SYCLOPS

Deliverable 5.1 – CUDA to SYCL porting tool

GRANT AGREEMENT NUMBER: 101092877

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

D5.1 – CUDA to SYCL porting tool

Executive Summary: This deliverable focuses on “Task 5.1: CUDA to SYCL porting tool” in WP5. This task aimed at developing and demonstrating a robust toolchain to facilitate the migration of existing CUDA applications to the open standard SYCL. The work done in SYCLOPS has achieved this goal by building two core components: SYCLomatic, an open-source command-line tool, and SYCLcompat, a compatibility library that bridges the gap for complex or proprietary CUDA features by providing SYCL-implemented functions that mimic CUDA behaviour. This synergistic approach significantly streamlines the migration workflow, as SYCLomatic typically translates around 85% of CUDA code successfully and highlights remaining complex sections for manual refinement.

WP:	5		
Author(s):	Joe Todd, Kumudha Narasimhan		
Editor:	Raja Appuswamy		
Leading Partner:	EUR		
Participating Partners:	CPLAY		
Version:	1.0	**Status:**	Draft
Deliverable Type:	Other	**Dissemination Level:**	PU
Official Submission Date:	06-Oct-2025	**Actual Submission Date:**	30-Sep-2025

Disclaimer

The SYCLOPS consortium consists of the following partners:

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

Document Revision History

Version	Description	Contributions
0.1	Structure and outline	EUR
0.2	Updated description of SYCLomatic	CPLAY
0.3	Updated description of SYCLDB	EUR
1.0	Final draft	EUR

Authors

Author	Partner
Joe Todd	CPLAY
Kumudha Narasimhan	CPLAY
Raja Appuswamy	EUR

Reviewers

Name	Organisation
Aleksandar Ilic	INESC
Vincent Heuveline	UHEI
Stefan Roiser	CERN
Nimisha Chaturvedi	ACC
Martin Bozek	CSIP

Statement of Originality

This deliverable contains original unpublished work except where clearly indicated otherwise.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

Table of Contents

Executive Summary

This deliverable focuses on “Task 5.1: CUDA to SYCL porting tool” in WP5. This task aimed at developing and demonstrating a robust toolchain to facilitate the migration of existing CUDA applications to the open standard SYCL. The work done in SYCLOPS has achieved this goal by building two core components: SYCLomatic, an open-source command-line tool, and SYCLcompat, a compatibility library that bridges the gap for complex or proprietary CUDA features by providing SYCL-implemented functions that mimic CUDA behaviour. This synergistic approach significantly streamlines the migration workflow, as SYCLomatic typically translates around 85% of CUDA code successfully and highlights remaining complex sections for manual refinement. .

The conversion tools were also used to convert CUDA to SYCL code and assist the development of software components in SYCLOPS and beyond. In SYCLOPS, the primary internal demonstration of this toolchain involved the SYCLOPS Use Case: SYCLDB, the project’s in-house data analytics library. The tools were applied to port the state-of-the-art Crystal GPU Join hash join operator from CUDA to SYCL. The evaluation on SYCLOPS EMDC confirmed that the resulting SYCL DB join kernel, after minor performance optimizations, was capable of matching the performance of hand-crafted CUDA counterparts and demonstrated portability across a wide variety of hardware, including x86-64 CPUs, NVIDIA GeForce L40S GPUs, RISC-V CPUs, and the SYCLARA RISC-V accelerator developed in SYCLOPS

A significant demonstration beyond the SYCLOPS project scope utilized SYCLomatic and SYCLcompat to generate the SYCL backend for Llama.CPP, a high-performance C/C++ inference engine for large language models (LLMs). This external use case provided an essential testing ground to benchmark the correctness and performance of the auto-generated SYCL code against the existing CUDA backend. This successful conversion showcased the utility of the toolchain in the rapidly evolving AI application vertical.

Ultimately, the project successfully demonstrated that combining the automatic conversion capabilities of SYCLomatic with the broad compatibility offered by SYCLcompat allows developers to quickly generate functional and performant SYCL applications from existing CUDA code, accelerating the adoption of heterogeneous computing. The toolchain is available as open source software and has already seen wide-spread adoption.

1. Introduction

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

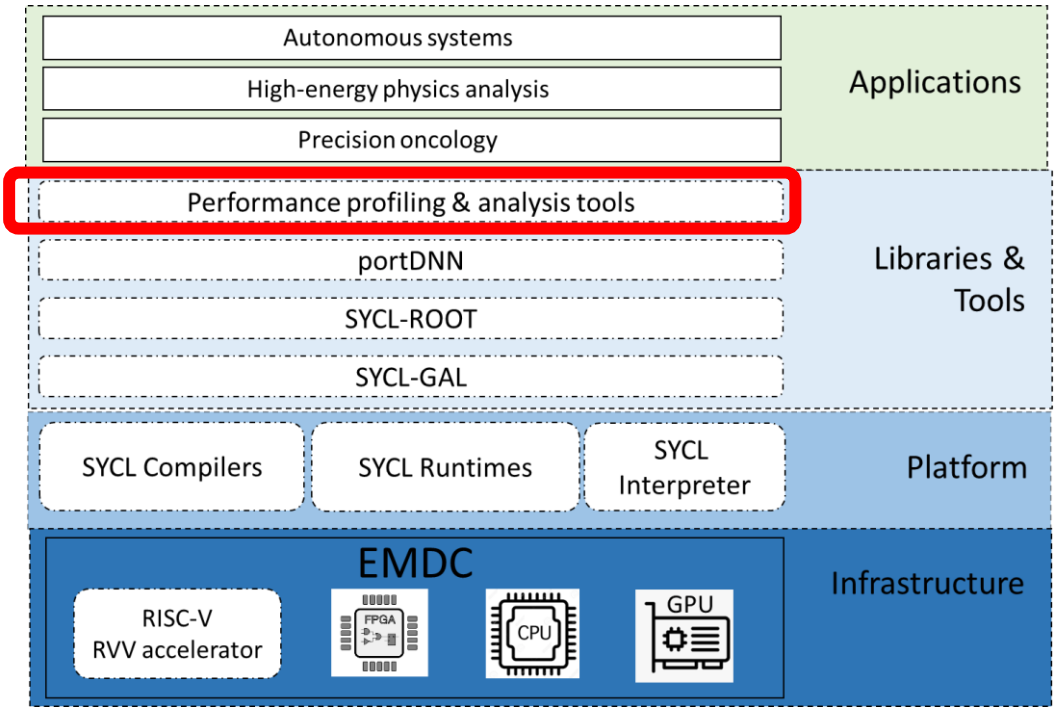

Figure 1. SYCLOPS architecture

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

Platform layer: The platform layer, provides the software required to compile, execute, and interpret SYCL applications over processors in the infrastructure layer.

Application libraries and tools layer: The libraries layer enables API-based programming by providing pre-designed, tuned libraries for various deep learning methods for the PointNet autonomous systems use case (SYCL-DNN), mathematical operators for scalable HEP analysis (SYCL-ROOT), and data parallel algorithms for scalable genomic analysis (SYCL-GAL). This layer also contains conversion tools, to facilitate porting of CUDA applications into SYCL, and profiling tools to enable the analysis of cross-architecture SYCL applications.

This deliverable presents the work carried out to enable **CUDA-to-SYCL porting tool** as highlighted in Figure 1 in the context of Task 5.1 of the SYCLOPS project. The work in this task “Objective 5.1: Develop porting tool to convert existing open-source CUDA into standard SYCL”. More specifically, this document provides a high-level overview of the work done on the SYCLomatic conversion tool and the SYCLcompat compatibility library in the context of SYCLOPS. We also concretely demonstrate the use of these tools in SYCLOPS project, by showing how it was used to convert CUDA-based relational query processing kernels to SYCL in the context of SYCLDB—the in-house data analytics library developed in SYCLOPS. We also explain how these tools can be used beyond SYCLOPS by showing we generated Llama.CPP’s SYCL backend.

2. SYCLomatic

SYCLomatic[1] is an open-source command-line tool designed to help developers migrate CUDA source code to SYCL. It streamlines the porting process by automatically converting key CUDA constructs, such as kernels, data types, and API calls, into their SYCL equivalents. This tool is especially useful for accelerating the modernization of codebases to support heterogeneous computing across different hardware architectures, including GPUs, CPUs, and FPGAs, without being tied to a single vendor's proprietary technology.

SYCLomatic is explained in this YouTube video: https://www.youtube.com/watch?v=isZyevA6jZM.

2.1 Key Features and Functionality

SYCLomatic's primary function is to **automate the translation** of CUDA code, reducing the manual effort and time required for porting. It handles a wide range of CUDA features, including:

- **Kernel Code:** It translates CUDA kernels, which are the core parallel computations, into SYCL kernels.
- **Data Types:** The tool converts specific CUDA data types and memory management functions (like cudaMalloc and cudaMemcpy) to their SYCL counterparts.
- **API Calls:** SYCLomatic identifies and replaces CUDA API calls with corresponding SYCL APIs, ensuring functional equivalence.

The tool provides a comprehensive report after the conversion, highlighting any code that couldn't be automatically translated. This allows developers to focus their efforts on the remaining complex sections, thereby increasing overall productivity. By providing a solid foundation for the migration, SYCLomatic significantly lowers the barrier to entry for adopting the SYCL standard.

2.2 Strategic Importance

For organizations, SYCLomatic represents a strategic asset for achieving hardware independence and future-proofing their high-performance computing applications. By migrating to SYCL, companies can:

- Reduce Vendor Lock-in: Eliminate reliance on proprietary technologies, allowing for greater flexibility in hardware procurement.
- Increase Code Portability: Enable applications to run on a broader range of hardware platforms from different vendors, maximizing performance and efficiency.
- Enhance Developer Productivity: Automate a significant portion of the porting process, allowing development teams to focus on innovation rather than tedious manual conversions.

This tool is a critical component of a strategy aimed at leveraging the full potential of diverse computing architectures and ensuring that software investments remain viable in a rapidly evolving hardware landscape.

[1] https://www.intel.com/content/www/us/en/developer/articles/technical/syclomatic-new-cuda-to-sycl-code-migration-tool.html

3. SYCL Compat

SYCLcompat[2] is a library that acts as a complementary piece to the SYCLomatic migration tool. While SYCLomatic automatically **translates CUDA code** to SYCL, some CUDA features, especially complex or vendor-specific APIs, don't have a direct one-to-one equivalent in the SYCL standard. SYCLcompat steps in to provide a **compatibility layer** that helps bridge these gaps.

3.1 How SYCLcompat Works

SYCLcompat provides a set of headers and functions that mimic the behaviour of common CUDA APIs. When SYCLomatic performs a migration, it may replace a CUDA function call with a corresponding SYCLcompat function. This allows the migrated code to compile and run without needing a complete manual rewrite for every single CUDA feature. The SYCLcompat functions are themselves implemented using standard SYCL, which ensures that the resulting application remains portable across different hardware platforms.

3.2 The Synergy Between SYCLomatic and SYCLcompat

SYCLomatic and SYCLcompat work together to create a streamlined migration workflow. The process typically looks like this:

1. **SYCLomatic Migration**: The developer uses the SYCLomatic tool to automatically convert their CUDA source code to SYCL.

2. **SYCLcompat Integration**: During this process, SYCLomatic inserts calls to the SYCLcompat library for CUDA APIs that are not part of the core SYCL specification.

3. **Manual Refinement**: The developer then addresses any remaining issues flagged by SYCLomatic, focusing on complex logic and performance tuning rather than basic API translation.

4. **Final Compilation**: The migrated code, which now includes SYCL and SYCLcompat calls, is compiled with a SYCL compiler.

Essentially, **SYCLomatic is the migration engine**, while **SYCLcompat is the compatibility layer** that makes the migrated code functional and portable. This two-part approach significantly simplifies the porting process, reducing the amount of manual work and accelerating the adoption of heterogeneous computing.

3.3 Code comparison with CUDA

Putting it all together, the following code snippet shows an implementation of the “Slope intercept kernel” in CUDA as an example.

[2] https://intel.github.io/llvm/syclcompat/README.html

```
// Slope intercept form of a straight line equation: Y = m * X + b
__global__ void slope_intercept(float *Y, float *X, float m, float b, size_t n) {
    // Block index
    size_t bx = blockIdx.x;
    // Thread index
    size_t tx = threadIdx.x;

    size_t i = bx * blockDim.x + tx;
    if (i < n)
        Y[i] = m * X[i] + b;
}
```

Device memory allocation and copying functions that provide data input to the kernel in CUDA are shown below.

```
// Slope intercept form of a straight line equation: Y = m * X + b
template <int BLOCK_SIZE>
void slope_intercept(float *Y, float *X, float m, float b, size_t n) {

  // Block index
  size_t bx = syclcompat::work_group_id::x();
  // Thread index
  size_t tx = syclcompat::local_id::x();

  size_t i = bx * BLOCK_SIZE + tx;
  // or  i = syclcompat::global_id::x();
  if (i < n)
    Y[i] = m * X[i] + b;
}
```

Finally, the kernel launch in CUDA is performed as follows.

```
std::cout << "Computing result using CUDA Kernel... ";
slope_intercept<<<blocksPerGrid, threadsPerBlock>>>(d_Y, d_X, m, b, n_points);
cudaDeviceSynchronize();
std::cout << "DONE" << "\n";
```

The equivalent kernel in SYCLcompat is illustrated below.

```
// Allocate device memory
float *d_X = nullptr;
float *d_Y = nullptr;
cudaMalloc((void **)&d_X, mem_size);
cudaMalloc((void **)&d_Y, mem_size);
CHECK_MEMORY(d_X);
CHECK_MEMORY(d_Y);

// Copy host memory to device
cudaMemcpy(d_X, h_X, mem_size, cudaMemcpyHostToDevice);
```

Corresponding device memory allocation and copying functions that provide input to the SYCL kernel in SYCLcompat are shown below.

```
// Allocate device memory
float *d_X = (float *)syclcompat::malloc(mem_size);
float *d_Y = (float *)syclcompat::malloc(mem_size);
CHECK_MEMORY(d_X);
CHECK_MEMORY(d_Y);

// copy host memory to device
syclcompat::memcpy(d_X, h_X, mem_size);
```

Finally, kernel launch in SYCLcompat is done as follows.

```
std::cout << "Computing result using SYCL Kernel... ";
syclcompat::launch<slope_intercept<32>>(grid, threads, d_Y, d_X, m, b,
                                        n_points);
syclcompat::wait();
std::cout << "DONE" << "\n";
```

4. Integration & SYCLOPS Use Case: SYCLDB

SYCLDB is a library of relational primitives developed in SYCLOPS in the context of Task 5.4 (SYCL-GAL). Our goal in developing SYCLDB was to take the first steps towards investigating the utility of SYCL in developing performance-portable database engines. SYCLomatic was used to develop initial versions of relational kernels in SYCLDB, which were then subsequently optimized. Here, we focus on the hash join operator as an example. We first provide an overview of a state-of-the-art hash join algorithm implemented in CUDA by the Crystal library[3] targeting GPUs. Then, we detail how we ported it from CUDA to SYCL using SYCLomatic.

4.1 Crystal GPU Join

The novelty of Crystal lies in its tile-based implementation strategy. The idea behind tiling comes from the observation that threads in a GPU are grouped into thread blocks (in CUDA terminology) such that threads within a thread block can communicate through shared memory and synchronize through barriers. The set of data elements that can be collectively processed by a thread block is referred to as a tile. The basic compute unit in Crystal is a tile, which is a sub slice of the input data. This approach makes it possible to write kernels in terms of block-wide functions that take work with a set of tiles as units of input and output. Each function uses vector instructions for memory accesses, and registers for storing values. %For instance, the BlockLoad function can be invoked in a kernel to load, for any given tile, all elements of the tile in a vectorized fashion from global memory into thread-local registers.

Using block-wide functions, Crystal implements a no partitioning join, which uses a non-partitioned global hash table. The join operator comprises two kernels, a build kernel and a probe kernel. The build kernel populates the hash table with the tuples of the smaller, build relation. Crystal implements a linear probing strategy due to its simplicity, with the hash table being implemented as a simple array of slots with each slot containing a key and a payload without any pointers. The probe kernel uses the other relation to search for matches in parallel. Each thread block loads a tile from the probe table, and each thread computes the local sum for a subset of tile elements that meet the predicate condition. Then, all local values are aggregated in a hierarchical fashion, first for all threads within a block, and then across all thread blocks.

4.1 Porting CUDA to SYCL

In order to port the Crystal join from CUDA to SYCL, we start with SYCLomatic that aims to convert CUDA code to SYCL at syntax level, recognizing the main CUDA constructs and converting them to their SYCL equivalent. We use this tool to convert the Crystal hash join implementation together with necessary block-wide functions from CUDA to SYCL. Our goal in using the compatibility tool is to understand and document issues in converting various aspects like data movement, kernel parameterization, atomics and synchronization from CUDA into SYCL, in order to assist in future migration of current CUDA-based GPU database engines.

SYCLomatic takes a .cu file as input and produces its SYCL counterpart. Thus, we apply the command to all .cu file of the project. At the source level, the overall translation is quite accurate. SYCLomatic automatically adds necessary boilerplate such as headers and compiler directives required for enabling SYCL compilation. Similarly, SYCLomatic preserves and converts templatized functions that correspond to block primitives and join kernels of the Crystal library for most part, with some minor syntactic modifications. At the programming model level, SYCLomatic replaces CUDA kernel launches with an *nd_range -- parallel_for* kernel.

Further, CUDA data management calls that move data from host to device memory, or assign specific values to device allocated memory regions, are replaced with appropriate SYCL calls. Despite its utility,

[3] https://github.com/anilshanbhag/crystal

one cannot expect SYCLomatic to blindly convert everything automatically and correctly. The first issue concerns kernel dimensions. CUDA programming model requires kernel dimensions to be specified in terms of number of threads in a thread block, and the number of thread blocks per grid. Moreover, both thread blocks and grids can be multidimensional. Similarly, SYCL uses the notion of work-item and work-group. Thus, a CUDA thread block roughly corresponds to a SYCL work-group, and a CUDA thread gets mapped to a work-item in SYCL. SYCL also provides an *nd_item* object to enable index lookup in a *nd_range* kernel. It represents the index of each work-item.

The compatibility tool converts the two CUDA join kernels - build and probe - in two SYCL *nd_range parallel_for* kernels, and automatically adds the *id_item* as parameters of all functions called in the kernel code. However, despite the fact that the original code implements a 1D kernel, SYCLomatic converts it into 3-dimensional kernel. As consequence, all accesses to the threads indexes (local-id, global-id, group-id) within the kernel code were wrong and needed to be rewritten.

Second, synchronization primitives and low-level constructs were not ported correctly. For example, in the original code, threads in the probe-kernel have to compute the sum of the product for all entries that match the query predicate. This involves a certain number of local sum computations performed by each thread that are first aggregated at the tile level by all threads within a thread block, and then aggregated across all thread blocks. This involves the use of memory barriers, atomics, and synchronisation at various kernel execution stages. More precisely, all threads in a warp compute aggregate their value using a low level primitive (*shuffle_down*) that allows inter-thread communication without any cost. The value computed by each warp is saved in local memory. A tree-reduction pattern is used to compute the aggregate sum per thread block. Finally, after all thread blocks compute their local sum, the global sum is computed using atomic instructions in the global memory.

While SYCLomatic is able to convert the memory barrier and the atomic variables from CUDA to SYCL, it was not able to replace the warp-level functions which are a central piece of the Crystal tile-based probe kernel. Thus, we had to reimplement the logic. SYCL already provides a set of functions that implement the main data-parallel patterns at the work-group level. Thus, we map the concept of a tile from Crystal to a work-group in SYCL and use the *reduce()* function of the *work_group* class to perform tile-level reduction directly without having to implement warp-level shuffles and block-level tree reduction manually.

Finally, in some cases, even when the SYCL conversion is semantically correct, it might be suboptimal in terms of performance. An example is the call to the memory barrier function. SYCLomatic converts it automatically into a memory fence in both global memory and local memory which are very expensive. However, in this specific case, a memory fence in the local memory of each work-group was sufficient. Thus, we optimized the code generated by SYCLomatic.

4.2 Evaluation

We ran the resulting SYCL join on a wide variety of hardware, many of which was developed and deployed in SYCLOPS, through our work done in the Infrastructure Layer (WP3). We provide a short summary of the hardware below.

- A x86-64 server equipped with a 24-core Intel(R) Core(TM) i9-10920X Cascade Lake CPU clocked at 3.50GHz and 128GB of DDR4 RAM. The system runs Ubuntu 22.04.5 LTS. We compile SYCLDB with ACPP compiler (v25.02.0) developed in SYCLOPS.
- Two NVIDIA GeForce L40S GPUs accessed via a PCI-Express Gen4 x16 interface, which provides a theoretical peak bandwidth of 31.5 GB/s in each direction. We use the same ACPP v25.02.0, built with NVCC v12.4 support, to target the GPUs.
- A System-on-Chip equipped with a 64-core SG2042 RISC-V CPU clocked at 2GHz and 128 GB of DDR4 RAM.
- The SYCLARA RISC-V accelerator developed in SYCLOPS. We use the DPC++ compiler (v2024-06-03) to cross compile SYCLDB, and communicate with the device via OCK v4.0.0.

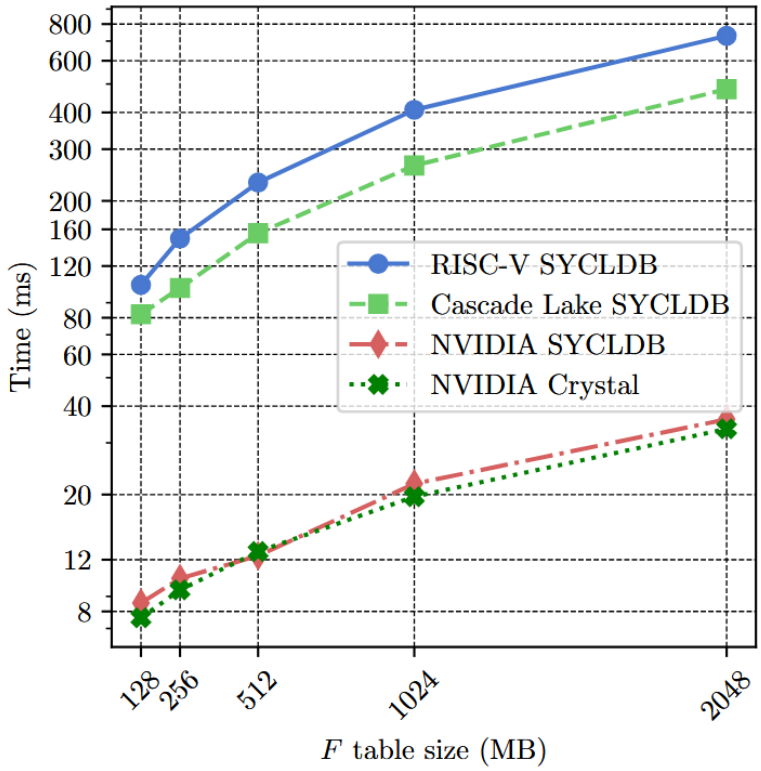

Figure 2. Execution time of SYCLDB join kernel under variety of hardware

For the workload, we generate a synthetic database containing two tables *F* and *D*. Table *F* is a large fact table containing two columns f1 and f2 both of which are 4-byte floating point values. Table *D* is a small dimension table containing two columns d1 and d2 which are also 4-byte floating point values. The join query that performs SELECT SUM (f1 × d1) FROM F, D WHERE f2 = d2. For all experiments, the D table is fixed at 128MB and the F table is 2GB.

Figure 2 shows the execution time of SYCLDB and Crystal on various CPU and GPU backends. There are several important observations to be made. First, comparing SYCLDB with Crystal, we see that their performance is identical. Prior work comparing SYCL versus CUDA for database acceleration had shown CUDA-based implementation to be 4× faster than a SYCL-based one. Our result shows that this gap has been closed, thanks to improvements in SYCL compiler toolchains. Second, comparing CPU and GPU, we can see that the NVIDIA GPU is 11× faster than the x64 CPU, which in turn, is 1.4× faster than the RISC-V CPU for the join query. This can be explained by the fact that the memory on the NVIDIA GPU has an order of magnitude higher memory bandwidth compared to the x64 and RISC-V CPUs. Similarly, SG2042 is one of the earliest RISC-V CPU tape outs available in the market. Given this, it is understandable that its performance is not competitive with the more mature x64 CPU.

To summarize, our results show that SYCLCompat is a useful tool in assisting developers to port CUDA to SYCL code. The code generated by SYCLCompat, with minor optimizations, is capable of matching the performance of hand-crafted CUDA counterparts, while being portable across diverse processor architectures.

5. Use Case Beyond SYCLOPS: LLAMA.CPP

In addition to the use of SYCLCompat in the context of SYCLDB described earlier, we have also performed experiments to show case its utility in AI application verticals. Llama.cpp is a high-performance C/C++ inference engine for large language models (LLMs). It was originally developed to run Meta's Llama models on consumer hardware, particularly without requiring a powerful GPU. Its core mission is to provide an efficient and portable way to run LLMs on various platforms, from laptops and mobile devices to embedded systems.

SYCLomatic & SYCLcompat were essential components of Llama.cpp's SYCL backend, which was partly automatically generated from the existing CUDA backend. Details of the project to port Llama.cpp from CUDA to SYCL are described in detailed blogs by Ruyman Reyes which was shared on the SYCLOPS website (https://www.syclops.org/updates/2024/07/31/porting-ai-codes-from-cuda-to-sycl-and-oneapi-one-llama-at-a-time-part-one,https://www.syclops.org/updates/2024/08/13/part-two-porting-ai-codes-from-cuda-to-sycl-and-oneapi-one-llama-at-a-time).

6. Conclusion

The work conducted demonstrated a successful approach for migrating CUDA applications to the open standard SYCL, relying on the combined strengths of the SYCLomatic conversion tool and the SYCLcompat compatibility library. This synergy allows developers to rapidly generate functional and performant SYCL applications based on existing CUDA code. A key metric of success is that SYCLomatic typically translates approximately 85% of CUDA code successfully, thereby dramatically reducing the manual effort required. Furthermore, the tool intelligently highlights its own translation limitations, making it straightforward for developers to identify and focus on the remaining complex sections that require refinement. Importantly, the resulting kernel demonstrated high performance and portability across diverse processor architectures, including NVIDIA GPUs, x64 CPUs, RISC-V CPUs, and the SYCLARA RISC-V accelerator developed within SYCLOPS and deployed in the SYCLOPS EMDC.

Finally, the tools proved their utility beyond the project's scope through the successful generation of the SYCL backend for Llama.CPP, a high-performance inference engine for large language models (LLMs). This external use case provided an ideal testing ground, leveraging the existing CUDA backend as a crucial benchmark to verify the correctness and performance of the automatically generated SYCL code.

In conclusion, the project successfully demonstrated that the SYCLomatic and SYCLcompat toolchain provides a vital means for accelerating the adoption of heterogeneous computing by providing a streamlined, efficient, and performance-competitive method for migrating high-performance computing applications away from proprietary CUDA ecosystems.

All work done on SYCLomatic and SYCLcompat has already been made open source in their public Github repositories.

References

[1] https://codeplay.com/portal/blogs/2024/07/31/porting-ai-codes-from-cuda-to-sycl-and-oneapi-one-llama-at-a-time-part-one

[2] https://intel.github.io/llvm/syclcompat/README.html

[3] https://www.youtube.com/watch?v=isZyevA6jZM