Deliverable 4.1 – RISC-V Compiler Backends

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**

D4.1 – RISC-V Compiler Backends

Executive Summary: This deliverable supplements the software releases of DPC++ and AdaptiveCpp compiler toolchains. In a nutshell, the work done on these toolchains in SYCLOPS has demonstrated that (i) We are able to compile and execute SYCL applications using both DPC++ and AdaptiveCpp on real RISC-V host CPUs, (ii) We are able to offload SYCL kernels to RISC-V accelerators using the oneAPI construction kit, and (iii) using new functionalities like kernel fusion in DPC++, dynamic functions in AdaptiveCpp, and others, we are able to improve SYCL application performance on GPUs from several vendors. Both compilers are publicly available. We have also disseminated our work via blogs on the SYCLOPS website, and technical talks on the Youtube channel.

WP:	4		
Author(s):	Uwe Dolinsky, Kumudha Narasimhan, Mehdi Goli, Aksel Alpay, Raja Appuswamy		
Editor:	Raja Appuswamy		
Leading Partner:	UHEI		
Participating Partners:	CPLAY, UHEI, EUR		
Version:	1.0	**Status:**	Draft
Deliverable Type:	Other	**Dissemination Level:**	PU - Public
Official Submission Date:	30-06-2024	**Actual Submission Date:**	01-07-2024

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	Document outline & structure	**EUR**
0.2	Version with OCK & DPC++	**CPLAY**
0.3	Version with AdaptiveCpp	**UHEI**
1.0	Final version	**EUR**

Authors

Author	Partner
Uwe Dolinsky, Kumudha Narasimhan, Mehdi Goli	**CPLAY**
Aksel Alpay	**UHEI**
Raja Appuswamy	**EUR**

Reviewers

Name	Organisation
Aleksandar Ilic	INESC
Vincent Heuveline	UHEI
Stefan Roiser	CERN
Nimisha Chaturvedi	ACC
Martin Božek	CSIP
Mehdi Goli	CPLAY

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

List of Figures

Executive Summary

The aim of this deliverable (D4.1 RISC-V Compiler Backends) is to supplement the software releases of the two compiler toolchains involved in SYCLOPS, namely, DPC++ (formerly ACORAN) and AdaptiveCpp (formerly hipSYCL), and to provide a high-level overview of work done with respect to (i) support for RISC-V (as host CPU and as accelerator), and (ii) support for advanced functionalities and optimizations in SYCL.

In a nutshell, the work done in SYCLOPS has demonstrated that (i) We are able to compile and execute SYCL applications using both DPC++ and AdaptiveCpp on real RISC-V host CPUs (the 64-core SOPHON SG2042 RISC-V CPU deployed in the SYCLOPS platform), (ii) We are able to offload SYCL kernels to RISC-V accelerators (FPGA-based RISC-V platform containing A730 CSIP core available in the SYCLOPS platform) using the oneAPI construction kit, and (iii) using new functionalities like kernel fusion in DPC++, dynamic functions in AdaptiveCpp, and others, we are able to substantially improve SYCL application performance on GPUs from several vendors. Both our compilers are publicly available, and we have also disseminated our work via technical blogs on the SYCLOPS website, and technical talks on the SYCLOPS Youtube channel.

1 Introduction

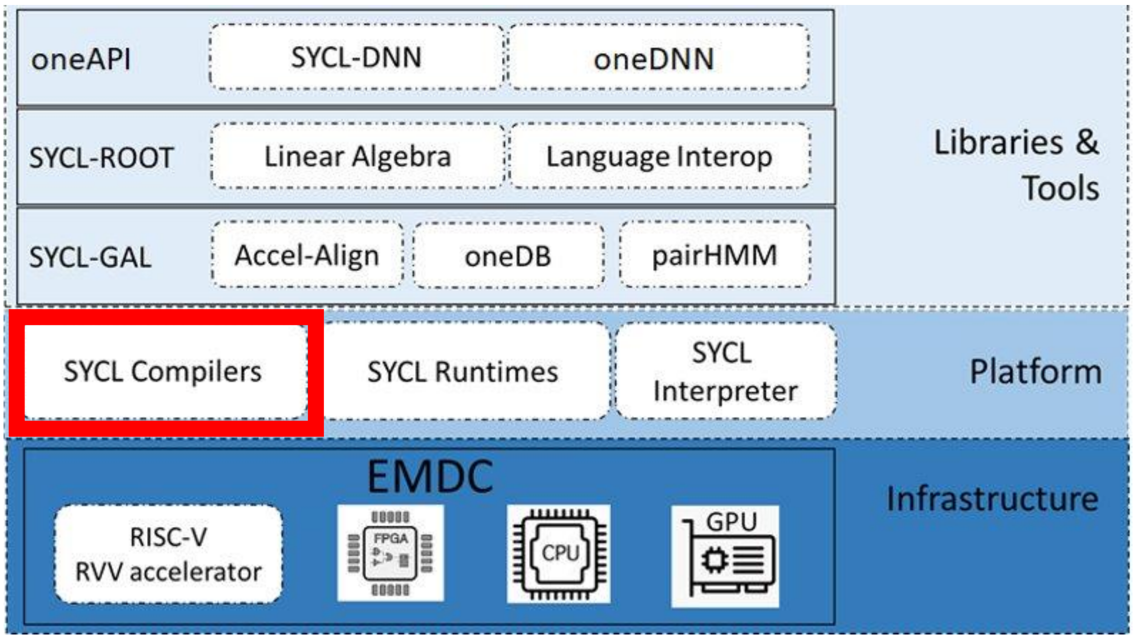


Figure 1. SYCLOPS architecture

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.

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. Deliverable *"D3.1 SYCLOPS Reference Platform v1.0"* describes various accelerators available in the current SYCLOPS platform at M18.

Platform 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. Cling is a state-of-the-art C++ interpreter that is being used as an interactive code development environment for exploratory analysis.

Application libraries and tools layer: While the platform layer described above enables direct programming in SYCL, 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 covers the **SYCL compilers** part of the stack as highlighted in Figure 1. In the M1-M18 period of SYCLOPS, several new functionalities have been added to both DPC++ and AdaptiveCpp in the context of "*Task 4.1: Compiler support for RISC-V"* in *WP4* (M3-M33). These functionalities have already been merged and integrated in several public releases of DPC++ and AdaptiveCpp that have been made during the project, or is being staged for integration and release in the upcoming months. This deliverable is a summary of this work with a special focus on (i) support for RISC-V hardware available in the SYCLOPS platform at

M18, and (ii) support for advanced SYCL functionalities that will be used to improve the performance of application libraries later in the project.

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++ & oneAPI

After the official start of project SYCLOPS, Intel acquired our partner CPLAY. Hence, CPLAY's proprietary ACORAN ComputeCPP compiler described in the SYCLOPS proposal has been discontinued. This has replaced by open-source DPC++ as one of the SYCL implementation used in SYCLOPS. DPC++ is an open-source, state-of-the-art SYCL compiler and runtime with a much larger developer and user community than ComputeCPP. DPC++ and SYCL form the core of oneAPI[1]-- an open, cross-architecture programming model allowing developers to use a single codebase across multiple accelerator architectures such as GPUs and FPGAs.

With AI adoption increasing rapidly, it has become increasingly common for hardware vendors to create specialist AI processors that run inference and/or training more efficiently than would be possible with commercial-off-the-shelf hardware. While these custom processors can offer the advantage of performance, they come with challenges for developers as it often involves porting software to proprietary and non-standard programming models. oneAPI Construction Kit (OCK) has been created by CPLAY to solve this problem by bringing all the benefits of oneAPI and SYCL to new and custom hardware. OCK was developed out of ComputeAorta-- a part of the ACORAN toolchain developed by CPLAY, and its goal is to expose the full performance potential of heterogenous hardware and to provide standards compliant interfaces for developers.

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.

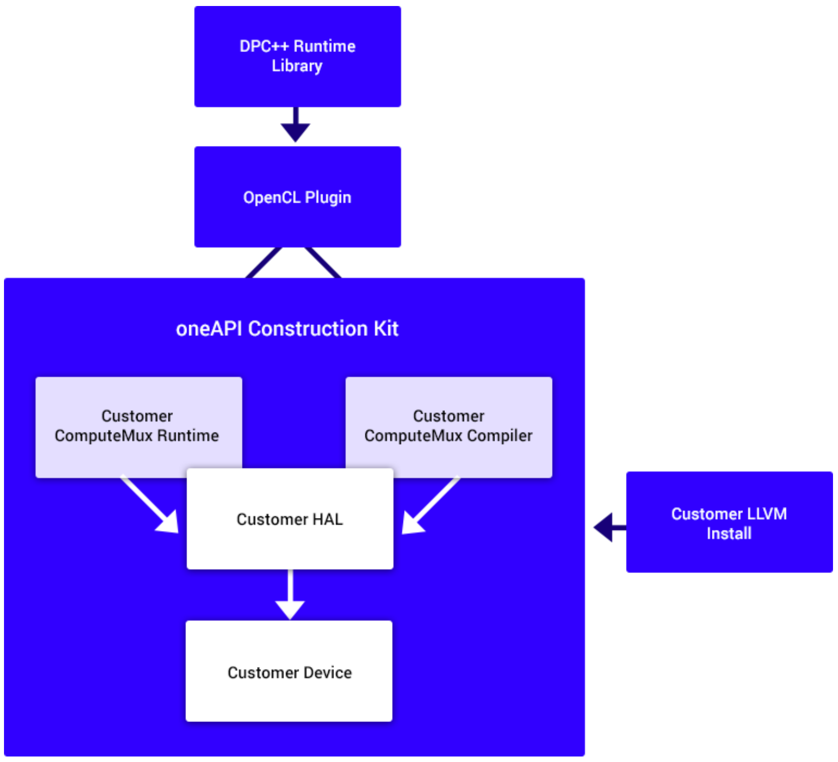


Figure 2: Components of the oneAPI construction kit

[1] https://uxlfoundation.org/

The DPC++ runtime allows OpenCL to be used as a plugin. OCK supports this interface and provides runtime and compiler modules. Support for new accelerators is done by developing a new custom target. A custom target is made up of three key parts:

- Runtime code (ComputeMux Runtime)
- Compiler code ComputeMux Compiler)
- A HAL (Hardware Abstraction Layer)

The runtime code will run on the host device and will interface with the target device. It will handle aspects such as allocation of and reading/writing memory, queuing of commands and executing kernels on the device. The compiler code is typically based on a number of LLVM passes that will turn the original kernels into something matching the interfaces required to run a kernel on the device. The optional HAL gives static information and simplified runtime interfaces to a device to make getting started easier.

During this first half of the SYCLOPS project, we have worked on completely open sourcing OCK (https://github.com/codeplaysoftware/oneapi-construction-kit). Detailed information about OCK is available on the developer documentation website (https://developer.codeplay.com/products/oneapi/construction-kit/home/). Further, in the context of SYCLOPS, we have extended DPC++ and OCK to support the two RISC-V targets that have been deployed in the SYCLOPS hardware platform, namely, the FPGA board with CSIP A730 RISV-V soft core processor, and the MilkV Pioneer with SOPHON SG2042 64-core RISC-V CPU. A detailed description of this hardware is available in deliverable *D3.1*.

2.1 OCK & CSIP RISC-V FPGA Board

To support the FPGA, we developed a remote HAL, which is essentially a shim layer to enable offloading kernel code to a device that may be located remotely (Relevant PRs: https://github.com/codeplaysoftware/oneapi-construction-kit/pull/437, https://github.com/codeplaysoftware/oneapi-construction-kit/pull/434). The remote HAL server uses a socket connection to communicate with a remote client. This can be cross-compiled as needed. It is a small program which can be modified to use different ways of transmission or different underlying HAL interfaces. The server will be paired with a built oneapi-construction-kit OpenCL interface using the HAL socket client.

When run, the HAL server requires a port which will be listened on. This can be any free user port. It also will only accept connections from specified nodes. This can be an IP address or host name e.g. "127.0.0.1" or "localhost". This defaults to "127.0.0.1", assuming that ports will be forwarded if the client is not run on the same machine. The server will accept one connection, which when completed will end the program. This is to avoid programs putting the cpu into a bad state. The HAL client by default supports a socket connection only to the current node and it expects the server to be running on the same machine. To be able to run the server on a different machine (for example where we have a RISC-V device running Ubuntu), one can use ssh port forwarding. Running the client is done similar to any normal OCK OpenCL target (or via SYCL OpenCL plugin).

We worked together with EUR and CSIP in getting the remote HAL server executed on the CSIP FPGA board. With this, we could treat the RISC-V soft core on the FPGA board as an accelerator and use OCK to offload kernels to it. For each kernel launch OCK runs under the hood a tool called vecz which is a whole function vectorizer to combine a number (typically the vector width supported by the hardware) of work items into a vectorized kernel. If that vectorization succeeded the original kernel launch over a range/ND range in SYCL is then adapted by OCK LLVM passes to launch the vectorized kernel instead over the range divided by the vector width. One the device side OCK has a math library called Abacus to provide builtins with the precision required by OpenCL. Details about end-to-end experiments performed to validate this setup are described in deliverable *D3.1*.

2.2 OCK, DPC++, & MilkV Board

On the MilkV board, we have two different compilation and execution paths for a SYCL program. The first path is to natively compile a program to run on the RISC-V CPU. To enable this, we have built the latest version of DPC++ on MilkV targeting the SOPHON SG2024 64-core RISC-V CPU, and were able to demonstrate building SYCL applications with kernels running on OpenCL CPU target, and even on the new still experimental DPC++ NativeCPU target which uses the OCK vectorizer and LLVM passes to adapt the work item loops. The second path is to use OCK as the target. To enable this, we have successfully configured and built OCK on the MilkV board. This build provides the OpenCL driver for RISC-V that enables running OpenCL and SYCL kernels on the CPU. Experimental results evaluating DPC++ on MilkV are described in deliverable *D3.1*.

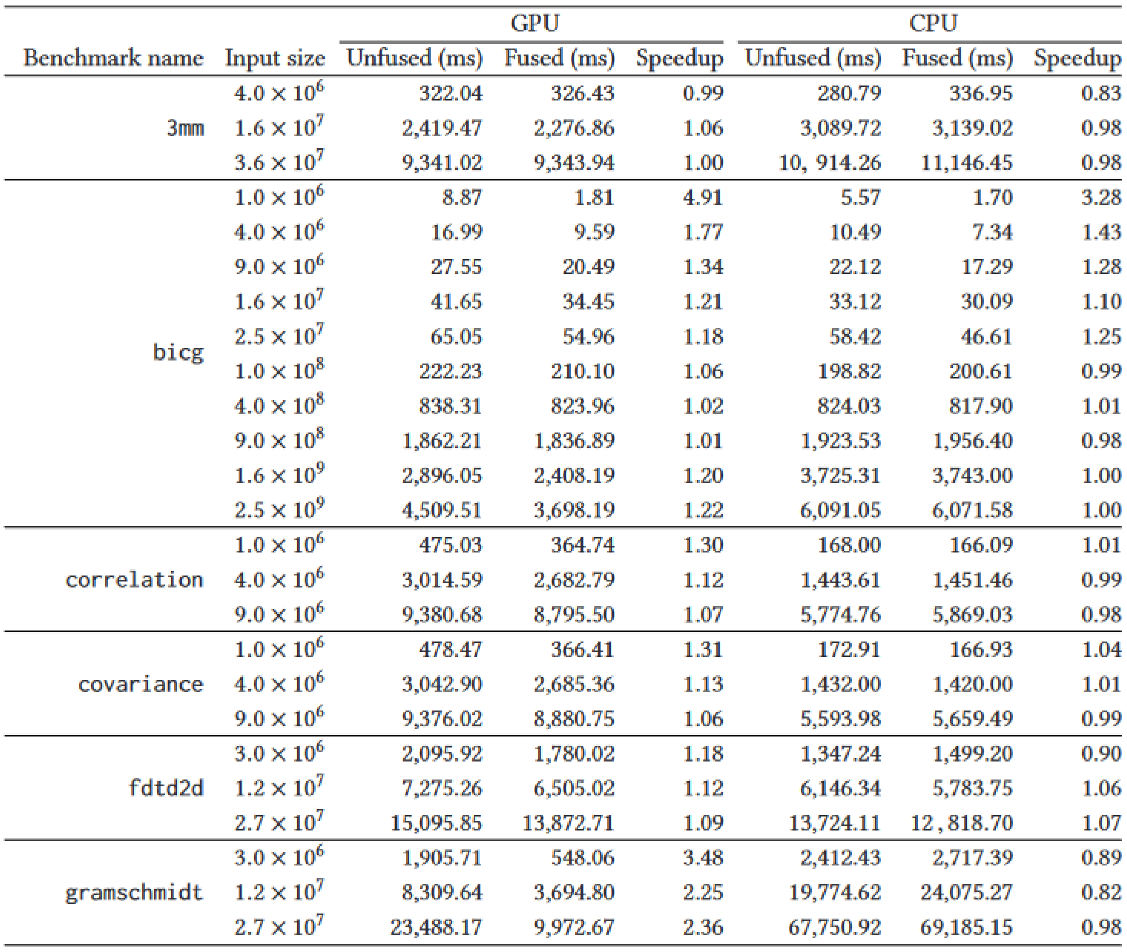
All work on OCK to enable RISC-V support has been made available on the OCK Github (https://github.com/codeplaysoftware/oneapi-construction-kit.git). Fixes related to RISC-V code generation were submitted as pull requests for the main llvm repository (https://github.com/llvm/llvm-project) from where they will flow into the OCK and DPC++ repositories. We have created internal Gitlab CI jobs to run the SYCL and OpenCL CTS on RISC-V (CPU), with the view to make the results publicly available. Internally, we had to update our CI to use the latest Ubuntu images to be able to support RISC fp16 in Gitlab CI (Note that we aim to target Github CI later in the project to be able to open the CI up to the community). In the second half of the project we will focus on performance optimisations for RISC-V, targeting RVV, improve threading, integrating JIT compilation, finalising cross-compilation.

2.3 DPC++ & Kernel Fusion

Today, many computational tasks require heterogeneous computing for their efficient solution. However, every kernel launch on an accelerator carries some runtime overhead for tasks such as synchronization. This effect is particularly pronounced for a sequence of very short-running device kernels. One solution to help achieve the benefit of accelerators is to fuse multiple smaller kernels into a single, larger kernel. The fused kernel is better able to amortize the overhead for device kernel launch, due to the better compute/overhead ratio.

We have developed an extension for the SYCL standard which lets the user decide when and which kernels to fuse, and then completely automates the creation of the fused kernel implementation in the SYCL runtime. This relieves the user of the burden to manually implement a fused kernel, and allows for fast performance exploration. Current DPC++ development builds and daily releases already by default include support for kernel fusion on Intel CPUs and GPUs, Nvidia GPUs and AMD GPUs.

Figure 3 shows the fused and unfused times when running benchmarks from SYCL-BENCH on an Intel CPU and an Intel iGPU. SYCL-Bench is a benchmark suite specifically designed for SYCL. In addition to allowing for a performance characterization of devices and different SYCL implementations, the different SYCL-specific benchmarks also present optimization opportunities for SYCL runtime implementations to exploit. When applying fusion optimizations to the SYCL-Bench benchmark suite, we found that six out of nine benchmarks that launch more than one kernel were amendable to the kernel fusion. As it can be seen, speedups are less notable on CPU in this case, getting little to no improvement in most cases, except for the bicg benchmark, also presenting speedups in the range of up to 4.91× with a mean of 1.59× on GPU. The correlation and covariance benchmarks both present speedups of around 1.30× for small input sizes when run on GPU. Also, benchmark gramschmidt presents no improvement on CPU, but an improvement of up to 3.48× on GPU. More information on how to use kernel fusion together with a detailed characterization of performance is documented in [4,5].



		GPU			CPU		
Benchmark name	Input size	Unfused (ms)	Fused (ms)	Speedup	Unfused (ms)	Fused (ms)	Speedup
3mm	4.0×10^6	322.04	326.43	0.99	280.79	336.95	0.83
	1.6×10^7	2,419.47	2,276.86	1.06	3,089.72	3,139.02	0.98
	3.6×10^7	9,341.02	9,343.94	1.00	10, 914.26	11,146.45	0.98
bicg	1.0×10^6	8.87	1.81	4.91	5.57	1.70	3.28
	4.0×10^6	16.99	9.59	1.77	10.49	7.34	1.43
	9.0×10^6	27.55	20.49	1.34	22.12	17.29	1.28
	1.6×10^7	41.65	34.45	1.21	33.12	30.09	1.10
	2.5×10^7	65.05	54.96	1.18	58.42	46.61	1.25
	1.0×10^8	222.23	210.10	1.06	198.82	200.61	0.99
	4.0×10^8	838.31	823.96	1.02	824.03	817.90	1.01
	9.0×10^8	1,862.21	1,836.89	1.01	1,923.53	1,956.40	0.98
	1.6×10^9	2,896.05	2,408.19	1.20	3,725.31	3,743.00	1.00
	2.5×10^9	4,509.51	3,698.19	1.22	6,091.05	6,071.58	1.00
correlation	1.0×10^6	475.03	364.74	1.30	168.00	166.09	1.01
	4.0×10^6	3,014.59	2,682.79	1.12	1,443.61	1,451.46	0.99
	9.0×10^6	9,380.68	8,795.50	1.07	5,774.76	5,869.03	0.98
covariance	1.0×10^6	478.47	366.41	1.31	172.91	166.93	1.04
	4.0×10^6	3,042.90	2,685.36	1.13	1,432.00	1,420.00	1.01
	9.0×10^6	9,376.02	8,880.75	1.06	5,593.98	5,659.49	0.99
fdtd2d	3.0×10^6	2,095.92	1,780.02	1.18	1,347.24	1,499.20	0.90
	1.2×10^7	7,275.26	6,505.02	1.12	6,146.34	5,783.75	1.06
	2.7×10^7	15,095.85	13,872.71	1.09	13,724.11	12 , 818.70	1.07
gramschmidt	3.0×10^6	1,905.71	548.06	3.48	2,412.43	2,717.39	0.89
	1.2×10^7	8,309.64	3,694.80	2.25	19,774.62	24,075.27	0.82
	2.7×10^7	23,488.17	9,972.67	2.36	67,750.92	69,185.15	0.98

Figure 3: Results of kernel fusion under SYCL-BENCH benchmark suite.

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 foocus 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.

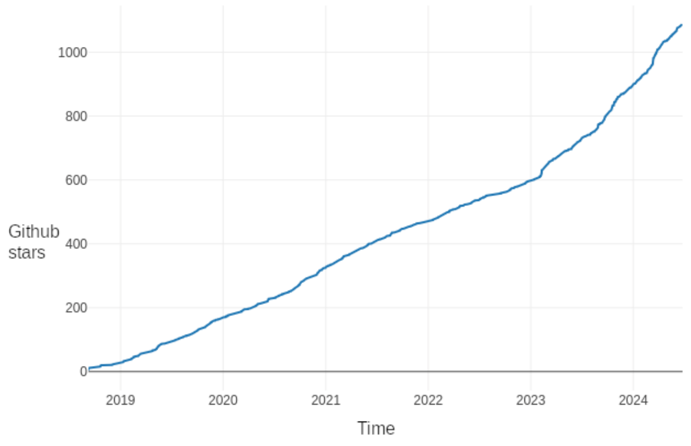


Figure 4: Github stars for AdaptiveCpp

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

Finally, we worked together with other partners in building and configuring AdaptiveCpp on the SYCLOPS hardware platform, and demonstrating that SYCL kernels can be executed on GPU and RISC-V accelerators. Experimental results validating AdaptiveCpp and benchmarking its performance with respect to DPC++ are described in deliverable *D3.1*. In the rest of this section, we explain technical changes introduced in AdaptiveCpp is more detail.

3.1 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

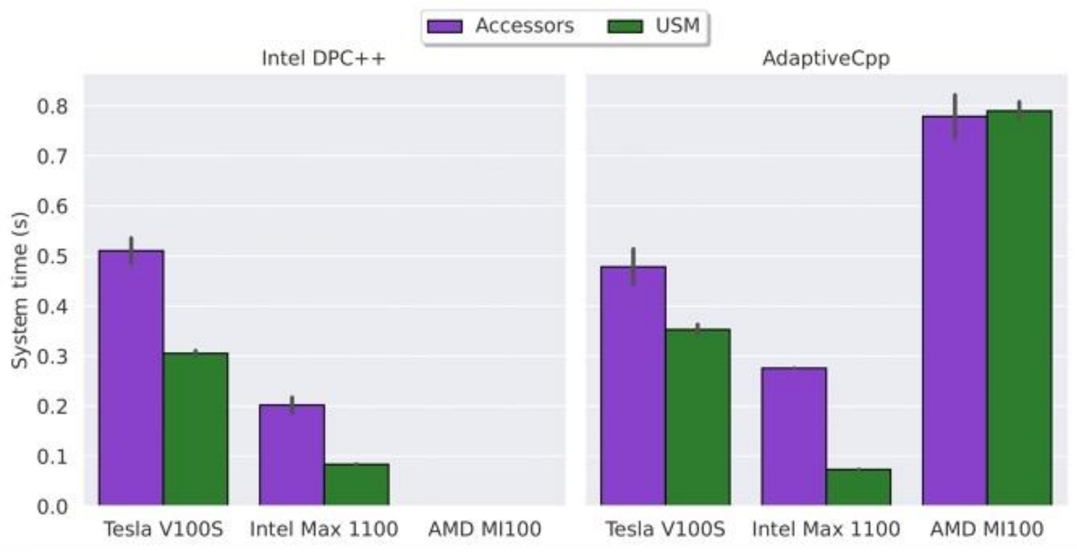


Figure 5: Kernel launch latency under various backends

aspects of the runtime. For example, Figure 5 (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.

3.2 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 and the SYCL 2020 reduction.

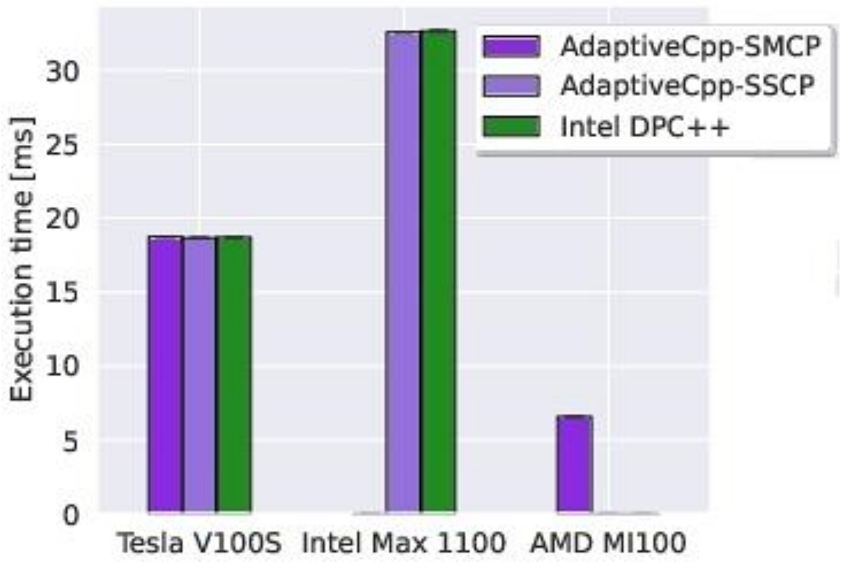


Figure 6: Atomic test results from SYCL-Bench

Atomics in the generic JIT compiler are implemented using a builtin interface that is implemented in backend-specific LLVM bitcode 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 (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.

3.3 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 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 consider 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 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++.

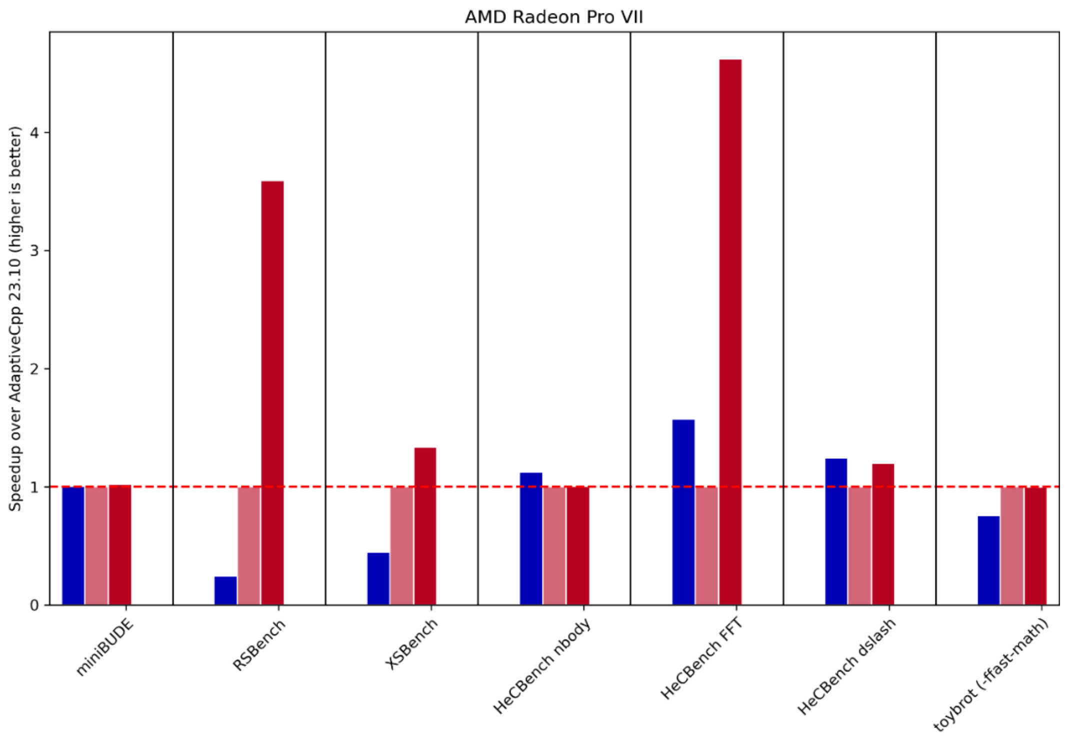


Figure 7: Performance comparison of AdaptiveCpp and DPC++ on AMD GPU

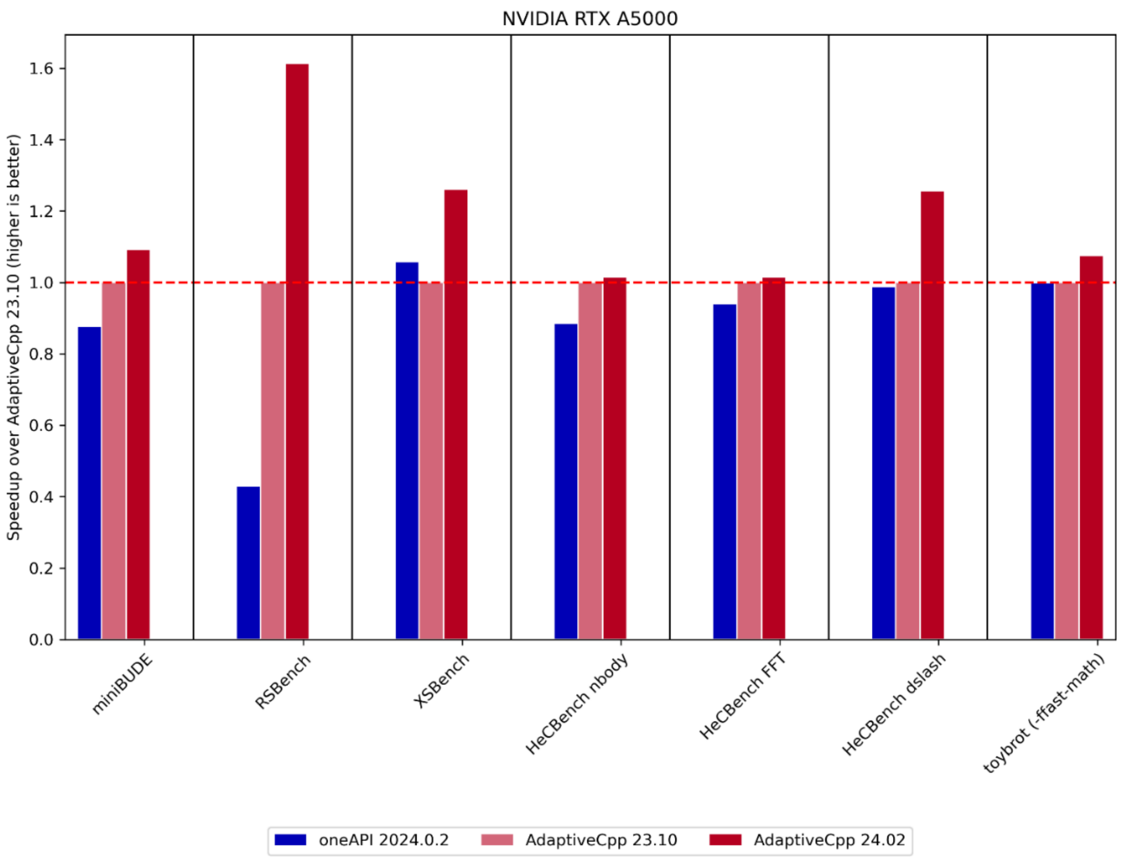


Figure 9: Performance comparison of AdaptiveCpp and DPC++ on NVIDIA GPU

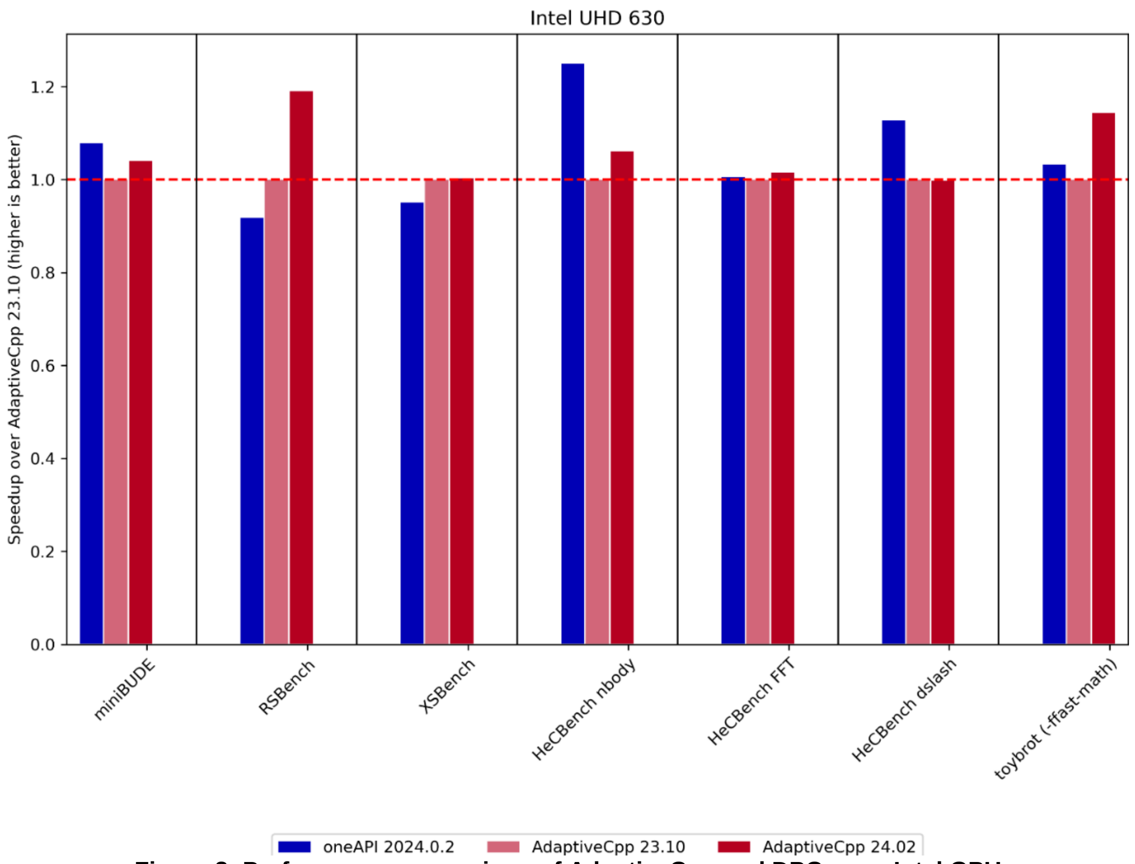


Figure 8: Performance comparison of AdaptiveCpp and DPC++ on Intel GPU

Recently, 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 very new, and the performance evaluation is still ongoing. The primary cost of this optimization is a slight additional kernel launch latency, as for every kernel launch, the kernel arguments must be investigated and processed to decide whether to attempt to launch a generic or a specialized kernel. Because a new kernel is only JIT-compiled when a kernel has already been invoked a substantial number of times with the same argument, the additional JIT overhead so far appears to not matter much for the averaged performance, especially over the course of multiple application runs. 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. For the future, additional optimizations at adaptivity level 2 are planned.

3.4 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. We therefore plan to propose this extension for standardization in future versions of SYCL.

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 behavior 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.

3.5 Standard C++ parallelism offloading support

Based on user feedback, and because we believe that the infrastructure will be useful for other SYCL-related work in the future, we have added support for offloading calls to C++ parallel STL algorithms on top of our SYCL compiler and runtime infrastructure.This allows users to formulate their program in terms of high-level C++ algorithms such as std::for_each, std::copy or std::transform_reduce, but still drop to the lower-level SYCL layer as more control is needed, e.g. for optimizations. This can be beneficial for programmer productivity in general, as well as accessibility of heterogeneous computing to programmers who lack deeper expertise. Because not everything can be expressed with standard C++ algorithms that can be expressed with SYCL, the SYCL layer is still very much needed and we see the two approaches as complementing each other. The AdaptiveCpp standard C++ algorithms are supported with our generic JIT compiler on all backends, including OpenCL via the construction kit, and thus eventually RISC-V hardware. Our publication on the matter [1] discusses the implementation detail.

Because C++ has a flat memory model, it is unaware that there might be multiple devices in the system with distinct memory spaces. Therefore, a primary challenge for the C++ standard parallelism model for offloading is that of making all system memory device-accessible. In some cases, e.g. when host and device are tightly integrated and share the same physical memory, or Linux HMM (heterogeneous memory management) is enabled, this might just work out of the box. In other cases, the compiler and runtime will have to remap all memory allocation and deallocation requests to device-aware variants. AdaptiveCpp does this by remapping allocations to sycl::malloc_shared, which returns allocations that may automatically migrate between host and device. This remapping is non-trivial, and we refer to [1] for details.

AdaptiveCpp employs additional optimizations that are not implemented by C++ standard parallelism offloading compilers from hardware vendors (NVIDIA's nvc++, AMD's roc-stdpar, Intel's icpx/DPC++). This includes automatically emitting prefetch operations to migrate data used by kernels, to elide unnecessary synchronization at the end of kernels, and an offloading heuristic that attempts to estimate whether it is worth offloading a C++ standard algorithm on the device, as opposed to the host.

Because in standard C++ parallelism math functions or other functionality like atomics needs to work in terms of functionality from the std:: C++ namespace, not the sycl:: namespace, we have added additional compiler transformations that remap C++ std math functions to

AdaptiveCpp math builtins and std atomic functionality to AdaptiveCpp atomic builtins. This allows using std:: math builtins and std::atomic or std::atomic_ref to be used in device code.

Figure 9 illustrates the AdaptiveCpp standard C++ parallelism performance for three mini-apps relative to the respective hardware vendor compilers and their support for standard C++ parallelism offloading. The red line indicates performance parity, and the blue line performance within 20%. XNACK refers to a hardware feature on AMD GPUs that is required to enable proper support for automatically migrating memory between host and device. Without, the ROCm stack runs in a degraded mode. In our experience, XNACK is rarely available on most production systems because it is not supported by all AMD GPUs, and requires non-standard Linux kernel boot parameters.
AdaptiveCpp outperforms the vendor compilers on all platforms for two out of three mini-apps, sometimes by an order of magnitude. As can be seen from the results, it is also less dependent on the availability of XNACK on AMD hardware. Overall, the results indicate that AdaptiveCpp can generate highly competitive code on all three platforms in the standard C++ parallelism model, and is the only standard parallelism solution that can target them all robustly. These results were obtained using the generic JIT compiler on all hardware.

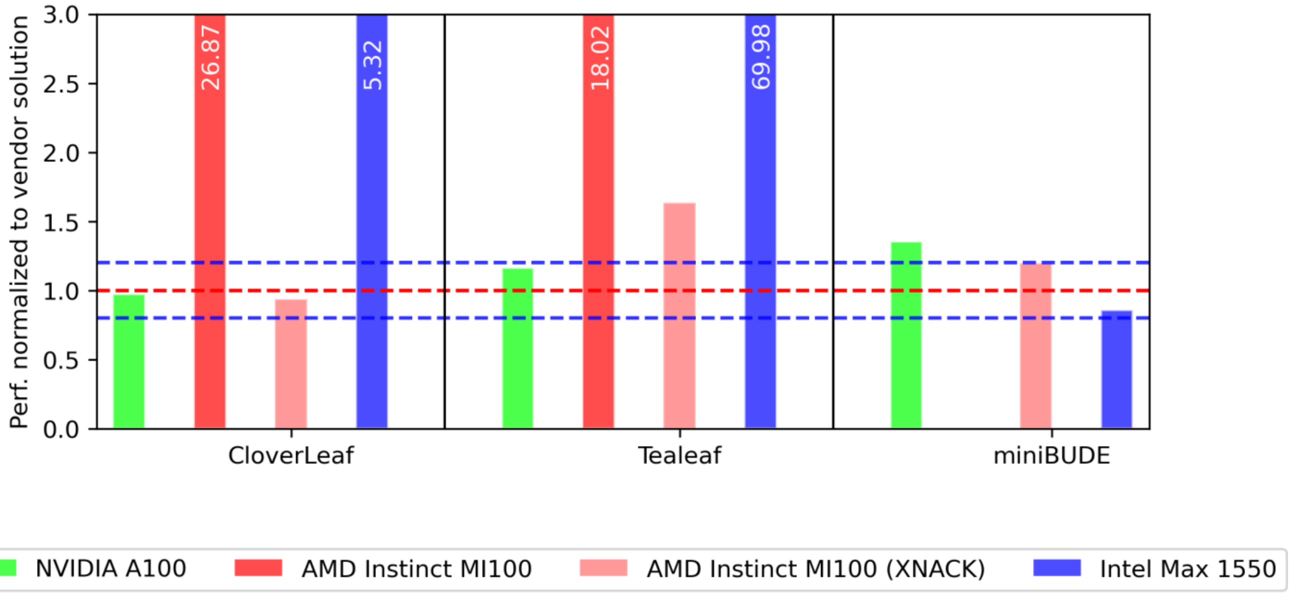


Figure 10: Performance of AdaptiveCpp's C++ parallelism on several GPU backends

3.6 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 is scheduled to be released in early July. In order to better structure development and provide stronger guarantees for users, a fixed for month release schedule was adopted, with the next release after 24.06 being planned in late October (version 24.10).

4 Conclusion

At M18, both our compiler toolchains (DPC++ and AdaptiveCpp) have been significantly advanced both in their support for RISC-V and in terms of new SYCL features. We have integrated both compilers and performed end-to-end evaluation using v1.0 of SYCLOPS platform as described in deliverable *D3.1*. 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://codeplay.com/portal/blogs/2023/06/21/using-the-sycl-kernel-fusion-extension-a-hands-on-introduction
[5] https://dl.acm.org/doi/10.1145/3571284