



Porting LULESH to SYCL: practical insights and performance analysis

Wenyuan Liang¹ · Hengzhong Liang¹ · Han Huang¹ · Xianwei Zhang¹

Received: 12 August 2025 / Accepted: 13 April 2026 / Published online: 8 September 2026
© China Computer Federation (CCF) 2026

Abstract

High-performance computing (HPC) systems are increasingly heterogeneous, requiring developers to balance platform-specific optimizations with code portability. SYCL (a standard, single-source C++ programming model) addresses this by abstracting hardware diversity into single-source code that can run across CPUs, GPUs, and other accelerators. Therefore, porting a classical HPC benchmark to SYCL and assessing the challenges, limitations, and performance are essential to understand the practical implications of adopting SYCL in real-world scientific applications. While previous porting work has largely focused on using automated tools like the Intel DPC++ Compatibility Tool (DPCT), less attention has been paid to the practical challenges of manual porting. In this paper, we evaluate the portability and performance of SYCL by porting the LULESH (Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics) proxy application from CUDA to SYCL. This study explores two migration strategies: the automated approach using DPCT and a manual code refactoring approach. We provide insights into the difficulties encountered during the porting process and the corresponding solutions. Performance results demonstrate that both SYCL implementations of LULESH achieve competitive performance compared to the original CUDA version on single-GPU configurations. We compare oneAPI and AdaptiveCpp using a manually refactored LULESH, and observe notable performance differences. Additionally, when running on CPUs with Message Passing Interface (MPI) across multiple processes, the SYCL versions achieve an average speedup of approximately $3.7\times$ over the OpenMP baseline. In contrast, initial multi-GPU tests reveal significant MPI overheads, indicating that efficient scaling across GPUs remains a challenge.

Keywords Performance evaluation · SYCL · OneAPI · AdaptiveCpp

1 Introduction

With the gradual decline of Moore's Law (Tuomi 2002), heterogeneous accelerators have become the mainstream trend in the field of HPC. The architectural diversification is clearly reflected in the November 2025 edition of the TOP500 list (Top500 and Org 2025). The top four systems

exemplify this vendor heterogeneity: El Capitan and Frontier (ranked first and second respectively) employ AMD accelerators, while Aurora and JUPITER Booster (ranked third and fourth) utilize Intel and NVIDIA accelerators. Meanwhile, these systems require various heterogeneous programming languages and module support. The diversity of accelerators poses significant challenges for developers of scientific computing applications. Developers need to refactor, optimize, and debug code for different hardware architectures to ensure high computational efficiency across various platforms.

Programming languages and models that support cross-platform compatibility offer a promising solution. Several frameworks have emerged to facilitate vendor-agnostic heterogeneous computing, including OpenCL (Stone et al. 2010), OpenMP (Dagum and Menon 1998), and SYCL (Howes and Rovatsou 2015). Although OpenCL is among the most popular frameworks employed in heterogeneous programming (Kim et al. 2015), its low-level programming

✉ Xianwei Zhang
zhangxw79@mail.sysu.edu.cn

Wenyuan Liang
liang79@mail2.sysu.edu.cn

Hengzhong Liang
lianghzh23@mail2.sysu.edu.cn

Han Huang
huangh367@mail2.sysu.edu.cn

¹ School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou 510006, Guangdong, China

model can be challenging, especially for less experienced developers. In contrast, SYCL is a modern programming model that introduces high-level abstractions and supports single-source programming, enabling developers to write unified code for both host and device. By leveraging standard C++ and its extensive libraries, SYCL reduces the learning effort and enables developers to concentrate on parallelism rather than language complexity.

Despite these advantages, more practical porting and evaluation efforts are still needed to validate SYCL's effectiveness in real-world HPC applications. Previous porting work (Weckert et al. 2023; Christgau and Steinke 2020; Costanzo et al. 2024) related to SYCL mainly used the DPCT (Intel Corporation 2025) tool provided by oneAPI (Intel Corporation 2025) for porting, optimization, and evaluation. One limitation of using DPCT for SYCL porting is its dependence on oneAPI-specific libraries, which may hinder portability and prevent successful compilation with other SYCL implementations like AdaptiveCpp (Alpay and Heuveline 2024). In order to achieve better compatibility across platforms, manual porting based on the SYCL 2020 standard enables applications to run on a wider range of SYCL implementations.

To this end, we explore key aspects of SYCL development from a developer's perspective, including porting, debugging, and performance evaluation. As a representative case study, we select LULESH, a widely adopted benchmark in high-performance computing that models simplified shock physics and is commonly used for evaluating performance portability across platforms. We port the CUDA version of LULESH to SYCL using two methods: an automated DPCT-based migration (referred to as LULESH-DPCT) and manual code rewriting (referred to as LULESH-BUF). LULESH-BUF and LULESH-DPCT use different memory management models. LULESH-BUF uses the buffer-accessor memory model, which relies on `sycl::buffer` to enable implicit data synchronization between the host and device. LULESH-DPCT adopts the USM (Unified Shared Memory) model, a pointer-based memory management approach introduced in SYCL 2020. During the porting and debugging process, we share common challenges developers may face and provide practical solutions. After LULESH is ported, performance evaluation becomes a key focus. We conduct tests on a selection of GPUs and CPUs from different vendors, analyze the performance differences, and explore optimizations to achieve better performance.

The contributions of this paper are as follows:

- We present a comprehensive analysis of the challenges encountered during the porting of LULESH from

CUDA to SYCL, detailing the issues arising from both DPCT-based automated migration and manual porting.

- We perform a comparative performance analysis of the LULESH-BUF implementation across heterogeneous architectures from multiple vendors.
- We investigate the impact of different compiler optimization levels on LULESH-BUF across various platforms and compare the performance of two SYCL implementations: Intel oneAPI DPC++ and AdaptiveCpp.
- We analyze the performance characteristics of the SYCL-based LULESH under MPI multi-process execution, identifying potential scalability bottlenecks and communication overheads.

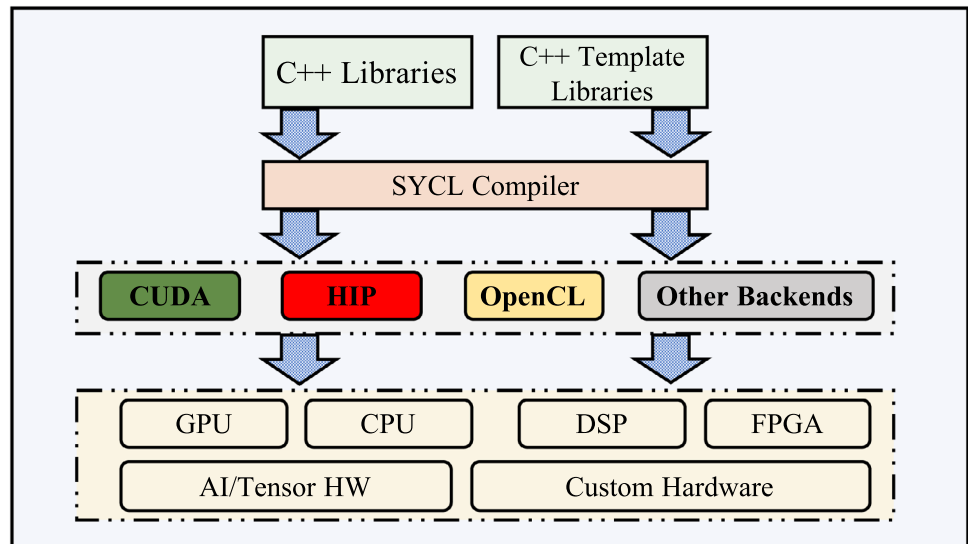
The structure of this paper is as follows: In Sect. 1, we provide a brief introduction to the entire paper. In Sect. 2, we introduce SYCL, the LULESH application, and the experimental background, highlighting the context and significance of our research. In Sect. 3, we describe the process of porting LULESH from CUDA to SYCL, outlining the challenges and issues encountered during both automated and manual porting, along with the solutions implemented. In Sect. 4, we conduct a comprehensive performance evaluation of the ported LULESH. In Sect. 5, we discuss related works, and finally, in Sect. 6, we conclude with a summary and suggestions for future work.

2 Background

2.1 SYCL

SYCL is an open standard developed by the Khronos Group that provides a single-source, cross-platform programming model for heterogeneous computing. It aims to achieve performance close to that of the native programming models while improving code portability and developer productivity. As shown in Fig. 1, SYCL serves as a high-level abstraction layer that enables C++ applications to target heterogeneous hardware platforms through a unified programming model. It supports various backends including CUDA, HIP, OpenCL, and others, while maintaining compatibility with standard C++ libraries. Through these different backends, SYCL allows single-source code to be executed across a wide range of hardware devices, including GPUs, CPUs, ARM processors, and more, thus achieving true cross-platform portability.

SYCL implementations Currently, several SYCL implementations are available, including industry-ready implementations like DPC++ and AdaptiveCpp, as well as research-oriented ones such as triSYCL (Cabellos, et al. 2025), neoSYCL (Tohoku University Takizawa Laboratory

Fig. 1 A simplified SYCL overview**Listing 1** An example of SYCL USM and Buffer-Accessor Pattern

```

1  // Initialize the device and queue
2  sycl::device dev {sycl::gpu_selector_v};
3  sycl::queue q(dev);
4  // Unified Shared Memory (USM) example
5  int* data = sycl::malloc_shared<int>(1024, q);
6  q.parallel_for(1024, [=](sycl::id<1> idx) {
7      data[idx] = idx; // Direct pointer access
8  }).wait();
9  sycl::free(data, q); // Deallocate USM memory
10
11 // Buffer-Accessor example
12 sycl::buffer<int, 1> buf {sycl::range<1>(1024)};
13 q.submit([&(sycl::handler& cgh) {
14     sycl::accessor acc {buf, cgh,
15         sycl::range<1>(1024),
16         sycl::read_write};
17     cgh.parallel_for(1024, [=](sycl::id<1> idx) {
18         acc[idx] *= 2; // Access via accessor
19     });
20 }).wait();
  
```

2025), and SimSYCL (Celerity and Team 2025). Notably, DPC++ and AdaptiveCpp stand out as the most widely adopted. DPC++ is Intel's SYCL implementation that conforms to the SYCL 2020 standard and serves as the main compiler within the oneAPI ecosystem. It supports different backends: CUDA for NVIDIA GPUs, HIP for AMD GPUs, and OpenCL for CPUs and other devices. AdaptiveCpp is an independent and open-source project maintained by the University of Heidelberg. It supports different ways to compile SYCL code. In practice, it relies on an LLVM-based toolchain to generate device code, with the exact compilation mode depending on the selected backend and build configuration. DPC++ and AdaptiveCpp represent alternative SYCL implementations, with DPC++ being the default toolchain in Intel's oneAPI ecosystem, while AdaptiveCpp provides a lightweight, backend-flexible solution. By comparing both implementations, we evaluate SYCL's performance portability across diverse platforms. *Memory management* Initially, SYCL adopted the buffer-accessor model for memory

management through automatic dependency tracking. By leveraging a directed acyclic graph (DAG) of requirements, this model reduces the developer's burden in cross-platform development and ensures data integrity without manual synchronization. The SYCL 2020 specification introduces enhanced features such as USM, improved kernel programming flexibility, and support for sub-groups, making it a robust choice for modern HPC (Deakin and McIntosh-Smith 2020) and artificial intelligence (AI) (Tanvir et al. 2022) workloads. Listing 1 shows an example of the USM and the buffer-accessor pattern. In the USM model, `malloc_shared` allocates memory accessible by both host and device, whereas `sycl::buffer` objects require explicit accessors. While buffers excel in simplifying memory orchestration, the industry has increasingly pivoted towards USM to prioritize performance-first optimization. USM provides direct pointer access and fine-grained control over data movement, granting developers the freedom to implement complex optimizations that are often constrained by

the higher-level buffer abstraction. Consequently, the selection of a memory model reflects a strategic trade-off: the buffer-accessor model favors development productivity and portability, while USM is preferred for squeezing maximum hardware efficiency through manual management.

SYCL porting workflow Figure 2 illustrates how a CUDA application, after being ported to SYCL, is abstracted and managed by different SYCL implementations. Subsequently, the code is mapped to various programming models such as CUDA, HIP, and OpenCL, enabling cross-platform execution from a single source code base. DPCT, also known as SYCLomatic, is a powerful tool designed to assist developers in porting the majority of CUDA application code to SYCL. It provides extensive mapping libraries for CUDA APIs, significantly reducing the time and effort required for code migration. However, as part of Intel's oneAPI ecosystem, DPCT often generates SYCL code that depends on libraries specific to DPC++, which may lead to compatibility issues with other SYCL implementations such as AdaptiveCpp. In contrast, manually ported versions typically avoid vendor-specific dependencies and adhere closely to the SYCL 2020 standard, making them more portable across different SYCL implementations. Therefore, for the migration of our benchmark application LULESH, we adopt both approaches to facilitate a comprehensive comparison.

2.2 LULESH

Complex scientific and engineering challenges require detailed physical modeling of materials, particularly in the context of multi-material hydrodynamics simulations. The physical modeling captures the deformation of materials under interactions between forces, effectively reflecting their physical states and playing a critical role in scientific computation. As an HPC benchmark, LULESH is widely

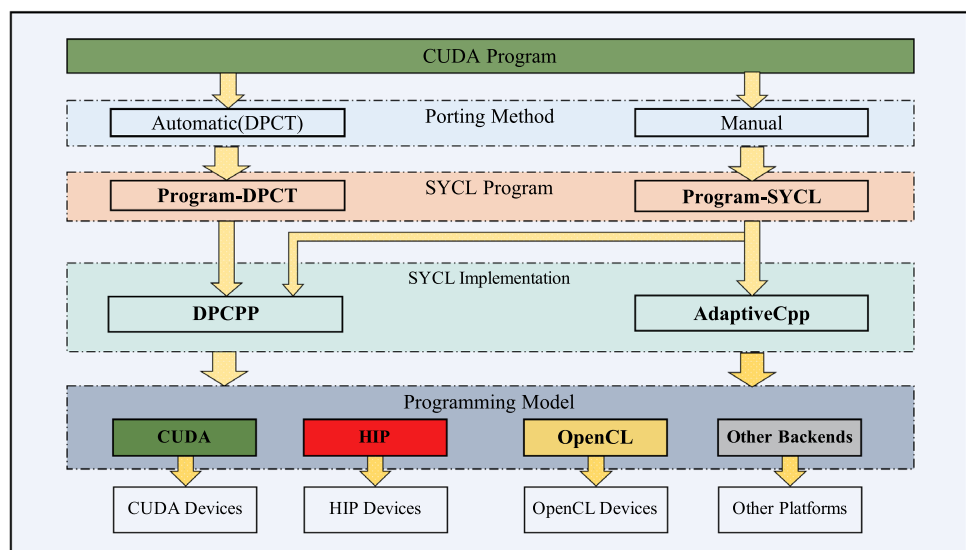
recognized as a representative proxy application in the field of HPC, particularly valuable for evaluating the performance, scalability, and efficiency of emerging HPC architectures and programming models. It is a representative problem in shock hydrodynamics, originally developed as one of the five challenge problems by Lawrence Livermore National Laboratory (LLNL). Over time, LULESH has become a widely studied proxy application in the U.S. Department of Energy's (DOE) exascale co-design efforts.

As a proxy application for shock hydrodynamics, LULESH is specifically designed to solve the Sedov blast problem. This problem involves analyzing the changes in physical states of a material caused by the shock from an explosion at a point in a uniform medium, as well as studying the propagation of the resulting shock wave. While LULESH is a simplified proxy application, it captures representative programming patterns, numerical algorithms, and data flow characteristics commonly found in scientific computing. Its modular structure, portability, and well-understood computational behavior make it a suitable benchmark for evaluating SYCL's applicability in heterogeneous HPC environments.

2.3 Motivation

Recent research efforts have explored the portability and performance of SYCL across heterogeneous platforms, often focusing on benchmark suites and micro-kernel evaluations. These studies provide valuable insights into compiler maturity, language compliance, and performance portability. However, they tend to emphasize simplified code patterns and do not fully capture the complexity of real-world HPC applications. In contrast, many large-scale scientific applications in HPC rely on MPI-based distributed-memory parallelism and involve complex communication and

Fig. 2 Two approaches to porting a CUDA program to SYCL



memory management patterns. Evaluating SYCL in such contexts is essential to assess its practical viability beyond benchmarks. To this end, we select LULESH, a widely used proxy application in the HPC community, as our case study. Unlike existing SYCL versions that exclude MPI support, we port LULESH from its original CUDA implementation using a combination of automated tools and manual code refactoring. This approach retains the key design features of the original code, such as asynchronous MPI communication and explicit memory management, providing a structurally faithful port to SYCL.

As a mature and versatile benchmark, LULESH has been developed across various programming models and optimized for multiple platforms since its inception. In addition to our CUDA-based SYCL port, LULESH includes versions for CPUs using OpenMP, as well as GPU-based implementations using CUDA, OpenCL, and AMP (Lawrence Livermore National Laboratory 2025). Beyond these implementations, the benchmark suite HeCBench developed by Jin and Vetter (2023) includes a SYCL version of LULESH. This version was ported from the OpenMP implementation. Although it retains the host-side communication code, the MPI communication code in the main program has been removed. In contrast, the original CUDA version of LULESH employs asynchronous communication. We ported LULESH to SYCL from its original CUDA implementation, using both the DPCT tool and manual code rewriting. By preserving the original CUDA logic, this porting approach enables a performance-oriented SYCL implementation that closely reflects real-world HPC practices.

3 Experiences with porting

In this section, we present the procedure of porting LULESH from CUDA to SYCL. This work aims to evaluate SYCL's performance and practicality in a scientifically

relevant benchmark. We choose LULESH, a widely used benchmark in the HPC community, as a representative case. We first describe the preparatory steps before the porting begins. Then, we explain two porting approaches in detail: automatic porting using DPCT and manual porting. Each approach is discussed separately, including its method, challenges, and solutions.

3.1 Experimental setup

Hardware We evaluate performance across heterogeneous systems from three major vendors: NVIDIA, AMD, and Intel. For single-GPU benchmarks, the following accelerators are employed: NVIDIA A100 (40GB) and V100, AMD MI100, and Intel Arc A770. To evaluate multi-GPU scalability, we employ clusters equipped with NVIDIA A100 (80 GB) and H100 GPUs, where each compute node contains 8 GPUs interconnected via NVLink. These specific clusters are selected because only they provide the scale necessary to accommodate LULESH's requirement for cubic numbers of MPI ranks (e.g., 8, 27, 64). In this paper, we report results with 8 ranks.

Software The SYCL implementations are compiled using Intel oneAPI 2024.2.0 and AdaptiveCpp 24.10.0 (built with LLVM 17.0.0). For performance benchmarking, the original CUDA implementation serves as the GPU baseline, while the native OpenMP version is used for CPU baseline comparisons. The complete configuration details of the systems are provided in Table 1.

Porting workflow As Fig. 3 shows, the porting workflow of LULESH from CUDA to SYCL includes two distinct paths. After analyzing the original CUDA code to understand the computational structure and data flow of LULESH, we proceed with the main porting process, which consists of five key steps: First, we perform profiling and identify computational hotspots to ensure the porting effort focuses on the most performance-critical regions. This enables more effective performance optimization during later stages. Next, we identify CUDA-specific APIs and begin adapting the code from CUDA to SYCL. This includes converting data structures, such as replacing `thrust::device_vector` with `sycl::buffer`. We then refactor the codebase to adapt to SYCL's single-source programming style, adjusting the kernel launch mechanisms and memory access patterns. After that, we carry out debugging and validation to ensure the numerical correctness of the results. Finally, we tune performance and test the application across different hardware platforms to assess both portability and stability. Notably, when using DPCT for porting, the process can skip directly to the fourth step of the major porting workflow. Compared to manual porting, this saves a significant amount of time but requires careful verification to ensure

Table 1 System hardware and software information

Host	Devices	Software stack
Intel Xeon Platinum 8358P	/	Ubuntu 22.04.3 LTS, oneAPI 2024.2.0, OpenMP 4.5, OpenMPI 5.0
Intel Xeon Gold 6132	4× NVIDIA V100 SXM2 16GB	CentOS 7.6, CUDA 11.2, Intel MPI 2021.13
Intel Xeon Platinum 8462Y+	8× NVIDIA H100 HBM3 80GB	Ubuntu 22.04.3 LTS, CUDA 12.2, OpenMPI 5.0
Intel Xeon Gold 6348	8× NVIDIA A100 SXM4 80GB	Ubuntu 22.04.3 LTS, CUDA 12.2, OpenMPI 5.0
AMD EPYC 7742	4× NVIDIA A100 PCIe 40GB	Debian 11, CUDA 12.2, Intel MPI 2021.13
AMD EPYC 7302	2× AMD Instinct MI100 32GB	Ubuntu 22.04.3 LTS, HIP 5.5.0, AdaptiveCpp 24
AMD Ryzen 5 5600	1× Intel Arc A770 16GB	Debian 12, oneAPI 2024.2.0

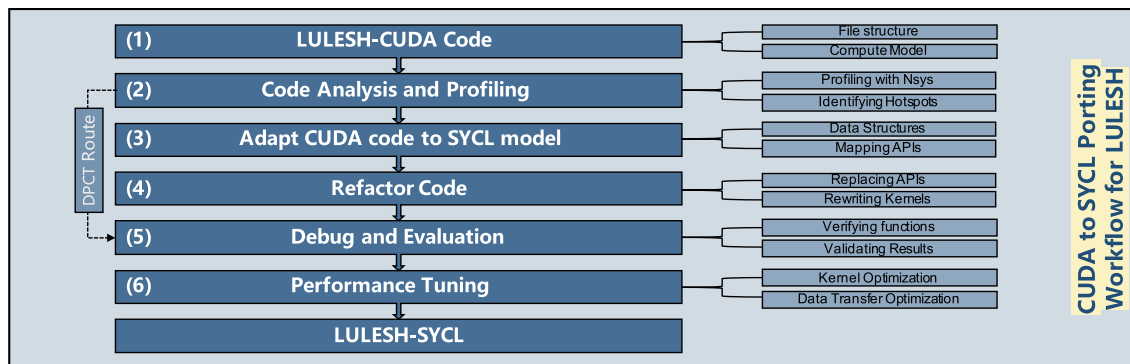


Fig. 3 The porting workflow of LULESH from CUDA to SYCL

the correctness of the ported code. A detailed discussion of the complete porting experience is provided in Sects. 3.2 and 3.3.

3.2 Automated migration with DPCT

DPCT offers an efficient solution for accelerating CUDA-to-SYCL migration, especially for developers who are unfamiliar with SYCL's memory model and programming paradigms. As a result, we begin the porting procedure of LULESH with the DPCT tool for automated migration. DPCT provides several options to customize the porting process based on the needs of the application. For example, developers can choose 1D or 3D work-groups depending on algorithmic structure. It also supports optional optimizations to improve performance. In addition, DPCT includes experimental features to handle more complex cases and give developers more flexibility. For LULESH, DPCT successfully automates the porting of about 90% of the CUDA code. Although a large portion of the code can be automatically ported, careful debugging and validation are still required to ensure the correctness and completeness of the migrated SYCL implementation. After porting, DPCT provides diagnostic messages to help identify and fix possible issues. These diagnostics are categorized into three primary classes as follows:

Warnings of potential runtime errors These warnings identify potential issues such as excessive local variable usage or work-group sizes that could result in register overflow. They also include prompts related to memory management and data synchronization, which require careful attention to prevent runtime errors. For example, due to the large number of parameters in LULESH's compute kernels, we encounter warnings related to work-group sizes exceeding device limits and local variable sizes surpassing hardware constraints. These issues highlight the need to adjust work-group and sub-group sizes to comply with device-specific limitations, which correspond to the concepts of threads and warps in CUDA. To avoid such issues

during LULESH's execution, we query device limits (e.g., maximum work-group size, supported sub-group sizes, and local memory limits) during initialization and adjust kernel launch parameters accordingly.

Alerts for mandatory manual intervention These warnings indicate areas where DPCT cannot directly migrate the code. In these cases, the tool provides suggestions to guide developers in making manual modifications for debug and fine-tuning. For example, in LULESH, there is a function called `cuda_init()` that performs a series of CUDA device checks and initializations. DPCT raises a warning indicating that this function might cause the application to fail in selecting the desired device. To address this, we refactor the function and add a CPU/GPU selector, allowing the correct device to be chosen during compilation. Such issues often lead to application failures or computational errors, so careful adjustments are necessary.

Guidance for performance optimization DPCT offers guidance on alternative functions and optimization strategies. However, it is important to note that these suggestions may not always bring improved performance and should be evaluated based on the specific context of the application. For example, in the function that computes the minimum time step in LULESH, a barrier operation is required. DPCT recommends using `item.barrier(sycl::access::fence_space::local_space)` instead of the default `item::barrier()` when the kernel is multidimensional. In our porting process, we evaluated both barrier options and observed no significant impact on LULESH's performance under the tested configurations. While DPCT offers helpful guidance and warnings during migration, it does not automatically handle all edge cases. Therefore, manual inspection, validation, and targeted performance testing remain necessary to ensure correctness and efficiency.

3.3 Manual migration

While DPCT can significantly reduce the time required for porting, careful validation is still necessary to ensure

correctness. During debugging, it can be difficult to determine whether issues are caused by incorrect porting or by the behavior of DPCT's support libraries. We encountered this issue during large-scale MPI multi-GPU runs, where computation errors are difficult to trace back to a specific source. As a result, only the manually ported version is usable for our MPI multi-GPU tests, since it allowed for more reliable debugging and validation. In contrast, manual porting offers full control over the code structure and logic, allowing for more precise handling of correctness and facilitating easier debugging in complex scenarios such as MPI communication.

Advantages Although automated migration significantly reduces manual porting effort, the DPCT-generated code's dependence on Intel-specific libraries creates implementation incompatibility. This results in compilation failures when using AdaptiveCpp, whereas manually ported code

maintains cross-platform compatibility by strictly adhering to the SYCL 2020 standard without extra libraries. Meanwhile, manual porting enables finer manual tuning of the program. While the migration tool provides a reliable syntactic translation from CUDA to SYCL, it is not designed to exploit the performance optimizations that arise from the unique semantic characteristics of the SYCL programming model. For example, in CUDA, kernel launches are typically configured using thread and block dimensions, while in SYCL, work-items and work-groups are used to define parallel execution. This semantic difference allows for more flexibility in optimizing memory access patterns and load balancing when porting manually, but a migration tool might overlook such opportunities, leading to suboptimal performance.

```

1 if (CONDITION) {
2     srcAddr = &commDataRecv[MSGSIZE]; // host data
3     d_srcAddr = &d_commDataRecv[MSGSIZE]; // device data
4     MPI_Wait(&recvRequest[MSGSIZE], &status); // wait for MPI receive
5
6     // SYCL: async memcpy and kernel launch
7     event evt = q.memcpy(d_srcAddr, srcAddr, DATASIZE);
8     q.submit([&](handler& cgh){
9         cgh.depends_on(evt); // wait for memcpy
10        accessor dest {*DEST, cgh, write_only};
11        cgh.parallel_for(nd_range<1>{
12            range<1>(DIMSIZE*BLOCK), range<1>(BLOCK)},
13            [=](nd_item<1> item) {
14                Kernel<0>(d_srcAddr, &DEST[0], ...);
15            });
16    });
17 }

```

Listing 2 SYCL memory transfer and kernel launch

```

1 if (CONDITION) {
2     srcAddr = &commDataRecv[MSGSIZE]; // host data
3     d_srcAddr = &d_commDataRecv[MSGSIZE]; // device data
4     MPI_Wait(&recvRequest[MSGSIZE], &status); // wait for MPI receive
5
6     // CUDA: async memcpy and kernel launch
7     cudaMemcpyAsync(d_srcAddr, srcAddr,
8                     DATASIZE, cudaMemcpyHostToDevice, stream);
9     Kernel<0><<<DIMSIZE, BLOCK, 0, stream>>>{
10         d_srcAddr, &DEST[0], ...);
11 }

```

Listing 3 CUDA memory transfer and kernel launch

Memory management As Listings 2 and 3 illustrate, the memory transfer and kernel launch processes in SYCL differ from those in CUDA. After the completion of MPI communication, data is transferred from the host to the device. In SYCL, this is performed using a `memcpy` followed by a `parallel_for` kernel, with an explicit dependency between them. In CUDA, the corresponding operations are implemented with `cudaMemcpyAsync` and a kernel launch. These examples demonstrate how both models express asynchronous data movement and execution. SYCL and CUDA adopt different strategies for memory management, which require corresponding adjustments when porting LULESH. A key challenge in manual porting is managing memory allocation. In the CUDA version of LULESH, the `thrust` library is used, employing `host_vector` and `device_vector` for allocating memory on the host and device, respectively. This design mandates explicit data transfers between host and device memory. In the manually migrated SYCL version, we use `sycl::buffer` as a unified data structure for both the host and device. The `sycl::buffer` data structure is a core abstraction in SYCL, offering an efficient mechanism for managing data memory and automatically handling data movements between the host and device. This greatly simplifies the data transfer code compared to the CUDA implementation. In our SYCL implementation, `sycl::buffer` objects are created once during initialization (e.g., `new sycl::buffer<Real_t>(range)`), and then reused across iterations to avoid repeated buffer construction overhead. Experimental observations reveal that frequent `new` operator calls within computational loops incur substantial performance overhead. Therefore, memory allocation operations are predominantly relocated to the initialization phase to mitigate runtime latency.

Challenges in data synchronization While the buffer-accessor model in SYCL simplifies memory management through automatic data movement and dependency tracking, it can introduce challenges in applications involving MPI communication or frequent data exchanges. In such scenarios, automatic synchronization may conflict with explicit communication patterns, potentially leading to race conditions or computational errors if not carefully managed. To mitigate these issues, we employ SYCL events to coordinate `memcpy` and `copy` operations in communication routines. This approach enables asynchronous execution while ensuring proper synchronization, avoiding the need for global `wait()` calls that would otherwise stall the entire queue and degrade performance. Furthermore, we explicitly annotate accessors with `read_only`, `write_only` and `read_write` modes to clarify data access patterns. These annotations help the SYCL runtime infer dependency relationships and optimize buffer synchronization accordingly. For debugging and fine-tuning, we introduce optional

`wait()` calls after each communication-related kernel, controlled via compile-time flags. This mechanism facilitates correctness verification and debugging, without affecting asynchronous behavior in optimized runs.

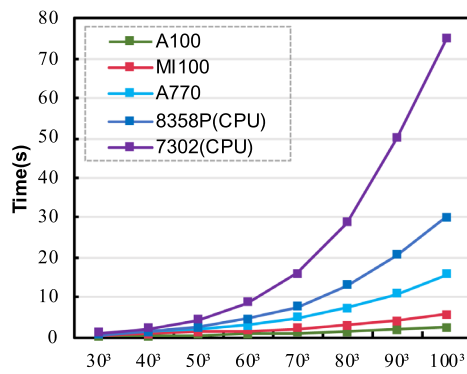
4 Performance results

In this section, we present an analysis of the experimental results, focusing on two versions of LULESH implemented in SYCL: the manually ported LULESH-BUF and the automatically ported LULESH-DPCT. The experimental evaluation is structured into three primary phases. Initially, we verify the cross-platform compatibility of SYCL on heterogeneous hardware. To delve deeper into performance, we compare the SYCL-port LULESH against its original CUDA implementation, followed by a detailed cross-compiler analysis (DPC++ and AdaptiveCpp) and an assessment of optimization strategies. Finally, we extend our scope to distributed environments through MPI-based experiments, highlighting key challenges encountered in high-performance multi-process execution.

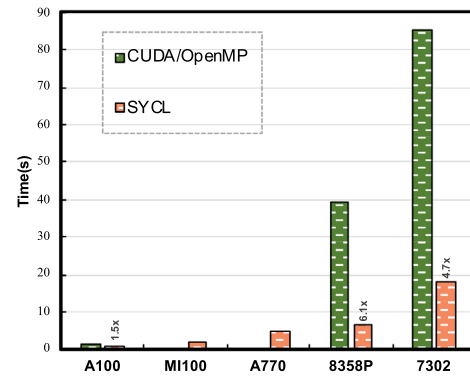
4.1 Cross-platform

After completing the manual porting process, we conduct cross-platform experiments of the DPC++ SYCL implementation on CPUs and GPUs from different vendors. Since the Intel A770 GPU does not support double-precision floating point (FP64), we use single-precision (FP32) to ensure consistency. As shown in Fig. 4a, computational efficiency decreases progressively across all devices as the problem size increases, due to cubic grid-size growth. On the GPU side, the A100 delivers the best performance, with average speedups of $2.29\times$ over the MI100 and $4.82\times$ over the A770. As the mesh size increases, the performance advantage of the A100 becomes more pronounced. At larger mesh sizes (100^3), the gap widens to $2.33\times$ and $6.40\times$, respectively. On the CPU side, using a single process with 32 cores, an Intel Xeon 8358P CPU outperforms the AMD EPYC 7302 CPU by an average of $2.06\times$, peaking at $2.48\times$ for a 100^3 grid.

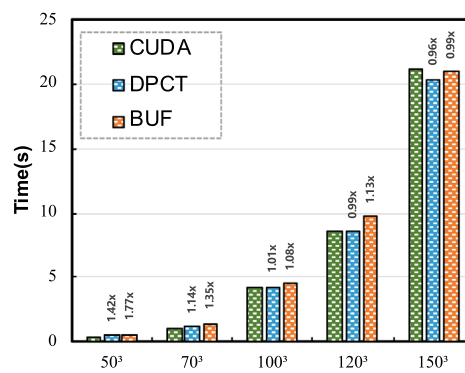
To further evaluate the performance portability of the manual SYCL implementation, Fig. 4b compares its execution time against native baselines (CUDA on the A100 GPU and OpenMP on CPU) across different hardware platforms with a fixed grid size of 70^3 . The baseline comparisons for the MI100 and A770 are not illustrated in the figures, as no publicly available native implementations exist to provide a direct performance reference. Specifically, on the A100 GPU, the SYCL implementation achieves a $1.5\times$ speedup over the native CUDA baseline. Most notably, on



(a) Cross-platform performance of LULESH-SYCL using FP32 precision.



(b) Performance comparison across different platforms (70^3 grid size, FP32).



(c) SYCL-to-CUDA execution time ratio on the NVIDIA A100 as a function of problem size (FP64).

Fig. 4 Performance evaluation of LULESH-SYCL across heterogeneous platforms

the $\times 86_64$ CPU platform (Intel Xeon 8358P and AMD EPYC 7302), the SYCL version significantly outperforms the OpenMP baseline by $6.1\times$ and $4.7\times$, respectively.

In this cross-platform evaluation, we deliberately refrain from tuning sub-group or work-group sizes for specific hardware platforms to preserve the single-source nature of the SYCL implementation. Instead, we use the original `blocksize` settings from the CUDA version without any changes. While this ensures fair and consistent comparisons across different devices, it may cause lower performance on some hardware that prefers different group sizes. For example, the NVIDIA A100 GPU natively supports a sub-group size of 32, while the AMD MI100 supports 64, and the Intel A770 supports 16. Consequently, the observed performance differences may reflect not only hardware capability, but also the degree of alignment between the original CUDA execution configuration and the architectural preferences of each backend. As a result, the performance may not be optimal on every platform. This shows a trade-off between

keeping the code portable and getting the optimal performance on each device.

4.2 Performance comparison

In this section, we present a series of performance comparison experiments. First, we evaluate and compare the performance of the three versions of LULESH under varying mesh sizes: LULESH-DPCT, LULESH-BUF and the original CUDA version. Next, we analyze the performance differences between two SYCL implementations, DPC++ and AdaptiveCpp, on both the A100 and MI100 GPUs. Finally, we study how different compiler optimization levels affect performance. These experiments allow us to comprehensively assess how SYCL porting affects the performance of LULESH and to explore the performance tuning opportunities across different platforms and compiler configurations.

4.2.1 CUDA and SYCL

An important aspect of SYCL porting is evaluating whether the ported code can achieve performance comparable to the original CUDA version. We benchmark all three versions of LULESH (CUDA, DPCT, and BUF) across various grid sizes to assess their relative performance using double-precision arithmetic. To ensure fairness and eliminate the impact of numerical precision differences, we compile all versions using the `-O3` optimization level and disable any precision-affecting compiler flags (such as `fast-math`). In the A100 GPU test shown in Fig. 4c, the performance gap between LULESH-DPCT, LULESH-BUF, and the native CUDA version narrows as the grid size increases. For grid sizes larger than 100^3 , both SYCL versions achieve similar performance to that of CUDA. When the grid size reaches 160^3 , LULESH-DPCT and LULESH-BUF slightly outperform the native implementation, with execution times of approximately $0.96\times$ and $0.99\times$ that of the CUDA version, respectively. In the following section, we compare the performance of LULESH-BUF under DPC++ and AdaptiveCpp across various platforms.

4.2.2 DPC++ and AdaptiveCpp

In this experiment, we use the abbreviations DPCPP (Intel oneAPI DPC++) and ACP (AdaptiveCpp) when referring to these SYCL implementations. We initially attempt to compile the LULESH-DPCT using ACP, but it fails due to conflicts with various DPCT-specific support libraries that LULESH-DPCT depends on. These libraries are tightly coupled with the Intel SYCL implementation and are not compatible with ACP. When using DPCT for porting, we tend to choose DPCPP as the SYCL implementation rather than ACP, since DPCT is part of the oneAPI ecosystem.

Therefore, in this comparative experiment, we use the manually ported version, LULESH-BUF. Since the porting is based on the SYCL 2020 standard, the code runs correctly on both DPCPP and ACP. By default, the DPC++ compiler `icpx` enables `fast-math` and `fp-contract=fast`, which may affect numerical precision. In contrast, ACP does not enable these flags by default. Therefore, to ensure a fair comparison, we explicitly disable these options when conducting performance experiments.

As shown in Table 2, DPCPP and ACP exhibit noticeable performance differences on the A100 and MI100 GPUs. We begin with the experiments conducted on the A100 GPU. When using the same compiler optimization level (`-O3`), DPCPP achieves an average speedup of $2.53\times$ over ACP across various mesh sizes. Notably, even with `-O1`, DPCPP outperforms ACP compiled with `-O3`. However, this trend differs on the MI100 GPU. When using the same `-O3` optimization level, ACP begins to run faster than DPCPP for grid sizes larger than 64^3 . The performance gap grows as the mesh size increases, achieving a peak speedup of $1.08\times$ for a mesh size of 160^3 . This discrepancy may be attributed to differences in kernel scheduling and execution strategies between the two implementations. Research has shown (Breyer et al. 2024) that DPCPP's data-parallel kernels generally outperform those of ACP, largely due to its highly optimized compiler and runtime environment. However, ACP demonstrates efficient performance when combined with hierarchical data-parallel kernels, especially on specific hardware configurations such as AMD GPUs. This is consistent with our results, where ACP shows increasing advantages on the MI100 with larger grid sizes, indicating its kernel strategy may better suit AMD hardware.

Therefore, when choosing between DPCPP and ACP, we must carefully weigh their trade-offs based on the specific application requirements and hardware environment.

Table 2 Elapsed time of LULESH-BUF with DPC++ and AdaptiveCpp at different optimization levels and mesh sizes. The numerical suffixes in `-O1`, `-O2`, and `-O3` denote the respective compiler optimization levels. Time is measured in seconds

Mesh size	32x32x32	48x48x48	64x64x64	96x96x96	128x128x128	160x160x160
NVIDIA A100						
CUDA	0.15	0.34	0.81	3.67	10.96	27.07
DPCPP-O1	0.29	0.46	0.81	3.76	11.25	27.66
DPCPP-O2	0.27	0.43	0.75	3.64	10.83	26.55
DPCPP-O3	0.27	0.43	0.75	3.55	10.88	26.63
ACPP-O1	0.61	1.75	4.64	21.57	67.17	163.88
ACPP-O2	0.59	1.76	4.76	22.27	69.20	170.33
ACPP-O3	0.37	0.84	2.37	10.55	31.83	77.01
AMD MI100						
DPCPP-O1	0.88	2.12	5.27	23.03	69.44	171.11
DPCPP-O2	0.85	1.98	4.96	20.90	62.81	152.02
DPCPP-O3	0.85	1.99	4.92	20.63	62.21	151.13
ACPP-O1	0.93	2.13	5.20	20.75	62.15	150.11
ACPP-O2	0.93	2.07	4.92	19.40	57.70	139.57
ACPP-O3	0.94	2.13	4.90	19.82	57.66	139.38

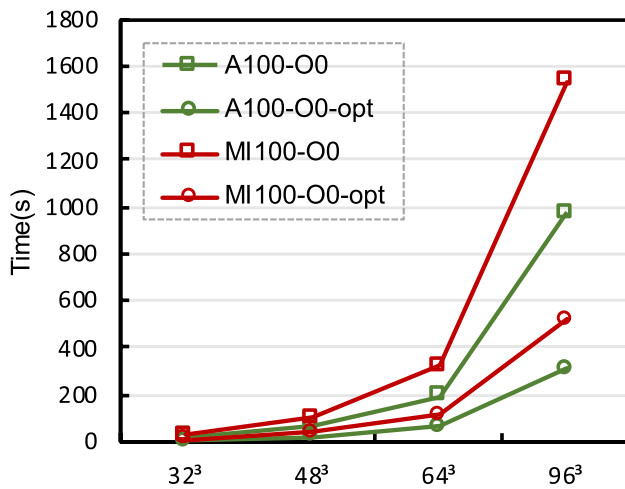


Fig. 5 Performance evaluation of LULESH-BUF on an A100 GPU and MI100 GPU across compiler optimization levels (`-O0`, `-O0-opt`) for problem sizes ranging from 32^3 to 96^3 . (`opt` is the `fsycl-optimize-non-user-code` compiler flag)

Table 3 Performance comparison between `O1` and `O2` optimization levels in `AdaptiveCpp` on A100, V100, and MI100

Mesh size	64x64x64	96x96x96	128x128x128	160x160x160
NVIDIA A100				
ACPP-O1	4.64	21.57	67.17	163.88
ACPP-O2	4.76	22.27	69.20	170.33
NVIDIA V100				
ACPP-O1	9.95	44.65	135.11	327.94
ACPP-O2	10.65	46.87	144.16	351.46
AMD MI100				
ACPP-O1	5.20	20.75	62.15	150.11
ACPP-O2	4.92	19.40	57.70	139.57

We now turn to evaluating how compiler optimization levels and specific flags impact performance across these implementations.

4.2.3 Compiler optimization

SYCL, as a high-level abstraction layer, enables cross-platform compilation to diverse device backends. However, this flexibility often requires the compiler to generate efficient code, making compiler optimization an important factor in SYCL program performance. Therefore, this experiment evaluates how SYCL compiler behavior influences LULESH's performance on heterogeneous hardware platforms. To investigate the impact of compiler optimizations on LULESH's performance, we conduct experiments comparing two scenarios: (1) all optimizations disabled (`-O0`), (2) `-O0` with selective non-user code optimizations enabled via the `-fsycl-optimize-non-user-code` (refer to `opt`) flag.

As shown in Fig. 5, single-GPU runtime measurements on A100 and MI100 devices are performed across varying

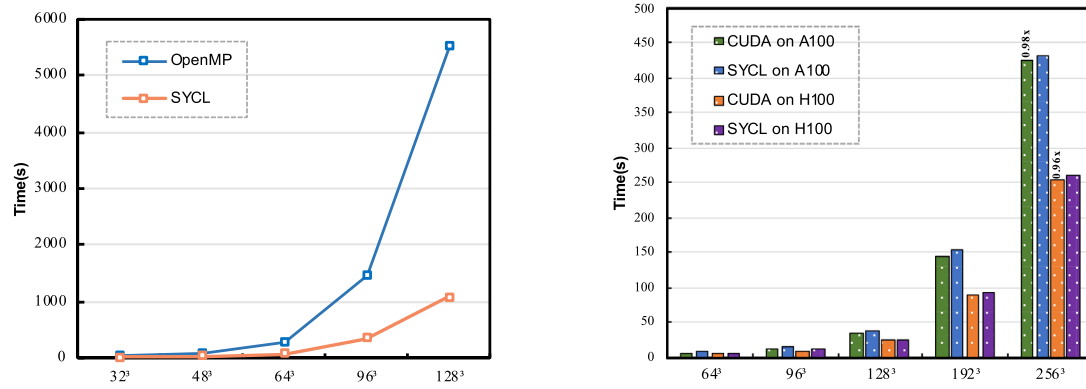
grid sizes to evaluate these configurations. Under the `opt` flag (`O0-opt`), the A100 and MI100 GPUs achieve average speedups of $3.11\times$ and $2.66\times$, respectively, over the baseline `O0` configuration. Although `O0-opt` offers a significant performance improvement over plain `-O0`, a substantial gap remains compared to the `O1` optimization level. For example, under a grid size of 96^3 on the A100 GPU, the `-O1` flag completes in approximately 3.67 s, while `O0` and `O0-opt` require 972.77 s and 310.21 s, respectively. This illustrates the considerable impact of compiler optimization levels on performance.

The comparison of standard optimization levels (`-O1`, `-O2`, `-O3`) is presented in Table 2. As indicated by the results, MI100 GPUs benefit from increasing optimization levels across both DPCPP and ACPP implementations. For the A100 GPU, the ACPP shows a slight performance regression when moving from `O1` to `O2`, with an average slowdown of 3.4% for grid sizes larger than 96^3 . To further examine this behavior, Table 3 presents performance measurements on the V100 GPU under both `O1` and `O2` optimization levels. Similar to the results on the A100, a performance degradation is observed when compiling with `O2`, compared to `O1`. However, this result is not observed on the MI100. Since both A100 and V100 use the CUDA backend, this suggests that the `O2` optimization level in `AdaptiveCpp` may introduce unfavorable optimizations for CUDA GPU.

Overall, this comparative analysis reveals that the SYCL compiler's optimization mechanisms exert a critical influence on application performance. When compiler optimizations are disabled, SYCL kernels may experience reduced computational throughput, limiting their ability to fully leverage the underlying hardware capabilities.

4.3 MPI communication

Setup While the preceding evaluations focus exclusively on single-process executions, real-world HPC applications often operate in distributed-memory environments. To assess communication performance of our SYCL implementation, this section presents a multi-process evaluation of the manually ported LULESH-BUF version using MPI. Our experimental setup employs 8 MPI processes. On the CPU side (Intel Xeon 8358P), each process is bound to a single physical core; thus 8 processes use 8 cores. The SYCL and OpenMP baselines use the same core budget for fairness. All multi-GPU experiments are conducted on a single compute node equipped with 8 NVIDIA GPUs (A100-80GB or H100). To maximize intra-node communication efficiency, the system leverages a fully interconnected NVLink fabric, providing high-bandwidth, low-latency point-to-point (P2P) data transfers between all GPU pairs.



(a) MPI performance comparison between LULESH-BUF and the OpenMP-based LULESH on the Intel Xeon 8358P CPU

(b) MPI scalability of LULESH-BUF on an NVIDIA A100 and H100 GPU cluster.

Fig. 6 MPI-based performance evaluation of LULESH-BUF on multi-process CPUs and multi-GPU clusters

This configuration significantly benefits LULESH by accelerating the frequent halo exchanges required for boundary synchronization, thereby mitigating communication bottlenecks often encountered with standard PCIe interconnects. Notably, we utilize CUDA-aware OpenMPI 5.0 for all GPU evaluations to enable direct device-to-device communication. Each MPI process is pinned to a dedicated GPU and a specific physical CPU core to ensure optimal data locality and minimize resource contention. This configuration represents the standard high-performance building block of modern GPU clusters, ensuring that the communication-heavy halo exchanges in LULESH are evaluated under optimal hardware interconnect conditions. *Evaluation results* We first evaluate the CPU performance, as shown in Fig. 6a, comparing the OpenMP version of LULESH with the manually ported LULESH-BUF under five grid sizes. At the smallest grid size (32³), LULESH-BUF already outperforms the OpenMP version by about 2.4 $\times$. As the grid grows, the advantage becomes more pronounced. For the 128³ case, the speedup reaches roughly 5 $\times$. Averaged over all five problem sizes in the MPI multi-process experiments, LULESH-BUF is about 3.7 $\times$ faster than the OpenMP baseline. We repeated the experiments and confirmed that both versions delivered stable execution times across runs. However, the performance gap narrows significantly during multi-GPU MPI evaluations. As illustrated in Fig. 6b, LULESH-CUDA initially exhibits a 1.8 $\times$ performance advantage over LULESH-BUF at the 32³ scale on the A100; however, this performance gap narrows significantly as the grid size increases. Specifically, the performance overhead of LULESH-BUF relative to the CUDA baseline decreases from 19.6% at 96³ to 6.3% at 192³, eventually converging to near-parity at 256³, where LULESH-BUF achieves 98% of CUDA's performance (0.98 $\times$ runtime ratio). A

consistent trend is observed on the H100, where the initial 1.73 $\times$ speedup at the smallest scale diminishes to a marginal 1.03 $\times$ at 256³.

To further analyze the reasons for the performance difference, we profile the 64³ and 128³ grid sizes. At the 64³ scale, a stark contrast emerges in the execution profile: the time-reduction kernel accounts for a mere 2% of the aggregate kernel execution time in the CUDA implementation, yet it surges to 22.6% in the SYCL version. We carefully compared the implementations of both versions. While CUDA exploits implicit warp-level synchronization via volatile shared memory, the SYCL port incurs substantial instruction stalls due to mandated barriers at every reduction stage. Furthermore, the reliance on the buffer-accessor model introduces non-trivial host-side latency during command group ingestion. These fixed overheads dominate the execution profile at lower grid sizes (32³). However, as the computational intensity scales cubically with the problem size, these runtime-induced overheads are progressively amortized. This transition is clearly visible at the 128³ grid size, where the kernel distribution of both implementations begins to converge. At this scale, the top three kernels by execution time are identical for both versions, and the proportion of the time-reduction kernel in SYCL drops to 3.1%, closely approaching the 2.6% observed in the native CUDA implementation.

Challenges LULESH's MPI communication follows a halo exchange pattern, involving frequent boundary data transfers between neighboring processes. In the original CUDA implementation, memory transfers are handled manually using proxy data structures, and separate buffers are used for sending and receiving data. After porting to SYCL, LULESH adopts the buffer-accessor model for memory management. On the device side, `accessors` are used to

access data in device memory. The buffer-accessor model helps the runtime build a dependency graph between kernels, enabling asynchronous scheduling. To maintain data consistency between host and device, `wait()` must be inserted at appropriate points. Alternatively, `host_accessors` can be used to access `sycl::buffer` contents on the host, introducing implicit synchronization. Although the combination of `buffer` and `accessors` simplifies memory management and kernel scheduling, it also weakens explicit control over host-device synchronization. This may lead to inconsistent data states when used alongside MPI, where precise control of communication timing is essential.

To address these challenges, careful memory management and synchronization strategies are necessary when using `buffer` in MPI-intensive scenarios. Analyzing the complexity of data communication and selectively using `accessors` helps reduce unnecessary synchronization overhead. Compared to the buffer-accessor model, USM offers finer control through methods like `malloc_device` and `malloc_shared`. This makes it more suitable for managing memory explicitly in MPI-based GPU communication, helping to avoid synchronization issues and improve performance. Although both DPCPP and ACPP have adopted USM as a preferred alternative to the buffer-accessor model since SYCL 2020, its use remains optional rather than mandatory. Future work could further investigate the trade-offs between these memory models in large-scale, communication-intensive HPC applications, especially under different SYCL implementations and backend configurations.

5 Related work

LULESH is widely used as a benchmark in HPC research (Ferreira and Levy 2020; Levy and Ferreira 2018; Liu and Kulkarni 2015). It represents a typical scientific computing application and is available in various versions developed with different programming models. While most research efforts focus on optimizing performance for specific hardware platforms without considering cross-platform compatibility, some studies have explored novel approaches to enhance LULESH's versatility.

Singanaboina et al. (2024) ported the OpenMP version of LULESH to HPX and HPXMP, leveraging the parallel features of HPX to enhance LULESH's performance. Their work includes performance testing on CPUs from different vendors. Alpay and Heuveline (Alpay and Heuveline 2024) used the AdaptiveCpp compiler to offload the OpenMP version of LULESH onto accelerators for heterogeneous computing. The offloading mechanism utilizes vendor-specific compilers to enable execution on hardware from different

vendors, allowing for flexibility in deploying the application across various devices.

With the increasing diversity of heterogeneous hardware and the introduction of the automated migration tools such as DPCT, many studies have started utilizing DPCT to migrate applications to SYCL. For instance, Weckert et al. (2023) used DPCT to port the Altis Benchmarking Suite to SYCL and conducted a series of tests. Similarly, Rangel et al. (2023) migrated a large cosmology application to SYCL and evaluated its performance on GPUs from three different vendors. Nguyen et al. (2023) focused on porting and optimizing batched iterative solvers for Intel GPUs using SYCL, demonstrating significant performance gains over CUDA on NVIDIA GPUs. However, the optimizations rely on the oneAPI ecosystem and are not tested on other SYCL implementations like AdaptiveCpp. Additionally, Thoman et al. (2022) studied compilation times across different SYCL implementations, combining large-scale tests with real-world application validation.

6 Conclusion and future work

In this work, we port LULESH from CUDA to SYCL using two approaches: automated porting and manual porting. We share the challenges encountered during both processes and the solutions we develop to address them. After completing the porting, we conduct a comprehensive performance evaluation of the two SYCL versions of LULESH. Experimental results demonstrate that both LULESH-DPCT and LULESH-BUF exhibit performance portability in single-GPU experiments when compared to the original CUDA implementation. However, achieving this portability requires careful optimization, as we observe that the SYCL compiler's behavior significantly impacts application performance during the debugging process. Our evaluation shows that the SYCL implementations demonstrate different optimization characteristics across platforms. The choice between DPC++ and AdaptiveCpp requires careful consideration of application requirements and target hardware environments. Moreover, delving into the optimization levels and options of the compiler can aid in enhancing program performance.

Beyond single-device execution, our multi-GPU analysis via MPI reveals that for workloads mirroring LULESH's computational motifs, the performance delta between CUDA and SYCL is largely dictated by problem scale, synchronization overheads, and the efficiency of the computational kernels. As these factors are intrinsically linked to the underlying computing and communication patterns of a specific application, our results should be interpreted

as a case-specific observation instead of a universal benchmark. To generalize these findings, subsequent studies must explore a wider spectrum of algorithmic behaviors and massive-scale architectures, thereby providing a holistic assessment of SYCL's potential in modern, large-scale production environments.

Author contributions Xianwei Zhang managed the project. Wenyan Liang was responsible for the porting, debugging, and performance evaluation of the application. Hengzhong Liang and Han Huang contributed to the design and implementation of the experimental setup. Wenyan Liang and Xianwei Zhang analyzed the data. The first draft of the manuscript was written by Wenyan Liang and all authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

Funding No funds, grants, or other support was received.

Data availability statement The source code and experimental data supporting this study are available from the corresponding author upon reasonable request.

Declarations

Conflict of interest The authors have no financial or proprietary interests in any material discussed in this article.

Ethics approval and consent to participate Not applicable.

References

- Alpay, A., Heuveline, V.: Adaptivecpp stdpar: C++ standard parallelism integrated into a sycl compiler. In : Proceedings of the 12th International Workshop on OpenCL and SYCL, pp 1–12 (2024)
- Breyer, M., Van Craen, A., Pflüger, D.: Evaluation of sycl's different data parallel kernels. In : Proceedings of the 12th International Workshop on OpenCL and SYCL, pp 1–4 (2024)
- Cabellos, R., et al.: triSYCL: SYCL implementation based on standard C++. <https://github.com/triSYCL/triSYCL>. Accessed: 2025-06-04 (2025)
- Celerity SYCL Team: SimSYCL: A functional SYCL runtime simulator. <https://github.com/celerity/SimSYCL>. Accessed: 2025-06-04 (2025)
- Christgau, S., Steinke, T.: Porting a legacy cuda stencil code to oneapi. In : 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp 359–367 (2020). IEEE
- Costanzo, M., Rucci, E., García-Sánchez, C., Naiouf, M., Prieto-Matías, M.: Assessing opportunities of sycl for biological sequence alignment on gpu-based systems. *J. Supercomput.* **80**(9), 12599–12622 (2024)
- Dagum, L., Menon, R.: Openmp: an industry standard api for shared-memory programming. *IEEE Comput. Sci. Eng.* **5**(1), 46–55 (1998)
- Deakin, T., McIntosh-Smith, S.: Evaluating the performance of hpc-style sycl applications. In : Proceedings of the International Workshop on OpenCL, pp 1–11 (2020)
- Ferreira, K.B., Levy, S.: Evaluating mpi message size summary statistics. In: Proceedings of the 27th European MPI Users' Group Meeting, pp 61–70 (2020)
- Howes, L., Rovatsou, M.: Sycl integrates opencl devices with modern c++. Khronos Group (2015)
- Intel Corporation: SYCLomatic: Migrating CUDA to SYCL. <https://github.com/oneapi-src/SYCLomatic>. Accessed: 2025-06-04 (2025)
- Intel Corporation: Intel oneAPI Toolkits. <https://www.intel.cn/content/www/cn/zh/developer/tools/oneapi/base-toolkit.html>. Accessed: 2025-06-04 (2025)
- Jin, Z., Vetter, J.S.: A benchmark suite for improving performance portability of the sycl programming model. In : 2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp 325–327 (2023). IEEE
- Kim, J., Dao, T.T., Jung, J., Joo, J., Lee, J.: Bridging opencl and cuda: a comparative analysis and translation. In : Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp 1–12 (2015)
- Lawrence Livermore National Laboratory: LULESH: Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics. <https://asc.lnl.gov/codes/proxy-apps/lulesh>. Accessed: 2025-06-04 (2025)
- Levy, S., Ferreira, K.B.: Using simulation to examine the effect of mpi message matching costs on application performance. In : Proceedings of the 25th European MPI Users' Group Meeting, pp 1–11 (2018)
- Liu, C., Kulkarni, M.: Optimizing the lulesh stencil code using concurrent collections. In : Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, pp 1–10 (2015)
- Nguyen, P., Nayak, P., Anzt, H.: Porting batched iterative solvers onto intel gpus with sycl. In : Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, pp 1048–1058 (2023)
- Rangel, E.M., Pennycook, S.J., Pope, A., Frontiere, N., Ma, Z., Madan-anth, V.: A performance-portable sycl implementation of crk-hacc for exascale. In : Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, pp 1114–1125 (2023)
- Singanaboina, S.Y., Wei, W., Tsaousis Seiras, I., Syskakis, P., Richardson, B., Cook, B., Kaiser, H.: Accelerating lulesh using hpx-the c++ standard library for parallelism and concurrency. In : Practice and Experience in Advanced Research Computing 2024: Human Powered Computing, pp 1–8 (2024)
- Stone, J.E., Gohara, D., Shi, G.: Opencl: A parallel programming standard for heterogeneous computing systems. *Comput. Sci. Eng.* **12**(3), 66 (2010)
- Tanvir, M., Narasimhan, K., Goli, M., El Farouki, O., Georgiev, S., Ault, I.: Towards performance portability of ai models using sycl-dnn. In : Proceedings of the 10th International Workshop on OpenCL, pp 1–3 (2022)
- Thoman, P., Molina Heredia, F., Fahringer, T.: On the compilation performance of current sycl implementations. In : Proceedings of the 10th International Workshop on OpenCL, pp 1–12 (2022)
- Tohoku University Takizawa Laboratory: neoSYCL: A SYCL implementation for HPC platforms. <https://github.com/Tohoku-University-Takizawa-Lab/neoSYCL>. Accessed: 2025-06-04 (2025)
- TOP500.org: November 2025 TOP500. Accessed: 2025-11 (2025) <https://www.top500.org/lists/top500/2025/11/>
- Tuomi, I.: The lives and death of Moore's law. First Monday (2002). <https://doi.org/10.5210/fm.v7i11.1000>
- Weckert, C., Solis-Vasquez, L., Oppermann, J., Koch, A., Sinnen, O.: Altis-sycl: Migrating altis benchmarking suite from cuda to sycl for gpus and fpgas. In : Proceedings of the SC'23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis, pp 547–555 (2023)

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.



Wen yuan Liang received the B.S. degree in Applied Physics from Huazhong University of Science and Technology in 2022, and the M.S. degree from Sun Yat-sen University in 2026, where he was with arcSYSU (ARCHitecture and SYStem Upscaling @ SYSU) under the supervision of Assistant Professor Xianwei Zhang. His research interests include high-performance computing and scientific workflows.



Hengzhong Liang received the B.S. degree in Computer Science and Technology from Sun Yat-sen University, Guangzhou, China, in 2023 and the M.S. degree in Computer Technology from Sun Yat-sen University, Guangzhou, China, in 2026. His research interests include HPC and Computer Architecture.



acceleration architectures.



computing systems.

Han Huang is currently a Ph.D. candidate with the School of Computer Science and Engineering, Sun Yat-sen University, co-advised by Associate Professor Xianwei Zhang and Professor Yutong Lu. He received the B.S. degree in Computer Science and Technology from Sun Yat-sen University in 2023. His research interests include high-performance computing and matrix

Xianwei Zhang is currently an Associate Professor with the School of Computer Science and Engineering at Sun Yat-sen University. He received the Ph.D. degree in Computer Science from University of Pittsburgh in 2017. Prior to joining Sun Yat-sen University in 2020, he worked at AMD Inc. (2017-2020) and Nvidia Corporation (2016). His research interests include computer architecture, high-performance computing, and intelligent