Celerity Evaluation Project Report

Experience of using Celerity runtime and API with hipSYCL to implement HPC kernels

Ioannis Vardas

January 10, 2021

Contents

1 Introduction

For the purposes of this project I have implemented two kernels using the Celerity runtime and the hipSYCL implementation of SYCL. I decided to implement a lu decomposition and a matrix multiplication kernel. As the implementation of the lu kernel is not quite right I decided to include the matrix multiplication kernel which I implemented **first** in order to test my understanding of the Celerity runtime. I should also say that I have never used SYCL and the only GPU programming I have worked on is a very basic level of CUDA. This report documents my steps from the beginning to the end of this endeavour.

1.1 Environment

I chose hipSYCL since it is an open source project and in principle, I prefer compiling from source rather than using ready-binaries due to performance optimization and other issues. Also I found the documentation very clear and generally easy to install in any Linux machine. This project was run on the CPU instead of the GPU since the architecture of my GPU was not supported by AMD ROCm.

1.1.1 Machine + OS

- AMD Ryzen 5 3600x 6 Core 12 Threads Processor (x86 64)
- Gentoo Linux distribution with the latest kernel 5.10.4

1.1.2 Software

- LLVM + Clang 11.0.0
- gcc (g++) 10.2.0 with OpenMP
- Open MPI 4.0.4

1.2 hipSYCL implementation of SYCL

hipSYCL compiled and run without any issue using cmake and ninja. I run some simple tests on my CPU to confirm its runtime integrity.

1.3 Celerity

Although the installation of hipSYCL was successful and tests were run successfully on the CPU, Celerity framework had trouble identifying the devices and using the CPU. Thus, I was unable to run any of the tests provided by Celerity. As this project is relatively new I could not find information about this particular issue. To solve this issue I decided to do a "dirty hack" in Celerity's source code.

1.3.1 The dirty hack in Celerity source code

I changed the following line in device queue.cc file:

```
const auto devices = cl::sycl::device::get_devices(cl::sycl::info::device_type::gpu);
```

After confirming that the following device type is available I changed it to:

```
const auto devices = cl::sycl::device::get_devices(cl::sycl::info::device_type::cpu);
```

I am aware that this is just a temporary solution literally, a hack. This issue could possible be related to hipSYCL or a wrong configuration by me but the objective was not to fix hipSYCL for my system. It should also be noted that if the Celerity was not open source I could not have hacked it and it would be unusable.

2 Celerity Application tutorial

After setting up the environment, I followed the complete application tutorial for celerity. However, the tutorial presented in `https://celerity.github.io/docs/tutorial` was not actually working for me. More precisely the last part which was about copying the result from the kernel to a file did not compile as the method `get_pointer()` could not be found. Eventually I managed to compile the tutorial by adding `edge_buf.get_access<cl::sycl::access::mode::read, cl::sycl::access::target::host_buffer>` to the `edge_buf` accesor. Although I begun to understand a bit more about the Celerity framework but also I now understood that I knew nothing about SYCL. I believe it was necessary for me to understand some basics about SYCL and thus, I searched for some documentation on SYCL, I include the links in the 6 section.

3 Matrix Multiplication kernel

My final goal was to implement a LU decomposition kernel but due to it being more complex I first chose to implement a matrix multiplication kernel and using that as a frame build the LU decomposition. Using the Tutorial section code as a frame I created a matrix multiplication kernel. For simplicity I assume square matrices.

3.1 Implementation

I believe that the most challenging part was understanding and using the range mappers, correctly. For this reason I would not consider the implementation of this kernel as trivial. I found it suitable to use the `slice` range mapper for the input matrices and the `one_to_one` for the output matrix. Documentation suggests using the `slice` range mapper for dense matrix multiplication and I kept the `one_to_one` range mapper from the tutorial code for the output matrix. The second most challenging task was to write the actual computation kernel. Compared to the previous kernel, I used another for loop to calculate the element e of the output matrix where:

$$e_{ij} = \sum_{k=0}^{N-1} a_{ik} b_{kj}$$

I also used a function to generate random floats for the two matrices, the array with the result is initialized with 0. The verification is done with a host task with the master node tag. The documentation sections about range mappers and host-tasks as well as pitfalls were proven invaluable for the implementation of this kernel.

4 LU decomposition kernel

As I mentioned earlier, the implementation of the LU decomposition kernel was not successful. However, I included it since I worked on it and I consider it the most important part of this project. In a way this implementation might include what I may have missed or misunderstood so far.

4.1 Implementation

I implemented the kernel in a separate function and I kept only the printing functions for the matrices. I did not implement the error checking as the matrix multiplication as it was evident that in almost every case this kernel would produce an incorrect result. I tried to simplify the kernel code making it clear that in the first for loop the lower part is being calculate and in the second loop the upper part.

4.2 What did go wrong

Again the first issue I would mention is the range mappers. LU decomposition (or at least the algorithm that I had in mind) has a more complex access pattern that matrix multiplication and produces two matrices instead of one. As I was not sure which range mapper should be used, I used the `all` range mapper, although intuitively I used the `slice` range mapper, at first. The most important issue is that the accesses between Lower and Upper matrices are asynchronous and the code I present has dependencies between the two matrices. But without synchronization the result will not be correct in most cases.

5 Conclusion and Remarks

It was an educational but also entertaining experience using all these new frameworks and runtimes, from SYCL to Celerity. Hacking the code of Celerity was surely one of the most fun parts and also learning to use the range mappers of celerity. In the end, this endeavour proved to be more challenging than I expected but also more entertaining and educational. I am very happy with this result even though I did not implement LU decomposition kernel correctly. I could say that matrix multiplication reflects what I managed to capture and understand whereas LU reflects what I missed or misunderstood. Parallelism is something that requires meticulous steps with many details and there **always** is much more behind every line of code. The Celerity framework has very clear documentation but I believe it should have more, with examples and/or pictures. Of course, I understand that Celerity is a research project and it does not cover all the cases. But the beauty of research, is to have something to explore. All in all, I believe I have attained a better understanding of this framework and I would like to know more and maybe solve my problem in the future.

6 Resources

- https://developer.codeplay.com/products/computecpp/ce/guides/sycl-guide/hello-sycl
- https://mmha.github.io/SYCL-Docs/sycl/sycl.pdf