the tasks. This behavior ensures a predictable execution flow, simplifying reasoning about task dependencies.

In contrast, out-of-order queues allow tasks to execute as soon as their dependencies are resolved, regardless of the order in which they were submitted. This capability enables greater concurrency, as independent tasks can run in parallel. Dependencies between tasks are specified explicitly using events. [5]

Synchronization and Dependency Management

Regardless of the queue type, SYCL provides mechanisms to synchronize tasks and manage dependencies:

- **Events**: Each task submission generates an event object, which can be queried to determine whether the task has completed. Events can also be passed to subsequent tasks to define dependencies explicitly.
- **Queue-Level Synchronization**: Developers can call `queue.wait()` to block the host until all tasks in the queue have completed execution. This approach ensures that all previously submitted tasks are finalized before proceeding with further operations on the host.

[5]

2.2.3 Memory Management and Data Flow. Memory management is a critical aspect of SYCL, with two main approaches: **buffers and accessors** and **Unified Shared Memory (USM)**.

Buffers and Accessors: Buffers encapsulate data that is shared between host and device memory. They automatically handle data transfers and synchronization. In the example, the buffer wraps the `data` array, ensuring that the device kernel can access and modify it. An accessor is used to specify the desired access mode (e.g., read, write, or read-write) and to provide safe access to the buffer's data within the kernel [5].

Unified Shared Memory (USM): Introduced in SYCL 2020, USM offers a more flexible memory model. Unlike buffers, USM allows developers to allocate memory directly shared between the host and device. There are three types of USM:

(1) **Device Memory**: Allocated on the device, accessible only from the device. Explicit memory transfers are required to move data between the host and device.
(2) **Host Memory**: Allocated on the host, accessible from both host and device, but with performance overhead for device access. Data is typically transferred over a bus, such as PCIe, which can introduce latency.
(3) **Shared Memory**: Allocated in a shared memory space accessible by both host and device. The runtime handles data transfer and reallocation behind the scenes, staging data on the device before kernel execution to minimize latency during computation.

Unified Shared Memory simplifies the integration of SYCL into existing applications by allowing developers to allocate memory using familiar C++ constructs such as `malloc` and `new`. This reduces the need to rewrite application logic to accommodate SYCL's abstractions. Furthermore, USM is particularly advantageous for applications requiring indirect memory access. For example, in pointer-based data structures like linked lists or graphs, USM enables device kernels to follow pointers naturally without requiring additional data packing or offset calculations [5, 11].

2.3 AdaptiveCpp: A SYCL implemenation

The following introudction is based on the work of Aksel Alpay and Vincent Heuveline in their paper *AdaptiveCpp Stdpar: C++ Standard Parallelism Integrated Into a SYCL Compiler* [2]. AdaptiveCpp (formerly *hipSYCL*), available under an open-source license at https://github.com/AdaptiveCpp/AdaptiveCpp, is a SYCL implementation designed for performance portability across CPUs, NVIDIA GPUs, and AMD GPUs. Unlike traditional SYCL toolchains, AdaptiveCpp's architecture emphasizes a **single-source, single-pass compiler** that generates **universal binaries** capable of targeting multiple accelerators without recompilation [2].

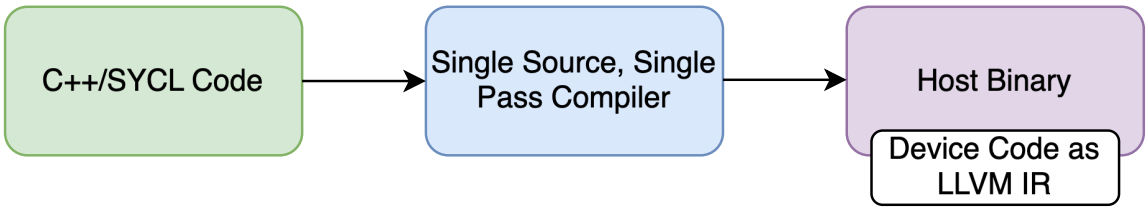

Fig. 1. AdaptiveCpp Compiler Workflow

Compiler Architecture. The AdaptiveCpp compiler processes SYCL code in a single pass, generating a unified LLVM Intermediate Representation (IR) that embeds both host and device code. This wofklow is shown in Figure 1. Device code regions (e.g., SYCL kernels) are annotated with metadata such as `__attribute__((sycl_device))` to distinguish them from host logic. Crucially, this IR remains hardware-agnostic, deferring target-specific code generation to runtime. During execution, the Just-in-Time (JIT) compiler extracts device code from the IR and compiles it to platform-specific instruction sets (e.g., NVIDIA PTX, AMD GCN, or CPU multithreaded code) using different backends. The available device code generators are shown in Figure 2. This enables **universal binaries** that dynamically adapt to the available hardware, avoiding upfront commitment to a specific accelerator architecture [5].

Memory Management. AdaptiveCpp implements SYCL's memory model with a hybrid approach combining automatic and explicit strategies. For a detailed description of Unified Shared Memory (USM) and other memory types, please refer to subsection 2.2.3. The Buffer-Accessor Model uses `sycl::buffer<T>` to track memory regions across host and

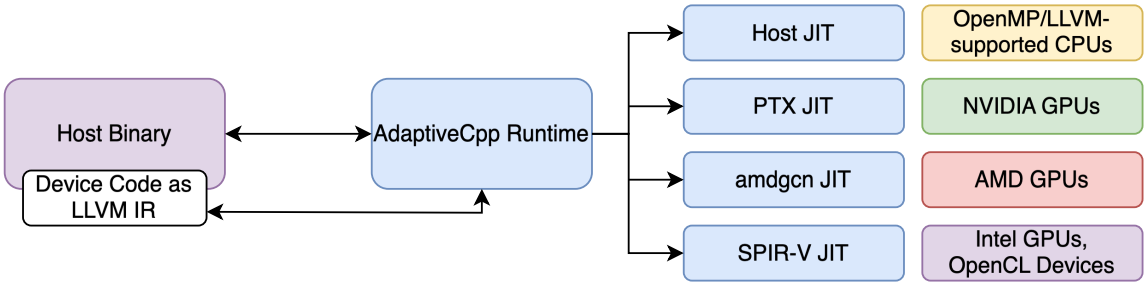

Fig. 2. AdaptiveCpp Device Code Generators [1]

device, while `sycl::accessor` objects define access modes (read/write) and trigger implicit data transfers. The runtime constructs a directed acyclic graph (DAG) of kernel dependencies, synchronizing memory automatically. For instance, overlapping writes to a buffer are resolved through runtime-managed locks or atomic operations.

Under the hood, AdaptiveCpp's runtime delegates memory operations to backend-specific APIs (e.g., CUDA streams for NVIDIA GPUs), but ensures these details are abstracted from developers. This minimizes manual memory management while retaining performance close to native frameworks [5].

Compilation Example. At a basic level, AdaptiveCpp provides a wrapper script (e.g., `acpp`) for building single-source programs that produce portable binaries. For instance:

```
acpp --acpp-targets=generic -o test test.cpp
```

This command invokes the single-pass compiler to create a *generic* universal binary that adapts dynamically to the available hardware. Alternative flags can specify explicit usage of vendor compilers (e.g., `cuda-nvcxx`) when targeting NVIDIA GPUs, allowing further customization and performance tuning.

AdaptiveCpp not only supports SYCL's programming model but the C++ standard parallelism library (stdpar) as well. This integration is explored in the subsequent sections starting with an introduction of the C++ standard parallelism programming model.

3 INTRODUCTION TO C++ STANDARD PARALLELISM (STDPAR)

The evolution of C++ has consistently aimed to enhance performance and efficiency. A significant milestone in this journey was the introduction of standard parallelism, commonly referred to as **stdpar**, in the C++17 standard. This feature provides developers with a standardized and portable approach to writing parallel code, facilitating the efficient utilization of modern multi-core processors and accelerators. This introduction is baed on work of V'yukova, et al [13].

3.1 Background and Motivation

Prior to stdpar, parallel programming in C++ often relied on external libraries or platform-specific extensions, such as OpenMP or Intel's Threading Building Blocks (TBB). While effective, these solutions lacked standardization, leading to portability challenges across different compilers and platforms. The need for a unified approach to parallelism in C++ became evident, prompting the inclusion of parallel execution policies in the C++17 standard [13].

3.2 Overview of stdpar

stdpar introduces execution policies that dictate how standard library algorithms are executed:

- **`std::execution::seq`**: Specifies sequential execution.
- **`std::execution::par`**: Enables parallel execution across multiple threads.
- **`std::execution::par_unseq`**: Allows parallel execution with vectorization, combining multithreading and SIMD (Single Instruction, Multiple Data) optimizations.

These policies can be applied to standard algorithms to control their execution behavior [13].

3.3 Example Usage of stdpar

Listing 2 shows a simple example demonstrating the use of `std::execution::par` to parallelize a standard algorithm:

```
#include <algorithm>
#include <execution>
#include <vector>

int main() {
    std::vector<int> data(100000, 1);
    std::for_each(std::execution::par, data.begin(), data.end(), [](int& x) { x *= 2; });
}
```

Listing 2. Parallel execution using `std::execution::par`

In this example, `std::for_each` applies the provided lambda function to each element in the vector `data` in parallel, effectively doubling each element's value.

3.4 Execution Model, Hardware Mapping, and Memory Management

The execution policy specified (e.g., `std::execution::par`) serves as a hint to the compiler and runtime about how the algorithm should be executed. The actual execution can vary depending on the system's hardware and the compiler's implementation. For instance, parallel execution may be achieved through multi-threading on a CPU or offloaded to a GPU if supported. This abstraction allows developers to write parallel code without delving into the specifics of the underlying hardware [13].

Memory management in stdpar is typically handled by the compiler and runtime system. When parallel algorithms are executed, the necessary data is usually allocated in host memory. If the computation is offloaded to an accelerator, such as a GPU, the compiler and runtime manage data transfers between the host and the device. For instance, NVIDIA's implementation of stdpar utilizes CUDA Unified Memory, allowing allocated memory to be visible on both the host and the device, thereby simplifying memory management for the programmer. However, depending on the implementation, explicit memory transfers may still be required to optimize performance for specific workloads [9].

3.5 Performance Considerations and Limitations

While stdpar offers several advantages, such as portability and ease of use, there are considerations and limitations to keep in mind:

- **Backend-Dependent Performance**: The efficiency of parallel execution depends on the compiler's implementation and the target hardware. Not all compilers optimize stdpar for all architectures. For instance, NVIDIA's

`nvc++` compiler provides limited support for C++ Parallel Algorithms on Pascal architecture GPUs, as they lack independent thread scheduling necessary to properly support the `std::execution::par` policy [9].

- **Lack of Fine-Grained Control**: Compared to low-level parallelism models like CUDA or OpenMP, stdpar provides less control over thread management and workload distribution. This abstraction can lead to suboptimal performance in scenarios requiring fine-tuned parallelism.
- **Compiler Support Variability**: Not all compilers fully support stdpar, and the level of optimization can vary between compiler implementations. For example, as of certain versions, Clang does not support C++17 execution policies [13].
- **Language Feature Restrictions**: Certain C++ language features and constructs should be used with caution to ensure safe and efficient parallel execution. For example, the use of function pointers within parallel algorithms may be restricted or unsupported, depending on the compiler and target hardware. Additionally, exception handling within parallel regions is often limited, as many implementations do not support exceptions in GPU code [9].

The following sections will explore how SYCL can be leveraged as a backend for stdpar to address some of these limitations and provide a more portable and efficient parallel programming model.

4 USING SYCL AS AN BACKEND FOR C++ STANDARD PARALLELISM

4.1 Design Goals

The following section is based on the work of Aksel Alpay and Vincent Heuveline in their paper *AdaptiveCpp Stdpar: C++ Standard Parallelism Integrated Into a SYCL Compiler* [2]. Their approach to integrating SYCL as a backend for C++ Standard Parallelism (*stdpar*) is guided by three main design goals: (1) support for a wide range of hardware, (2) maintaining an open-source implementation, and (3) achieving tight integration with the compiler for improved performance.

Wide Range of Hardware Support. A fundamental goal of AdaptiveCpp Stdpar is to provide a backend capable of supporting multiple hardware architectures. Unlike vendor-specific solutions, such as NVIDIA's `nvc++` compiler, which restricts offloading to NVIDIA GPUs, AdaptiveCpp extends support to a broader range of devices, including CPUs, Intel GPUs, AMD GPUs, and NVIDIA GPUs. By leveraging SYCL's backend abstraction, the implementation ensures portability across different hardware vendors while enabling the use of *stdpar* without modifying source code.

Open-Source Implementation. Another key goal is to provide an open-source alternative to proprietary solutions. Open-source availability not only fosters community-driven development and collaboration but also allows researchers and developers to extend and customize the implementation. AdaptiveCpp is developed as part of an open ecosystem, enabling transparency in its optimizations and ensuring that it remains adaptable to future hardware architectures and software changes.

Tight Integration with the Compiler. Unlike NVIDIA's *stdpar* implementation, which relies primarily on a runtime library for offloading, AdaptiveCpp integrates tightly into the compiler toolchain. This approach allows for more efficient code generation, improved memory management, and reduced runtime overhead. By embedding device-specific optimizations directly into the compilation process, AdaptiveCpp minimizes the need for manual tuning and achieves performance levels closer to native SYCL or CUDA implementations.

These three goals form the foundation of AdaptiveCpp Stdpar's design and influence its approach to execution and memory management, which will be explored in the following sections.

4.2 Challenges

The following section discusses key challenges in integrating SYCL as a backend for C++ Standard Parallelism (stdpar). This analysis is based on the work of Aksel Alpay and Vincent Heuveline in their paper *AdaptiveCpp Stdpar: C++ Standard Parallelism Integrated Into a SYCL Compiler* [2]. The integration of stdpar with SYCL presents several difficulties due to fundamental differences in execution models, memory handling, and synchronization mechanisms.

Execution Model Differences. C++ Standard Parallelism (stdpar) provides a high-level abstraction of parallel execution using execution policies such as `std::execution::par` and `std::execution::par_unseq`. These policies allow algorithms to be expressed in a portable way, but they do not impose a specific execution model. In contrast, SYCL requires an explicit task-based execution model using command groups, queues, and kernel enqueuing. This fundamental mismatch poses challenges in mapping stdpar's implicit parallel execution semantics onto SYCL's explicitly managed execution graph.

Memory Management Mismatch. stdpar assumes implicit memory management, where data movement and allocation are handled transparently by the compiler and runtime. In contrast, SYCL requires explicit memory handling using buffers, accessors, or Unified Shared Memory (USM). stdpar does not define explicit data transfer mechanisms, leading to potential mismatches when interfacing with SYCL's explicit memory model. Managing memory allocation and ensuring correct data movement between host and device adds complexity to the integration.

Synchronization and Task Scheduling. stdpar does not provide explicit synchronization mechanisms; it relies on the underlying execution backend to ensure correct ordering of operations. However, SYCL employs an explicit dependency model where task execution must be synchronized using events, dependency graphs, or barriers. This difference introduces challenges in enforcing correct execution ordering when translating stdpar algorithms into SYCL kernels.

Device Execution Constraints. stdpar is designed to work seamlessly with standard C++ execution environments, but SYCL imposes constraints on device execution. Notably, device kernels in SYCL have restrictions such as the absence of dynamic memory allocation, function pointers, and exception handling. These limitations create difficulties in porting certain stdpar constructs to SYCL-compatible execution models.

Performance Overhead. stdpar is optimized for CPU execution, where compilers can efficiently schedule operations across multiple cores. However, SYCL introduces additional overhead due to explicit kernel enqueuing and execution on heterogeneous devices. The need to manage execution queues, memory transfers, and synchronization mechanisms can lead to performance bottlenecks, making efficient stdpar execution on SYCL devices a challenging task.

These challenges highlight the complexity of integrating stdpar with SYCL. The following sections will explore how these issues are addressed to enable seamless execution of stdpar algorithms on heterogeneous hardware through SYCL. Performance optimizations will be discussed in section 5.

4.3 Providing stdpar Execution Model Using SYCL as Backend

The following section describes how C++ Standard Parallelism (stdpar) execution is mapped to SYCL, allowing stdpar algorithms to offload computations to heterogeneous hardware via AdaptiveCpp. The integration requires automatic

translation of stdpar execution policies into SYCL's explicit task-based execution model. This work is based on the research of Aksel Alpay and Vincent Heuveline in their paper *AdaptiveCpp Stdpar: C++ Standard Parallelism Integrated Into a SYCL Compiler* [2].

Automatic Detection of Code for Device Execution. stdpar does not explicitly specify which code should execute on a device. Instead, the AdaptiveCpp compiler analyzes stdpar algorithms and detects execution regions suitable for device offloading. When an execution policy such as `std::execution::par` is used, the compiler determines if a function can be offloaded and automatically transforms it into a SYCL-compatible kernel. Unlike traditional SYCL programming, where explicit kernel definitions are required, AdaptiveCpp eliminates this requirement by generating SYCL kernels dynamically.

A key challenge in automatic detection is ensuring that only the correct portions of an algorithm are offloaded. Certain C++ constructs, such as those involving indirect memory access or dynamic dispatch, may be problematic on accelerators. AdaptiveCpp must analyze function dependencies and ensure that only compute-intensive and parallelizable operations are translated into SYCL kernels while keeping host-side operations intact. This requires sophisticated compiler transformations to split execution logic efficiently.

Queue-Based Execution and Kernel Dispatch. stdpar does not expose an explicit queueing model, whereas SYCL requires that all device execution be submitted through a queue. AdaptiveCpp bridges this gap by mapping stdpar algorithm calls to SYCL kernels, which are enqueued into a SYCL queue. If stdpar operations have dependencies on previous computations, execution order must be preserved, requiring an implicit synchronization mechanism within the queue.

To ensure efficient execution, AdaptiveCpp uses a queue management system that dynamically selects an optimal SYCL queue based on the device type and workload characteristics. This avoids unnecessary overhead when executing stdpar operations on heterogeneous devices. Moreover, the queueing mechanism ensures that multiple stdpar operations submitted in sequence are optimally scheduled, reducing kernel launch latency.

Mapping Parallel Execution to SYCL Kernels. stdpar abstracts parallel execution through execution policies such as `std::execution::par`, whereas SYCL requires explicit kernel invocation via `parallel_for`. The AdaptiveCpp compiler translates stdpar execution policies into appropriate SYCL parallel constructs. For example, a call to `std::for_each(std::execution::par, ...)` is transformed into a SYCL `parallel_for` execution.

An important consideration when generating SYCL kernels is handling workload distribution efficiently. AdaptiveCpp performs compile-time and runtime analysis to determine the best way to partition work across available processing units. This involves selecting between different SYCL execution models, including `nd_range` kernels for fine-grained control or implicit work partitioning when possible.

Handling Work-Group and Work-Item Distribution. stdpar does not provide direct control over work-group and work-item distribution, but SYCL requires this information for kernel execution. AdaptiveCpp automatically selects an appropriate `nd_range` or implicit work division for SYCL kernels based on the target hardware. The compiler queries device capabilities (e.g., maximum work-group size, available compute units) and determines an efficient execution strategy.

For large workloads, AdaptiveCpp optimizes work distribution using device-specific heuristics. On GPUs, it ensures that the number of work-items per work-group is tuned to achieve high resource utilization. On CPUs, AdaptiveCpp

adapts its execution strategy to leverage SIMD execution where applicable, ensuring that performance benefits are retained across different architectures.

Synchronization between stdpar Operations in SYCL. stdpar does not enforce explicit synchronization between operations, whereas SYCL requires explicit event handling for execution dependencies. AdaptiveCpp ensures correct execution order by introducing SYCL events where necessary. If an stdpar algorithm has a dependency on a previous computation, AdaptiveCpp generates an appropriate `sycl::event` to ensure proper sequencing. Unnecessary synchronization is avoided by applying graph-based dependency tracking, reducing execution overhead.

To further optimize performance, AdaptiveCpp can elide certain synchronization points when it detects that dependent operations do not conflict. This ensures that execution proceeds efficiently without excessive barriers that could degrade parallel performance.

This section outlines the key transformations required to map stdpar execution semantics to SYCL's explicit execution model. The next section will explore memory management strategies required to support stdpar execution in a SYCL backend.

4.4 Handling of Memory Allocations and Data Transfers

4.4.1 Memory Models in stdpar and SYCL. The integration of SYCL as a backend for C++ Standard Parallelism (stdpar) introduces significant challenges due to differences in memory models. stdpar assumes an implicit memory management approach, where memory allocation and transfers are handled transparently by the compiler and runtime system. In contrast, SYCL requires explicit memory management, where developers must define how data is allocated, transferred, and accessed between the host and device.

A detailed introduction to SYCL's memory model has already been provided in Section 2.2.3. This subsection will build upon that foundation by focusing on the specific challenges that arise when mapping stdpar's implicit memory assumptions onto SYCL's explicit memory management.

Memory Model in stdpar. stdpar abstracts memory operations from the programmer, providing a high-level interface where data movement is implicitly managed. When executing an algorithm using `std::execution::par` or `par_unseq`, the underlying implementation determines whether memory needs to be allocated on an accelerator and performs the necessary data transfers automatically. This approach simplifies development but requires the compiler and runtime to infer memory access patterns efficiently.

stdpar's implicit memory model eliminates the need for explicit memory handling, allowing users to focus on algorithmic development rather than low-level memory operations. However, this abstraction introduces challenges when integrating stdpar with SYCL's explicit memory management model.

Memory Model in SYCL. SYCL provides a flexible but explicit memory model, giving developers fine-grained control over data movement and synchronization. It offers two primary approaches to memory management: **buffers/accessors** and **Unified Shared Memory (USM)**, both of which must be explicitly managed.

Buffer-Accessor Model: This model requires memory to be allocated as `sycl::buffer<T>` objects. Access to the buffer contents is provided through `sycl::accessor` objects, which specify whether the memory is read-only, write-only, or read-write. The SYCL runtime tracks dependencies between memory accesses and ensures correct synchronization of buffers between host and device.

Unified Shared Memory (USM): As previously discussed in Section 2.2.3, USM allows developers to allocate memory that can be accessed by both the host and device without requiring explicit buffer management. This model simplifies memory handling and is particularly useful for stdpar integration, as it aligns with stdpar's expectation of automatic memory management.

The explicit nature of SYCL's memory model presents challenges when integrating stdpar, as stdpar assumes automatic memory handling while SYCL requires explicit memory specification. The next section will explore how AdaptiveCpp bridges this gap to ensure seamless integration of stdpar execution within SYCL's memory management framework.

4.4.2 Managing Data Transfers in AdaptiveCpp. The integration of stdpar with SYCL requires efficient memory transfers between the host and the target device. Since stdpar does not expose an explicit memory management model, AdaptiveCpp must ensure that data movement is handled correctly without requiring user intervention.

Intercepting stdpar Memory Operations. AdaptiveCpp intercepts stdpar memory operations and translates them into SYCL-compatible memory management calls. This ensures that memory allocations and transfers conform to SYCL's explicit memory model while preserving the abstraction provided by stdpar. Depending on the execution context, AdaptiveCpp dynamically decides whether to use USM or explicit buffer-accessor transfers.

Ensuring Correct Data Transfers. stdpar abstracts away memory movement, requiring AdaptiveCpp to infer data transfer needs at runtime. When USM is used, memory migration is handled transparently by the SYCL runtime. However, in cases where explicit buffers are used, AdaptiveCpp ensures correct data placement between the host and device by managing data dependencies and tracking memory accesses.

By leveraging SYCL's memory management features, AdaptiveCpp enables stdpar applications to execute efficiently on heterogeneous devices while maintaining stdpar's high-level abstraction of memory handling.

4.4.3 Synchronization and Memory Consistency. stdpar provides an implicit synchronization model in which memory consistency and execution ordering are handled by the compiler and runtime. This abstraction simplifies development by ensuring that memory updates are propagated correctly without requiring explicit user intervention. However, when mapping stdpar to SYCL, these assumptions must be explicitly managed since SYCL requires a more fine-grained approach to synchronization.

Explicit Synchronization in SYCL. SYCL introduces explicit synchronization mechanisms to manage dependencies between tasks and ensure correct execution order. Unlike stdpar, which abstracts synchronization, SYCL requires the use of:

- **Events (`sycl::event`):** Used to define dependencies between kernel executions, ensuring that a task does not start before its prerequisites are completed.
- **Memory fences:** Necessary to guarantee memory consistency between different processing units, ensuring that data written by one kernel is visible to subsequent tasks.
- **Explicit host-device synchronization:** Required when data is accessed on both the host and device, avoiding race conditions and undefined behavior.

Ensuring Correct Synchronization in AdaptiveCpp. To bridge the gap between stdpar's implicit model and SYCL's explicit requirements, AdaptiveCpp implements automatic synchronization management by:

- **Generating event dependencies:** AdaptiveCpp dynamically inserts `sycl::event` objects to enforce execution order where needed.
- **Ensuring memory consistency:** AdaptiveCpp ensures that host-device memory consistency is maintained without redundant synchronization, improving correctness.

4.5 Conclusion

The integration of SYCL as a backend for C++ Standard Parallelism requires careful handling of memory management, data transfers, and synchronization. Unlike stdpar, which abstracts memory operations, SYCL enforces an explicit memory model that necessitates strategies to maintain correctness across heterogeneous execution environments.

AdaptiveCpp effectively bridges this gap by automating memory management through USM and buffer-accessor models while ensuring correct synchronization between host and device execution. The performance impact of these design choices will be evaluated in Section 5, where optimization strategies for execution and memory handling are discussed in detail.

5 PERFORMANCE OPTIMIZATION

The following section is based on the work of Aksel Alpay and Vincent Heuveline in their paper *AdaptiveCpp Stdpar: C++ Standard Parallelism Integrated Into a SYCL Compiler* [2].

5.1 Reducing Kernel Launch Overhead

One of the key challenges in using SYCL as a backend for C++ Standard Parallelism (stdpar) is the overhead associated with kernel launches. Unlike stdpar, which abstracts execution details, SYCL requires explicit kernel enqueuing and synchronization. This additional complexity can introduce latency, particularly when launching many small kernels or when excessive synchronization points are introduced.

Problem: Latency in Kernel Submission. stdpar enables parallel execution through high-level execution policies such as `std::execution::par`, without requiring the user to manage kernel launches explicitly. In contrast, SYCL mandates that each kernel be enqueued into a command queue, leading to additional overhead in terms of event tracking and command group management. If every stdpar operation translates directly into an individual SYCL kernel submission, the resulting overhead can significantly impact performance.

Optimization: Instant Kernel Submission. AdaptiveCpp optimizes stdpar execution by reducing redundant kernel launch overhead. Instead of enqueuing each operation separately, it implements an *instant submission model*, which minimizes intermediate operations. By bypassing unnecessary buffering layers in the compilation pipeline, kernel launches are optimized for lower latency.

Optimization: Reducing Event Dependencies. SYCL requires explicit dependencies between kernels to ensure correct execution ordering. However, excessive synchronization can lead to performance bottlenecks. AdaptiveCpp dynamically analyzes execution dependencies and removes redundant event tracking when safe to do so. This optimization ensures that kernel submission remains efficient without unnecessary dependency chains.

Optimization: Efficient Queue Management. Another source of kernel launch overhead is inefficient queue utilization. AdaptiveCpp dynamically selects execution queues based on workload characteristics, distributing tasks across available

queues to minimize bottlenecks. This prevents individual queues from becoming overloaded while ensuring that computational resources are fully utilized.

By addressing these challenges, AdaptiveCpp significantly reduces kernel launch latency, making stdpar execution more efficient in SYCL-based heterogeneous computing environments.

5.2 Memory Management Optimizations

Memory management plays a critical role in the efficient execution of stdpar workloads using SYCL. Unlike stdpar, which abstracts memory allocation and transfer, SYCL requires explicit memory handling. Inefficient memory management can introduce excessive allocation overhead and redundant data transfers, leading to performance degradation.

Problem: Explicit Memory Management in SYCL. stdpar assumes an implicit memory model where memory allocations and transfers are managed automatically. However, SYCL requires explicit allocation using Unified Shared Memory (USM) or buffers/accessors. The additional burden of managing memory explicitly can lead to increased execution overhead, particularly if allocations and deallocations are performed dynamically within performance-critical code paths.

Optimization: USM Memory Pooling. AdaptiveCpp addresses memory allocation overhead by implementing a USM memory pooling strategy. Instead of dynamically allocating and deallocating memory for each operation, AdaptiveCpp preallocates a pool of memory that can be reused across multiple stdpar executions. This significantly reduces allocation latency and prevents fragmentation, improving overall memory management efficiency.

Optimization: Avoiding Redundant Host-Device Transfers. To prevent unnecessary memory copies between the host and device, AdaptiveCpp employs a tracking mechanism to detect when data modifications occur. By transferring only modified data instead of performing full synchronizations, AdaptiveCpp reduces bandwidth overhead and improves execution efficiency. This approach ensures that stdpar operations execute with minimal data movement while maintaining correctness.

Optimization: Efficient Buffer-Accessor Handling. For workloads that rely on SYCL's buffer-accessor model, AdaptiveCpp optimizes buffer lifetimes to avoid unnecessary memory allocations and deallocations. By maintaining buffer reuse policies and ensuring efficient memory access patterns, AdaptiveCpp enhances performance in applications where frequent data movement occurs.

By implementing these optimizations, AdaptiveCpp effectively bridges the gap between stdpar's implicit memory management model and SYCL's explicit memory handling, ensuring that memory operations are executed with minimal overhead in heterogeneous computing environments.

5.3 Synchronization Optimizations

Synchronization is a crucial factor in achieving efficient execution when using SYCL as a backend for stdpar. stdpar assumes implicit synchronization, whereas SYCL enforces explicit synchronization points to ensure correct execution order. Excessive synchronization can introduce performance bottlenecks, leading to unnecessary execution stalls.

Problem: Explicit Synchronization in SYCL. stdpar applications typically execute in a way that abstracts synchronization, allowing the compiler and runtime to handle dependencies. In contrast, SYCL requires explicit use of `sycl::event`

objects, memory fences, and command group dependencies to synchronize execution. This fundamental difference can lead to unnecessary serialization of computations if not optimized properly.

Optimization: Graph-Based Dependency Tracking. AdaptiveCpp optimizes synchronization by implementing a graph-based dependency tracking system. Instead of introducing global barriers after each operation, AdaptiveCpp constructs a task dependency graph that identifies the minimal required synchronization points. This reduces the number of unnecessary synchronization barriers while ensuring execution correctness.

Optimization: Lazy Synchronization. To minimize the overhead introduced by excessive synchronization, AdaptiveCpp applies lazy synchronization techniques. Instead of synchronizing after every operation, AdaptiveCpp delays synchronization until it is absolutely necessary. This approach maximizes parallel execution opportunities while maintaining correctness.

Optimization: Reducing Barrier Overhead. In cases where stdpar's execution model introduces unnecessary memory fences or barriers, AdaptiveCpp eliminates redundant synchronization points. By detecting scenarios where memory consistency does not require immediate enforcement, the compiler removes unnecessary barriers, further improving execution efficiency.

By addressing these synchronization challenges, AdaptiveCpp ensures that stdpar execution in SYCL environments remains highly efficient while maintaining the expected correctness guarantees of the C++ standard library.

5.4 Conditional Offloading

stdpar assumes that execution will take place on the best available hardware resource; however, offloading small workloads to a GPU may introduce excessive overhead, reducing performance instead of improving it. AdaptiveCpp addresses this issue through conditional offloading, which dynamically determines whether a given stdpar workload should be executed on a GPU or remain on the CPU.

Problem: Inefficient Offloading of Small Workloads. Offloading small workloads to a GPU can result in higher latency due to kernel launch overhead and data transfer costs. In many cases, execution on the CPU can be more efficient for smaller task sizes, as it avoids the need to initialize GPU resources and transfer memory.

Optimization: Adaptive Offloading Heuristics. AdaptiveCpp implements a runtime heuristic that predicts the performance impact of offloading based on workload size. If a computation is determined to be too small to benefit from GPU execution, AdaptiveCpp keeps it on the CPU.

Optimization: Performance Monitoring for Offloading Decisions. By tracking execution performance over time, AdaptiveCpp refines its offloading decision-making process. It evaluates historical execution times and dynamically adjusts the threshold at which offloading is beneficial. This ensures that workloads are executed on the most suitable hardware resource.

Through these optimizations, AdaptiveCpp improves overall execution efficiency by ensuring that workloads are allocated to the appropriate execution resource, avoiding unnecessary GPU offloading when it is not beneficial.

6 PERFORMANCE EVALUATION

The performance evaluation of AdaptiveCpp's stdpar backend is based on the benchmarking results provided in Aksel Alpay and Vincent Heuveline's paper *AdaptiveCpp Stdpar: C++ Standard Parallelism Integrated Into a SYCL Compiler* [2].

Due to the lack of a complete independent test setup, this section summarizes the key findings from the original paper and discusses their implications. Readers interested in a detailed description of the experimental methodology, hardware configurations, and benchmarking tools are referred to the original work.

6.1 Comparative Performance Analysis

A key aspect of evaluating AdaptiveCpp's stdpar backend is its performance relative to existing implementations. The experimental results highlight the following comparisons:

Overall Speedup. The execution of stdpar workloads using AdaptiveCpp's SYCL backend achieves notable speedups when compared to traditional CPU execution. Compared to NVIDIA's proprietary `nvc++` stdpar backend, AdaptiveCpp demonstrates competitive performance while maintaining broader hardware compatibility.

Kernel Launch Efficiency. The reduction in kernel launch overhead, facilitated by AdaptiveCpp's instant kernel submission model, results in improved execution times. By minimizing redundant synchronization and event dependencies, kernel execution latency is reduced, leading to a more efficient mapping of stdpar operations to SYCL kernels.

Memory Optimization Benefits. The introduction of USM pooling in AdaptiveCpp significantly reduces memory allocation and transfer overhead. By ensuring that memory is preallocated and reused efficiently, AdaptiveCpp minimizes host-device memory transfer bottlenecks, particularly in workloads that involve frequent memory access.

Synchronization Efficiency. The optimizations in event dependency management and lazy synchronization further contribute to reduced execution stalls. The paper's results confirm that the removal of redundant synchronization barriers improves overall execution throughput without compromising correctness.

6.2 Scalability Analysis

The scalability of AdaptiveCpp's stdpar backend is evaluated across various hardware configurations and dataset sizes. The findings indicate the following trends:

Hardware Scalability. Performance improvements scale effectively across different GPU architectures and CPU-based parallel execution environments. AdaptiveCpp enables seamless execution across heterogeneous devices, demonstrating strong adaptability to varying compute resources.

Workload Scaling. For large workloads, AdaptiveCpp's optimizations ensure sustained performance improvements over traditional CPU execution. However, for small workloads, the effectiveness of GPU offloading is more variable, reinforcing the importance of the conditional offloading heuristic that keeps small computations on the CPU.

6.3 Summary of Performance Findings

The experimental evaluation confirms that AdaptiveCpp's stdpar backend provides an efficient mapping of high-level parallel algorithms onto SYCL's explicit execution model. The key takeaways from the performance analysis include:

- AdaptiveCpp achieves competitive execution performance compared to vendor-specific stdpar implementations while maintaining hardware portability.
- Kernel launch overhead is minimized through optimizations in task submission and event dependency tracking.

- Memory transfer overhead is significantly reduced due to USM pooling and optimized buffer management strategies.
- Synchronization optimizations effectively eliminate redundant barriers, leading to improved execution efficiency.
- Performance scaling across different hardware architectures validates AdaptiveCpp's adaptability to various compute environments.

These findings demonstrate the effectiveness of AdaptiveCpp's approach in bridging the gap between stdpar's high-level abstraction and SYCL's explicit parallel execution model. Future work may further explore additional compiler optimizations to enhance performance in specific workload scenarios.

7 CONCLUSION

This work has explored the integration of C++ Standard Parallelism (stdpar) with SYCL through the AdaptiveCpp compiler, demonstrating how a tightly integrated compiler approach can enable efficient parallel execution on heterogeneous hardware. By leveraging SYCL as a backend, AdaptiveCpp enables stdpar-based applications to execute seamlessly across CPUs, GPUs, and other accelerators, maintaining both performance and portability.

Summary of Findings. The study has shown that AdaptiveCpp's compiler-based approach provides several key advantages:

- **Tighter integration with the compiler provides significant advantages**, allowing for more efficient optimizations compared to purely runtime-based solutions such as NVIDIA's `nvc++`.
- **Optimized execution**, reducing kernel launch overhead, improving memory management, and enhancing synchronization, leading to more efficient parallel execution.
- **Seamless memory handling**, as USM pooling and optimized buffer-accessor usage significantly reduce data transfer overhead and improve execution efficiency.
- **Conditional offloading is a general strategy** that ensures workloads are executed on the most suitable hardware, avoiding inefficient GPU offloading for small computations.

Implications for the Future of Parallel Computing. The success of AdaptiveCpp in integrating stdpar with SYCL highlights several broader implications:

- **Lowering the bar for the use of modern hardware accelerators**, making parallel computing more accessible to developers without specialized hardware knowledge.
- **Enhancing the C++ ecosystem** by demonstrating how SYCL can serve as a robust backend for standardized parallel programming models, fostering greater adoption of heterogeneous computing in modern C++ development.
- **Encouraging future compiler optimizations** that further improve execution performance and hardware adaptability across diverse architectures.

While this work has demonstrated the feasibility of using SYCL as a backend for stdpar, additional optimizations and refinements can further improve execution efficiency and scalability. The following section discusses possible directions for future research and enhancements.

8 EPILOGUE

As we look toward the future of C++ and its capabilities in parallel computing, two forthcoming features stand out: **Senders/Receivers** and **`std::simd`**. These additions aim to enhance asynchronous programming and data-parallel operations, respectively.

8.1 Senders/Receivers: A New Paradigm for Asynchronous Programming

The Senders/Receivers framework introduces a standardized approach to asynchronous operations in C++. This model is built around three primary abstractions:

- **Senders**: Represent asynchronous computations that will eventually produce a result.
- **Receivers**: Define how to handle the outcome of these computations, including successful results, errors, or cancellations.
- **Schedulers**: Manage the execution context for these operations, determining where and when tasks run.

This framework offers several advantages:

- **Composability**: Developers can build complex asynchronous workflows by composing simpler operations.
- **Efficiency**: By providing a unified model, it reduces the overhead associated with managing asynchronous tasks.
- **Extensibility**: The modular nature allows for easy integration with various execution environments and libraries.

For a comprehensive overview, refer to the proposal:

- **P2300R10: `std::execution`**: This document details the design and rationale behind the Senders/Receivers framework [12].

8.2 `std::simd`: Enhancing Data-Parallel Computation

The introduction of `std::simd` brings standardized support for SIMD (Single Instruction, Multiple Data) operations to C++. This feature allows developers to write data-parallel code that can be efficiently executed across different hardware architectures.

Key aspects of `std::simd` include:

- **Portability**: Provides a consistent interface for SIMD operations, ensuring code can run efficiently on various platforms.
- **Performance**: Enables explicit vectorization, allowing developers to harness the full potential of modern CPUs and accelerators.
- **Integration**: Seamlessly integrates with existing C++ codebases, facilitating the adoption of data-parallel programming practices.

For detailed information, consider the following resources:

- **`std::experimental::simd` Documentation**: Offers insights into the experimental implementation and usage of SIMD types in C++ [4].
- **P3024R0: Interface Directions for `std::simd`**: Discusses the design considerations and future directions for the `std::simd` interface [6].

The integration of these features into the C++ standard signifies a substantial advancement in the language's support for parallel and asynchronous programming. As these capabilities become mainstream, developers will be better equipped to write efficient, scalable, and maintainable code for modern computing environments.

REFERENCES

[1] AdaptiveCpp Contributors. Adaptivecpp: Implementation of sycl and c++ standard parallelism for cpus and gpus from all vendors. https://github.com/AdaptiveCpp/AdaptiveCpp. Accessed: 2025-02-02.

[2] Alpay, A., and Heuveline, V. Adaptivecpp stdpar: C++ standard parallelism integrated into a sycl compiler. In *Proceedings of the 12th International Workshop on OpenCL and SYCL* (New York, NY, USA, 2024), IWOCL '24, Association for Computing Machinery.

[3] (AMD), A. M. D. *ROCm Documentation*, 2024. Accessed: 2024-03-01.

[4] cppreference.com. std::experimental::simd, 2023. Accessed: 2024-03-01.

[5] Group, K. S. W. Sycl 2020 rev 4 specification. Standard, Khronos Group, Inc, Beaverton, OR, USA, 2021. Available at: https://www.khronos.org/registry/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf.

[6] Kretz, M. P3024r0: Interface directions for std::simd. Tech. rep., ISO/IEC JTC1/SC22/WG21, 2023.

[7] Meyerovich, L. A., and Rabkin, A. S. Empirical analysis of programming language adoption. *SIGPLAN Not. 48*, 10 (Oct. 2013), 1–18.

[8] Munshi, A., Gaster, B., Mattson, T. G., Fung, J., and Ginsburg, D. *OpenCL Programming Guide*, 1st ed. Addison-Wesley Professional, 2011.

[9] NVIDIA. *C++ Parallel Algorithms*, 2020.

[10] NVIDIA. *CUDA C Programming Guide*, 2024. Accessed: 2024-03-01.

[11] Reinders, J., Ashbaugh, B., Brodman, J., Kinsner, M., Pennycook, J., and Tian, X. *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems Using C++ and SYCL*. Apress, 2020.

[12] Shoop, K., Baker, L., Howes, L., Garland, M., Niebler, E., Lelbach, B. A., Brown, G., Avermann, M. J., Domani, H., Vitale, M., and Wong, M. P2300r10: std::execution, 2024. ISO/IEC JTC1/SC22/WG21 Proposal.

[13] V'yukova, N., Galatenko, V., and Samborskii, S. Support for parallel and concurrent programming in c++. *Programming and Computer Software 44* (2018), 35–42.