SYCLOPS

Deliverable 2.3 – Use case integration, validation, and demonstration report

GRANT AGREEMENT NUMBER: 101092877

This project has received funding from the European Union's HE research and innovation programme under grant agreement No 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**

D2.3 – Use case integration, validation, and demonstration report

Executive Summary: This deliverable is the final technical deliverable of the SYCLOPS project and summarizes the work done in integrating various components of the SYCLOPS hardware—software stack and deploying them in the context of the three use cases.: Particle Acceleration (HEP), Genomics, and Autonomous Systems. This document is a detailed account of our integration work that concretely demonstrates that the SYCLOLPS project has far exceeded the original KPI targets for each use case, and in doing so, has successfully demonstrated the power of performance-portable, open-standard hardware acceleration.

WP:	2		
Author(s):	Jan Kastil, Martin Bozek, Fred Buining, Nimisha Chaturvedi, Aleksander Illic, Maksymilian Graczyk, Monica Dessole, Devajith Valaparambil Sreeramaswamy, Raja Appuswamy		
Editor:	Raja Appuswamy		
Leading Partner:	CERN		
Participating Partners:	All		
Version:	1.0	**Status:**	Draft
Deliverable Type:	R	**Dissemination Level:**	PU
Official Submission Date:	31-Dec-2025	**Actual Submission Date:**	31-Dec-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	Skeleton template and outline	EUR
0.2	Particle acceleration use case description	CERN
0.3	Genomics use case description	ACC, INESC, EUR
0.4	CPLAY use case description	EUR
0.5	Infratructure integration	HIRO, CSIP
1.0	Completed draft	EUR

Authors

Author	Partner
Jan Kastil	CSIP
Martin Bozek	CSIP
Fred Buining	HIRO
Nimisha Chaturvedi	ACC
Aleksander Illic	INESC
Maksymilian Graczyk	CERN
Monica Dessole	CERN
Devajith Valaparambil Sreeramaswamy	CERN
Raja Appuswamy	EUR

Reviewers

Name	Organisation
Raja Appuswamy	EUR
Aleksander Illic	INESC
Vincent Heuveline	UHEI
Nimisha Chaturvedi	ACC
Stefan Roiser	CERN

Table of Contents

Executive Summary

This deliverable is the final technical deliverable of the SYCLOPS project and summarizes the work done in integrating various components of the SYCLOPS hardware—software stack and deploying them in the context of the three use cases: Particle Acceleration (HEP), Genomics, and Autonomous Systems.

In the HEP use case led by CERN, we have integrated SYCL support into the ROOT framework, and the Cling interpreter, enabling interactive, hardware-accelerated data analysis in Jupyter Notebooks. The newly developed GenVectorX library demonstrated up to 3.5x speedup on heterogeneous backends and reduced energy consumption by ~73.9%.

In the genomics use case led by ACCELOM, we have developed SYCL-GAL, a library of accelerated primitives that reduced total execution time for the GATK germline variant calling pipeline by ~4.6x. Preprocessing stages saw an 11x improvement, and the core pairHMM computation in variant calling stage was accelerated by two orders of magnitude, effectively removing a major industry bottleneck.

In the autonomous systems use case led originally by CPLAY, and now by EUR after CPLAY's exit, we unified the portability of portDNN with the oneDNN ecosystem, allowing the PointNet architecture to run across diverse hardware without code changes. Convolution kernels optimized for RISC-V vector extensions achieved performance gains of up to 18.8x.

This document is a detailed account of our integration work that concretely demonstrates that the SYCLOLPS project has far exceeded the original KPI targets for each use case, and in doing so, has successfully demonstrated the power of performance-portable, open-standard hardware acceleration.

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.

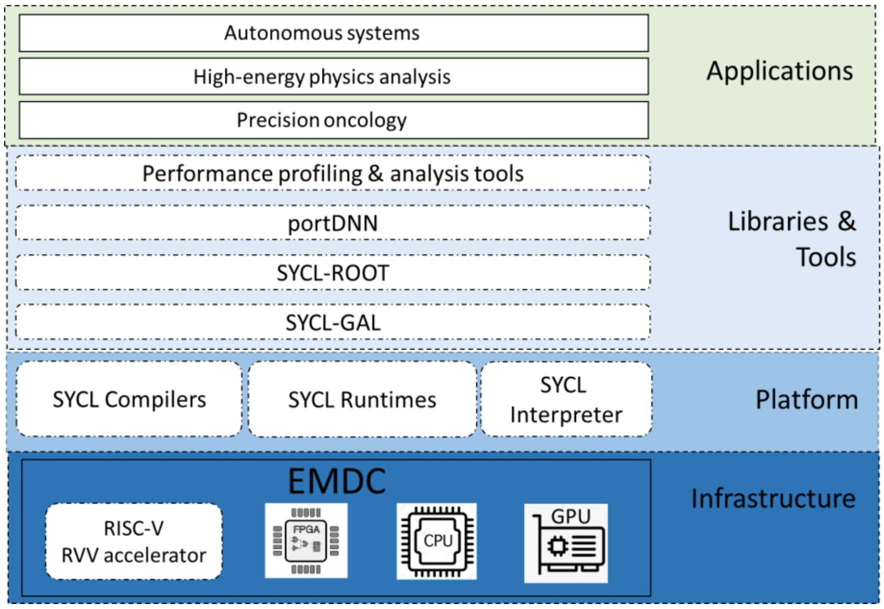


Figure 1: SYCLOPS architecture

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 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 from UHEI, and the Cling interpreter from CERN.

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 presents the work carried out in in the context of “Task 2.3: Use cases & SYCLOPS validation” of the SYCLOPS project. This task focuses on end-to-end validation of the SYCLOPS stack using the three use cases. In this task, the use case partners performed full integration of the libraries developed in WP5 into their respective pipelines. All partners then collaborated on executing these pipelines, using runtime tools developed in WP4, on SYCLOPS hardware developed in WP3, with the goal of using the full end-to-end evaluation methodology developed in task 2.2 to compare at a global level (i) the benefit of hardware acceleration in various use cases compared to their non-accelerated solutions, (ii) the difference in performance/energy efficiency various accelerators in SYCLOPS, (iii) the efficacy of open, standards-based SYCLOPS stack compared to other proprietary solutions.

This deliverable is structured as follows. Sections 2, 3, and 4 provide an overview of integration work with respect to each of the three use cases in SYCLOPS, and spans the top three layers of the SYCLOPS stack shown in Figure 1. Section 5 details the integration effort at the hardware level in the Infrastructure layer of SYCLOPS stack.

2 Particle Acceleration Use Case

2.1 State-of-the-art Before SYCLOPS

High Energy Physics (HEP) research is characterised by the need for processing and analysing huge amounts of particle collision data coming from the accelerators. ROOT is a popular tool for storing, analysing and visualising physics data regarding particle collisions. These collision events are expressed as operations on particles, represented as 4-dimensional time-space vectors, also known as Lorentz Vectors. Within ROOT, the GenVector package contains classes for specialised vectors in 2, 3 and 4 dimensions, and their operations, providing models and capabilities tailored to HEP analysis. The largest source of such data is the Large Hadron Collider (LHC), hosted at CERN in Switzerland, which since its start has collected more than 2 EB of data generated from physics events that need to be stored and accessed by the physics community to be analysed. The need for compute power and data storage is expected to increase dramatically in the next years, as in 2030 the High Luminosity LHC (HL-LHC), an upgraded hardware configuration of the LHC particle accelerator, will start operating with a demand of computational resources that is estimated more than 10 times with respect to the current use. In this scenario, in which scientists need to run their analysis on computing facilities exhibiting different hardware configurations and composition, performance portability plays a crucial role. Such considerations motivate the need for developing performance portable software tailored to the HEP use case. In particular, the KPIs addressed in this use case are the following:

- KPI 6: A new interpreted SYCL execution environment will enable Jupyter-Notebook-based, hardware-accelerated, ad-hoc data analytics that is at least 2x faster than the CPU-based execution environment.
- KPI 8: A new hardware-accelerated library of linear algebra operators will enable HEP acceleration.
- KPI 10: Three applications domains will successfully demonstrate that their end-to-end pipelines that integrate SYCL libraries can be portably deployed, with no code change in application logic, on several accelerators.
- KPI 11: Detailed evaluation will demonstrate that (i) Use cases can portably run their SYCL-based accelerated pipelines on multiple processors and achieve at least 2x improvement in latency and/or throughput and/or energy efficiency (depending on the accelerator used) compared to non-accelerated versions, (ii) SYCL-based libraries can achieve performance comparable to their CUDA counterparts when appropriate.
- KPI 14: Establish close ties with research, academic, industrial partners, and conferences with tutorials, submitted papers/presentations.

2.2 Cling Integration

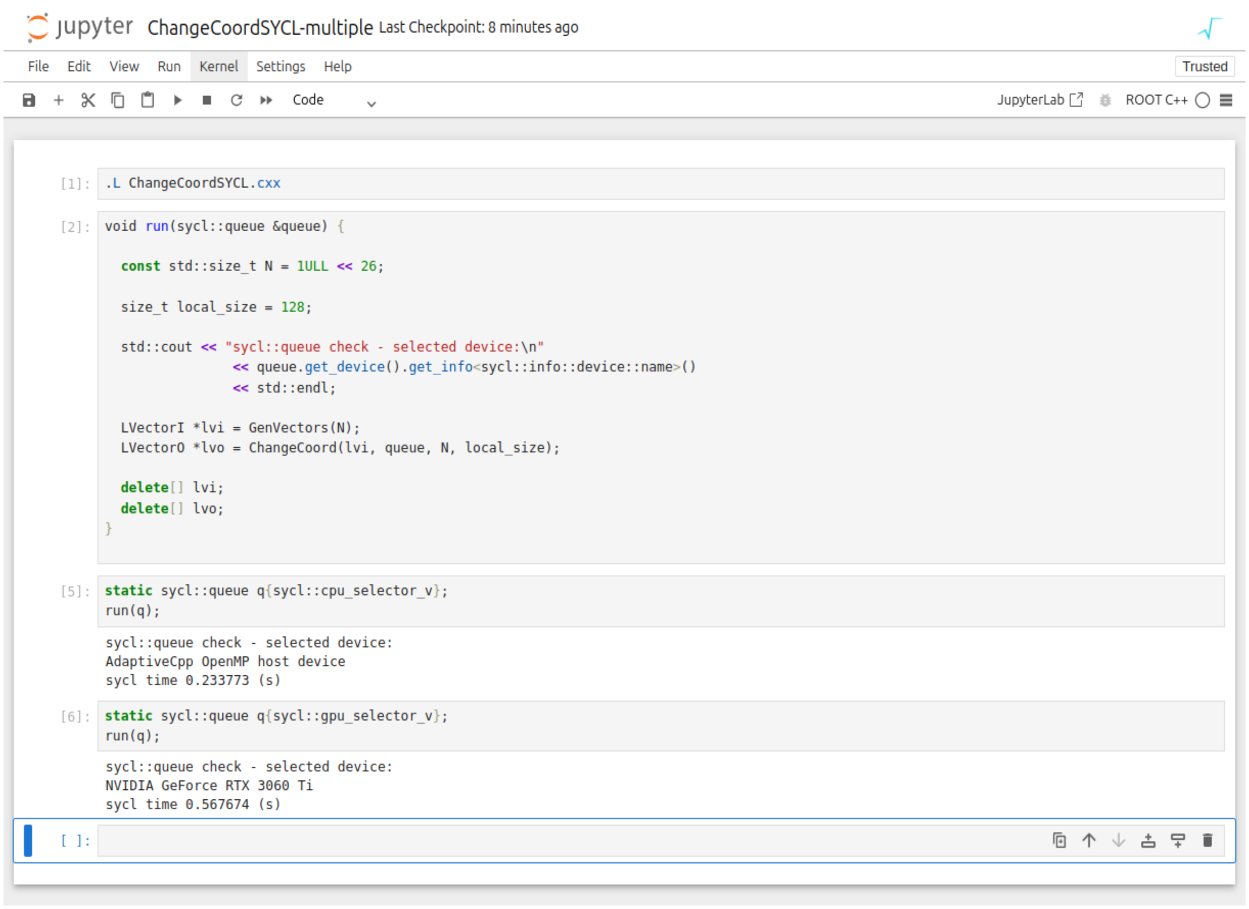
Cling, the interactive C++ interpreter that sits at the centre of ROOT, is essential for exploratory data analysis in High Energy Physics. Before the SYCLOPS project, Cling was tied to an older LLVM toolchain and did not have proper support for modern heterogeneous programming models such as SYCL. CUDA was supported, but only in a limited and not very flexible way. As a result, users were mostly restricted to CPU workflows, and there was no straightforward way to experiment interactively with SYCL or accelerator oriented C++ inside ROOT or inside Jupyter.

During SYCLOPS, a large part of the work focused on modernising this foundation so that ROOT could eventually support interactive heterogeneous computing. The first major task was to upgrade the LLVM backend in Cling, first to **LLVM 18** and later to **LLVM 20**. This was a difficult change because ROOT relies on a very large test suite, with more than three thousand tests accumulated over many years. These tests caught subtle regressions and long standing assumptions about older LLVM behaviour. The upgrade required changes in the JIT engine, in the incremental compilation logic, and in ROOT's dictionary generation pipeline. Since all of these depend closely on LLVM internals, some upstreaming effort was also needed. This work was necessary in order to make ROOT compatible with newer C++ standards such as C++20 and also to prepare the ground for AdaptiveCpp, which requires a modern LLVM toolchain for SYCL code.

Once this modernisation was in place, we introduced SYCL support directly in Cling as part of the SYCLOPS effort, and this support has now been merged into ROOT starting with version 6.38, thus achieving KPI 6. The feature is available when ROOT is built with the **-Dexperimental_adaptivecpp** flag. With this, users can write and run SYCL kernels interactively inside ROOT, which also means inside Jupyter notebooks. We prepared simple tutorials to help new users get started.

Functional testing and performance checks were carried out for both CPU (OpenMP) and GPU (CUDA) backends through the interpreter. The behaviour was consistent across both targets. We also validated the portability of the approach by building and running SYCL-Cling on RISC-V systems. This confirmed that the new infrastructure can serve as a general platform for heterogeneous development.

This reduces the entry barrier for physicists who want to try GPU acceleration without leaving the ROOT environment. The work also opens the door to teams at CERN and outside who are actively investigating heterogeneous computing models and need an environment where they can iterate quickly.


jupyter ChangeCoordSYCL-multiple Last Checkpoint: 8 minutes ago
File Edit View Run Kernel Settings Help
Trusted
Code
JupyterLab
ROOT C++

```
[1]: .L ChangeCoordSYCL.cxx

[2]: void run(sycl::queue &queue) {

       const std::size_t N = 1ULL << 26;

       size_t local_size = 128;

       std::cout << "sycl::queue check - selected device:\n"
                 << queue.get_device().get_info<sycl::info::device::name>()
                 << std::endl;

       LVectorI *lvi = GenVectors(N);
       LVectorO *lvo = ChangeCoord(lvi, queue, N, local_size);

       delete[] lvi;
       delete[] lvo;
     }

[5]: static sycl::queue q{sycl::cpu_selector_v};
     run(q);

     sycl::queue check - selected device:
     AdaptiveCpp OpenMP host device
     sycl time 0.233773 (s)

[6]: static sycl::queue q{sycl::gpu_selector_v};
     run(q);

     sycl::queue check - selected device:
     NVIDIA GeForce RTX 3060 Ti
     sycl time 0.567674 (s)

[ ]:
```

Figure 2: An example from GenVectorX running in a Jupyter Notebook on multiple backends (OpenMP and CUDA) without a change in application logic.

2.3 SYCL-ROOT Integration

Within the SYCLOPS project, we extended GenVector to GenVectorX, an accelerated library that provides both a CUDA and a SYCL implementation of the Lorentz Vector classes that facilitate computations with physical vectors. GenVectorX has been merged into ROOT starting with version 6.38, thus achieving KPI 8. With reference to KPI 11, we analysed the impact of manual code specialisation upon developers with regards to code maintenance, with the explicit scope of minimising code duplication and maximising code reuse in order to promote code sustainability and portability. We evaluated code divergence (a measure of similarity between codebases ranging from 0 to 1) of GenvectorX versus GenVector for the Invariant Masses test case, for both SYCL and CUDA backends. We highlighted that the SYCL and CUDA variants share almost all the code with the CPU implementation, nonetheless the CUDA implementation can only target NVIDIA GPUs. Also, we compared the performance of some of the most common operations involving Lorentz Vectors on multiple platforms. We carried out an extensive test campaign on NVIDIA GPUs, with particular focus on the performance gap between native CUDA and SYCL code execution. Focusing again on the Invariant Mass computation problem, we studied scaling and demonstrate that reach performance portability, for almost all sizes of inputs. Moreover, two SYCL implementations - OneAPI and AdaptiveCPP – have been taken into account, as well as two memory access strategies, i.e. Device Pointers (PTR) and Buffers (BUF). We considered the following computational environments:

1. NVIDIA L4 using CUDA 12.3 (CERN)
2. NVIDIA A100 40GB PCIe using CUDA 12.2 (CERN)

3. AMD MI250X using ROCm 5.3.3 (LUMI supercomputer)
4. Risc-V platform Milk-V (EURECOM)

These results are detailed in “GenVectorX: A performance-portable SYCL library for Lorentz Vectors operations”, M. Dessole, J. Chen, A. Naumann, to appear on Journal of Physics, ACAT’24 Proceedings, 2024.

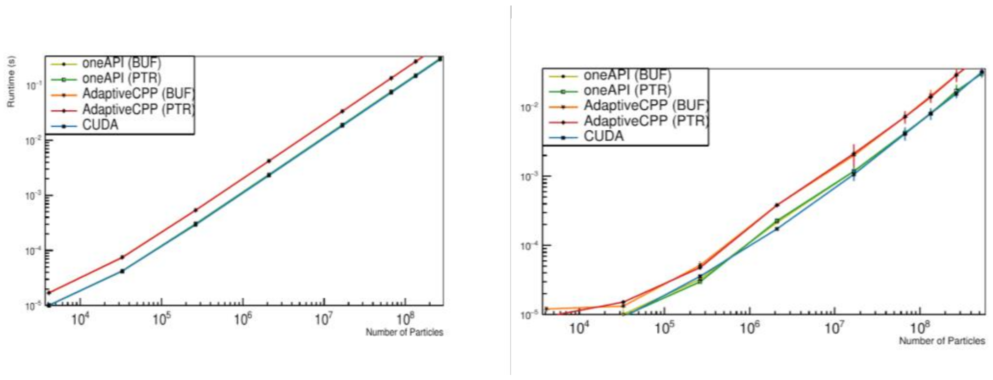


Figure 3: ***Scaling evaluated on NVIDIA GPUs L4 (left) and A100 (right) for the invariant masses test case. OneAPI and native CUDA implementations perform similarly, while AdaptiveCPP implementation is slightly less performant.***

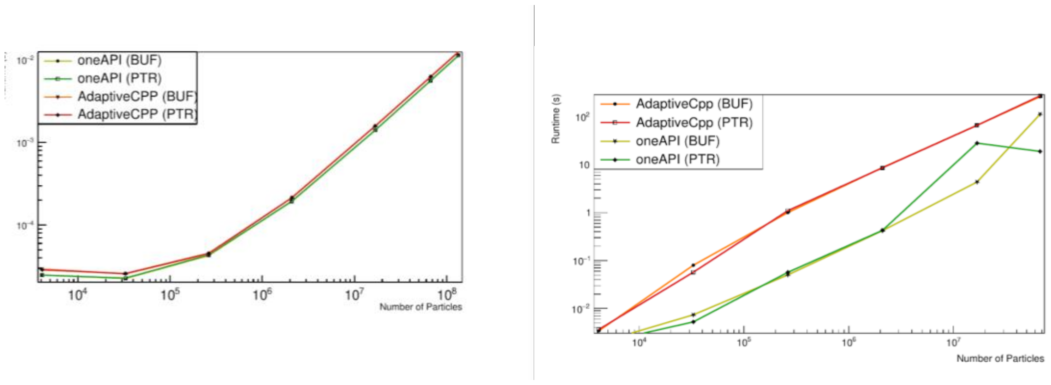
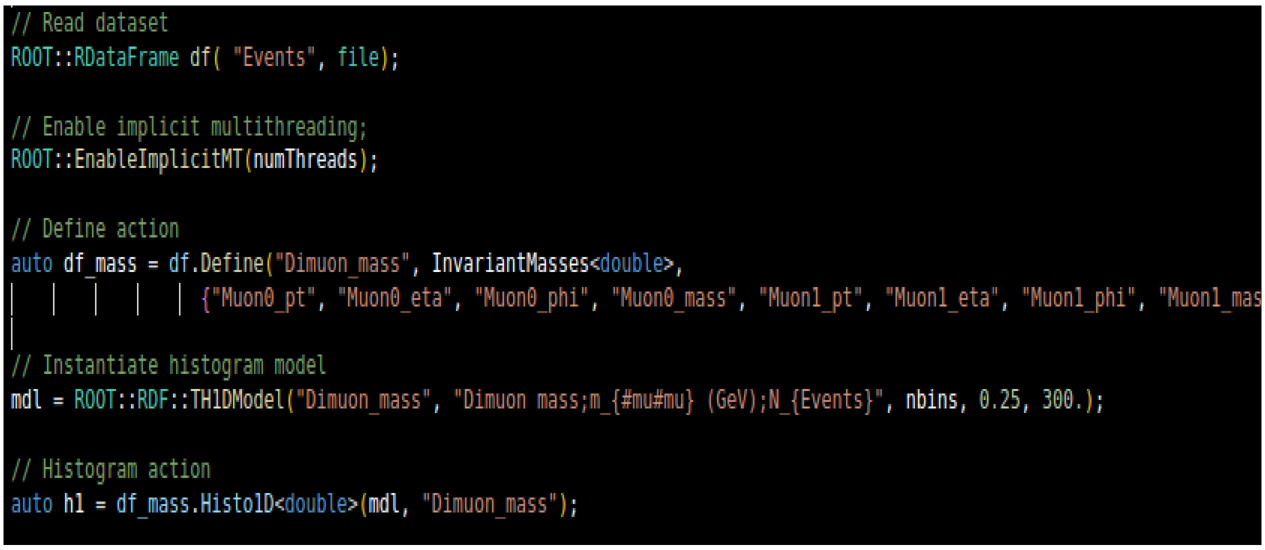


Figure 4: ***Scaling evaluated on AMD GPU MI250X (left) and Risc-V platform Milk-V (right) for the invariant masses test case. OneAPI and AdaptiveCPP implementations perform similarly on AMD GPU, while AdaptiveCPP implementation is slightly less performant on Milk-V.***

One of the main components of ROOT is RDataFrame, the high-level data analysis interface. RDataFrame represents physics data in a columnar format, where a row defines a collision event and the columns describe various characteristics for each event. A HEP analysis generally consists of iterating over the data from different events to apply filters and evaluating actions, like computing distributions (histogramming) and derive new columns (define). RDataFrame interface is designed to have easy-to-enable parallelism and portability, without requiring extensive knowledge of parallel computing and/or programming from its users. Currently, RDataFrame contains support for implicit parallelism in multi-threaded and multi-node distributed environments. Within SYCLOPS, we investigated the use of SYCL for enabling heterogeneous computing within RDataFrame. thus achieving KPI 10. In the current ROOT release (6.38), the actions are processed event-by-event, but recent developments include bulk-by-bulk processing of events. This provides a natural unit of data to offload to accelerators. The implicit parallelism mentioned previously is across bulks, but parallelism within bulks has not been implemented yet. In this project, we based our work on RDataFrame with the bulk API. Offloading the histogramming action alone has likely not enough computational intensity to fully benefit from GPU usage, as shown in “Migrating CUDA to SYCL: A HEP Case Study with ROOT RDataFrame”, J. Chen, M. Dessole, A.L. Varbanescu, IWOCL '24: Proceedings of the 12th International Workshop on OpenCL and SYCL, 2024.

Therefore, we prototyped a fused Define+Histogramming action. The GenVectorX library is used to provide the computational kernel to evaluate the new column. Moreover, the Define+Histogramming action can exploit CPU multithreading via RDataFrame engine for operations such as IO.



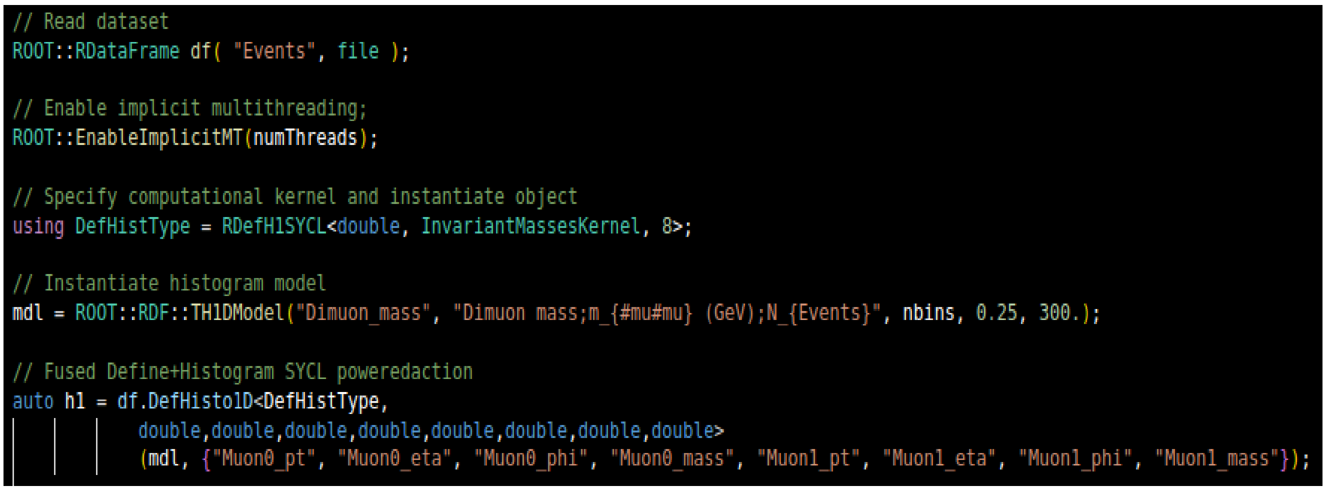
```
// Read dataset
ROOT::RDataFrame df( "Events", file);

// Enable implicit multithreading;
ROOT::EnableImplicitMT(numThreads);

// Define action
auto df_mass = df.Define("Dimuon_mass", InvariantMasses<double>,
|    |    |    |    | {"Muon0_pt", "Muon0_eta", "Muon0_phi", "Muon0_mass", "Muon1_pt", "Muon1_eta", "Muon1_phi", "Muon1_mas
|
// Instantiate histogram model
mdl = ROOT::RDF::TH1DModel("Dimuon_mass", "Dimuon mass;m_{#mu#mu} (GeV);N_{Events}", nbins, 0.25, 300.);

// Histogram action
auto h1 = df_mass.Histo1D<double>(mdl, "Dimuon_mass");
```

Figure 5: RDF API: DiMuon analysis example.



```
// Read dataset
ROOT::RDataFrame df( "Events", file );

// Enable implicit multithreading;
ROOT::EnableImplicitMT(numThreads);

// Specify computational kernel and instantiate object
using DefHistType = RDefH1SYCL<double, InvariantMassesKernel, 8>;

// Instantiate histogram model
mdl = ROOT::RDF::TH1DModel("Dimuon_mass", "Dimuon mass;m_{#mu#mu} (GeV);N_{Events}", nbins, 0.25, 300.);

// Fused Define+Histogram SYCL poweredaction
auto h1 = df.DefHisto1D<DefHistType,
|    |    |    double,double,double,double,double,double,double,double>
|    |    |    (mdl, {"Muon0_pt", "Muon0_eta", "Muon0_phi", "Muon0_mass", "Muon1_pt", "Muon1_eta", "Muon1_phi", "Muon1_mass"});
```

Figure 6: Prototype RDF API: DiMuon analysis example.

2.4 Adaptyst and Energy Consumption Analysis Integration

Adaptyst is an early-phase comprehensive performance analysis tool developed as the response to the fragmentation of the performance analysis world and the increasing need for going beyond purely algorithmic optimisations in computer systems. It is architecture-agnostic and aims to address software, hardware, and system needs of users in a future-proof way.

The project is explained in more detail in deliverable D2.2. However, it should be noted that since the submission of that deliverable, the new modular design of Adaptyst has been released. This version of the tool features a possibility of adding support for novel system and hardware components flawlessly via external modules. We provide two of these ourselves:

- linuxperf: encompassing the original functionality of Adaptyst, i.e. on-CPU and off-CPU profiling with process/thread tracing, support for low-level “perf” events (such as cache misses and retired instructions), and CPU cache-aware roofline modelling through the CARM Tool
- nvgpu: analysing performance of programs running on NVIDIA GPUs, currently by tracing NVIDIA CUDA API calls

Adaptyst is gaining more and more attention in various areas of CERN: apart from SYCL-ROOT and Cling with SYCL, there is an interest in using the tool in the Madgraph5 event generator used alongside ROOT for Large Hadron Collider data analysis, data processing in all four major LHC experiments, IT storage services etc.

In the context of the CERN use case in SYCLOPS, the tool has successfully profiled CPU-wise the DiMuonInvMasses.cxx ROOT script run in a single execution with ROOT/Cling with AdaptiveCpp (the Adaptyst system file used for this is in Figure 7). The dataset was pre-calculated earlier using Filter() in the same script. Two program variants were analysed: one without GenVectorX (arguments to DiMuonInvMasses(): 5, 20, false) and one with GenVectorX (arguments to DiMuonInvMasses(): 5, 20, true). The arguments provided to the script were crafted in a way to ensure repeated computations for reducing the impact of noise on measured runtimes.

The runtime (excluding the profiling result post-processing stage) of the non-GenVectorX version under Adaptyst was 813.388 s (vs 807.208 s with no profilers and no power/energy measurements) and the same runtime of the GenVectorX version was 197.426 s (vs 192.113 s with no profilers and no power/energy measurements). In percentage terms, the overall runtime of GenVectorX is ~75.7% faster under Adaptyst and ~76.5% faster with no profilers. The machine used was a bare-metal server with an Intel Xeon Silver 4216 CPU @ 2.10 GHz and an NVIDIA Tesla T4 GPU.

The produced main thread flame graphs for both DiMuonInvMasses.cxx variants are shown in Figures 8 and 9 (parts of the flame graphs are compressed automatically and shown in light purple to save rendering resources and reduce graph sizes). The primary function is DiMuonInvMasses(). The highlighted regions show the computation (i.e. vector addition) parts with their sampled runtimes indicated in Figure 10. It can be seen that the program is compute-bound, but it also features I/O such as ReadVectors(). The computation itself was speeded up by ~92% thanks to SYCL. These comparisons are approximate due to the sampling-based nature of profiling.

```
entities:
  abc:
    options:
      handle_mode: local
      processing_threads: 16
    nodes:
      cpu:
        modules:
          - name: linuxperf
            options:
               freq: 20
```

Figure 7: Adaptyst system file used for profiling. It indicates that there is a single entity (computer), where the CPU-related activity should be analysed through the linuxperf module (with on-CPU sampling frequency changed from the default value to 20 Hz). Mor

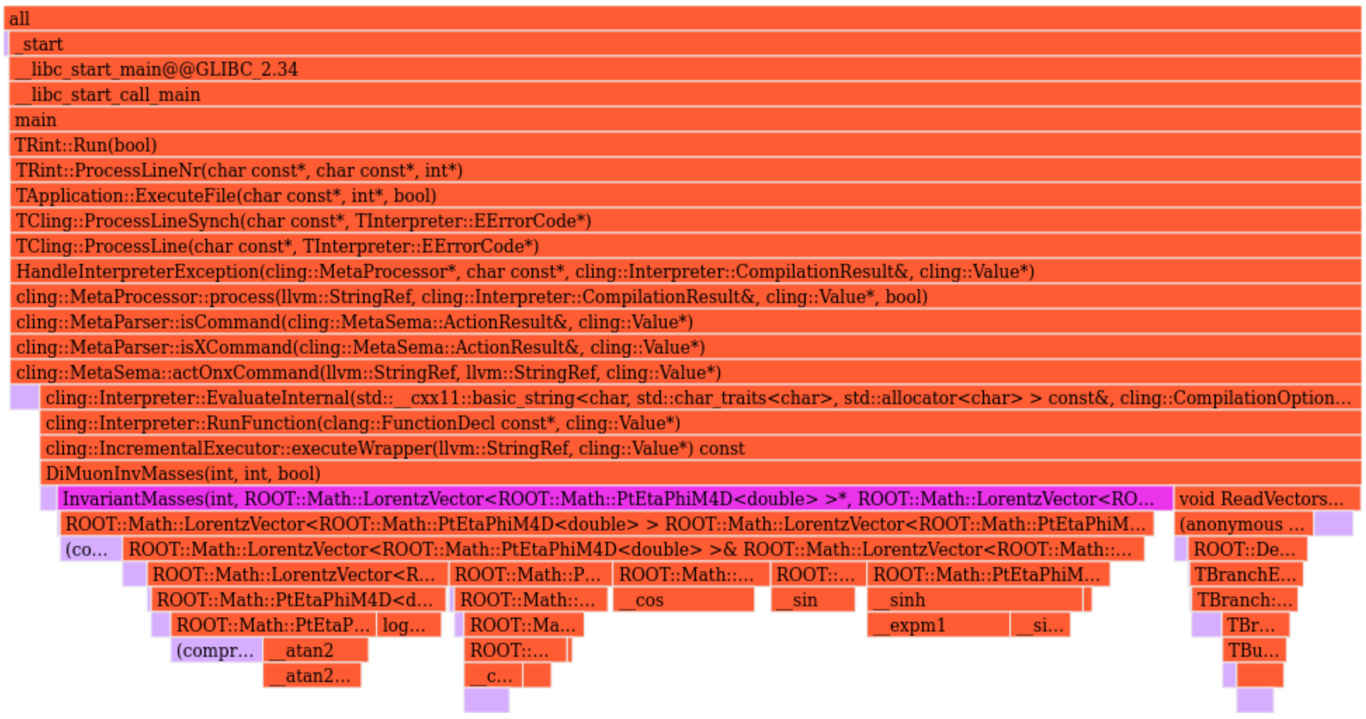


Figure 8: Non-GenVectorX main thread flame graph with the computation part highlighted in purple. Some flame graph elements are compressed and shown in light purple. The more blue a block is, the more off-CPU it is. The more red a block is, the more on-CPU it is

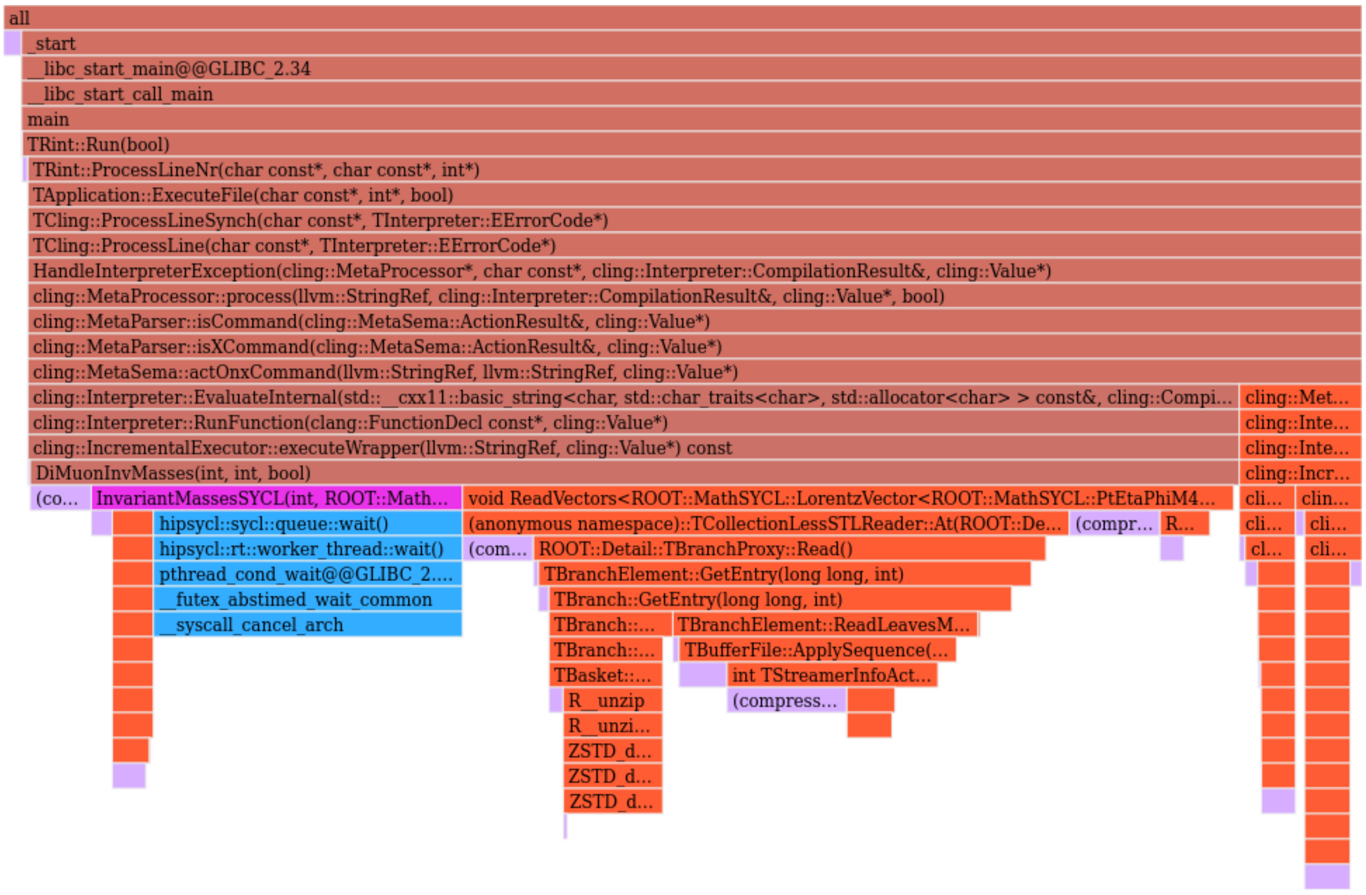


Figure 9: GenVectorX main thread flame graph with the computation part highlighted in purple. Some flame graph elements are compressed and shown in light purple. The more blue a block is, the more off-CPU it is. The more red a block is, the more on-CPU it is.

Non-GenVectorX	GenVectorX
668.350 s (84.49% of DiMuonInvMasses())	53.780 s (30.71% of DiMuonInvMasses())

Figure 10: Approximate computation times of the non-GenVectorX and GenVectorX versions.

Alongside profiling with Adaptyst, energy consumption measurements were made when running the non-GenVectorX and GenVectorX versions without Adaptyst on the same bare-metal machine. This was done by attaching "perf stat" with the power/energy-pkg/ counter to the script for the non-GenVectorX and GenVectorX CPU consumption and instructing nvidia-smi to sample the GPU power consumption every 500 ms in the background slightly before, during, and slightly after the GenVectorX script execution.

For the non-GenVectorX version, the CPU and thus total energy consumption was 61510.18 J. For the GenVectorX version, the total energy consumption was 16049.305 J, where the CPU consumption was 14457.38 J and the GPU consumption was approximately 1591.925 J, calculated from the graph shown in Figure 11. The GenVectorX energy consumption was ~73.9% lower than the non-GenVectorX one.

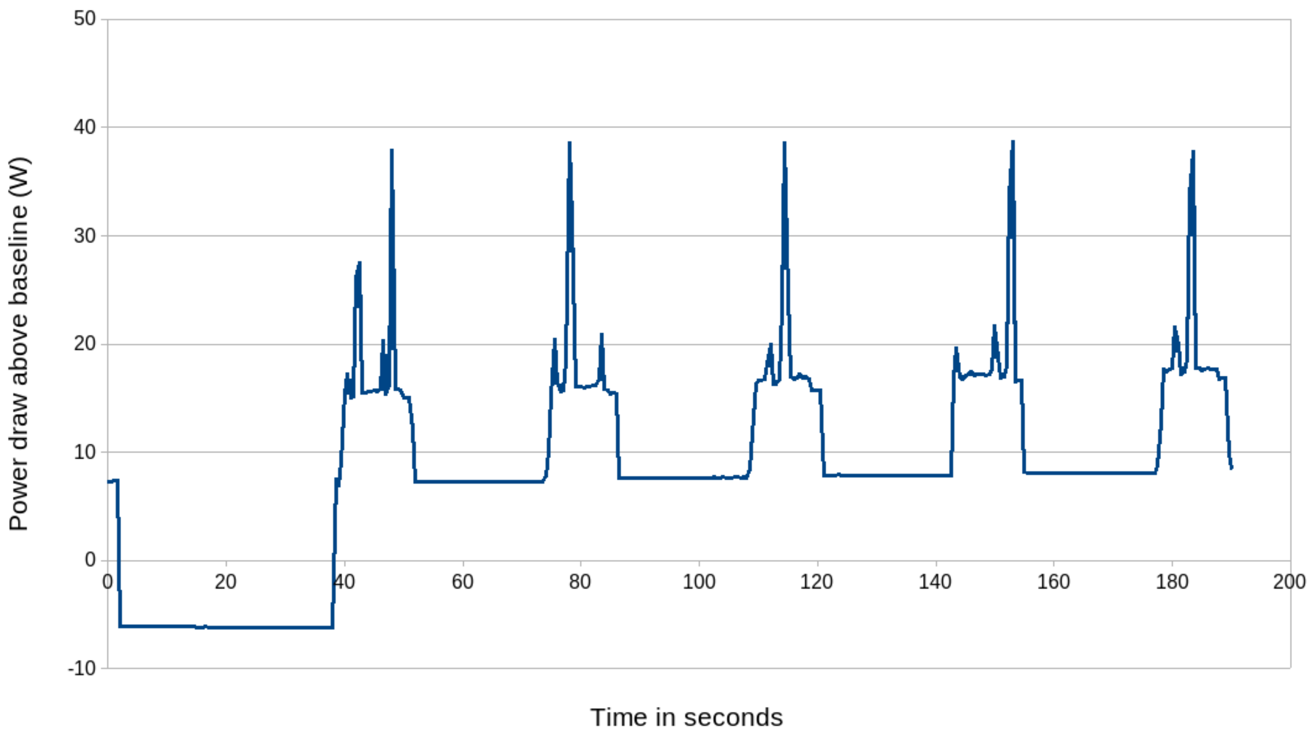


Figure 11: GPU power consumption above the baseline during the execution of the GenVectorX version of the script. The baseline is the average consumption from 5 samples before and 5 samples after the program execution.

Adaptyst has also been tested on the RISC-V-based machine with RVV 1.0 (DeepComputing DC-ROMA Laptop II with the SpacemiT X60 CPU): the "join" SYCLDB benchmark with AdaptiveCpp and the OpenCL backend was profiled. The program used oneAPI Construction Kit (OCK) as the OpenCL implementation. Two variants were analysed: one with OCK compiled without RVV support and one with OCK compiled with RVV support. The code was structured to run repeated computations, minimising the impact of noise on measured runtimes.

With Adaptyst attached, the runtime of the non-RVV version was 1510.576 s while the runtime of the RVV version was 1009.444 s, ~33.2% lower. The main thread flame graphs generated by Adaptyst are shown in Figures 12 and 13 (light purple elements are compressed). The primary function is main(). The highlighted regions show the “wait for SYCL computation to be finished” part with their sampled runtimes indicated in Figure 14. Their time length was reduced by ~38.2% thanks to RVV. As before, the approximation is due to the sampling-based nature of profiling.

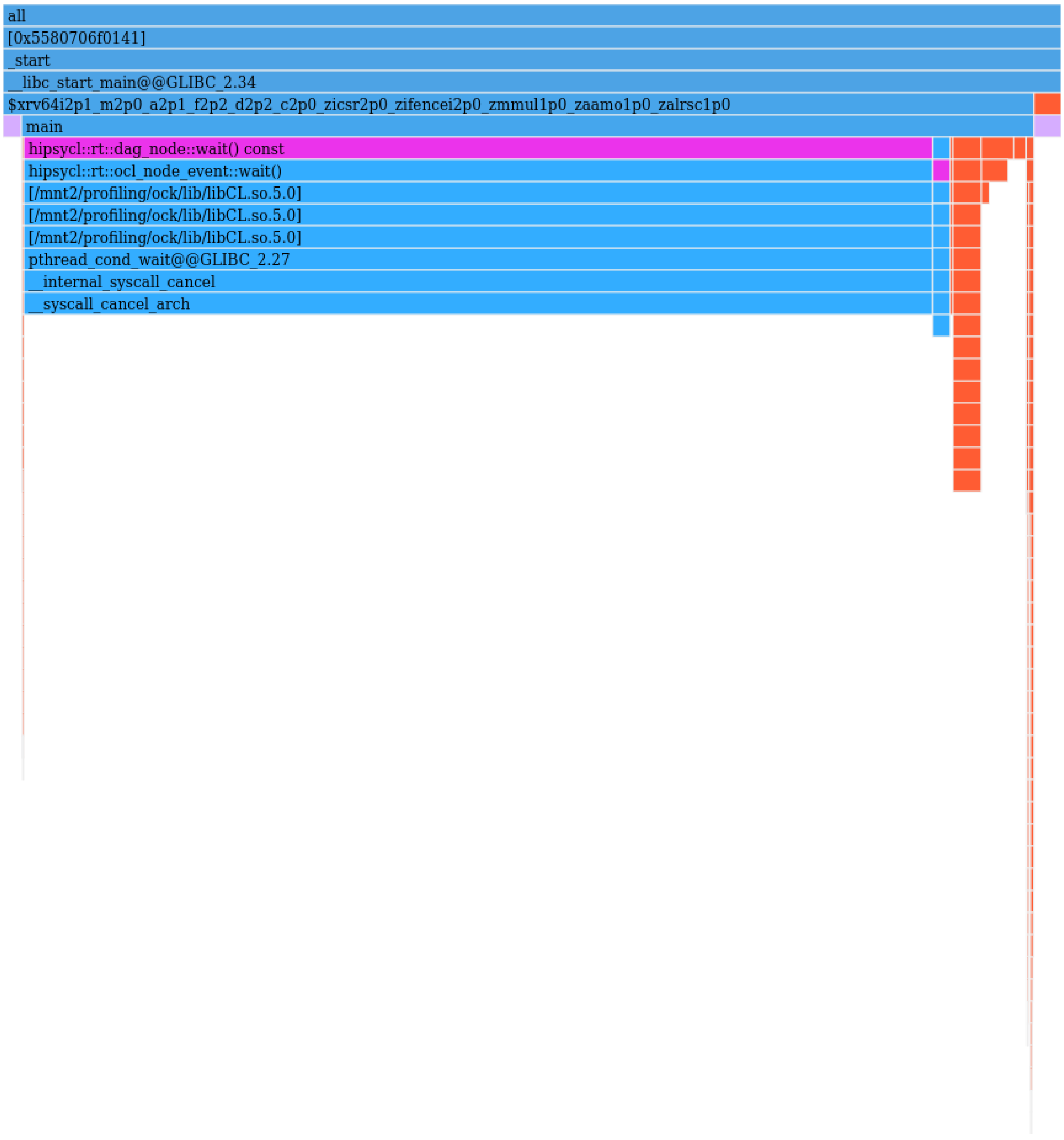


Figure 12: Non-RVV SYCLDB main thread flame graph with the “wait for SYCL computation to be finished” part highlighted in purple. Some flame graph elements are compressed and shown in light purple. The more blue a block is, the more off-CPU it is. The more red a

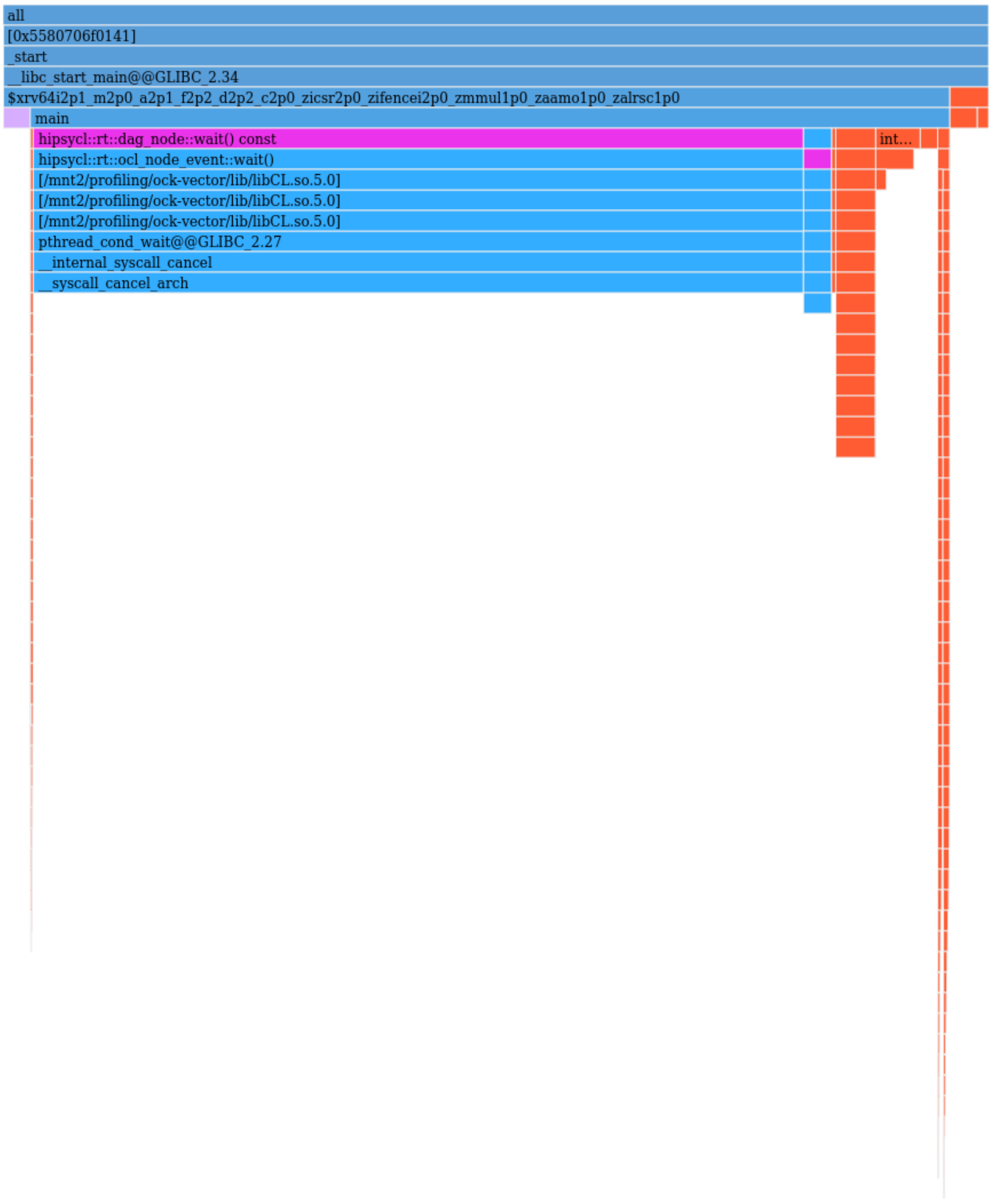


Figure 13: RVV SYCLDB main thread flame graph with the "wait for SYCL computation to be finished" part highlighted in purple. Some flame graph elements are compressed and shown in light purple. The more blue a block is, the more off-CPU it is. The more red a bloc

Non-RVV	RVV
1321.684 s (91.48% of main())	817.316 s (86.81% of main())

Figure 14: Approximate "wait for SYCL computation to be finished" times of the non-RVV and RVV SYCLDB versions.

The versions of the software used are as follows:

- Adaptyst on x86-64: dev branch, commit 51d6ff131e040ed56954752f3a2dd9c2b0c75058
- Adaptyst on RISC-V: dev branch, commit 2a3d413c3639db6e52853ab9b92f743336fae92e
- ROOT: v6.38.00-rc1
- AdaptiveCpp on x86-64: integrated with ROOT v6.38.00-rc1
- AdaptiveCpp on RISC-V: v25.10.0
- oneAPI Construction Kit: v5.0.0
- perf for energy consumption analysis: 6.17
- nvidia-smi: 580.95.05
- CUDA: 13.0

2.5 State-of-the-art After SYCLOPS

The test case taken into account carries out a DiMuon analysis, where the original code is available at https://root.cern/doc/v636/df102__NanoAODDimuonAnalysis_8C.html. In this simple, but representative example, the user calculates the invariant mass of all events with exactly 2 muons with opposite charge. As the filtering of particles has not been tackled in this project, it is not taken into account in final measurements. All events are packed in bulks to be transferred to the device for processing: 8 doubles (two 4-dimensional particles) are transferred, then invariant mass is computed to fill a bin in one histogram. Last, the result is moved back to the host. Figure below shows the speedup of the multithreaded CPU execution together with a SYCL offloading of the Define+Histogramming action versus the multithreaded CPU only execution. On the x-axis, the 2-logarithm of the bulk size is shown, which corresponds to the kernel size. Here, AdaptiveCpp has been used as SYCL implementation and device pointers have been used for data transfers. The computational environment is equipped as follows: AMD Ryzen 7 5700G 16 cores CPU, NVIDIA GeForce GTX3060 GPU. We observe that 8 and 16-threaded execution can benefit of a speedup up to ~3.5x, thus achieving KPI 11.

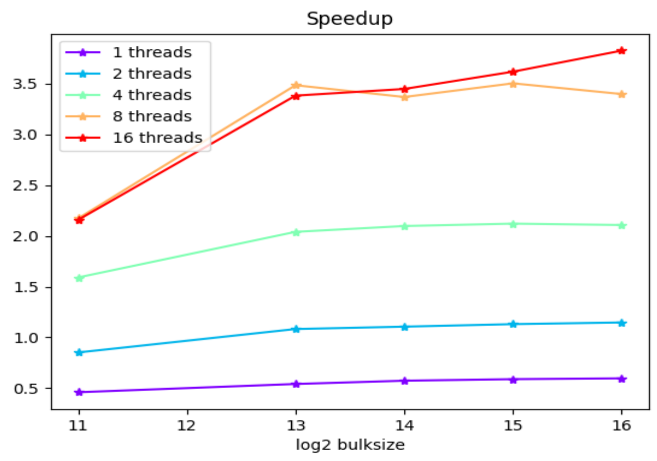


Figure 15: Speedup of the multithreaded CPU execution together with a SYCL offloading of the Define+Histogramming action versus the multithreaded CPU only execution. Different thread numbers are shown in different colors.

3 Genomics Use Case

This section details the integration carried out by our partner ACCELOM, which is an SME specializing in AI-based genomic data analysis. The primary objective of this use case was to accelerate the "Gold Standard" GATK germline variant calling pipeline using open hardware acceleration. Through the development and integration of the SYCL-GAL library, SYCLOPS has successfully demonstrated a ~4.6x reduction in total execution time compared to the standard non-accelerated GATK pipeline for the genomics use case. Specifically, the integration achieved an 11x improvement in preprocessing time and a 1.5x improvement in variant calling time, exceeding key project KPIs regarding latency and throughput. In the rest of this section, we will provide an overview of state-of-the-art pipelines used by ACCELOM before SYCLOPS.

3.1 State-of-the-art Before SYCLOPS

ACCELOM is an SME specializing in AI-based genomic data analysis, offering bespoke software and statistical expertise for identifying complex interactions in large-scale genomic datasets. In SYCLOPS, ACCELOM is focusing specifically on germline variant calling which is critical for many clinical use cases, such as Rare Disease Diagnosis (identifying *de novo* or recessive variants in family trios) and Childhood Cancer screening (analyzing inheritance patterns of cancer-associated variants). The figure below shows an example pipeline, similar to what has been used by ACCELOM before for performing germline variant calling, in a hypothetical paediatric cancer diagnostic setting where 2 parents and a child get their DNA sequenced with the goal of getting information about cancer-associated germline variants, their inheritance patterns, and the role of multigenic interactions in tumorigenesis. Three samples, two from the parents and one from the child, are sequenced to produce reads. These sequencing reads are checked for quality by aligning them to a reference and verifying with QC tools. If quality check fails, resequencing is performed. Once all three sequences pass quality check, the core computational pipeline is kicked off, which is the trio variant calling pipeline from GATK.

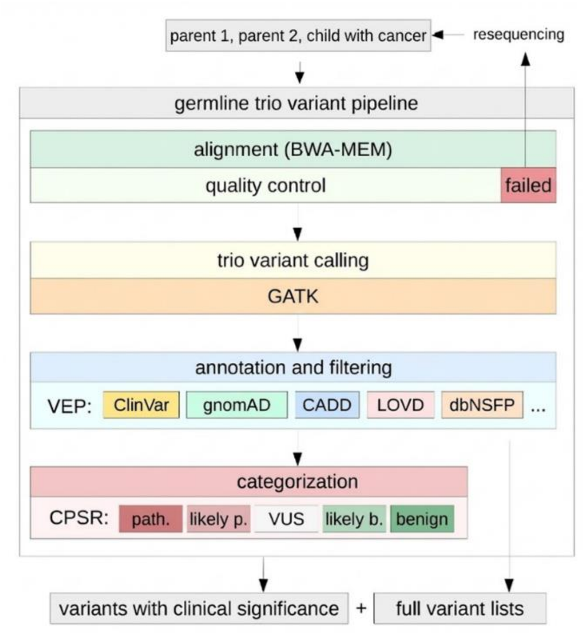


Figure 16: Germline variant calling pipeline example

GATK germline variant calling pipeline is the gold-standard for trio analysis. It takes as input all three sequenced reads in FASTQ format and produces a set of variants in a VCF file. Following this, the variants identified by GATK are annotated and filtered with tools like VEP. VEP predicts the consequence of variants at the molecular level, reports known phenotype associations, and offers predictions of deleteriousness using a variety of sources like ClinVar, gnomAD, etcetera. VEP also includes filtering options to prioritize variants and rank them. Following VEP, the annotated variants are then passed to the tool CPSR which performs automated variant interpretation in known cancer predisposition genes and produces a report. This report is then used by clinicians to make treatment decisions.

As shown above, the GATK germline variant calling pipeline is a key part of this analysis. The figure below shows the two main stages of this pipeline, which are preprocessing and variant discovery. The preprocessing stage consists of 3 main substages, which are Sorting, Marking Duplicates, and Recalibrating Base Quality Scores (BQSR). In the variant discovery stage, the most important component is the HaplotypeCaller tool.

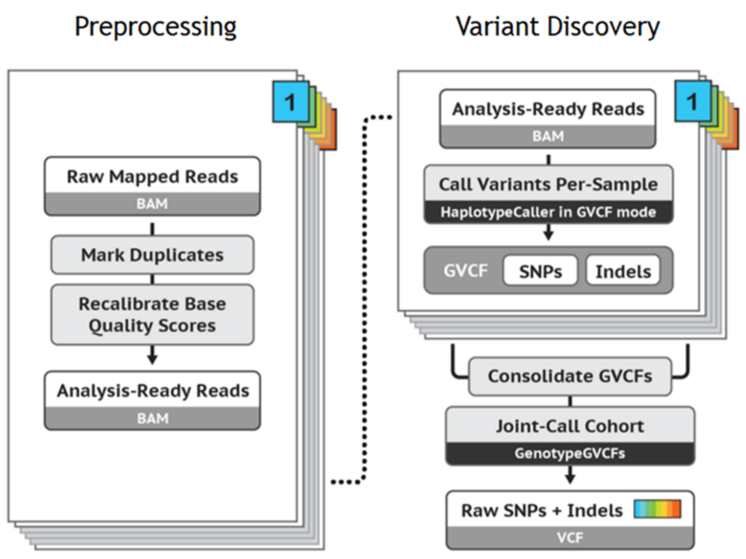


Figure 17: GATK pipeline stages

The central problem, and the core of the work in SYCLOPS, as relevant to ACCELOM's use case, is the fact that it is well known that this GATK germline variant calling pipeline is extremely computationally intensive. A recent report by Intel showed that running this pipeline on a 36-core Intel CPU to process just one whole-genome sequencing sample (which was around 130GB in size) took nearly 4 days, with the preprocessing stage take 3 days and HaplotypeCaller taking 20 hours. The analysis we did in SYCLOPS at the beginning of the project showed similar results.

Goal: The goal of SYCLOPS was to accelerate the pre-processing and variant calling stages to overcome this computational bottleneck.

Outcome: To achieve this, ACCELOM collaborated with EURECOM on preprocessing, and INESC on variant calling. The outcome of this work was SYCL-GAL, a library of primitives for accelerating various substages of the GATK germline variant calling pipeline.

KPIs: The KPIs addressed in this use case are the following:

- KPI 9: A new library of parallel algorithms (SYCL-GAL) will enable genomic data analysis acceleration.

- KPI 10: Three applications domains will successfully demonstrate that their end-to-end pipelines that integrate SYCL libraries can be portably deployed, with no code change in application logic, on several accelerators.
- KPI 11: Detailed evaluation will demonstrate that (i) Use cases can portably run their SYCL-based accelerated pipelines on multiple processors and achieve at least 2x improvement in latency and/or throughput and/or energy efficiency (depending on the accelerator used) compared to non-accelerated versions, (ii) SYCL-based libraries can achieve performance comparable to their CUDA counterparts when appropriate.

In the rest of this section, we provide a brief overview of what was achieved individually with respect to accelerating preprocessing (Section 3.2) and variant calling (Section 3.3). Then, we describe how we integrated SYCL-GAL in the context of a modified GATK pipeline used by ACCELOM (Section 3.4), and did an end-to-end performance and accuracy testing (Section 3.5).

3.2 Pre-processing Acceleration

In this section, we provide an overview of the work done in accelerating the preprocessing stage of the GATK germline variant calling pipeline.

As mentioned earlier, this stage contains three sub operations, namely, Sorting, Marking Duplicates, and BQSR. Our first insight was the fact that Sorting and Mark Duplicate stages share several algorithmic similarities with database queries. For instance, relational databases perform fast sorting of structured data, and SQL constructs like GROUP BY and UNIQUE are used to perform aggregation and eliminate duplicates. Over the past few years, GPU acceleration has become extremely interesting both academically and industrially for relational databases. Several commercial GPU databases exist today that provide orders of magnitude better performance for analytical SQL queries compared to their CPU counterparts. Motivated by this, we asked the question as to whether portable acceleration of SQL database workloads can be achieved using SYCL and RISC-V?

To answer this question, we developed SYCLDB, a library of relational primitives (like sort and join) that can be used to build database engines on open hardware. SYCLDB was proven to work across Intel, AMD, and NVIDIA GPUs, as well as RISC-V CPUs, matching the performance of the CUDA-based counterparts while remaining vendor-neutral. SYCLDB was also used throughout SYCLOPS as a vehicle to demonstrate integration at various levels. For instance, SYCLDB relied on functionality (like kernel fusion) provided by SYCL compilers DPCPP and ACPP developed in SYCLOPS. Similarly, SYCLDB was run on RISC-V CPUs and GPUs deployed in SYCLOPS EMDC and showed that RISC-V vector extensions can be successfully exploited by SYCL compilers. A peer-reviewed publication about SYCLDB appeared in HeteroPar 2024.

Having built SYCLDB, we directly used the insights from optimizing SYCLDB, to accelerate sorting and mark duplicate stages in SYCL-GAL. In particular, unlike GATK, which reads and writes data to disk at every stage (creating massive I/O overhead), SYCL-GAL maintains data in-memory using a columnar layout similar to SYCLDB. This structure improves GPU cache utilization and allows for efficient vectorization. The sorting stage of the GATK pipeline was optimized using a two-stage sorting algorithm based on a columnar layout. This allows for rapid ordering of reads by genomic coordinates, a prerequisite for duplicate marking. Standard algorithms for Mark Duplicates step often use hash maps, which are inefficient on GPUs due to collisions. We observed this during the design of SQL JOIN operations in SYDLDB. Thus, SYCL-GAL replaces this with a Merge Sort + Duplicate Marking approach (conceptually similar to SORT UNIQUE in databases). This scans sorted reads to identify duplicates without the overhead of hash table management.

With sorting and mark duplicates optimized, the final step left was BQSR. The BQSR stage builds a statistical error model to correct systematic errors from the sequencing machine. It does in two phases. In the first phase, BQSR computes auxiliary arrays (covariates like nucleotide context and cycle position). In the second phase, BQSR perform model construction, where it updates a global recalibration table. We developed a fully GPU-based version of BQSR in SYCL-GAL which performs both these phases entirely in GPU memory. More information about our implementation is available in deliverable D5.4. But with this, we have fully implemented an accelerated version of the preprocessing stage of the GATK germline variant calling pipeline in SYCL. Deliverable D5.4 also contains an isolated evaluation of this accelerated preprocessing stage and showed that SYCL-GAL preprocessing pipeline can achieve a 10x improvement over the GATK baseline.

3.3 Variant calling acceleration

As mentioned earlier, the core component of the variant calling stage of the GATK pipeline. Preliminary analysis in the project of HaplotypeCaller revealed that the Pair Hidden Markov Model (pairHMM) algorithm is the main component of variant calling, responsible for calculating the probability that a specific DNA read was generated by a candidate haplotype. In the standard GATK pipeline, this single operation for over 70% of the total execution time, making it the primary target for acceleration. However, accelerating pairHMM is notoriously difficult because it relies on dynamic programming with data dependencies that typically force sequential processing, and its computational complexity scales quadratically with sequence length.

To overcome these limitations, INESC developed a novel parallelization strategy named "Endeavor." Unlike previous GPU implementations that rely on processing anti-diagonals (which leads to irregular workloads), Endeavor mathematically redefines the pairHMM steps to expose row-level parallelism. This allows the algorithm to map efficiently to the GPU's hardware structure. Specifically, the implementation processes read-haplotype pairs at the warp level rather than the thread level. Threads within a warp calculate elements of a matrix row in parallel, storing necessary intermediate values in shared memory to synchronize across the warp.

This architectural shift allows endeavor to handle a vast range of sequence lengths dynamically:

- Short Sequences (<1024 bases): Processed by a single warp using thread registers.
- Medium Sequences: Processed by multiple warps leveraging shared memory.
- Long Sequences (>131,072 bases): Processed by multiple thread blocks

This flexibility effectively removes the memory bandwidth bottleneck that plagues existing solutions. Roofline model analysis confirms that while the state-of-the-art gpuPairHMM is memory-bound, the Endeavor implementation is compute-bound, meaning it fully utilizes the raw processing power of the accelerator.

More information about Endeavor is available in deliverable D5.4 which also contains an isolated evaluation of pairHMM that showed that Endeavor can achieve two orders of magnitude speedup over a CPU-based AVX512 implementation.

3.4 Integration and State-of-the-art After SYCLOPS

To recap, SYCL-GAL library contains individual acceleration of various steps in the preprocessing stage, and an accelerated pairHMM implementation for the variant calling stage.

3.4.1 Pipeline Integration

For ACCELOM's specific use case, the SYCL-GAL components were integrated into a cohesive, accelerated pipeline as follows

1. Preprocessing Integration: The standard GATK sorting and BQSR tools were directly replaced with the SYCL-GAL library which supports accelerated sorting, mark duplicates, and BQSR steps.
2. C++ HaplotypeCaller Development: The standard HaplotypeCaller is written in Java. This made direct integration of the GPU code written in SYCL extremely tedious. Preliminary effort at integration with GATK HaplotypeCaller showed that the overhead of integration effectively negated the performance optimizations achieved by SYCL pairHMM. So, we developed a new C++ HaplotypeCaller as a prototype. It should be noted that reimplementing the HaplotypeCaller is a major endeavor that was not an original part of the SYCLOPS project planning. Yet, we undertook this initiative as a good-will effort, and as an activity that we plan to continue well beyond the end of the SYCLOPS project, primarily to demonstrate the potential of pairHMM in an integrated pipeline. We have integrated the SYCL pairHMM with the prototype C++ HaplotypeCaller, thus making an accelerated variant calling stage.
3. Deployment: The accelerated pipeline was extended by adding BWA-MEM2 sequence aligner and deployed on the SYCLOPS EMDC, demonstrating successful portable deployment, thus achieving KPI 9 and KPI 10.

3.4.2 Pipeline Evaluation

The integrated SYCL-GAL pipeline was rigorously evaluated to validate its performance and accuracy against the industry-standard GATK baseline. The benchmarking utilized the publicly available CEPH Utah Trio dataset, which contains sequencing data obtained from a Utah-resident family of Northern and Western European ancestry, as a part of the Genome In a Bottle datasets. The input consists of approximately 50GB of raw sequencing data. We focused on one sample (NA12878) to study performance and accuracy of three different pipelines:

1. State-of-the-art before SYCLOPS: GATK germline variant calling (GATK post processing + HaplotypeCaller variant calling)
2. Developed in SYCLOPS: SYCL-GAL-accelerated preprocessing + variant calling pipeline
3. Commercial alternative and CUDA solution: NVIDIA Parabricks preprocessing + Parabricks HaplotypeCaller

The figures below shows the comparison of SYCL-GAL and GATK on our SYCLOPS EMDC server that is equipped with a Intel Xeon CPU and an NVIDIA L40S GPU. GATK is single threaded and runs on 1 CPU while accelerated SYCL-GAL uses the GPU. Clearly, we can see a dramatic reduction in processing latency as the total execution time for the pipeline was reduced from 113.62 minutes using the standard GATK workflow to just 24.58 minutes with the SYCL-GAL integration, representing a total speedup of approximately 4.6x. These results show that we exceeded the planned KPI11 target of 2x improvement. This improvement was most pronounced in the preprocessing stages, where the time required dropped from 89.00 minutes to 8.26 minutes—an 11x improvement that effectively eliminated the preprocessing bottleneck. Variant calling also saw significant gains, with execution time falling from 27.32 minutes to 18.00 minutes, achieving a 1.5x speedup despite the complexity of the HaplotypeCaller integration.

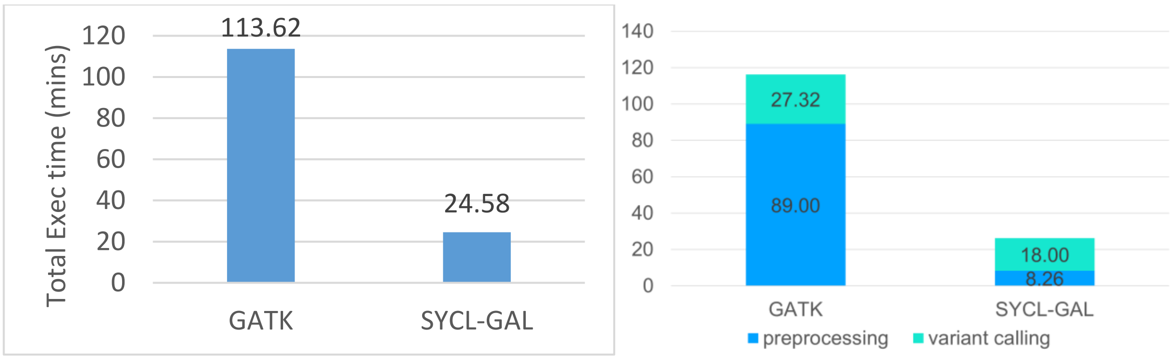


Figure 18: Performance of GATK vs SYCL-GAL

To gauge the maturity of the solution, the SYCL-GAL pipeline was also compared against NVIDIA Parabricks, a highly optimized, proprietary, and closed-source solution. Figure below shows the results from this comparison.

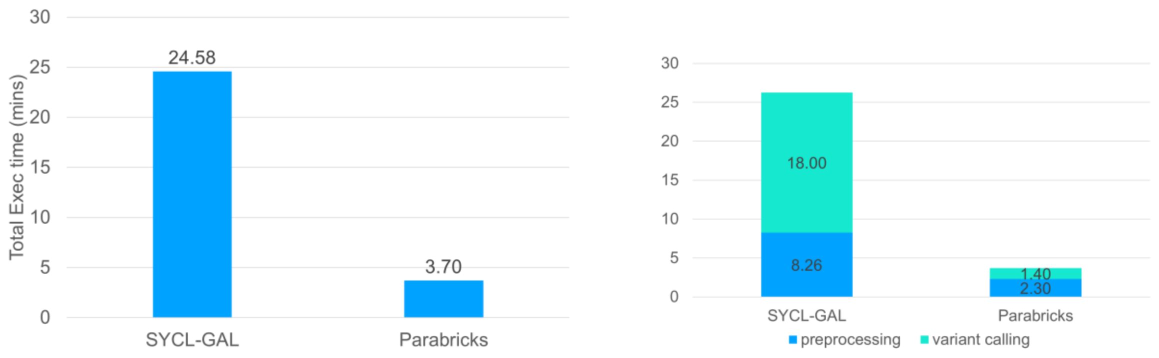


Figure 19: Performance of SYCL-GAL vs Parabricks

While Parabricks achieved a faster total execution time of 3.70 minutes, the evaluation revealed that this gap is largely driven by Input/Output (I/O) architecture rather than raw computational deficiency. The current SYCL-GAL implementation reads and writes large BAM files to disk between stages, incurring significant I/O penalties, whereas Parabricks pipes data directly between stages in memory. When isolating the computational kernels (Mark Duplicates and BQSR), SYCL-GAL was found to be highly competitive, running only 1.7x slower than the mature Parabricks solution. This indicates that the open-standard SYCL algorithms are approaching the efficiency of vendor-locked alternatives.

Finally, we also evaluated the accuracy of the accelerated pipeline by comparing the output Variant Call Format (VCF) files generated by the SYCL-GAL pipeline with those from the GATK baseline. We found that 98.2% of variant calls matched. We further analyzed the discrepancies and found that accelerated components in SYCL-GAL had no issues. Rather, the discrepancies that were observed were attributed to edge cases within the newly developed C++ HaplotypeCaller prototype. Thus, further work is required on the new HaplotypeCaller to iron out these differences.

Overall, the evaluation confirms that ACCELOM successfully integrated a portable, open-standard acceleration library that delivers massive speedups over CPU baselines and offers a viable, vendor-neutral alternative to proprietary genomic analysis stacks. In doing so, we have achieved KPI 9 by developing SYCL-GAL, KPI 10 by integrating it in a use case pipeline, and exceeded KPI11 by demonstrating a 4.6x speedup.

4 Autonomous Systems Use Case

4.1 State-of-the-art Before SYCLOPS

The autonomous systems use case within the SYCLOPS project centers on PointNet, a deep learning architecture designed to process 3D point cloud data efficiently. Figure below shows an example of point clouds, which are unordered sets of points.

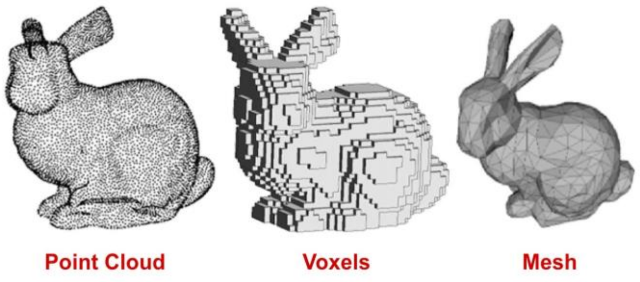


Figure 20: Point cloud example

PointNet[1] is a foundational neutral network architecture that demonstrated feature learning on point clouds. Unlike traditional methods that convert data into structured formats like voxel grids or images, PointNet processes point clouds directly. This capability is critical because point clouds are unordered sets of data where the permutation of points does not alter the spatial information they represent. To handle this "permutation invariance," PointNet utilizes a symmetric function, specifically max pooling, to aggregate features after applying transformations to individual points. This architecture has become foundational for 3D perception tasks such as object detection and semantic segmentation in autonomous driving scenarios.

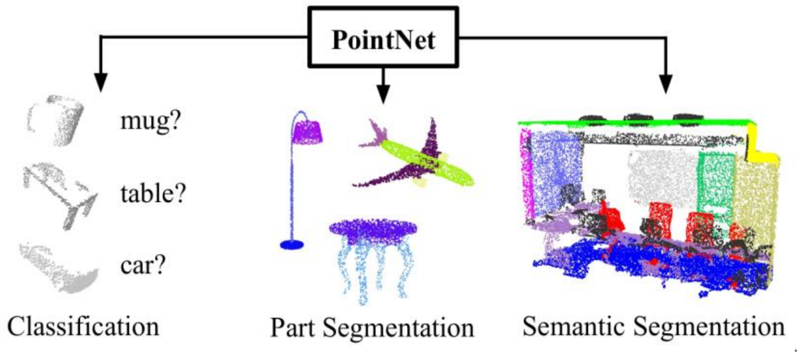


Figure 21: Pointnet example

Prior to the SYCLOPS project, the landscape of Deep Neural Network (DNN) acceleration libraries presented a stark trade-off between portability and performance. portDNN[2] (formerly known as SYCL-DNN) existed as an open-source library that offered portability across different hardware architectures—including CPUs, GPUs, and FPGAs—through the use of generic SYCL kernels. However, its utility was limited because it only supported a small set of common operators and lacked key primitives required for PointNet, such as 1D convolutions, batch normalization, and concatenation. Conversely, oneDNN[3] (oneAPI Deep Neural Network Library) provided a unified interface for DNN operations with the goal of enabling developers

[1] https://arxiv.org/abs/1612.00593v2
[2] https://github.com/codeplaysoftware/portDNN
[3] https://github.com/uxlfoundation/oneDNN

to write code once and deploy it anywhere. While robust, oneDNN achieved high performance primarily through vendor-specific backends, meaning it lacked a truly portable generic path that could easily support new or non-traditional hardware architectures without specific vendor libraries.

Goal: The goal of SYCLOPS was to bridge the gap between the two libraries and enable SYCL-based, accelerated point cloud analysis.

Outcome: To achieve this, CPLAY worked on extending and combining the functionalities of the two disparate SYCL DNN libraries to produce a single, state-of-the-art PointNet implementation in SYCL.

KPIs: The KPIs addressed in this use case are the following:

- KPI 7: Significantly enhanced SYCL-DNN will enable scalable object detection for autonomous systems.
- KPI 10: Three applications domains will successfully demonstrate that their end-to-end pipelines that integrate SYCL libraries can be portably deployed, with no code change in application logic, on several accelerators.
- KPI 11: Detailed evaluation will demonstrate that (i) Use cases can portably run their SYCL-based accelerated pipelines on multiple processors and achieve at least 2x improvement in latency and/or throughput and/or energy efficiency (depending on the accelerator used) compared to non-accelerated versions, (ii) SYCL-based libraries can achieve performance comparable to their CUDA counterparts when appropriate.

4.2 SYCLomatic and SYCLcompat

To bridge the gap between existing high-performance CUDA implementations and the open SYCL standard, the project employed a robust toolchain consisting of SYCLomatic and SYCLcompat. SYCLomatic is an open-source command-line tool designed to automate the migration of CUDA code to SYCL, typically achieving an automatic translation rate of approximately 95%. It handles the conversion of kernels, data types, and API calls, while identifying complex sections that require manual intervention. Complementing this, SYCLcompat serves as a compatibility library that provides SYCL-implemented functions to mimic specific CUDA behaviours that lack a direct one-to-one mapping in the SYCL standard.

This toolchain played a pivotal role in enabling the PointNet architecture within the portable library ecosystem. The project utilized these tools to assist in the development of missing operations in portDNN that were essential for PointNet. Specifically, the tools facilitated the porting and implementation of complex primitives such as concatenation, broadcasted binary operations, and 1D convolutions. By automating the translation of standard constructs and providing compatibility layers for vendor-specific APIs, the project successfully extended portDNN's operator support to fully encompass the requirements of the PointNet model. More details about SYCLomatic and SYCLcompat are available in deliverable D5.1.

4.3 Merging portDNN and oneDNN

A defining achievement of the SYCLOPS project was the unification of the portability found in portDNN with the extensive ecosystem of oneDNN. The strategy involved introducing a generic SYCL GPU backend into oneDNN, which was achieved by migrating the generic SYCL kernels originally developed for portDNN directly into the oneDNN repository. This integration effectively endowed oneDNN with a portable execution path that complements its existing vendor-optimized backends.

Consequently, the merged infrastructure allows the PointNet model to be executed through oneDNN on diverse hardware accelerators without requiring any changes to the source code. The implementation leverages oneDNN primitives to encapsulate operations like convolutions and pooling, while engines and streams abstract the computational device and execution context. This unification ensures that the optimizations and portability enhancements developed within portDNN are now available within the widely adopted oneDNN ecosystem, resolving the prior trade-off between performance and hardware flexibility.

4.4 State-of-the-art After SYCLOPS

Following the integration, the SYCLOPS project demonstrated that the new SYCL-based implementations could achieve performance parity with native backends. Detailed benchmarking results that demonstrate the performance and portability of our solution are available in deliverable D5.4.

The project extended beyond standard benchmarking to a practical research use case involving Unmanned Aerial Vehicles (UAVs). Collaborating with the RAPID project, SYCLOPS researchers applied the PointNet architecture to process aerial point clouds generated by 76-81GHz W-band radar for Beyond Visual Line of Sight (BVLOS) operations. To address the stringent Size, Weight, and Power (SWaP) constraints of UAVs, the team used portDNN to refine core operations, most notably optimizing the model by transforming 1D convolutions into Matrix Multiplications to reduce computation time. This optimized framework successfully distinguished between multiple classes of aerial and ground objects, enhancing the situational awareness of autonomous drones. Further details about this collaboration are available in a peer-reviewed publication[4].

To demonstrate integration with RISC-V hardware developed in SYCLOPS, we performed rigorous testing of hand-crafted 1D convolution and matrix multiplication kernels written in SYCL, two kernels that are heavily used in portDNN, on both the CSIP RVV soft core and the SYCLARA platform. For this, we used the oneAPI Construction Kit developed in WP4 and described in deliverable D4.2 to offload kernel execution to the RVV accelerators.

Platform	**Operation**	**Size**	**Data Type**	**Scalar [ms]**	**VF=16 [ms]**	**Speedup**
Codasip	Convolution	8K	Int32	5643	1665	3.389189
Codasip	Convolution	8K	Float	3296	1230	2.679675
Codasip	Convolution	8K	Double	5594	1651	3.38825
ARA	Convolution	8K	Int32	10404	2511	4.143369
ARA	Convolution	8K	Float	7178	2100	3.418095
ARA	Convolution	8K	Double	47499	2524	18.81894

Figure 22: RVV vs scalar performance

The table above shows the results for scalar convolution (before SYCLOPS) and RVV convolution with CSIP or ARA (after SYCLOPS) for Int32, Float, and Double data types. As

[4] https://arxiv.org/abs/2311.03221

can be seen, we observe substantial performance gains when utilizing RISC-V vector extensions on either hardware platform, with CSIP achieving up to 3.38x improvement and ARA an 18.8x improvement. It can also be seen that CSIP RVV solution provides much better performance than ARA in all cases.

The table below shows the results for matrix multiplication. Similar results can be seen here as well, with RISC-V vector extensions providing up to 3.07x improvement with CSIP platform and 5.9x with ARA.

Platform	**Operation**	**Size**	**Data Type**	**Scalar [ms]**	**RVV [ms]**	**Speedup**
Codasip	Matrix multiplication	(150,300) X (300,600)	Int32	11491	4397	2.613373
Codasip	Matrix multiplication	(150,300) X (300,600)	Float	15696	5098	3.078854
Codasip	Matrix multiplication	(150,300) X (300,600)	Double	20565	8569	2.39993
ARA	Matrix multiplication	(150,300) X (300,600)	Int32	25947	4586	5.657872
ARA	Matrix multiplication	(150,300) X (300,600)	Float	26465.3	5307	4.986866
ARA	Matrix multiplication	(150,300) X (300,600)	Double	36571.3	6156.33	5.940439

Figure 23: RVV vs scalar performance of matmul

These results confirm that the SYCL compiler toolchains and libraries developed in SYCLOPS can effectively exploit RISC-V vector accelerators to enhance the performance of deep learning primitives essential for autonomous systems. In doing so, we have achieved KPI 7 by developing oneDNN library, KPI10 by using it to train a PointNet model with open weights, and KPI11 by running the SYCL-based PointNet implementation on a variety of hardware accelerators either end-to-end, or just key kernels (depending on the capability of the underlying hardware), and demonstrating concrete performance improvements.

5 Infrastructure & Platform Tools Integration

So far in this document, we presented the integration and validation work done on a per use case basis. In this section, we specifically focus on integration at the infrastructure layer of the SYCLOPS stack that involves the RVV accelerator from CSIP and the EMDC from HIRO.

In deliverable "D3.2: EMDC v2.0 with RVV accelerator release", we provided a detailed evaluation of the two RVV accelerator platforms developed in SYCLOPS: (i) the FPGA platform from CSIP, and (ii) SYCLARA platform developed by EUR. Similarly, In deliverable D3.2, HIRO described their effort in developing a CXL-enabled, PCIe 6.0 switch for their next-generation EMDCs, and the CXL research testbed that has been put in place to test software developed in SYCLOPS.

In this section, with respect to the CSIP RVV accelerator, we will focus on an analysis of energy efficiency gains that can achieved by using RVV over scalar execution in Section 5.1. Following this, we present the work done in integrating and using the CARM profiling tool to evaluate specialized RISC-V accelerators in Section 5.2. Finally, we provide an update with respect to the switch, and present results obtained by running SYCLDB on the CXL server in Section 5.3.

5.1 CSIP RVV Accelerator Integration

5.1.1 CSIP RVV Power estimation analysis

To estimate the energy requirements of the RVV accelerator prototype, we run reduced tasks in the RTL simulation to generate switching activity file for every analysed benchmark. The simulated RTL was synthesized by Cadence Genus tool, and this tool was also used to provide power estimation for every benchmark. Each benchmark was evaluated in both scalar and vector implementations. However, both variants run on the vector -enable core. No comparison with scalar core is done as part of this report.

The A730 family implements a private L1 cache, while the L2 cache is shared between cores in a multicore system. The focus of the SYCLOPS project was on the enhancing the RISC-V core with the RVV support. The multicore systems are much larger by definition which negatively affect the simulation and synthesis runtime. Therefore, we implemented only single core solution for the SYCLOPS project. Since L1 cache is private, it is considered part of the core for the purposes of this analysis. Level 2 cache is the resource shared with other potential cores and therefore its power consumption is not addressed by the analysis. During the simulation, we simulated the whole memory subsystem with the L2 cache to ensure that the behaviour of the core is the same as it would be in the real system. However, we computed the power estimation only for L1 cache and the core.

While the RVV Accelerator targets 1.2GHz operation frequency, the experiments were done on much more relaxed target of 600MHz due to the prototype constraints. TSMC 7nm FinFET technology was used as a synthesis target.

The following benchmarks were evaluated.

Vector Addition

Two predefined vectors of length 1024 and 4096 elements of 32-bit floating point datatype were added element wise.

Variant	Vector Length	Clock cycles	Average power [mW]	Consumed Energy [nJ]
SCALAR	1024	9126	68.1296	1036.25
VECTOR	1024	7168	68.1144	813.74
SCALAR	4096	26541	68.1366	3014.02
VECTOR	4096	18984	68.1150	2155.16

Figure 24: Power/energy consumption of RVV for vecadd

It can be seen from the table that vector implementation can bring significant energy savings of **21% (1024)** and **28% (4096)**, primarily due to the computation speed up of 21% and 28%, respectively. However, the average power is slightly reduced in the vector implementation, which can be attributed to simplified load/store operations.

Vector Multiplication

Two predefined vectors of the 32-bit floating point elements were multiplied elementwise.
The evaluated vector lengths were 1024 and 4096.

Variant	Vector Length	Clock cycles	Average power [mW]	Consumed Energy [nJ]
SCALAR	1024	9139	68.1307	1037.74
VECTOR	1024	7168	68.1154	813.75
SCALAR	4096	26530	68.1389	3012.88
VECTOR	4096	18984	68.1162	2155.2

Figure 25: Power/energy consumption of RVV for vecmul

The algorithm and implementation of the vector multiplication benchmark are very similar to the vector addition benchmark. Only difference lies in the arithmetic operation applied to the vector elements. Multiplication is slightly more complex than addition; however, as shown in the tables, the actual performance difference is negligible.

Dot product

The dot product of two predefined vectors was computed. The implementation uses a floating-point register as an accumulator, representing a family of algorithms that require data transfers between scalar and vector parts of the system.

Variant	Vector Length	Clock cycles	Average power [mW]	Consumed Energy [nJ]
SCALAR	1024	12745	68.1094	1446.76
VECTOR	1024	7820	68.1305	887.97
SCALAR	4096	39854	68.1089	4524.02
VECTOR	4096	19922	68.1415	2262.52

Figure 26: Power/energy consumption of RVV for dotproduct

The table shows that the vector implementation consumes slightly more power than the scalar implementation in average, which differs from other vector benchmarks. This is due to the dot product algorithm using a floating-point register as an accumulator. After each MACC instruction, data must be transferred from and to the floating-point register, introducing additional overhead.

Matrix multiplication

Two square matrices of 20 columns were multiplied together. Several implementations were evaluated. The ANSI_C implementation refers to the classic approach using three nested loops in C,.with only compiler optimizations applied. The BLASFEO implementations targets the GENERIC configuration of the Blasfeo library. Both these implementations do not use vector instructions. In addition, four other implementations were developed in vector assembly., differing only in the LMUL size.

Variant	Instruction count	Clock cycles	Average power [mW]	Consumed Energy [nJ]
[ANSI_C	64261	76799	6.81110	8718.09
BLASFEO	29403	38587	68.1116	4380.37
LMUL1	27846	50033	68.1614	5683.87
LMUL2	20646	44920	68.1594	5102.87
LMUL4	17046	47218	68.1470	5362.94
LMUL8	13446	51610	68.1422	5861.36

Figure 27: Power/energy consumption of RVV for matmul

The matrix multiplication benchmark demonstrates the complexity of optimization. When analysed by the instruction count, we can see the expected trend: the pure C implementation takes the longest time, followed by the optimized but scalar library implementation, while vectorized implementations are the fastest. The higher LMUL, the faster the code according to the IA model.

However, when the clock cycles are considered, the situation changes. Increasing LMUL for the vectorized implementation from LMUL1 to LMUL2 introduces a speed up, but it is much lower than predicted by the IA model. The IA model predicted a 35% speed up, but the RTL simulation observed only 11%. This discrepancy occurs because the number of the computation elements in the RVV accelerator is optimized for the vector width of 128bits. For larger LMUL, the core executes the same operation over multiple consequent cycles for different segments of the enlarged register. Therefore, the observed speed up is primarily due to the more efficient pipeline utilization and reduced loop management.

From the table, it is clear that increasing LMUL from 2 up does not significantly accelerate the computation. This is due to the “small” size of the problem. Even though this is the longest-running benchmark in the set, it still operates on only 20x20 matrices. For LMUL2, the vector register contains 8 elements, which means that the row or column of the matrix is processed in three computation steps. For the LMUL4, the vector register stores 16 elements, but the remaining 4 elements still force a second iteration of the loop. Moreover, each iteration with a larger LMUL requires additional clock cycles to complete. We conclude, that to fully utilize large LMUL values, much larger datasets are needed.

The fastest implementation turned out to be the BLASFEO. While this implementation does not use vector instructions, it splits the input matrix into several smaller tiles to optimize the memory access alongside computation.

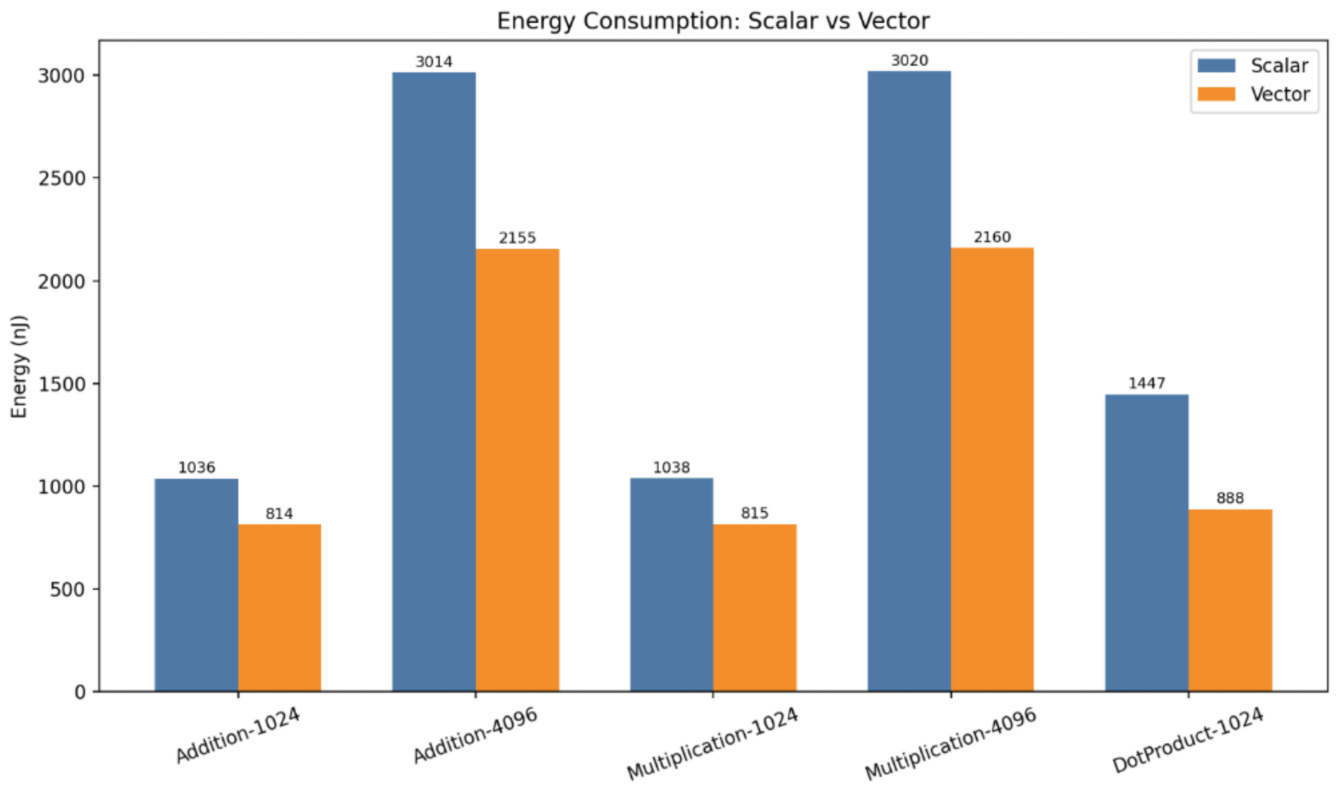


Figure 28: Energy Consumption: Scalar vs Vector (Shows energy savings for vector implementations across Addition, Multiplication and Dot Product benchmarks for 1024 and 4096 elements)

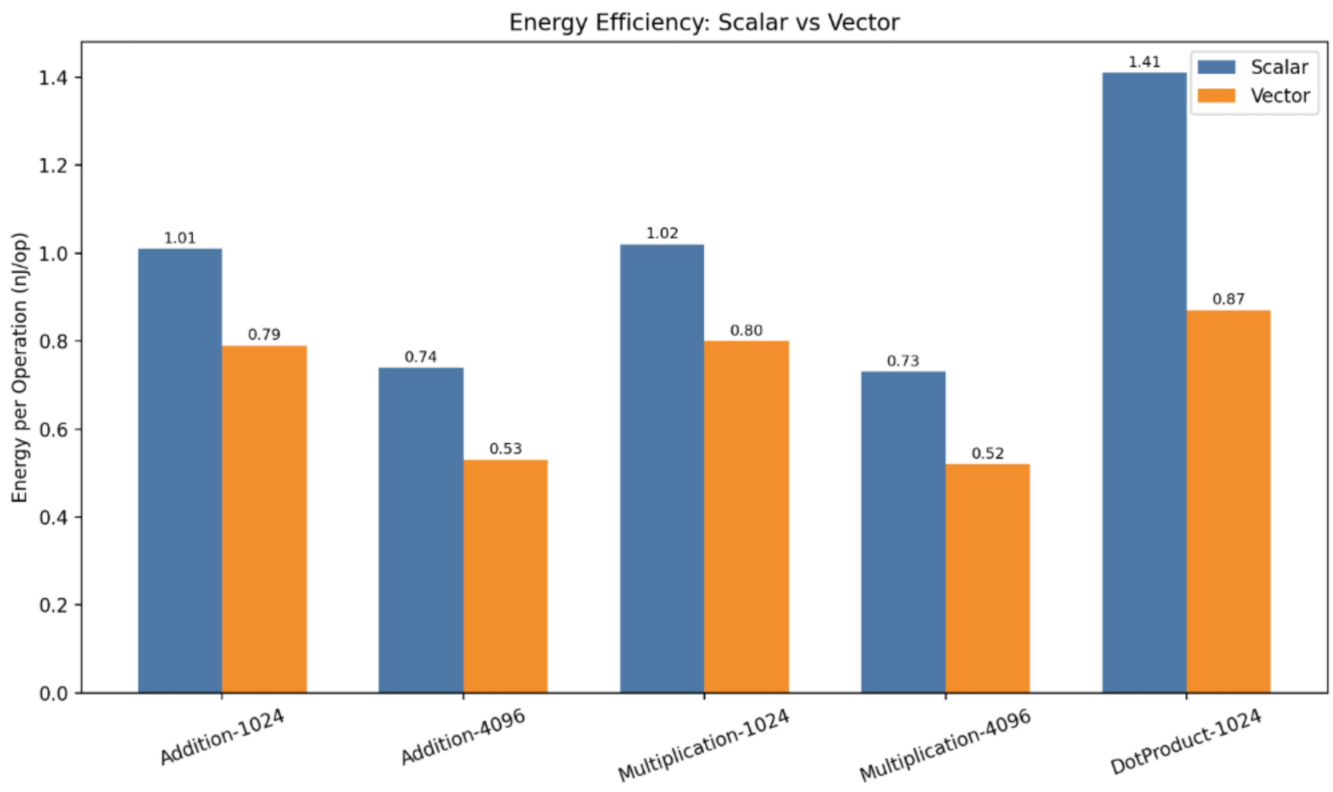


Figure 29: Energy Efficiency: Scalar vs Vector

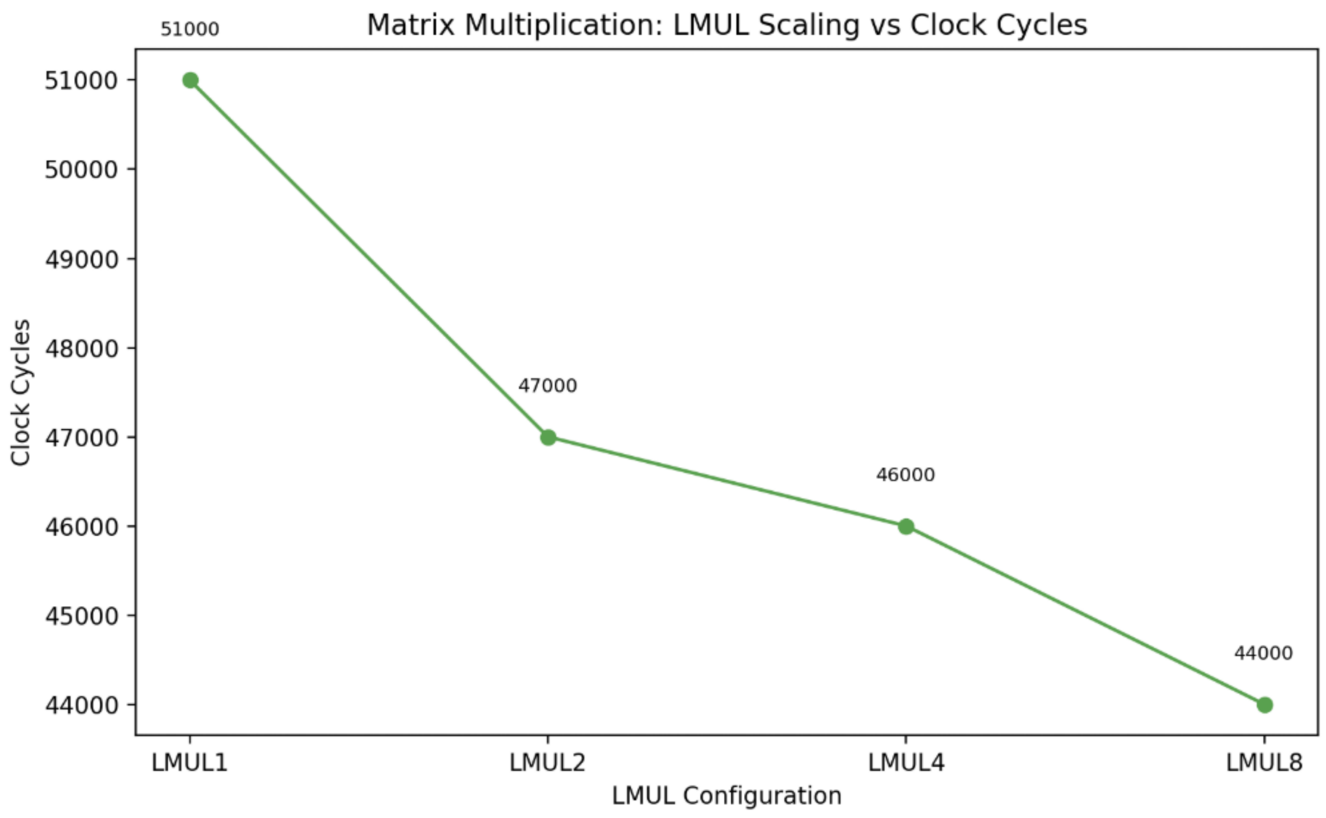


Figure 30: LMUL Scaling vs Clock Cycles during Matrix Multiplication (Shows diminishing returns for LMUL > 2 on small matrices, reinforcing need for larger datasets to exploit RVV fully)

5.2 CARM Tool—CSIP Integration

In our continued collaboration with Codasip, we have further developed our benchmarking tools to evaluate not only a greater variety of RISC-V cores, but also specialized accelerators. The support includes integer instructions (8, 16, 32 and 64-bit), custom fixed-point instructions for the MAC and DSP accelerators (Q7, Q15 and Q31 formats), floating-point instructions (32 and 64-bit), and bit-manipulation instructions part of the RISC-V Zbb extension. Furthermore, finer control over the benchmark instructions is now possible, mixing multiple arithmetic instructions, or targeting alternative ratios between loads and stores. This suite of roofline benchmarks was evaluated on the L31 and L110 cores, under a variety of microarchitectural configurations.

Codasip's L31 core provides several degrees of freedom regarding the configuration of the microarchitecture. In particular, the L1 cache can be configured in its size and number of ways. The change in size is reflected in the raw microbenchmark results as expected. As neither parameter has an influence on the memory bandwidth itself, the roofline plot remains unchanged. Given the single-issue nature of this embedded core, the theoretical peak floating-point performance is 2 operations per cycle, which the result tends towards. Compared to the floating-point benchmarks, the integer pipeline shows an identical memory bandwidth but a lower peak performance, i.e., close to the theoretical maximum of 1 operation per cycle, and the respective roofline can be observed in Figure 31.

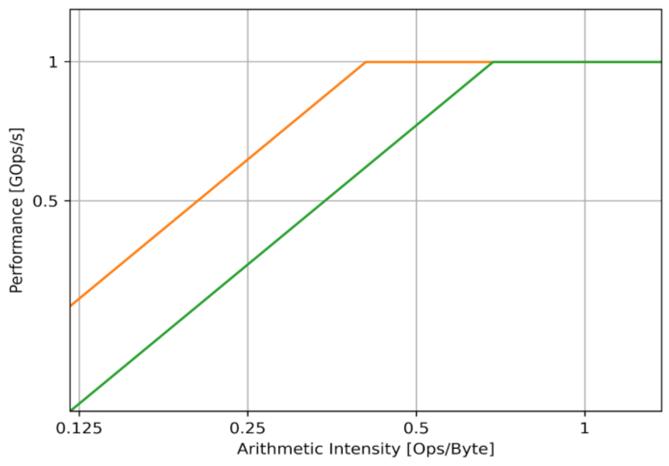


Figure 31: 32-bit integer CARM roofline of the L31 core

The L110 core can be equipped with a variety of Bounded Customisation accelerators, among which is a DSP accelerator, capable of performing dot products on large fixed-point vectors. The microbenchmarking required to evaluate the memory bandwidth of the accelerator differs significantly from typical CARM microbenchmarks, requiring custom instructions and a different code structure. As such, a specialised tool was developed to generate the microbenchmark code for this accelerator, which would be challenging to accommodate within a more general-purpose tool.

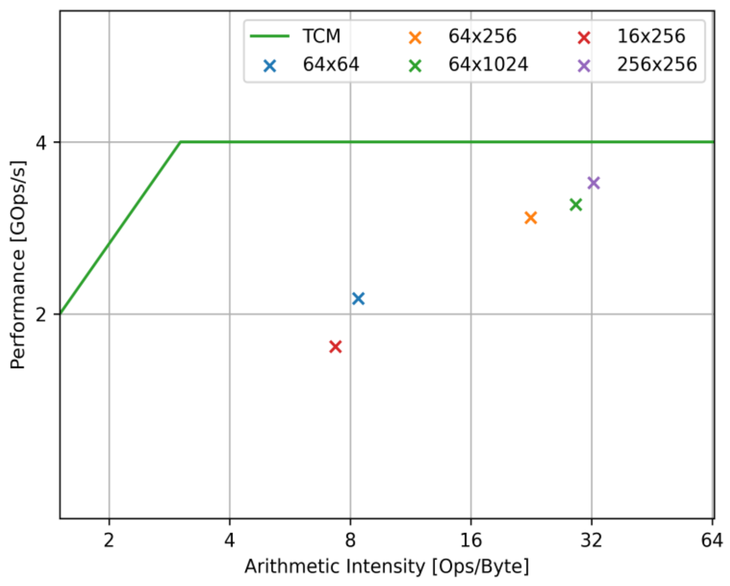


Figure 32: CARM roofline of the L110's DSP for the Q31 format and FIR filter performance (labelled as number of taps x block size)

As a case study, a Finite Impulse Response (FIR) filter application was instrumented and analysed using the CARM, accelerated using the L110 DSP. FIR filters have two key input parameters, which are the number of taps and the block size. These typically have a strong impact on the arithmetic intensity, with the number of taps also affecting the maximum size of the dot product. As shown in Figure 32, all tested configurations are under the compute-bound section of the roof, meaning the variation in performance is likely due to the variation in dot-product size. A lower number of taps reduces the maximum dot-product size, decreasing the number of operations per instruction, which lowers the hardware utilisation.

The Codasip Exploration Framework allows for the automated testing of software over a wide variety of cores and their configurations. Using this platform, we have benchmarked all compatible configurations, modelling them with the CARM. This allows for a fast performance assessment, and the identification of key parameters with a strong influence on performance. Figures 33 and 34 show part of the results of this exploration, covering multiple benchmark types and microarchitectural configurations, respectively.

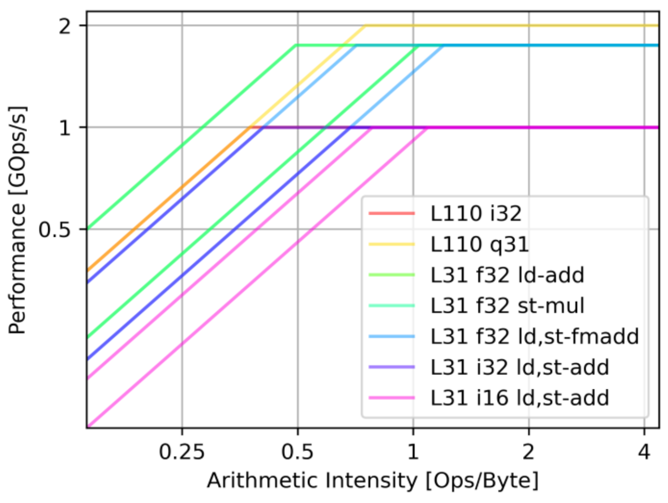


Figure 33: CARM rooflines of the L110 and L31 cores for a variety of precisions and instructions.

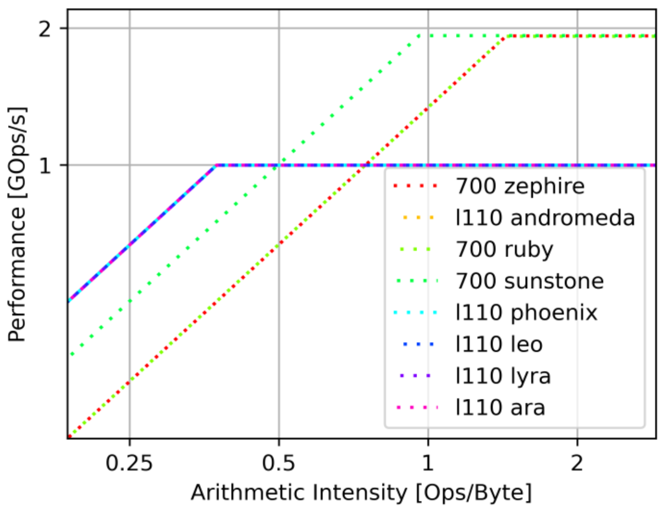


Figure 34: CARM rooflines for a variety of microarchitectural configurations of the L110 and the 700-series cores, for a 32-bit integer benchmark (load/store, add/mul)

5.3 HIRO EMDC Integration

5.3.1 PCIe Gen 6 Switch & CXL R&D for SYCLOPS

Edge-cloud AI workloads are becoming increasingly multi-agent, multi-model, and distributed across diverse accelerators. As the density of models and data grows the overall performance, scalability, and portability are no longer limited by compute throughput but by the movement of AI (hardware agnosticism, interoperability) and Data movement (bandwidth, smarter memory usage): shuttling model weights, KV-caches, embeddings, and intermediate tensors between storage, host memory, and heterogeneous devices. This challenge is amplified at the edge, where systems operate under strict latency, power, and DRAM constraints.

Three maturing technologies jointly address these bottleneck: (1) the SYCL programming model and (2) PCIe Gen 6 + CXL.

SYCL provides a unified, cross-architecture programming model that abstracts heterogeneous hardware and allows LLM pipelines, tokenizers, attention kernels, KV-cache updates, agent routing, and higher-level reasoning modules to execute across CPUs, GPUs, NPUs, and FPGAs without rewriting kernels or managing device-specific memory semantics. Its unified shared memory (USM), asynchronous execution, and emerging graph-execution extensions enable fine-grained scheduling, overlapping compute with data movement, and dynamic load balancing. SYCL becomes the software layer that can fully exploit the high-bandwidth, coherent interconnects emerging in the edge-cloud.

PCIe Gen 6 provides the next-generation physical layer needed for these workloads, delivering 64 GT/s PAM4 signalling (2× Gen5, 4× Gen4) and enabling low-latency, high-throughput paths between accelerators, storage, and host memory. PCIe Gen 6.0 is the first PCIe generation built for AI-native systems designed around the bottlenecks of distributed training, large-model inference, checkpointing, and memory movement at scale. PCIe not as a "peripheral interconnect" but as the primary compute fabric binding GPUs, NPUs, SmartNICs, CXL memory pools, and NVMe clusters into one coherent system.

CXL (Compute Express Link) solves the memory limitations of edge LLM systems by enabling dynamic memory expansion, shared memory pools, and coherent access to KV-caches and model state across heterogeneous devices.

Together, PCIe Gen6 + CXL 3.0 form a coherent, memory-centric fabric that enables capabilities impossible under traditional PCIe-only systems. The combination of SYCL + PCIe Gen6 + CXL is essential: AI applications become hardware-agnostic, can run on any mix of accelerators, and can fully leverage high-speed DMA transfers, peer-to-peer communication, shared context, pooled memory, and fast access to NVMe storage. PCIe Gen 6 ensures that SYCL's runtime, unified memory abstractions, and graph execution do not become constrained by legacy I/O limits unlocking heterogeneous, scalable, real-time LLM and multi-agent AI in the edge cloud.

In the SYCLOPS project, HIRO followed multiple R&D activities on PCIe gen6 and CXL:

1. Development of EMDC v1.0 of the RISC-V reference platform. (FPGA based) (1st reporting period)
2. Build an EMDC v2 testbed (not CXL capable) with multi-vendor accelerators to test transferability of a SYCL based application across GPUs from different vendors; (1st reporting period)
3. Develop and manufacture the PCIe Gen6 switch (Broadcom Atlas3 Gen 6 RDK/HIB, switch, and retimer) for EMDC. (**2nd reporting period**)
4. Set up a validation board testbed to validate the signal integrity of the design and when endpoints become available benchmark (**2nd reporting period**): (i) Maximum per-device throughput, (ii) Ensuring clean 64 GT/s PCIe Gen 6 signal integrity, (iii) Validating NVMe SSD → Host → Accelerator data paths, (iv) Achieving stable Gen 6 PAM4 operation across real cables and boards.
5. Research CXL 2.0 Memory pooling (Host CPU (Intel Xeon) + local DDR5 + Micron CXL memory module(s)) in EdgeMicroDataCenter testbeds, a CXL 2.0 capable testbed (**2nd reporting period**).

5.3.2 PCIe Gen 6 Switch Development

Competitive forces and the potential of PCIe Gen6 over PCIe Gen5 made Broadcom decide to speed up the market launch for Gen 6 switch Silicon. Synopsys published the most detailed early PCIe Gen 6 PHY results, demonstrating stable 64 GT/s signaling with PAM4 modulation and Forward Error Correction (FEC). Their benchmarks include eye diagrams, bit-error rate (BER) plots, and latency measurements across backplane and cable channels.
Key Data Points

- 64 GT/s PAM4 with $<10^{-15}$ BER after FEC
- Typical added latency: 20–30 ns per FEC decode stage
- Demonstrated channel loss tolerance up to 36 dB with adaptive DFE+CTLE

Broadcom is the only vendor showing system-level PCIe Gen 6 demonstrations with real components. (Teledyne LeCroy + Micron + Broadcom Gen 6 Demo (2025)). Because of the market volume the Hyperscale adoption of PCIe Gen6 has the priority of the OEM's with the Industrial/ Edge AI to follow in their footsteps.

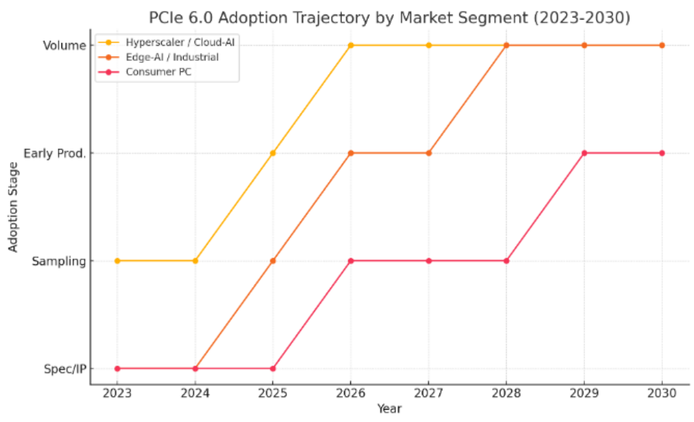
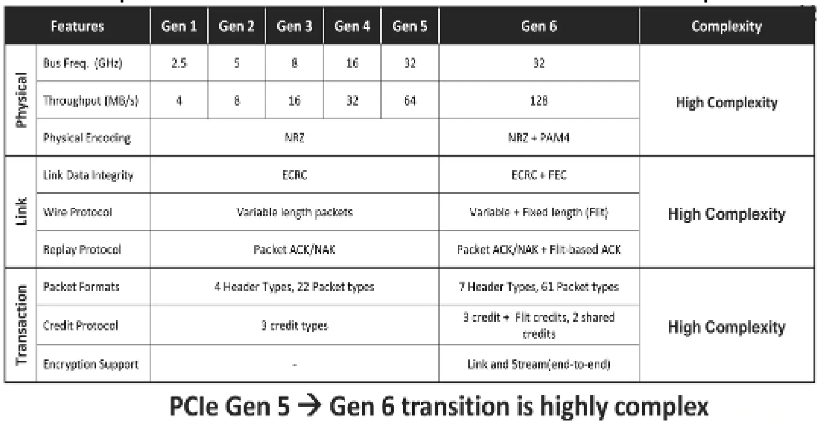


Figure 35: PCIe Gen 6 switch roll out and adoption in different domains.

HIRO decided to align with this market progress and aim for a Gen 6 switch development instead of the projected Gen 5 Switch. Gen 6 not only has more capacity but also more features which makes the development and utilisation of the Switch more complex.



	Features	Gen 1	Gen 2	Gen 3	Gen 4	Gen 5	Gen 6	Complexity
Physical	Bus Freq. (GHz)	2.5	5	8	16	32	32	High Complexity
	Throughput (MB/s)	4	8	16	32	64	128	
	Physical Encoding	NRZ					NRZ + PAM4	
Link	Link Data Integrity	ECRC					ECRC + FEC	High Complexity
	Wire Protocol	Variable length packets					Variable + Fixed length (Flit)	
	Replay Protocol	Packet ACK/NAK					Packet ACK/NAK + Flit-based ACK	
Transaction	Packet Formats	4 Header Types, 22 Packet types					7 Header Types, 61 Packet types	High Complexity
	Credit Protocol	3 credit types					3 credit + Flit credits, 2 shared credits	
	Encryption Support	-					Link and Stream(end-to-end)	

PCIe Gen 5 → Gen 6 transition is highly complex

Figure 36: PCIe transition complexity

Through SYCLOPS, HIRO was able to obtain evaluation boards and sample silicon for a PCIeGen 6 Switch to develop their switch in the form factor for EdgeMicroDataCenter. The signal integrity testing will be executed Jan 2026, using the data from the evaluation boards.

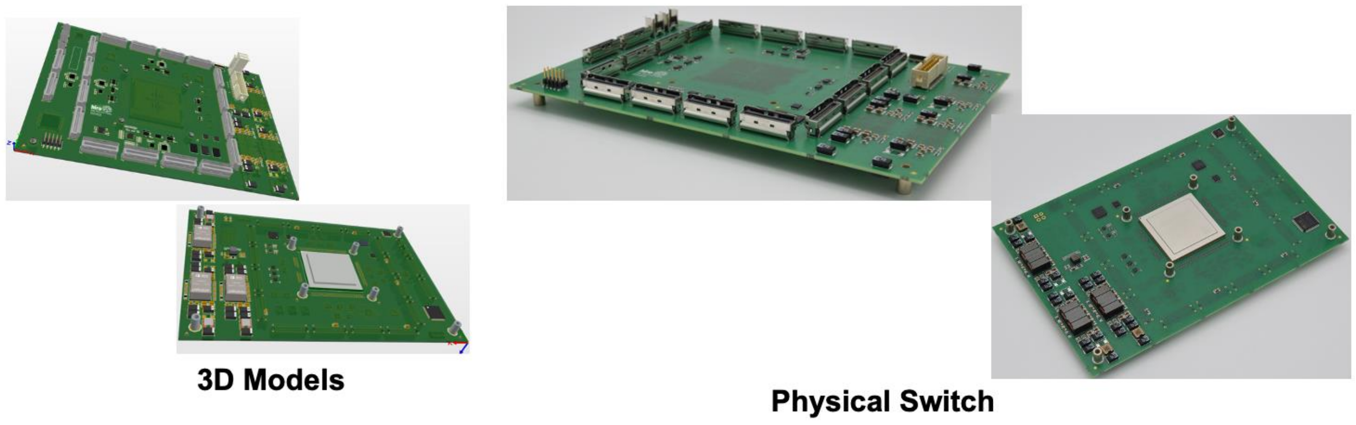


Figure 37: Models and physical design of PCIe 6.0 switch

5.3.2.1 CXL 2.0 capable testbed for switch testing

For a CXL 3.0 switch to be tested in CXL mode, you need, CXL 3.0 GPUs, CXL 3.0 memory expanders (first samples shipping Dec 2025), CXL 3.0 root complexes (CPUs with Gen 6 controllers). The critical fact: There are no CXL 3.0 endpoints available today, and first devices will become available 2026.

Also, Gen 6 retimers, switches, and connectors are significantly more complex through the introduction of PAM4 signalling and Forward Error Correction (FEC), which are required at 64 GT/s to maintain acceptable BER (bit-error rate). What our current evaluation board testbed what we can do is provide the measurements for the signal integrity testing of the manufactured switch for our microdatacenter. The next step in 2026 will be when cxl capable end point become available to test the performance of the evaluation board and the board designed for our microdatacenter.

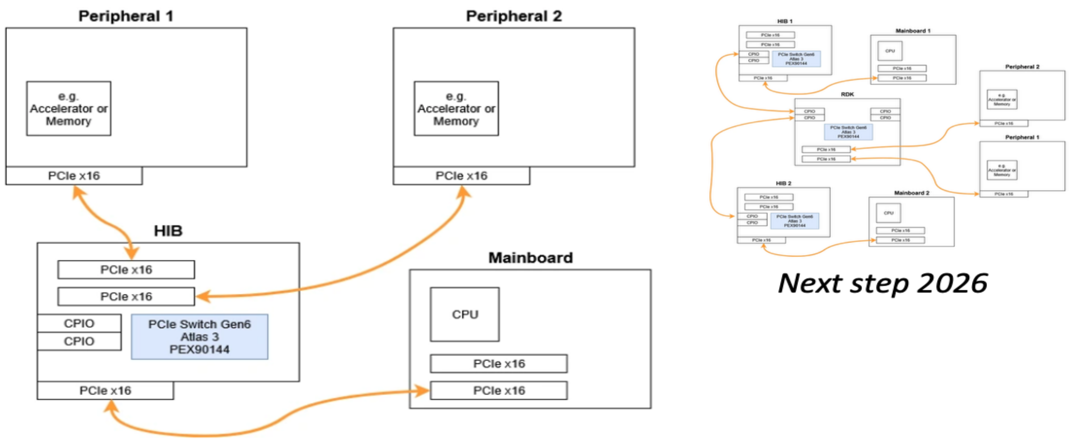


Figure 38: Evaluation Board Switch, testbed set-up

Signal integrity of PAM-4 components is critical for the overall system. Simulation-based verification was carried out using COM-HPC connector interfaces, and loss budgets were defined together with the COM-HPC standardization committee. Signal integrity measurements on real prototypes are planned for further evaluation⇒ Signal integrity is ensured.

5.3.3 CXL Research

As described in deliverable D3.2, HIRO has also put together an experimental testbed server equipped with a CXL memory module. AN experimental evaluation was conducted on this HIRO CXL testbed, which serves as a representative prototype of a next-generation heterogeneous edge–cloud compute node. The hardware platform is based on a dual-socket Intel Xeon 6740E system, equipped with local DDR5 memory per socket and external memory devices attached via PCIe/CXL. The configuration reflects early CXL-capable architectures that are expected to become common in modular edge micro-data-centers, enabling memory expansion, tiering, and future memory pooling scenarios. The system was configured in a NUMA-aware manner to allow explicit control over memory placement and access paths. With current hardware availability we were able to set up a Fabric Attached Memory File System (FAMFS).

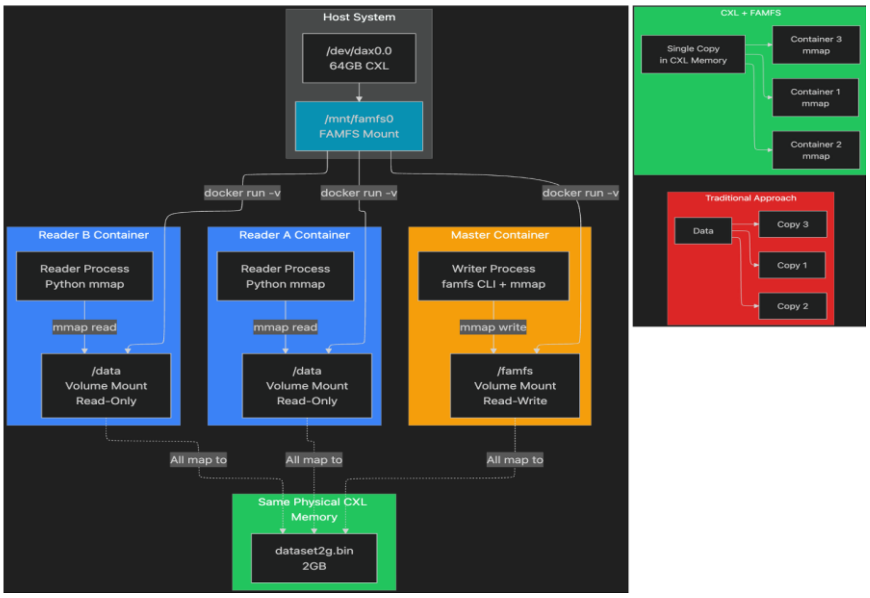


Figure 39: Multi-container FAMFS sharing architecture

On the software side, a Linux operating system with CXL support enabled was used. CXL memory was exposed to the operating system in two distinct modes: (i) NUMA mode, where CXL memory appears as an additional NUMA node and can be used as Tier-2 system memory, and (ii) devdax mode, which enables direct, page-cache-free access to CXL memory regions. Benchmarking tools, scripts, system configuration outputs, and result plots were collected and stored together with the experiments to ensure traceability and reproducibility of the measurements. This setup allows both low-level characterization and application-level evaluation, while remaining close to realistic deployment conditions.

5.3.3.1 Tests Performed

The primary low-level characterization was performed using a pointer-chasing microbenchmark. This benchmark measures average memory access latency by traversing a randomized linked list in memory, thereby minimizing the effects of caching and prefetching. The benchmark was executed against three memory classes: local DRAM, cross-socket DRAM, and CXL-attached memory. Each test was repeated under different load conditions, including no additional system load, light contention, and heavy contention involving multiple CPU cores performing concurrent read and write operations. This approach provides insight into both baseline latency and scalability under realistic contention scenarios.

In addition to microbenchmarks, an application-level experiment was performed using a Self-Organizing Maps (SOM) workload. SOM is a memory-intensive algorithm commonly used in data analysis and machine learning contexts. The application was configured to allocate its working data either from local DRAM or from CXL-attached memory, enabling a direct comparison of performance behaviour and validating the usability of CXL memory in realistic computational workflows rather than purely synthetic tests.

5.3.3.2 Outcomes

The measurements confirm the expected latency hierarchy across the memory tiers. Local DRAM access exhibited the lowest latency, with values on the order of approximately 120–190 nanoseconds depending on system load. Cross-socket DRAM access showed significantly higher latency, typically in the range of approximately 460–670 nanoseconds. Access to CXL-attached memory resulted in higher latency than DRAM, but remained well within the range expected for memory-semantic access rather than storage-class access.

Under heavy contention, latency increased for all memory classes, with CXL memory showing a more pronounced sensitivity to concurrent access, but without exhibiting instability or pathological behaviour.

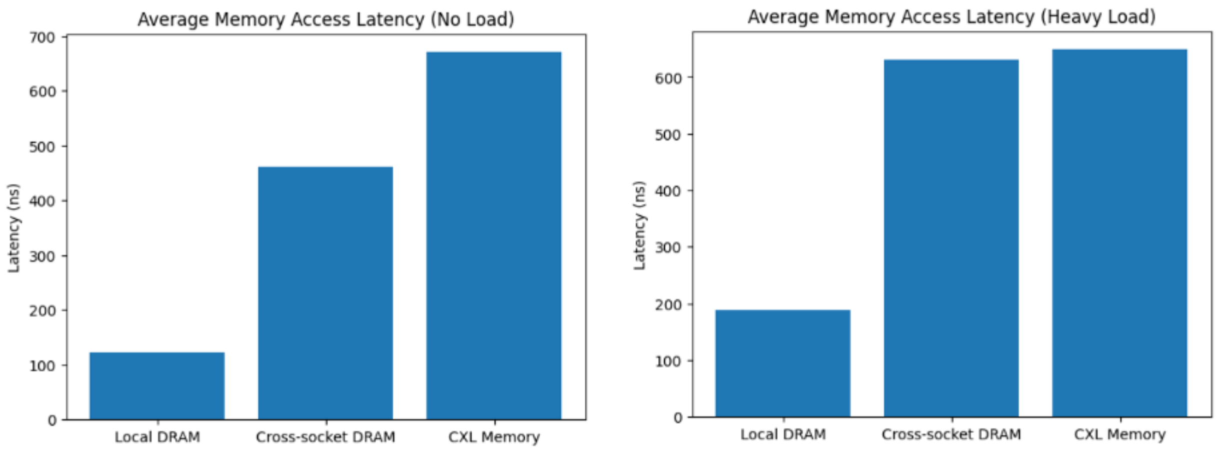


Figure 40: Average memory access latency under no-load conditions/ heavy load for local DRAM, cross-socket DRAM, and CXL-attached memory on the HIRO testbed

From a functional perspective, the operating system correctly enumerated and managed the CXL memory devices. NUMA policies and tools such as numactl were successfully used to bind application memory allocations explicitly to the CXL memory region. In devdax mode, the system enabled zero-copy access to shared memory regions, demonstrating the feasibility of using CXL memory as a shared resource across multiple processes or containers. All benchmark artefacts, including logs and plots, were successfully generated and archived, supporting reproducibility and further offline analysis.

5.3.3.3 Insights

The experiments demonstrate that CXL memory can be effectively used as a Tier-2 memory layer, providing capacity expansion beyond local DRAM while preserving load/store semantics. Although access latency is higher than for DRAM, it remains orders of magnitude lower than traditional storage solutions, making CXL memory suitable for large-footprint applications such as AI inference, in-memory analytics, and data-intensive edge workloads. The results also show that contention affects CXL memory performance more visibly than local DRAM, highlighting the importance of intelligent workload placement and memory-aware scheduling.

A key finding is the validation of two complementary CXL usage models within the same platform. NUMA-exposed CXL memory supports transparent memory expansion and tiering, while devdax-based access enables zero-copy shared memory usage across processes and containers. This dual capability significantly broadens the applicability of CXL in future edge-cloud systems, particularly in modular and composable architectures where resources are dynamically allocated. Overall, the maturity of the hardware–software stack is sufficient for meaningful experimentation, although integration complexity and management tooling remain areas for further development.

Recent external work (https://lnkd.in/g9eMHinw) has demonstrated how CXL 3.0–attached devices equipped with embedded controllers can be used not only for memory expansion, but also for intelligent, workload-aware memory management. In the context of large language model serving, this approach enables fine-grained KV-cache placement policies that significantly improve effective memory utilization and reduce end-to-end latency. While these results are not part of the present evaluation, they highlight the importance of coupling CXL-capable hardware with software and firmware-level intelligence. The current HIRO testbed and benchmarking setup provide a suitable foundation for exploring similar concepts in future work, including policy-driven memory tiering, controller-side decision making, and application-specific optimization strategies.

5.3.3.4 Integration Demonstration: SYCLDB on CXL server

In order to demonstrate integration of the EMDC hardware deployed by HIRO with rest of the SYCLOPS stack, we use SYCLDB once again as the vehicle. We focus exclusively on the CXL server as it has PCIe 5.0 interface and a fully functional hardware/software environment to get all SYCLOPS components integrated. We installed both DPC++ and ACPP on the CXL server. We compiled SYCLDB using both compilers, and ran a benchmarking workload to confirm that SYCLDB works in both cases, on both the CPU available in the CXL server and the L40S GPU. The figure below shows the execution time of SYCLDB under various queries from the Star Schema Benchmark, when the data is resident in CPU memory. As the server is a dual-socket configuration, and as the externally attached CXL memory is configured to be made visible as a separate NUMA domain, the data accessed by SYCLDB can be stored in one of three possible locations: (i) On the same NUMA node as the GPU, (ii) on the remote NUMA node, or (iii) on the CXL memory. The figure shows the performance of all three cases.

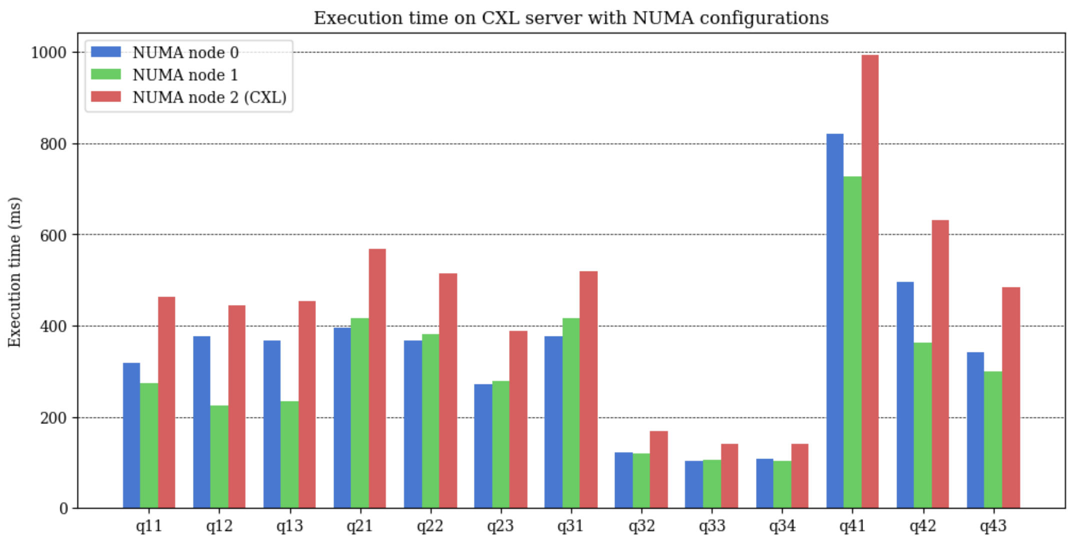


Figure 41: SYCLDB results on the CXL server

There are several important observations to be made from the figure. First, notice that we are demonstrating a fully functional, SYCL-based database query execution on GPU with the data sitting in CXL memory. This really brings together the work done in the entire SYCLOPS stack from the hardware level to the libraries level, and demonstrates clearly that the integration is fully functional. Second, as expected, the CXL memory (shown as NUMA node 2) has a much higher latency than the other two NUMA nodes. As SYCLDB kernels run on the GPU, they are bound by the PCIe bandwidth in case where the data sits in host memory (NUMA node 0 or NUMA node 1). When data sits in CXL memory, there is an additional penalty paid, which is expected, that is clearly visible in end-to-end query performance. This clearly motivates the need for further research on data placement policies for hybrid CPU-GPU query execution in the presence of an additional memory tier introduced by CXL. We plan to pursue such work in the near future continuing the strong collaborations put in place during the SYCLOPS project.

5.3.3.5 Future Research

Future work should extend the evaluation to more advanced CXL fabrics, including CXL 3.x and emerging CXL 4.0 capabilities, with a focus on multi-level switching, memory pooling, and multi-host access. Evaluating newer PCIe Gen6 and CXL-enabled switches is expected to provide insights into latency, bandwidth, and scalability improvements over the current generation. At the system-software level, further research is recommended on NUMA-aware schedulers, automated memory tiering policies, and tighter integration between CXL fabric management and platform management controllers.

From an application and orchestration perspective, additional benchmarks using full AI inference pipelines, analytics workloads, and multi-container workflows should be conducted to quantify end-to-end benefits. In particular, integration with higher-level orchestration frameworks and intent-driven provisioning mechanisms would allow exploration of how CXL resources can be dynamically allocated and reclaimed in operational environments. Finally, longer-term research should address reliability, isolation, and security aspects of shared CXL memory, as well as monitoring and telemetry mechanisms needed for production-grade deployment in edge micro-data-centers.

6 Conclusion

The SYCLOPS project has successfully fulfilled its mandate to scale extreme analytics through a unified, cross-architecture hardware-software stack based on open standards. By demonstrating performance-portable acceleration across three distinct and demanding domains—High Energy Physics, Genomics, and Autonomous Systems—the project has proven that vendor-neutral solutions can meet or exceed the performance of proprietary, vendor-locked ecosystems.

The integration of SYCL support into the ROOT framework and Cling interpreter has enabled interactive, hardware-accelerated data analysis within Jupyter Notebooks. The newly developed GenVectorX library demonstrated up to 3.5x speedup on heterogeneous backends while reducing energy consumption by approximately 73.9% for the Particle Acceleration (HEP) use case.

Through the SYCL-GAL library, the project achieved a 4.6x reduction in total execution time for the gold-standard GATK germline variant calling pipeline for the genomics use case. Notable highlights include an 11x improvement in preprocessing stages and accelerating the core pairHMM computation by two orders of magnitude.

Finally, SYCLOPS unified the portability of portDNN with the oneDNN ecosystem, allowing the PointNet architecture to run seamlessly across diverse hardware without application-level code changes for the autonomous systems use case. Optimized kernels for RISC-V vector extensions (RVV) achieved performance gains of up to 18.8x.

In conclusion, SYCLOPS has far exceeded its original KPI targets. It has provided a concrete demonstration that a software stack built on open standards like SYCL can deliver substantial speedups over CPU baselines and remain highly competitive with mature, proprietary solutions. This work ensures a more sustainable, energy-efficient, and vendor-independent future for high-performance computing in Europe and beyond.