POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

Toward portable HPC applications with SYCL: a molecular docking case study

Laurea Magistrale in Computer Engineering - Ingegneria Informatica

Author: Nicolò Scipione

Advisor: Prof. Gianluca Palermo

Co-advisor: Emanuele Vitali PhD

Academic year: 2021-2022

1. Introduction

The first four decades of computer architectures innovation were ruled by the well-known *"Moore's Law"*, but in the last twenty years, the advancement promised started to decay. New methods and architectures were designed to keep pushing the evolution and increase the computing power. This evolution gave birth to *heterogeneous systems*, which are complex systems that combine different types of architectures to get better performance. The most common combination that presents the best improvement in several recent types of workloads is composed of GPUs and CPUs. Very powerful and capable machines have been built around this type of system and are now used in many different fields, e.g., fluid dynamics, molecular docking, and AI/ML. These days the main vendor of GPUs is NVIDIA, and 7 out of 10 *supercomputer* inside the top 10 of the top500 [1] ranking are equipped with this type of hardware. The programming language used to better exploit this hardware is the NVIDIA proprietary language and toolkit CUDA. Soon, two other big players of computer hardware will supply GPUs to *supercomputers*. In particular, **AMD** already has three big contracts: *LUMI*, in Europe, *Frontier* and *El Capitain* in the USA. Likewise, **Intel** has already one contract to provide GPUs still in the USA for *Aurora*, that is expected to be the first exascale machine.

In this area, the emerging problem is that CUDA is proprietary and runs only on NVIDIA's platform. It means that all the available applications need to be rewritten or ported to some different framework or language. Nowadays, the most promising framework is called **SYCL**, an open standard C++ framework that promises to write code for any type of accelerator from any vendor, e.g., NVIDIA, AMD, and INTEL.

My work is to assess if SYCL can provide portable, easy to program, and performant code from CUDA. To evaluate my hypothesis, I considered a *molecular docking* application developed here at Politecnico di Milano and originally written in CUDA. A critical characteristic of the application is its high throughput thus any inefficiency is easy to spot and verify what causes it. Moreover, it has a sufficient number of different kernels in size and specific operations that let me provide insights into the conversion and test the performance achieved on **NVIDIA** and

[1] https://www.top500.org/lists/top500/2021/11/

AMD GPUs in a meaningful way.
This paper is divided into the following sections. Section 2 provides a rapid overview of the current state of the art of programming language in this field and SYCL related performance. Section 3 contains the steps that are necessary when considering porting a pre-existed application or writing a new one and the actual main point of the porting. Section 4 illustrates the experiment set up and results obtained. Section 5 shows my conclusion on the work done and future development.

2. State of The Art

Currently, there are multiple languages and standards used in the HPC world. All of them aim to obtain performance, but some also try to bring portability across hardware of different types and different vendors with various shortcomings. The most used language is CUDA because, even if it is not portable to other vendors' hardware, it is the one that brings the best performance on GPUs thanks to its special connection between hardware and software where the latter mapped perfectly the architecture. Other examples are *OpenCL* and *OpenACC*. For the latter, the most important shortcomings are that there are not many compilers available, and it is not flexible. In essence, directive-based programming languages do not allow a fine-grain tuning of the code because they do not expose low-level constructs to the programmer, and everything is in charge of the compiler. Moreover, the standard has been around for about ten years is not well supported yet. Almost every OpenACC compiler works on NVIDIA's hardware, while those for AMD GPU are still in an early stage. So from this point of view, it has the same problem as CUDA without all the benefits. On the other hand, OpenCL is a widespread open standard, but its syntax is very verbose and complex. It suffers from fragmentations from its available versions. Moreover, its implementation is based on old premises that do not stand very well today, e.g., it derives its syntax from C99. To conclude, it is a very portable language even if not always performances are equal to the CUDA ones [].
As the last open-standard born, SYCL has many advantages. It follows the same standard of coding as the iso-C++, which is a great advantage on portability and flexibility because developers don't have to learn a new language since CUDA is based on C/C++. Moreover, SYCL wants to allow researchers to obtain the best performance possible by exploiting any type of hardware, and it is already at the same level as CUDA in some cases [1] [2].
Performance-wise, there are not many complete studies that involve SYCL. To give some ideas of the level achieved so far and how it compares to the other frameworks, I refer to two different papers. The first [2] shows how SYCL, specifically *hipSYCL*, compares to CUDA and OpenACC and in [1] authors measure SYCL against OpenCL and native language (CUDA and HIP). Both the measurements of the above works are taken considering synthetic benchmarks, rarely with bigger applications, and in these tests, SYCL is in general equal to the other languages.
In this scenario, SYCL represents a promising framework, but its adoption is still limited for different reasons. First, it has different implementations all based on the newest SYCL2020 but with diverse toolchains and workflows making harder for the final user to choose. Second, the hardware support is fragmented among the implementations, with different level of maturity. All of them are still work in progress. The last reason is that tests about performances achieved in big real-world applications are still missing, so researchers cannot decide whether using SYCL for their research is worthy or not.

3. Contribution

Evaluating the feasibility of porting legacy code and doing it in the best possible way is not an easy task. The structure of SYCL allows researchers to make many decisions on how to structure the work and how to exploit it to have the best outcome, not only when starting from scratch but also in porting code from CUDA. This section elaborates the possible decisions and how I selected the ones used, it also gives some technical details on the most tricky part of the porting on NVIDIA and AMD GPUs.

General Consideration on Conversion

The main considerations before doing the port regard how to perform the conversion of the code, how to handle memory, and portability on

different architectures or highest performance. All of them are tight to each others.
The first decision is due to the fact that different SYCL implementations offer diverse approaches to the framework. In particular *Intel* inside its *OneAPI toolkit* offers an automatic tool that promises to convert the original CUDA code into SYCL. This proprietary tool uses specific headers that wrap functions instead of the standard one. It would make it more difficult to test other compilers, and it isn't very effective in the most complex part of the code to convert. Other implementations do not have a comparable feature. Luckily, all the companies and the university involved released complete documentation and wikis to help during the porting by hand. I decided to follow the last strategy because I want to test the code with all the compilers available, and I want to consider every detail and how to tune the code for SYCL.
The second general aspect to observe is how to handle memories of host and accelerators: *Unified shared memories*(usm) or *buffers and accessors*. The latter follows the general guidelines of modern C++ and represents the first implementation idea of SYCL. It is built upon the RAII(resource allocation is initialization) design pattern and lets the compiler handles when allocate/deallocate memory, and when transferring data from CPUs to GPUs. Moreover, it enables the compiler to create a task graph that can be useful to increase parallelism. On the other hand, *usm* is the standard way to handle memory in C++ and in CUDA, where developers have control over where is the data, how to move data and how much memory allocate. It is more accurate, but it could lead to errors more easily. Despite compilers allow to mix them I did not do it for code consistency. In my particular case, there are some specific situations that led to my choice. The code was already written using pointers, so I could convert them without having problems with memory allocation. Moreover, I have better handling of how and when allocating memory and how to align memory. E.g., in the application, the constructor allocates the memory granting that the code does not have to wait for allocation and initialization, which provides better performance to all the applications. This last example is particularly hard to code using *buffers and accessors*. So, even if the standard considers *buffers/accessors* to be safer I decided to use *Unified Shared Memory*.
The last decision is whether focus the effort on portability on different types of accelerators, e.g., FPGA and multicores on CPU, or focus on maximum performance achievable on GPUs. This decision is specific to SYCL because it allows writing code once and then compile it for the desired platform. The application considered is highly specific for GPUs, and my goal is to verify if SYCL can provide performance as good as CUDA. So I opted to use as many GPUs-specific constructs to obtain maximum performance.

Targetting NVIDIA GPUs

The conversion of kernels is straightforward in the part that concerns GPUs hardware identification as the running threads, block, and grid dimensions. It gets more complicated when dealing with specific GPU coding constructs, such as warp and reduction. In many of the application's kernels, e.g. *align_kernel* and *optmize_kernel*, arises different problems that can be sorted in four categories:

- Differences in structures of the framework. E.g., *shared memory* is handled differently. In SYCL, it is necessary to use special accessors, while in CUDA, it is a kernel parameter, and they compute differently also the size.
- Missing libraries. CUDA offers libraries such *cooperative_groups* that allow for finer control on the executions threads. In some kernels, this feature is used to compute one last reduction, and in SYCL are necessary some *tricks* to cope with this missing. Unfortunately, it slows down the computation.
- Features not yet implemented. SYCL is still in an early stage of development, so some features are included in the standard but not yet implemented in every compiler. E.g, the *CUDA texture memory* is foresee by SYCL standard and it is called *image*. Until now, compilers haven't got it yet for 3D images.
- Features that do not belong to the standard. CUDA and SYCL have different philosophies and there are many years of development between the two. This situation implies that CUDA has some features that

SYCL does not want to implement or be implemented, e.g., *dynamic programming*. When one of these features is present in the original code, there is the necessity to redesign before translating. This is happened in the last kernel of the *molecular docking* application. The only way possible to obtain the current results was to change it.

So far, the porting experience refers only to work needed to have a SYCL application running on NVIDIA's hardware and getting the correct results. To get the best performance out of it, I coded in SYCL trying to stay as close to the hardware as possible.

Targetting AMD GPUs

The first part of the conversion was done only with NVIDIA GPUs in mind because that was the platform available, and the native code is highly optimized for it. When the code was tested on AMD GPUs it did not work for endogenous and exogenous reasons. The former arose since the same type of accelerator does not imply the same architecture. AMD and NVIDIA GPUs follow the same design logic but are different in one key characteristic: the warp size(wavefront size in AMD terminology). Exploiting it guarantees better performance and less wasted resources that are particularly important in the HPC environment. At a practical level, this means that in some kernels I had to cope with this difference using various workarounds, e.g., introducing branches to exploit the correct size of the warp in reduce functions or adding a loop while changing the block size to compute the correct results. These classes of hurdles can be spotted and resolved in different ways but they highlight the importance of knowing the platform very well when trying to get maximum performance. Whereas the immaturity of compilers caused the other issue because available compilers have shortcomings using this platform. When the application was tested for the first time, DPC++and hipSYCL missed some low-level functions needed to get maximum performance from the platform.

The early stage of development of SYCL compilers and the nature of the porting cause these issues and in part the solution. I decided not to change too much the implementation of the kernels affected by the warp size problem to stay as close as possible to the original code, so I used some tricks to cope with it. The missing functions were added during my work on the AMD machine showing that are still in rapid development. On the brighter side, almost every kernel worked perfectly fine by simply letting the application tune itself and decide the size of block and warp, in this case running the application on different hardware is flawless.

4. Performance

The second part of my work is to verify if the performances obtained by the ported code are better, worst, or comparable with those achieved by the original one. I conducted the test on two different machines, one provided by Politecnico di Milano equipped with two NVIDIA A100 GPUs and the other one from CINECA that had four AMD MI100 GPUs. The number of GPUs does not have any repercussion since I set the code to use only one GPU at a time. I decided to test SYCL with both the compilers available *DPC++* and *hipSYCL*.

I conducted the tests by taking ten measurements of the kernels' performances with the tools provided by the hardware manufacturers in both cases. Here, I present only a selection of the most meaningful numbers. The molecule given as input to the application is always the same, and it is a specific one that ensures to stress the application and makes the measures meaningful. The application was tested with both single-precision and double-precision to understand if there are differences in SYCL code generation. For legacy reasons, the application needs to produce the results with double-precision computation. For this reason, and due to the limitation of space of this work, here I refer only to that.

The performances results are divided into two groups using the same logic on both machines. The first group represents kernels in which the performance results of SYCL are close to CUDA, while the second group is composed of those kernels that cause some problems.

Targetting NVIDIA GPUs

Figure 1 presents three different kernels. The two on the right are very simple ones, they are executed many times in the application which is why are presented here and represent all the other small kernels. The performances of

SYCL are very good compared to CUDA with all the compiler tested. For the kernel on the left, the situation is a little bit different, actually, *hipSYCL* is very close to native performance, while *DPC++* is quite far. The explanation for such difference is that compiling SYCL with DPC++ using double-precision cause an overspilling of registers from one kernel, which crashes the application at runtime. To overcome this issue, I have to limit the number of registers available for each block to 64. This imposed hard limitation blocks the performance of *ligand_is_bumping*. In fact, there is not much difference from CUDA performance using single-precision. Moreover, using NVIDIA Nsight Compute is possible to see that it would use 72 registers instead of 64, which causes the loss in performance.

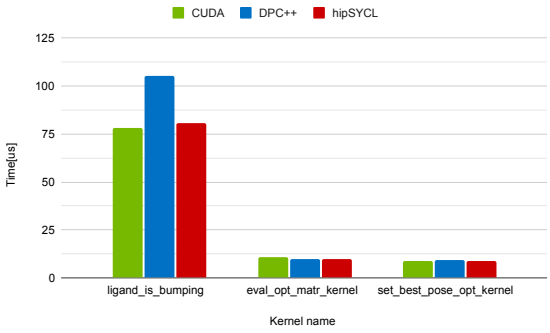

Figure 1: Performance comparison SYCL vs CUDA on three different kernels

Figure 2 presents a different situation. These kernels are the biggest and the most complex of the application, porting them required changing the code to cope with stronger limitations and differences from CUDA and sometimes make work-around to achieve the goal. The kernel in the middle shows promising results both SYCL versions are close to CUDA and *DPC++* performs even better. This is important due to the structure of the code. It is a long piece of code that makes functions call to other complex sorting algorithms, so it shows how SYCL is capable of delivering great performance in this scenario. The other two kernels tell a different story in both cases hipSYCL is too far from CUDA and also from DPC++, which means that for hipSYCL there are large margins of improvement. For DPC++the situation is slightly different, it takes longer than CUDA, but the results are encouraging. The real problem is the hard limit on the number of registers because also the "optimize kernel" would use more registers and it would lead to better performance.

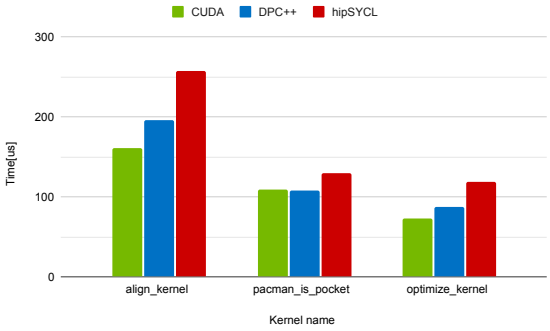

Figure 2: Performance comparison SYCL vs CUDA on three different big kernels

The overall SYCL slowdown using DPC++is about 15% and with hipSYCL is 48%.

Targetting AMD GPUs

To understand if code portability corresponds to performance portability, the runtime on AMD hardware is compared with the original times obtained with CUDA because it doesn't exist a native version of the code for AMD, as in the majority of HPC applications. The possibility of having the same application run on different hardware with few or no modifications is intriguing. Figure 3 provides encouraging results in favor of SYCL over CUDA. The three examples displayed on AMD require only to auto-tune the block sizes, to exploit better the hardware, and the results are close or even better than those of the native code. It shows that SYCL is a very good framework, capable of handling very well simple and complex code in both its implementations, even if support for AMD hardware is very early stage.
On the other hand, plot 4 presents a different perspective. The first two kernels are those that required a lot of effort and code changing to make them work effectively on AMD. The outcomes of these changes are not the expected performance-wise, in fact, the loss is too large. Looking only to *DPC++* bars, the loss compared to CUDA is huge and it makes the difference from results presented in Figures 1 and 2. Probably the numbers of branches added to cope with the new hardware and the early stage

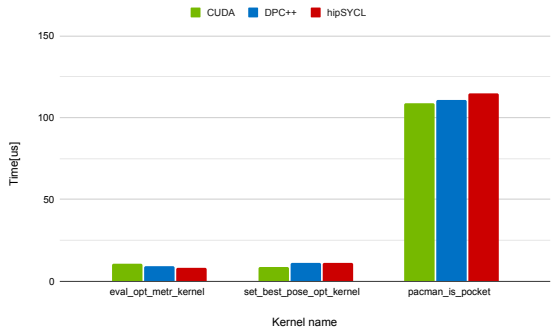

Figure 3: CUDA on A100 vs SYCL on AMD MI100 on three kernels with similar results using auto-tuning

of the platform support are the cause of these times. Looking at *hipSYCL* the situation is a little bit better, in *align_kernel* there is a huge difference from CUDA but in the other two, the distance is less, particularly *ligand_is_bumping* is very close to the native platform implementation. The difference detected in the *optimize kernel* is curious since the kernel was tweaked to make it work on AMD by adding a loop. This change guarantees the correct result, and it has a great effect on the performance if the compiler used is *hipSYCL*, while the performance using *DPC++* is way worst.

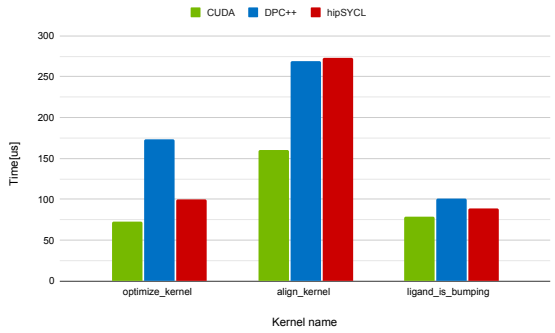

Figure 4: CUDA on A100 vs SYCL on AMD MI100 on three kernels where it struggles the most

The overall slowdown of SYCL on AMD compared to legacy code is of 100% using DPC++and 30% using hipSYCL.

5. Conclusion

The work reported in the thesis and the data presented indicate that SYCL reached a great level. All the following considerations must take into account that it is still in an early stage of development. Nonetheless, it is impressive. For what concerns portability is important to distinguish between two cases. SYCL indeed allows the same code to run on different types of accelerators and architectures thanks to its different implementations. It is possible to write code that runs on different architectures without modification. On the other hand, to obtain the best performance possible and to have comparable results with native code, it is necessary to tune the applications, so researchers still have to understand the underlining platform and how to code for it. Converting an existing application to SYCL across different architectures of the same accelerators is already possible. Limitations of the framework can be overcome by rewriting algorithms, or in many cases, its evolution will provide the missing functions. Performance-wise section 4 shows that SYCL is similar to CUDA performance in almost any case, especially if considering DPC++as compiler. Moreover, implementations are improving continuously, so it could close the gap very soon. On AMD's platform, the performances obtained are encouraging even though the distance in those two kernels is too much to be considered acceptable but the evolution of the compilers and tweaking algorithms to align them to the underline hardware will make it as fast as on NVIDIA, as it is possible to observe on all the other kernels.
To conclude, SYCL reduces a lot the difficulty of writing portable and performance-oriented code if targeting a single category of accelerators. In any case, performance doesn't come for free and it still requires that engineers know the underline architecture to obtain the best from it.
All the results presented in this work will be presented at *IWOCL&SYCLCon2022*

References

[1] Tom Deakin and Simon McIntosh-Smith. Evaluating the performance of hpc-style sycl applications. IWOCL '20, New York, NY, USA, 2020. Association for Computing Machinery.

[2] István Z. Reguly. Performance portability of multi-material kernels. In *2019 P3HPC*, pages 26–35, 2019.