SYCLOPS

Deliverable 4.2 – Compiler with auto-vectorization

GRANT AGREEMENT NUMBER: 101092877

Project acronym:	**SYCLOPS**
Project full title:	**Scaling extreme analYtics with Cross architecture acceLeration based on OPen Standards**
Call identifier:	**HORIZON-CL4-2022-DATA-01-05**
Type of action:	**RIA**
Start date:	**01/01/2023**
End date:	**31/12/2025**
Grant agreement no:	**101092877**

D4.2 – Compiler with auto-vectorization

Executive Summary: This deliverable outlines the work done in "Task 4.1: Compiler support for RISC-V" in WP4. Significant improvements have been to the two compiler toolchains developed in SYCLOPS, namely, DPC++ and AdaptiveCPP. On the DPC++ front, DPC++ and oneAPI Construction Kit were adapted to support both native building and cross-compilation for RISC-V platforms. Integration with the latest LLVM and OCK compiler pipeline enabled support for RISC-V Vector extensions version 1.0 (RVV 1.0), ensuring SYCLOPS use cases are deployable on RVV targets. On the AdaptiveCPP front, generic Single-Pass (SSCP) JIT compiler was improved making AdaptiveCpp the only SYCL implementation with this unified JIT compilation capability. A crucial new OpenCL runtime backend was added enabling compatibility with the OCK required for RISC-V hardware. Extensive JIT-time optimizations were introduced resulting in AdaptiveCpp outperforming NVCC and DPC++ by 30% /23% in geometric mean. Finally, PCUDA was recently introduced to allow for mixing of SYCL and CUDA/HIP code.

WP:	4		
Author(s):	Uwe Dolinsky, Kumudha Narasimhan, Aksel Alpay		
Editor:	Raja Appuswamy		
Leading Partner:	UHEI		
Participating Partners:	CPLAY		
Version:	1.0	**Status:**	Draft
Deliverable Type:	Other	**Dissemination Level:**	PU
Official Submission Date:	06-Oct-2025	**Actual Submission Date:**	30-Sep-2025

Disclaimer

The SYCLOPS consortium consists of the following partners:

No.	Partner Organisation Name	Partner Organisation Short Name	Country
1	EURECOM	EUR	**FR**
2	INESC ID - INSTITUTO DE ENGENHARIADE SISTEMAS E COMPUTADORES, INVESTIGACAO E DESENVOLVIMENTO EM LISBOA	INESC	**PT**
3	RUPRECHT-KARLS-UNIVERSITAET HEIDELBERG	UHEI	**DE**
4	ORGANISATION EUROPEENNE POUR LA RECHERCHE NUCLEAIRE	CERN	**CH**
5	HIRO MICRODATACENTERS B.V.	HIRO	NL
6	ACCELOM	ACC	FR
7	CODASIP S R O	CSIP	CZ
8	CODEPLAY SOFTWARE LIMITED	CPLAY	UK

Document Revision History

Version	Description	Contributions
0.1	Structure and outline	EUR
0.2	DPC++ technical contributions	CPLAY
0.3	AdaptiveCPP technical contributions	UHEI
1.0	Final draft	EUR

Authors

Author	Partner
Uwe Dolinsky	CPLAY
Kumudha Narasimhan	CPLAY
Aksel Alpay	UHEI

Reviewers

Name	Organisation
Aleksandar Ilic	INESC
Vincent Heuveline	UHEI
Danilo Piparo	CERN
Nimisha Chaturvedi	ACC
Martin Bozek	CSIP

Statement of Originality

This deliverable contains original unpublished work except where clearly indicated otherwise. Acknowledgement of previously published material and of the work of others has been made through appropriate citation, quotation or both.

Table of Contents

Executive Summary

This deliverable outlines the work done in “Task 4.1: Compiler support for RISC-V” in WP4 of the SYCLOPS project. The central objective of this task was developing SYCL compilers to facilitate the execution of SYCL applications on RISC-V and other accelerators provided by the SYCLOPS infrastructure layer. This has been achieved with significant improvements in the two compiler toolchains developed in SYCLOPS, namely, DPC++ and AdaptiveCPP.

On the DPC++ front, DPC++ and oneAPI Construction Kit were adapted to support both native building and cross-compilation for RISC-V platforms. Integration with the latest LLVM and OCK compiler pipeline enabled support for RISC-V Vector extensions version 1.0 (RVV 1.0), ensuring SYCLOPS use cases are deployable on RVV targets like the A730 core deployed in SYCLOPS EMDC v2.0. oneTBB was successfully integrated into the DPC++ NativeCPU device, allowing for powerful thread scheduling with NUMA support on multi-socket CPU platforms.

On the AdaptiveCPP front, generic Single-Pass (SSCP) JIT compiler, which is now the default, was substantially improved. This unified infrastructure allows a single compilation to generate a binary capable of dispatching to various devices (AMD GPUs, NVIDIA GPUs, and SPIR-V for OpenCL/RISC-V/OCK), making AdaptiveCpp the only SYCL implementation with this unified JIT compilation capability. A crucial new OpenCL runtime backend was added to AdaptiveCpp to target OpenCL devices using SPIR-V, enabling compatibility with the oneAPI Construction Kit required for RISC-V hardware. Extensive JIT-time optimizations were introduced, including a unified two-level kernel cache and the introduction of adaptivity levels. By leveraging these JIT-time opportunities, AdaptiveCpp was optimized to the point where it typically outperforms vendor compilers (e.g., outperforming CUDA by 30% and oneAPI by 23% in geometric mean). Finally, AdaptiveCpp 25.02 introduced PCUDA, an implementation of the CUDA and HIP programming languages within the generic JIT compiler. This allows for mixing SYCL and CUDA/HIP code while retaining the full cross-architecture portability of AdaptiveCpp's SYCL compilation.

1. Introduction

Figure 1 shows the SYCLOPS hardware-software stack consists of three layers: (i) infrastructure layer, (ii) platform layer, and (iii) application libraries and tools layer.

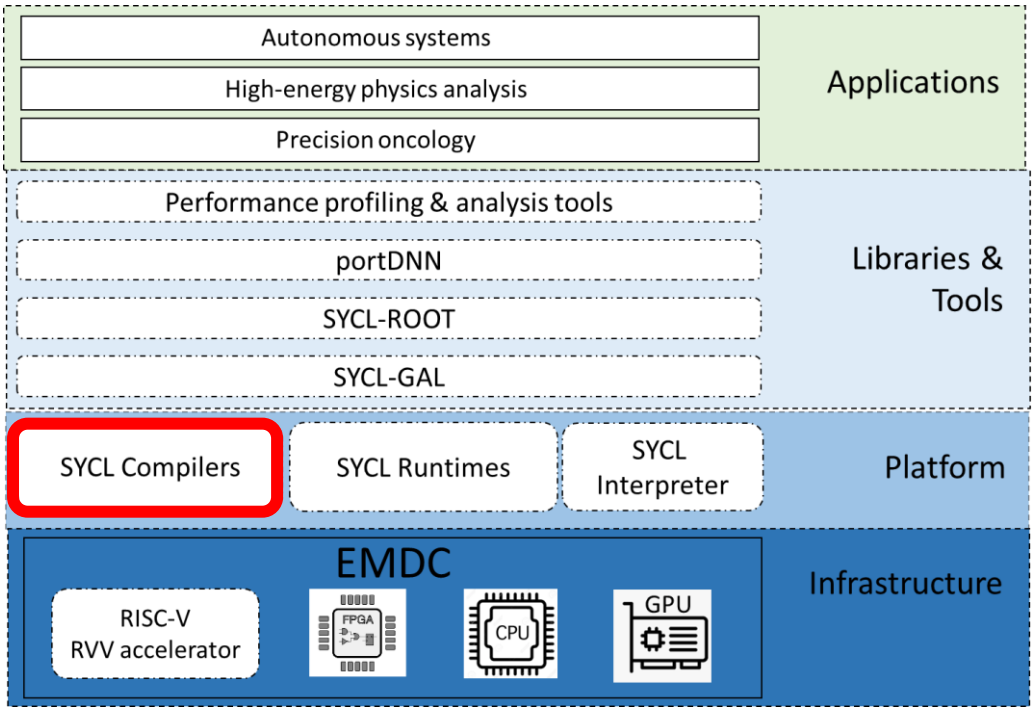


Figure 1. SYCLOPS architecture

Infrastructure layer: The SYCLOPS infrastructure layer is the bottom-most layer of the stack and provides heterogeneous hardware with a wide range of accelerators from several vendors.

Platform layer: The platform layer, provides the software required to compile, execute, and interpret SYCL applications over processors in the infrastructure layer. The second layer from the bottom, the platform layer, provides the software required to compile, execute, and interpret SYCL applications over processors in the infrastructure layer. SYCLOPS will contain oneAPI DPC++ compiler from CPLAY, and AdaptiveCpp, formerly known as hipSYCL, an open-source SYCL compiler toolchain from UHEI. In terms of SYCL interpreters, SYCLOPS will contain Cling from CERN.

Application libraries and tools layer: The libraries layer enables API-based programming by providing pre-designed, tuned libraries for various deep learning methods for the PointNet autonomous systems use case (SYCL-DNN), mathematical operators for scalable HEP analysis (SYCL-ROOT), and data parallel algorithms for scalable genomic analysis (SYCL-GAL).

This deliverable concerns the **SYCL compilers** part of the stack as highlighted in Figure 1, and covers the work done on the two compiler toolchains in the context of "*Task 4.1: Compiler support for RISC-V"* in *WP4* (M3-M33). This deliverable is a summary of this work with a special focus on (i) support for RISC-V vector hardware available in the SYCLOPS platform at M33 especially the oneAPI Construction Kit, and (ii) support for other advanced functionalities that can support applications beyond the scope of SYCLOPS. All functionalities described in this deliverable have already been merged and integrated in several public releases of DPC++ and AdaptiveCpp that have been made during the project. Both compiler toolchains are available as open source software for maximal transparency, reproducibility and maintainability.

This deliverable is structured as follows. Section 1 of this deliverable provides a high-level overview of the overall SYCLOPS architecture and positions this deliverable with respect to both components in the SYCLOPS stack and WP/tasks in the work plan. Section 2 describes updates to DPC++ and oneAPI. Section 3 describes updates to AdaptiveCpp.

2. DPC++ and oneAPI Construction Kit

After the official launch of project SYCLOPS, Intel acquired our partner CPLAY. As a result, CPLAY's proprietary ACORAN ComputeCPP compiler, originally described in the SYCLOPS proposal, was discontinued. It has since been replaced with the open-source DPC++ compiler, which now serves as one of the SYCL implementations used within SYCLOPS. DPC++ is a modern, open-source SYCL compiler and runtime supported by a significantly larger developer and user community. Together, DPC++ and SYCL form the foundation of oneAPI—an open, cross-architecture programming model that enables developers to maintain a single codebase across diverse accelerator platforms such as GPUs and FPGAs.

With the rapid growth of AI adoption, hardware vendors increasingly design specialized AI processors optimized for inference and/or training, delivering better efficiency than standard off-the-shelf hardware. However, these custom processors often present challenges for developers, as they typically require porting software to proprietary and non-standard programming models. To address this, CPLAY developed the oneAPI Construction Kit (OCK), which extends the benefits of oneAPI and SYCL to new and custom hardware. OCK evolved from ComputeAorta—part of the ACORAN toolchain originally created by CPLAY—and is designed to unlock the full performance of heterogeneous hardware while providing developers with standards-compliant interfaces.

The following diagram shows how OCK currently makes it possible to add new devices so that they can make use of the DPC++ SYCL compiler.

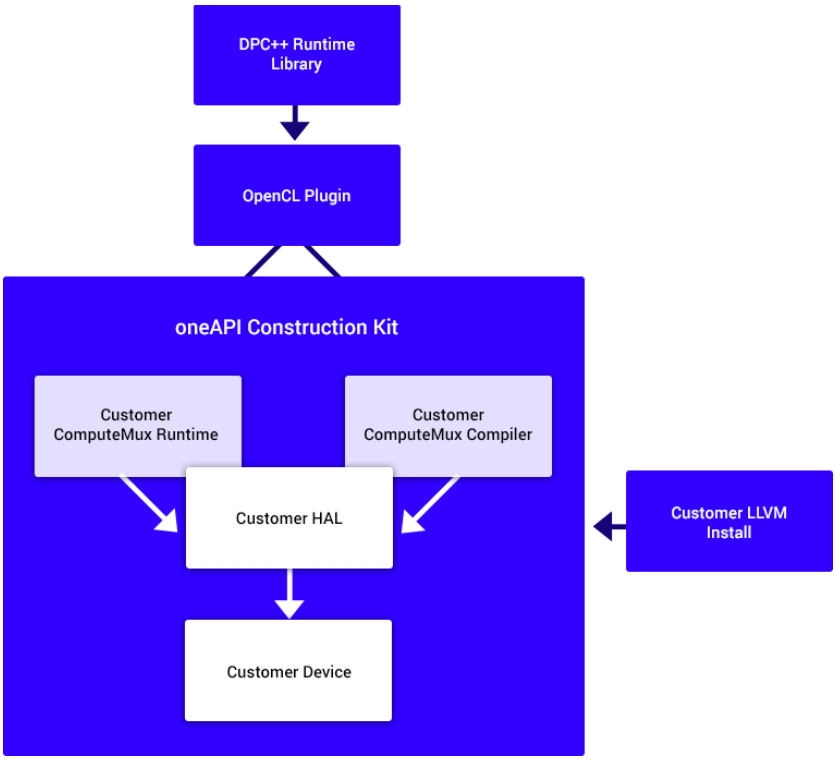


Figure 2: Components of the oneAPI construction kit

In deliverable "D4.1: RISC-V Compiler Backends", we presented the work done in M1—M18 in using the oneAPI construction Kit to support the RISC-V platforms that were deployed in EMDC v1.0 of SYCLOPS project at M18. In the following period M18-M33, oneAPI construction Kit and DPC++ have been extended in several ways.

First, the compilation process has been simplified to enable build of oneAPI construction Kit and DPC++ natively on RISC-V platforms. This was successfully tested on the MilkV RISC-V board hosted by EURECOM. Both projects have also been adapted to enable them to be cross-compiled for RISC-V to enable powerful build machines to compile RISC-V SYCL/OpenCL applications.

Second, to document the current status of the compiler support on RISC-V we have created public continuous integration test jobs on the oneAPI Construction Kit (OCK) Github (https://github.com/uxlfoundation/oneapi-construction-kit) for building and running the OpenCL and SYCL Conformance Test Suite (CTS) on RISC-V. The SYCL CTS is built (cross-compiled) and run via two targets: 1) the Native CPU target and 2) OpenCL where the OpenCL driver was built from OCK. This CI runs daily. For example, the log of the complete CI run from 29th July 2025 can be found at the following location: https://github.com/uxlfoundation/oneapi-construction-kit/actions/runs/16605113958

The following figure shows a snapshot of the SYCL_CTS run on RISC-V via NativeCPU (top right) and via OCK OpenCL (bottom right).

Figure 3: Snapshot of SYCL_CTS run on RISC-V via NativeCPU

Note that SYCL_CTS run via OpenCL is significantly slower than via Native CPU. This is because the OpenCL driver compiles the kernels at run-time, which is particularly slow in the current setup as the compiler runs on QEMU. This can be mitigated by adding a self-hosted github runner that uses a more powerful RISC-V device to build and run the kernels. The OpenCL CTS SYCL CTS (https://github.com/uxlfoundation/oneapi-construction-kit/actions/runs/16605113958/job/46986601757) pass rate on RISC-V is 100% (Note some tests were skipped due to missing capabilities of the RISC-V platform). The SYCL-CTS pass rate on RISC-V via 1) OpenCL is 99.7% https://github.com/uxlfoundation/oneapi-construction-kit/actions/runs/16605113958/job/46990593490 and 2) with NativeCPU 93.9% https://github.com/uxlfoundation/oneapi-construction-kit/actions/runs/16605113958/job/46990670653.

Third, we have integrated DPC++ with the latest the latest LLVM version to take advantage of the most recent RISC-V code generation improvements. Similarly, the NativeCPU device in DPC++ integrates the oneAPI Construction Kit compiler pipeline to take advantage of code transformations. oneTBB has been successfully integrated into the DPC++ NativeCPU device and tested on RISC-V. The net result of all this work is that we can now perform powerful thread scheduling with NUMA support, which benefits all SYCLOPS use cases and beyond on platforms with multiple CPU sockets, like the MilkV RISC-V board deployed at EURECOM.

Fourth, and most importantly, RISC-V Vector extensions version 1.0 has been enabled and tested in CI jobs mentioned previously. This will enable the OCK OpenCL driver and the OCK compiler pipeline within NativeCPU device to produce RVV1.0. We merged various PRs with fixes and updates to the OCK vectorizer targeting RISC-V (including issues exposed by using –march options). Through this work, we enable all SYCLOPS use cases to be deployable on RVV targets, like the updated A730 core provided by our partner CSIP that has been deployed at EURECOM as a part of EMDC v2.0 and described in deliverable “D3.2: EMDC v2.0 with RVV accelerator release”. Experimental results evaluating and validating DPC++ on EMDC v2.0 are also described in deliverable *D3.2*.

3. AdaptiveCPP

During the project time frame, a number of important changes were introduced to facilitate RISC-V support, and to improve the supporting compiler and runtime infrastructure in general.

Firstly, the hipSYCL project was renamed to AdaptiveCpp to better reflect the broadened focus of the project, since the old name was frequently misinterpreted as a focus on AMD hardware. This renaming was positively received. For example, the rate in which the project received stars on github (this roughly corresponds to github users marking a project as interesting to them) increased noticeably after the old name was abandoned in early 2023. This is illustrated in the figure below.

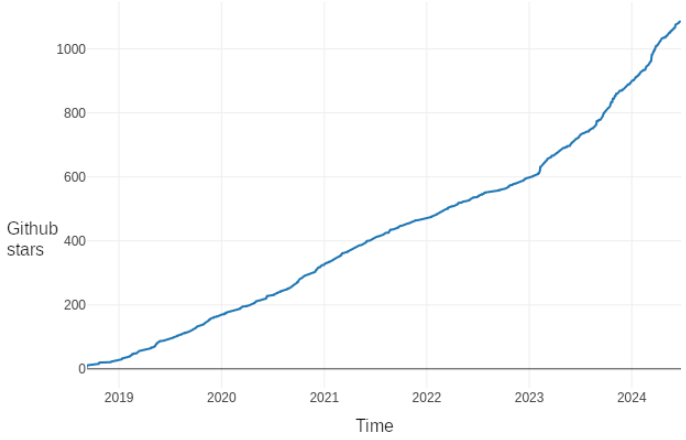


Figure 4: Github stars for AdaptiveCpp repo

On the technical side, substantial changes were introduced. Because the strategy for targeting RISC-V hardware relies on targeting Codeplay's oneAPI construction kit, AdaptiveCpp needed to be able to target OpenCL devices using SPIR-V, as this is how the oneAPI construction kit operates.

AdaptiveCpp supports a generic single-pass (SSCP) JIT compiler. This compiler embeds the device code at compile time as generic intermediate representation (IR) of the LLVM compiler infrastructure. At runtime, it can then generate amdgcn code for AMD GPUs, PTX code for NVIDIA GPUs, and SPIR-V for OpenCL devices (e.g. Intel GPUs or RISC-V). Recently, an additional backend was added to target the native host CPU. This design has the advantage that a single compilation can generate a binary that can dispatch to all of the devices supported by AdaptiveCpp, depending on what is found on the system.

AdaptiveCpp is the only SYCL implementation that can generate code for all these devices with a unified JIT compilation infrastructure.

Releases

Within the project time frame, there have been two releases:

- AdaptiveCpp 23.10 (Full release details: https://github.com/AdaptiveCpp/AdaptiveCpp/releases/tag/v23.10.0)
- AdaptiveCpp 24.02 (Full release details: https://github.com/AdaptiveCpp/AdaptiveCpp/releases/tag/v24.02.0)
- AdaptiveCpp 24.06 (Full release details: https://github.com/AdaptiveCpp/AdaptiveCpp/releases/tag/v24.06.0)
- AdaptiveCpp 24.10 (Full release details: https://github.com/AdaptiveCpp/AdaptiveCpp/releases/tag/v24.10.0)
- AdaptiveCpp 25.02 (Full release details: https://github.com/AdaptiveCpp/AdaptiveCpp/releases/tag/v25.02.0)

AdaptiveCpp 25.10 is scheduled to be released later this year in October. In order to better structure development and provide stronger guarantees for users, a fixed regular release schedule was adopted.

AdaptiveCpp was routinely presented, e.g. in tutorials at conferences (often as part of broader SYCL tutorials) such as at IWOCL' 23, IWOCL '24, IWOCL '25, ISC '23, ISC '24 as well as in dedicated talks, e.g. at PASC '24 [6] .

The International Workshop on OpenCL and SYCL '25 (IWOCL '25) was also hosted by our team at Heidelberg University.

1.1 OpenCL Runtime Backend

While the generic JIT compiler was already available at the beginning of the project, it was still experimental and did not yet support OpenCL.

AdaptiveCpp has a modular C++ interface for backends. Therefore, adding new runtime backends is fairly straight-forward as it mainly requires implementing these interfaces. We have thus added a new OpenCL runtime backend, which has since been shown to perform well (see e.g. for performance on Intel GPUs with OpenCL [1]).

Our publication on SYCL-Bench 2020 [2] contains microbenchmarks on some aspects of the runtime. For example, Figure 5 shown below (taken from [2]) shows the task scheduling latency for 50000 kernel launches on various hardware – the data on the Intel Max 1100 GPU was obtained using the new OpenCL backend. For the more modern USM memory management API in SYCL, AdaptiveCpp's OpenCL backend even outperforms the Intel oneAPI DPC++ compiler on the Intel GPU. Note that the data for the AMD MI100 GPU in the DPC++ case is missing due to excessive runtime.

Apart from the AMD case, AdaptiveCpp and oneAPI DPC++ perform similarly for this workload.

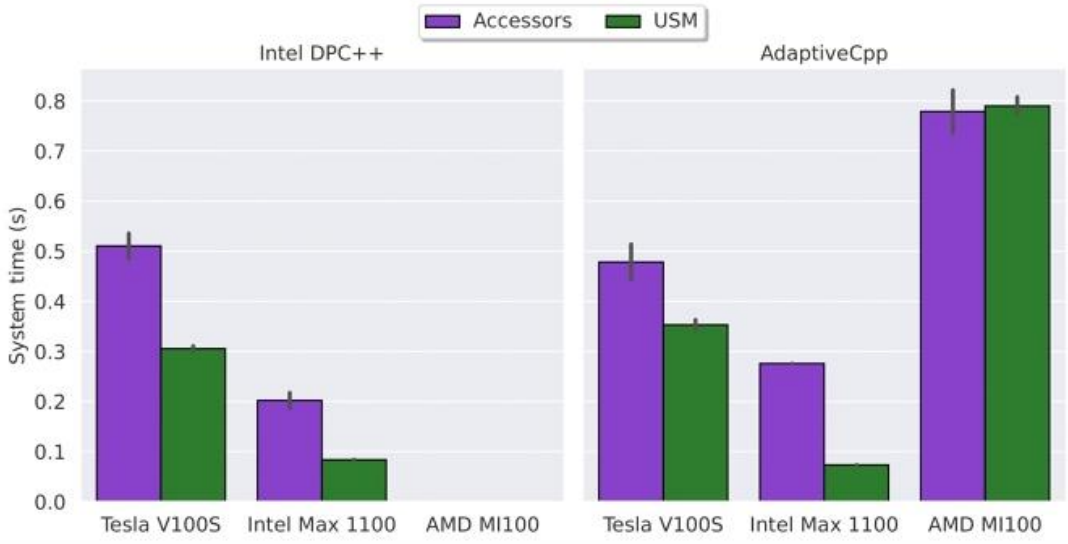


Figure 5: Task scheduling latency on various hardware

1.1.1 Code generation

SPIR-V generation through the generic JIT compiler is a complex process that was substantially improved and optimized over the course of the project. The original design of our compiler can be found in [3]. Firstly, SYCL features were implemented that at the start of the project were still missing in the generic JIT compiler. This in particular affects atomics, group algorithms and the SYCL 2020 reduction interface.

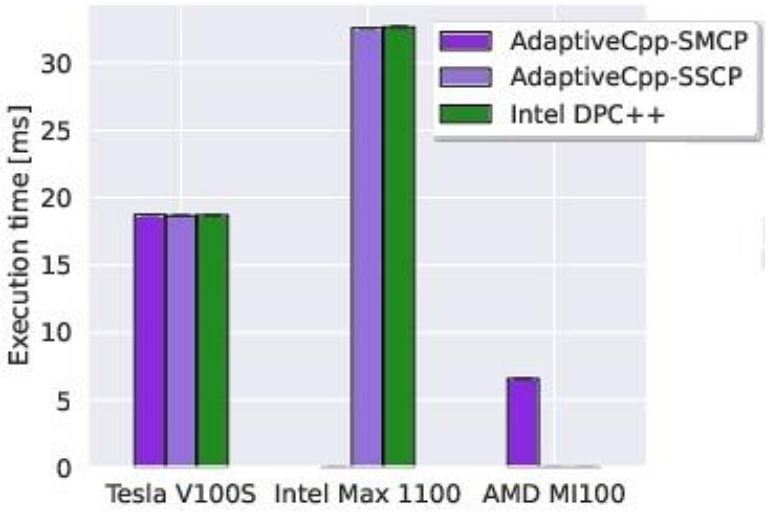


Figure 6: Task scheduling latency on various hardware

Atomics in the generic JIT compiler are implemented using a built-in interface that is implemented in backend-specific LLVM bit code libraries. Compare-and-swap loops as emulations of atomics are only used in cases where there is no native backend functionality to represent an atomic operation.

Figure 6 shown above (again taken from our paper [2]) shows microbenchmark results using the atomic test from SYCL-Bench 2020. The SSCP results refer to our new generic JIT compiler, and the SMCP results, where available, to our old compiler. As can be seen, the atomic implementation in the generic JIT compiler behaves very similarly compared to either DPC++ or our old (non-SPIR-V capable) compilers.

The SYCL 2020 reduction interface defines a flexible API that allows specifying reductions over arbitrary types (including user-defined types) and arbitrary reduction operators, as long as they are associative. This also include cases where no identity for the reduction is known.

This generality makes reductions challenging to implement, and to optimize. We have added an implementation that supports arbitrary data types and arbitrary reduction operators. Additionally, it employs an efficient caching scheme for scratch allocations that might be needed. Additional optimizations include assigning multiple reduction elements to each SYCL work item to better utilize memory bandwidth, as well backend-specific code paths. In particular, when running on the CPU, a different memory access pattern is employed. With the BabelStream benchmark, we find that our reduction implementation delivers performance in line with other compilers and programming models for the same problem, and in line with the hardware's memory bandwidth.

In the original implementation of the generic JIT compiler, the -ffast-math optimization flag was not yet correctly exploited. This flag is commonly used by applications which prefer speed over accuracy.

We have thus added proper fast-math handling, which includes a) relaxing numerical requirements when compiling user code and b) linking against bitcode libraries providing e.g. math builtins that employ similar optimizations. Furthermore, the default floating point model of the compiler was aligned to the defaults of other heterogeneous compilers (e.g. AMD's HIP compiler or DPC++) and now uses the clang option -ffp-contract=fast by default. This can result in a noticeable performance increase for applications not requesting a specific floating point model at compile time.

With the release of AdaptiveCpp 24.02, the generic JIT compiler was elevated to be the default compiler of AdaptiveCpp. This means that a simple compiler invocation (e.g. acpp -o test test.cpp) will by default generate a binary that can dispatch kernels to the host CPU, NVIDIA GPUs, AMD GPUs, Intel GPUs, as well as the oneAPI construction kit, and thus RISC-V hardware.

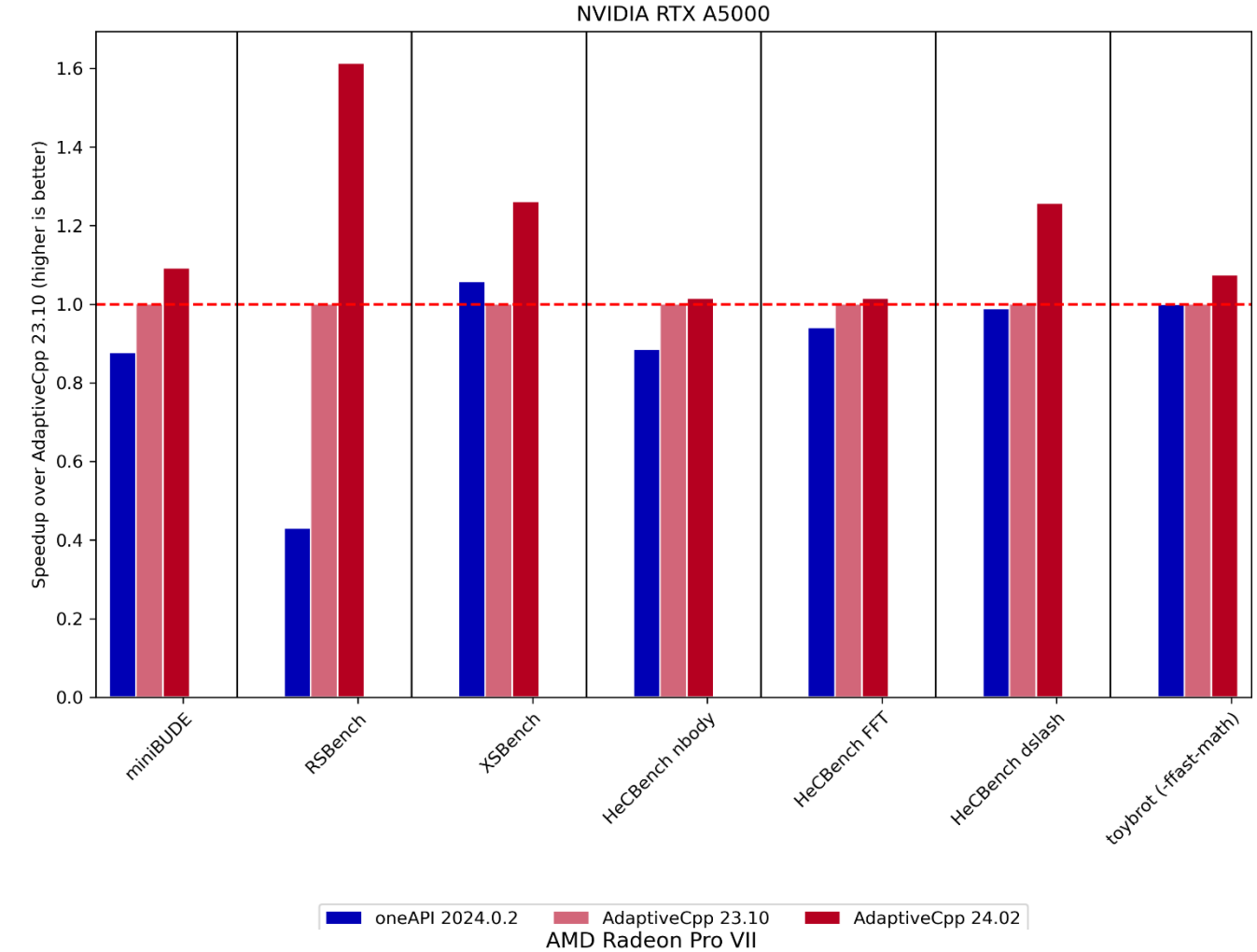
1.2 Adaptivity and JIT-time optimizations

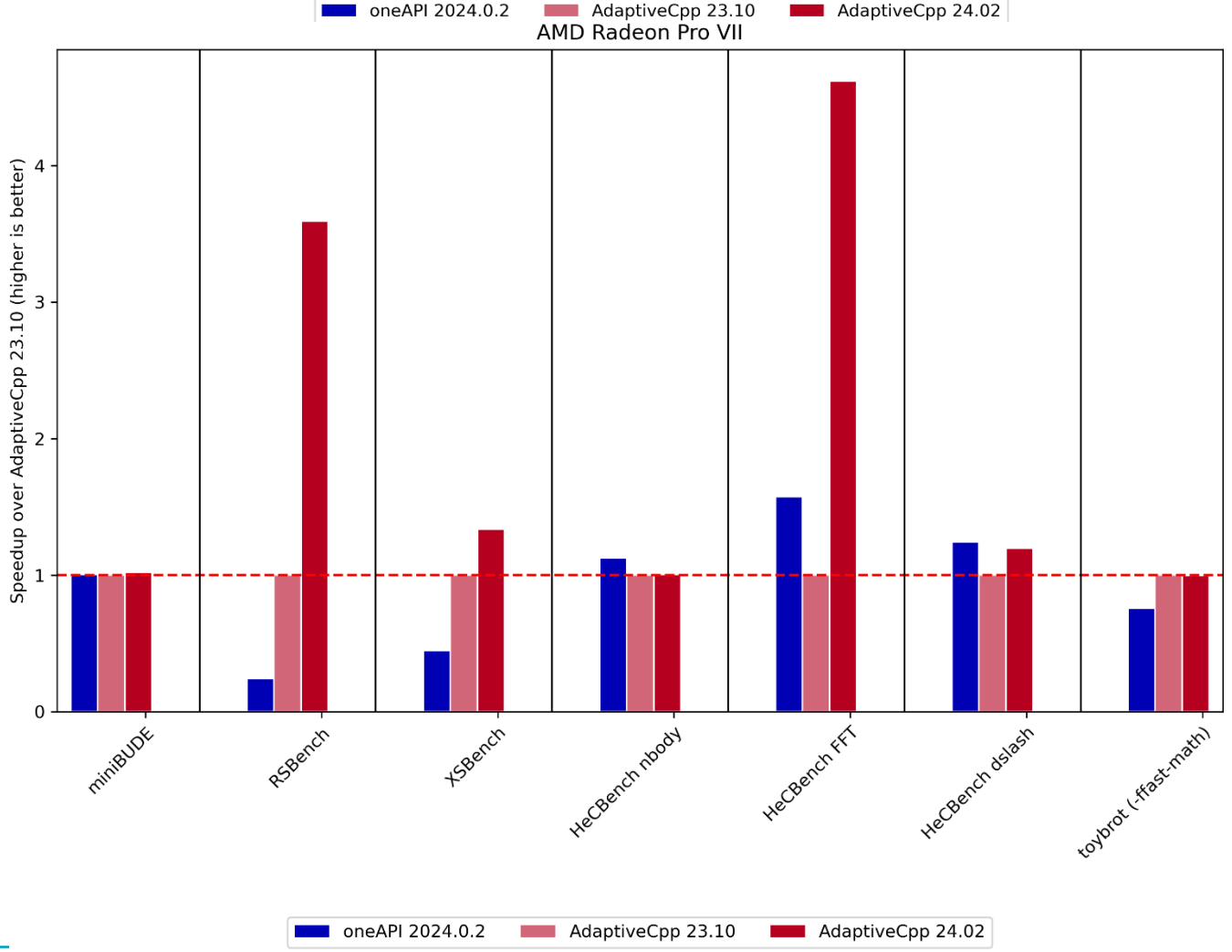
The AdaptiveCpp generic JIT compiler lowers and optimizes LLVM IR at runtime for the target backends. This opens the door for a wide array of runtime optimizations, but can cause additional overheads compared to directly generating e.g. SPIR-V at compile time. To mitigate this, AdaptiveCpp 24.02 has introduced a unified kernel cache across backends, with a persistent second-level cache on disk. To enable this two-level cache system, a new mechanism of uniquely identifying kernels was introduced: A 128-bit hash generated from all of configuration parameters specifying the current JIT compilation, such as target backend, device and target architecture, the translation unit that the kernel originates from, the kernel name, and others. Especially for future application runs, the persistent cache mitigates the impact of the additional step of processing the LLVM IR, which other compilers do not have to do.

With a JIT compiler, it is in principle possible to perform optimizations that are impossible in a static compilation model, since a JIT compiler can take into account information only known at runtime. A downside of performing such JIT-time optimizations is however that it might lead to additional JIT compilations.

With JIT overheads having been mitigated as a concern with the persistent cache, we have decided to implement additional JIT-time optimizations that are not commonly done by default in other production compilers. To this end, we have introduced the ACPP_ADAPTIVITY_LEVEL environment variable, which can be interpreted as a runtime optimization level: If the adaptivity level is 0, it will not perform any additional optimizations, trying to avoid additional JIT compilation steps. If the adaptivity level is 1, a set of optimizations is enabled that typically do not require many additional kernels to be generated and are thus relatively risk-free. This includes hardwiring the kernel work group size as a constant in the code, and also informing the backend optimizer about the kernel work group size, which can help register scheduling. Other optimizations include the detection of whether the problem size fits in 32-bit integers, and if so, not carrying out calculations e.g. to determine the global id of a work item in 64-bit.

The performance results in Figure 7 below show the performance improvements on NVIDIA, AMD and Intel that were achieved from the combined effect of the previously mentioned improvements at the time of the AdaptiveCpp 24.02 release, compared to the previous version 23.10. DPC++ results are provided as reference as well. As can be seen from these results, the performance increase from AdaptiveCpp 23.10 to 24.02 is noticeable on all backends, and it competes very well with DPC++.


NVIDIA RTX A5000
Speedup over AdaptiveCpp 23.10 (higher is better)
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
miniBUDE
RSBench
XSBench
HeCBench nbody
HeCBench FFT
HeCBench dslash
toybrot (-ffast-math)
oneAPI 2024.0.2
AdaptiveCpp 23.10
AdaptiveCpp 24.02


AMD Radeon Pro VII
Speedup over AdaptiveCpp 23.10 (higher is better)
0
1
2
3
4
miniBUDE
RSBench
XSBench
HeCBench nbody
HeCBench FFT
HeCBench dslash
toybrot (-ffast-math)
oneAPI 2024.0.2
AdaptiveCpp 23.10
AdaptiveCpp 24.02

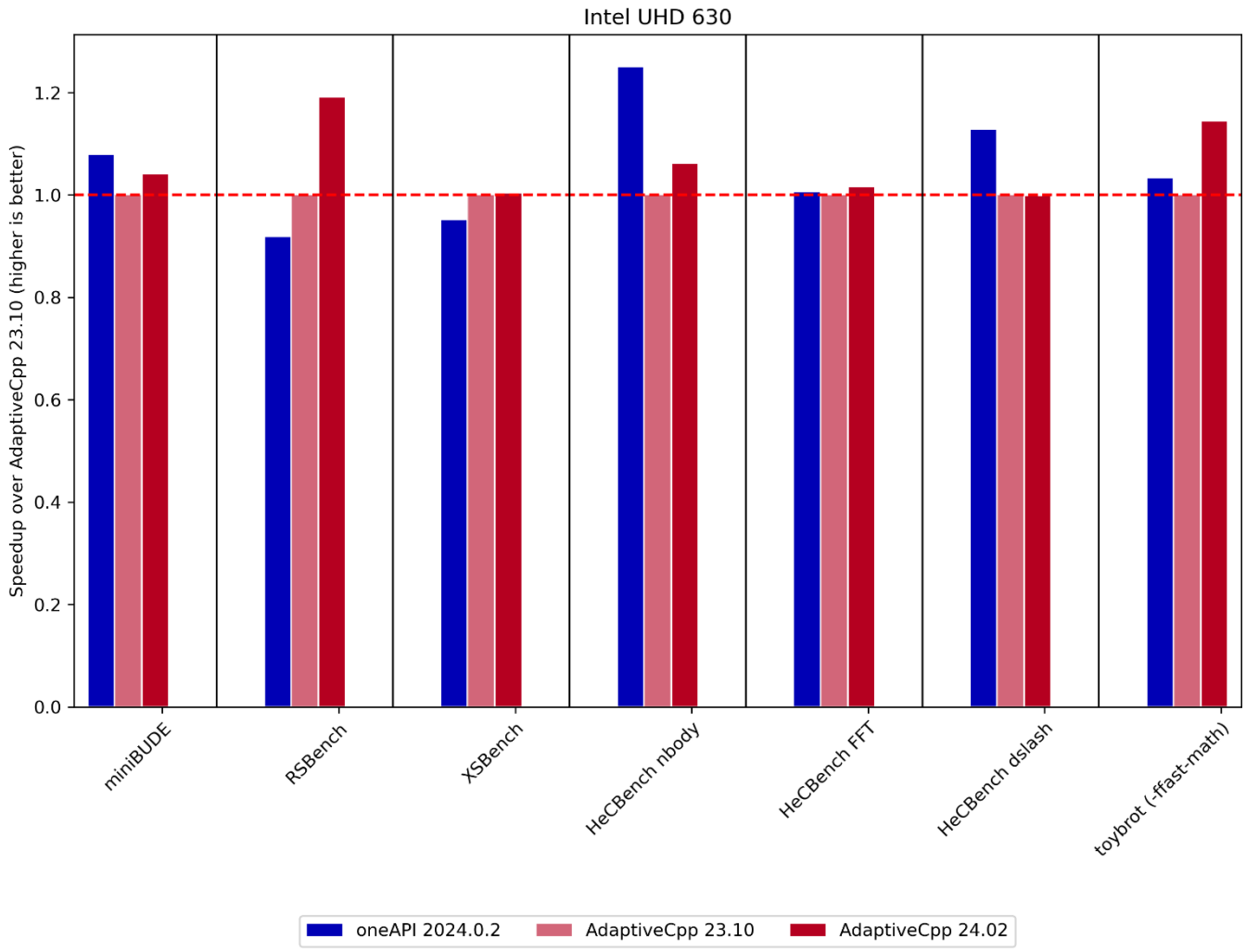


Figure 7: Performance results on NVIDIA, AMD, and Intel hardware

After the release of AdaptiveCpp 24.02, we have also started to introduce the first optimizations for a more aggressive setting of an adaptivity level of 2. At a level of 2, AdaptiveCpp is free to employ optimizations that are expected to come with additional JIT costs, kernel launch latencies or might need more application runs to achieve peak performance.

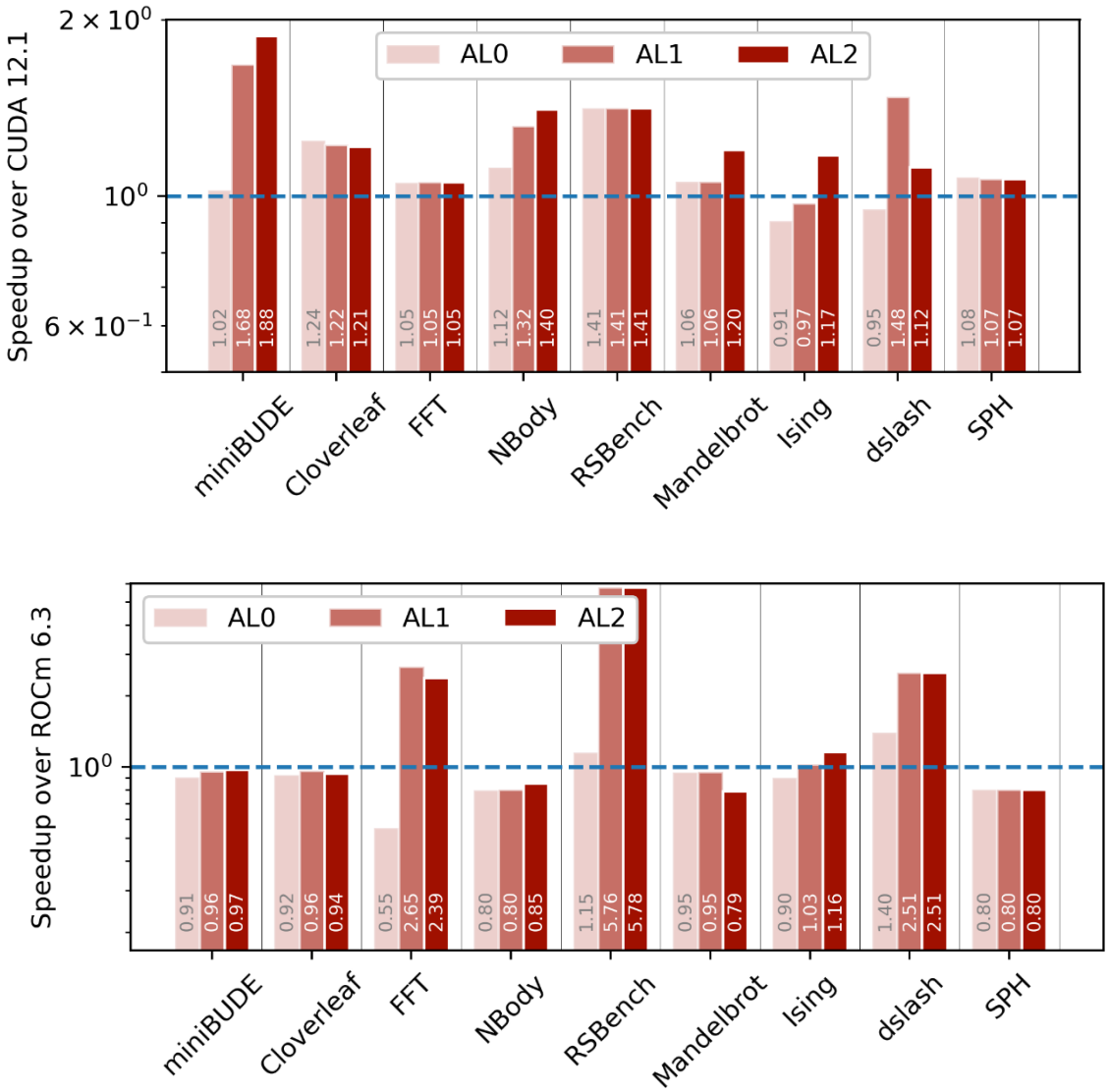
At this setting, AdaptiveCpp will analyze at runtime the usage patterns of arguments that get passed into kernels. If it detects that specific values are commonly used as arguments, it hardwires them as constants at JIT-time, compiling a new, specialized kernel. Because the LLVM optimization pipeline is only run after this process, the value will be propagated as a constant throughout the code, which could lead e.g. to dead code elimination or reduced register usage. This idea is similar to specialization constants from the SYCL 2020 specification, except that it happens automatically. Effectively, this feature enables not only constant propagation across the host-device boundary, but also the propagation of runtime values which are de-facto constants into device code as constants.

The detection of these common kernel arguments is enabled by storing a persistent, application-specific database containing statistical information on kernel invocations and kernel arguments on disk. Kernels are again identified using the 128-bit hash of the kernel configuration. This allows AdaptiveCpp to learn across multiple application runs which kernel argument values might be worth JIT-compiling a dedicated kernel for. Because it is in general impractical to store data on every different value that has ever been passed into the kernel, heuristics are employed to evict information on kernel arguments that have not been used in a while from the database.

This feature is now mature as of AdaptiveCpp 25.02. Performance-wise, the benefit of this feature seems to highly depend on the code and the target hardware. We see benefits especially for compute-bound applications, e.g. miniBUDE improving performance by ~10% on NVIDIA or ~30% on an Intel iGPU.

Furthermore, we have introduced additional optimizations at adaptivity level 1. This includes e.g. the ability to take into account the alignment of input pointers for kernels, and to automatically detect cases where input pointers don't alias. To this end, the AdaptiveCpp JIT compiler attempts to prove that a kernel does not perform indirect access, i.e. does not load pointers from memory. If no indirect access is present, then the set of allocations passed through kernel pointer arguments are all allocations that the kernel can access. If the allocation referred to by one pointer kernel argument is not referred to by any other pointer kernel argument, then the compiler can safely assume that no aliasing takes place. This has far-reaching consequences, and can allow the optimizer to reorder loads and stores, or cache loads etc.

In 2025, we have published a comprehensive description and evaluation of the adaptivity framework [4]. Figure 8 below from the publication shows the obtained results using the latest set of optimizations on NVIDIA RTX A500, AMD Radeon Pro VII and Intel UHD 630 respectively using a recent version of AdaptiveCpp. Results are normalized to the vendor native programming model and compiler (nvcc-compiled CUDA, hipcc-compiled HIP, icpx-compiled oneAPI).



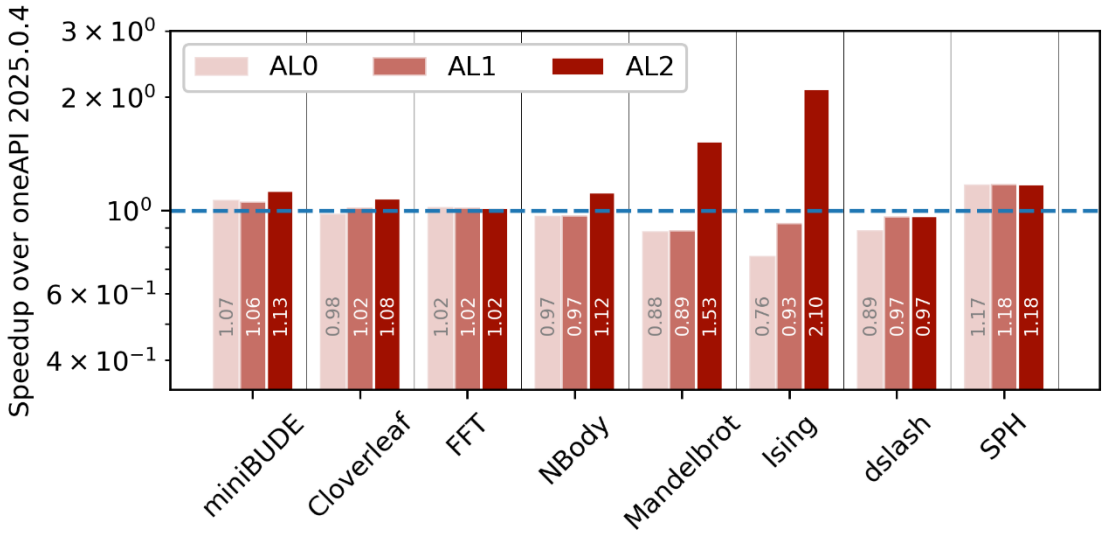


Figure 8: Performance results on NVIDIA, AMD, and Intel hardware

As can be seen, AdaptiveCpp typically outperforms vendor compilers. CUDA is outperformed by 30% in the geometric mean, and HIP and oneAPI by 44% and 23%, respectively. Potential JIT-overheads were investigated in detail, and were found to not play a role in typical cases and use cases (see [4] for details). This is primarily because of the effectiveness of the design of the JIT-cache, and because the optimizations are selected carefully so that they don't typically result in excessive amounts of generated kernels. In fact, for the investigated benchmarks we find that the amount of generated kernels stays the exactly same in almost all cases – therefore, the JIT-optimizations do not result in more overhead than any other JIT-based compiler that does not leverage such techniques.

1.3 User-driven runtime modification of kernels

In addition to these automatic JIT optimizations, we have also introduced APIs to leverage the JIT compiler more explicitly. This is targeted at users who wish more control for optimizations.

Firstly, wrapping a kernel argument in a new sycl::specialized type is interpreted as a hint to the runtime to generate a new kernel that has the value of this argument hardwired as a constant. This is a very similar use case as the SYCL 2020 specialization constant API. However, our API was developed to address shortcomings of the API in the SYCL specification:

- The SYCL API requires explicit get and set calls to set and retrieve and set the value, and all accesses need to be funneled through an additional kernel_handler argument that is passed to the kernel. This design makes it cumbersome both for the user to use and for the implementer to implement;
- Whenever a JIT compiler is unavailable (e.g. in an ahead-of-time compilation scenario) specialization constant support must be emulated. The additional indirection through the kernel_handler object can in this case lead to substantial performance overheads, especially since specialization constants are typically used in the hottest parts of the code.

Our sycl::specialized extension avoids both problems: It is very convenient and easy to use, and, if no JIT compiler is available, it incurs no additional overhead compared to a regular kernel argument. This extension has been presented in the Khronos SYCL working group for potential standardization in the future.

The second feature to expose the JIT compiler to users that we have added is the experimental ability to modify function calls at runtime. Users can at runtime instruct the JIT compiler to replace calls to a function A with another function B, or replace all calls to function A with a call sequence to other functions B and C. Once the calls have been replaced, there is no overhead compared to a regular function call.

This feature, which we call “dynamic functions”, effectively allows for a runtime assembly of kernels. Users can use it to implement a form of JIT-time polymorphism, where kernel behaviour needs to change based on runtime values. Since function calls can also be replaced by call sequences to other kernels, kernel-fusion-like semantics are also possible. This feature is currently available in an experimental state for users to evaluate.

1.4 AdaptiveCpp Portable CUDA (PCUDA)

Another feature that was added recently and released as part of AdaptiveCpp 25.02 is portable CUDA (PCUDA). PCUDA is an implementation of (a slight dialect of) the CUDA and HIP programming languages in the AdaptiveCpp generic JIT compiler. This allows users to mix-and-match SYCL code with CUDA (or HIP) code e.g. for the purpose of simplified, iterative porting. Because the CUDA code is compiled by our generic JIT compiler, the resulting program is just as portable as SYCL code compiled by AdaptiveCpp.

This has multiple benefits:

1. Code can be ported iteratively to SYCL; highly-optimized complex parts written in CUDA could also be left as-is to avoid risk;
2. Existing CUDA or HIP code might run on other hardware with no or little changes;
3. When making comparisons between CUDA and SYCL (e.g. performance or compile-time), it allows for fairer comparisons. Currently, such comparisons require changing multiple variables – input program, programming model and compiler. With PCUDA, the compiler can be kept the same between, thus simplifying distinguishing between effects due to different compilers, and effects due to the programming model or different code quality of the CUDA/SYCL ports of the input code.

To this end, support for key syntactic constructs in CUDA and HIP was added to AdaptiveCpp’s generic JIT compiler, and an implementation of the CUDA/HIP runtime was added to AdaptiveCpp, with interoperability with SYCL.

CUDA and HIP mostly differ from each other based on the naming scheme for runtime functions: CUDA functions are prefixed with `cuda` while HIP functions are prefixed with `hip`. PCUDA supports both naming schemes, depending on whether the CUDA or HIP headers are included.

Compilers that can compile CUDA or HIP code for other hardware already exist (e.g. the chipStar project). However, there no other solution to our knowledge currently provides full interoperability with SYCL on both a source and runtime level, and also enables the same portability as AdaptiveCpp’s SYCL support: Running kernels on CPUs, Intel GPUs, NVIDIA GPUs and AMD GPUs from a single binary.

There are some subtle differences between CUDA/HIP and PCUDA, which is why we generally call PCUDA a CUDA dialect. These differences are mostly related to the different compiler designs. NVIDIA’s nvcc compiler is an ahead-of-time compiler with distinct host and device passes, while AdaptiveCpp is a JIT design with unified host-device compiler. AMD’s hipcc is similar in this respect to nvcc. These differences necessitate some divergence e.g. in how code paths can be specialized for different targets.

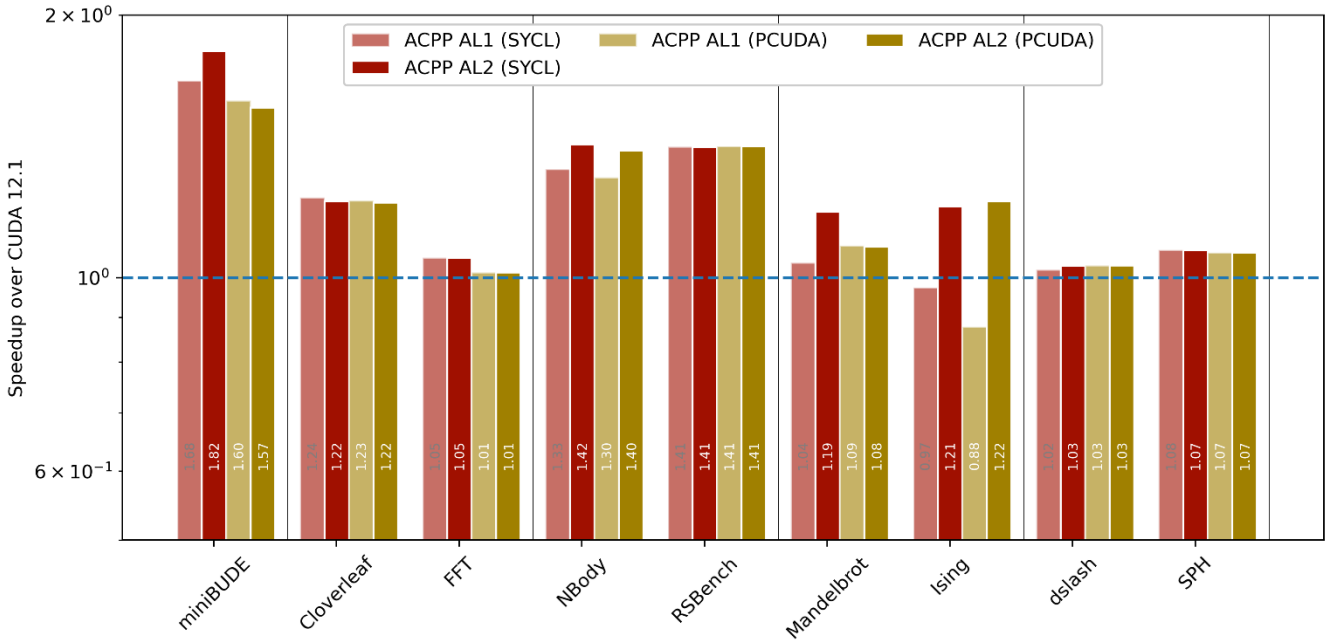
In nvcc’s CUDA, this can be accomplished by checking target macros defined by the compiler due to its ahead-of-time design. In a JIT-design however, it is generally not known on which hardware the code will be run until JIT compilation is triggered at runtime. Therefore, target macros cannot be available. Instead, AdaptiveCpp has a JIT-reflection mechanism that operates as part of control flow. We note that NVIDIA’s nvc++ uses similar mechanisms. A detailed discussion on how PCUDA diverges from CUDA or HIP can be found in the AdaptiveCpp documentation [5].

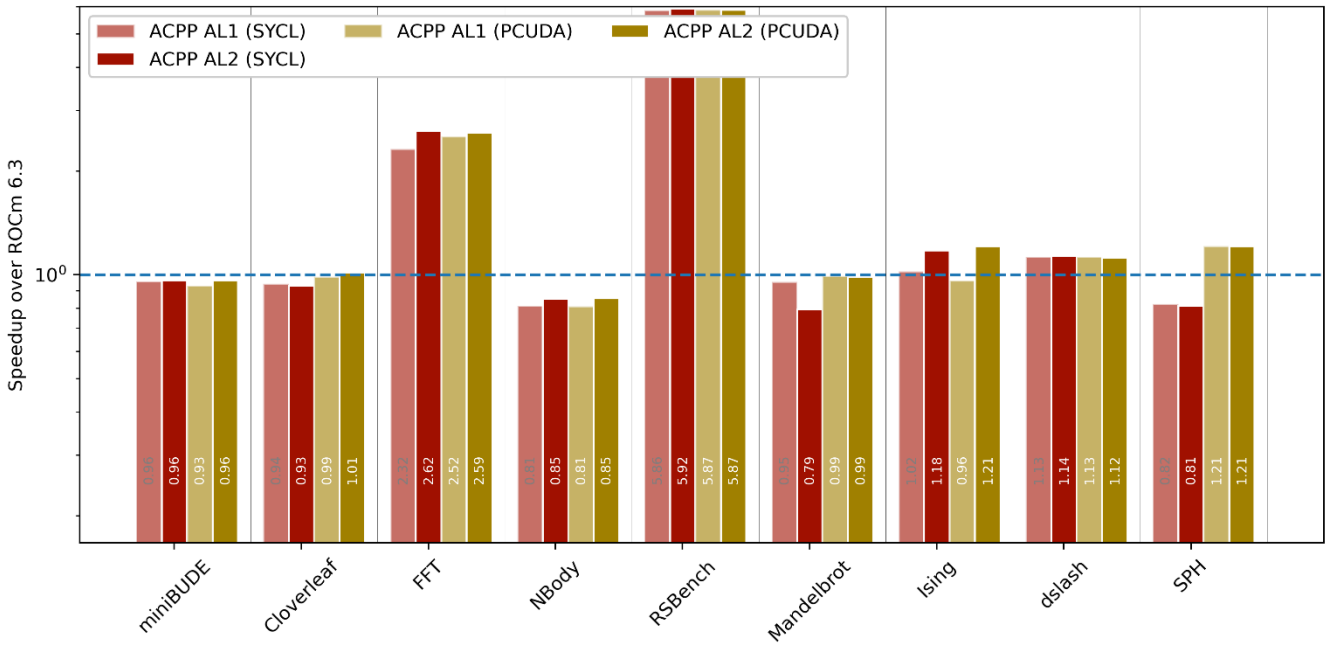
PCUDA is a very recent feature, therefore the list of supported CUDA APIs is still limited. However, support is sufficiently broad to evaluate PCUDA with common benchmark applications.

In Figure 9, performance comparisons can be seen between CUDA code compiled with AdaptiveCpp’s PCUDA support, and SYCL versions of the same application as compiled by AdaptiveCpp.

Note that the PCUDA results were obtained with the original CUDA versions of the code; changes were limited to enabling AdaptiveCpp in the build system and in one case, to fix a bug that was not noticed when compiling with nvcc (CloverLeaf pessimistic performance using function pointers unnecessarily).

These results were obtained on NVIDIA RTX A5000, AMD Radeon Pro VII, and Intel UHD 630 respectively. The performance numbers are normalized to the native model and native compiler on the platform, i.e. nvcc-compiled CUDA, hipcc-compiled HIP, and oneAPI icpx-compiled SYCL.





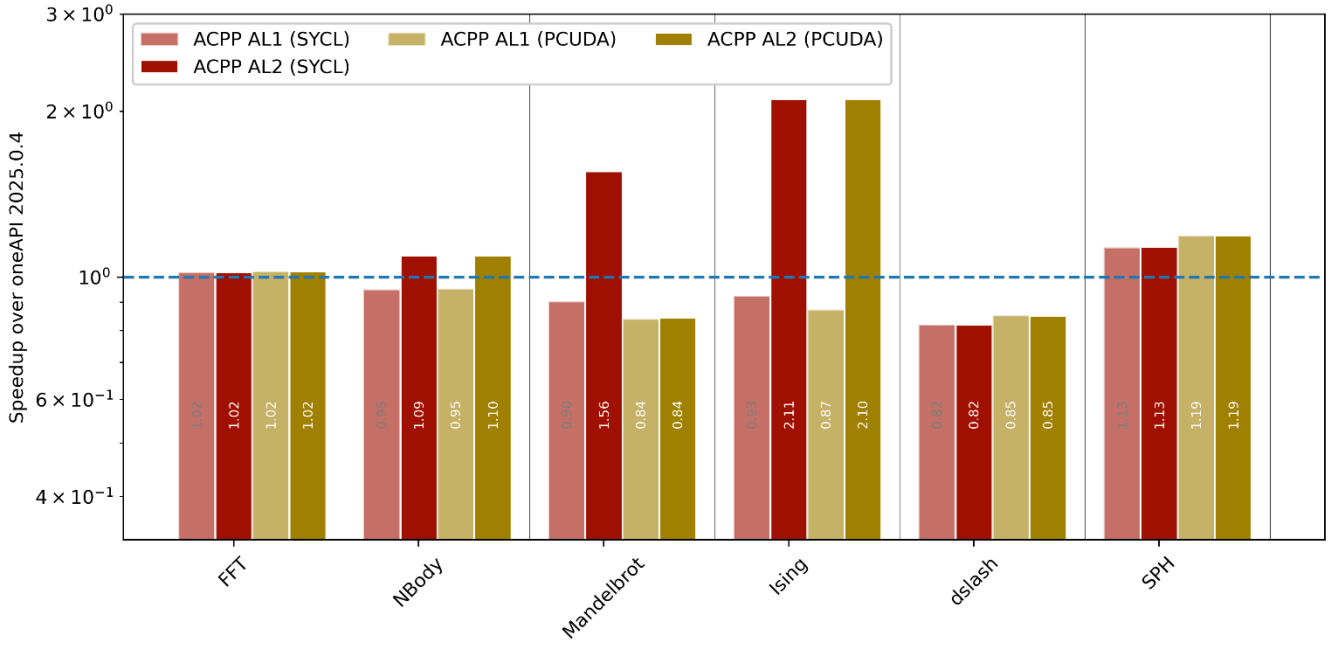


Figure 9: Performance comparison of PCUDA and SYCL code

As can be seen, performance of PCUDA and SYCL mostly match. In cases where differences were found, those could typically be traced back to differences in how well the different ports of the application used language features. For example, miniBUDE does not utilize local memory the same way in its CUDA port compared to SYCL. The close match in performance indicates that there is no inherent performance penalty due to the SYCL abstractions. Similarly, it shows that PCUDA is just as efficiently as SYCL – both typically outperform vendor compilers. This also implies that existing CUDA applications might see speedups by recompiling them with AdaptiveCpp PCUDA.

5. Conclusion

This deliverable concludes the work done in “Task 4.2: Compiler with auto vectorization” of WP4 in SYCLOPS project. We have made substantial improvements in both DPC++ and AdaptiveCPP compiler toolchains.

On the DPC++ front, we have established mature compiler support targeting RISC-V via the oneAPI Construction Kit and DPC++ projects. This enables standards-based acceleration via OpenCL and SYCL on RISC-V platforms. The latest version of LLVM was integrated to take advantage of most recent RISC-V/RVV support. This capability is demonstrated through public continuous integration test jobs visible on the OCK GitHub which show the SYCL/OpenCL conformance status on RISC-V.

Over the course of the project, significant improvements have also been made to AdaptiveCpp. These include a new OpenCL backend with the associated support both in the compiler and runtime and the maturing of the generic JIT compilation infrastructure that was still new at the beginning of the project. Furthermore, the compiler and runtime stack were optimized extensively, particularly by leveraging opportunities at JIT-time to the point where AdaptiveCpp typically outperforms vendor compilers. Support for PCUDA as an additional programming model simplifies porting code to SYCL, broadens use cases for AdaptiveCpp, and enables more programmer flexibility with respect to the development process.

We have integrated both compilers and performed preliminary evaluation using v2.0 of SYCLOPS EMDC platform as described in deliverable *D3.2*. All the work done on our compilers have already been made publicly available in their respective Github repositories mentioned in this document. We have also disseminated our work via technical blogs on OCK and AdaptiveCpp on the SYCLOPS website, and technical talks that can be found in the SYCLOPS YouTube channel.

References

[1] https://dl.acm.org/doi/10.1145/3648115.3648117

[2] https://dl.acm.org/doi/10.1145/3648115.3648120

[3] https://dl.acm.org/doi/abs/10.1145/3585341.3585351

[4] https://dl.acm.org/doi/10.1145/3731125.3731127

5] https://github.com/AdaptiveCpp/AdaptiveCpp/blob/develop/doc/pcuda.md

[6] A. Alpay, „The Community-driven AdaptiveCpp SYCL Compiler Project: From High-Level C++ Programming to the Automatic Synthesis of Specialized Kernels“, gehalten auf der Platform for Advanced Scientific Computing, 2024 (PASC24), ETH Zurich, Main Building, Zurich, Switzerland, Juni 18, 2025. doi: 10.5281/zenodo.15691718.[5] https://github.com/AdaptiveCpp/AdaptiveCpp/blob/develop/doc/pcuda.md