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

Tesi di Laurea Magistrale in
Computer Engineering - Ingegneria Informatica

Author: **Nicolò Scipione**

Student ID: 920281
Advisor: Prof. Gianluca Palermo
Co-advisor: Emanuele Vitali PhD
Academic Year: 2021-22

Abstract

My thesis aims to verify and assess the portability across hardware vendors and performance achievable on GPGPUs computing using the new C++ framework SYCL. To do it, I ported a complete high-throughput *molecular docking* application from CUDA code to SYCL, converting every kernel of the application to it and being sure that the new code produces the same results as the original one. Moreover, SYCL code has to be as similar to the native one as possible. During the conversion, I make analyses of the possibility available and the limitation that a non-proprietary framework has against native code. The performances were tested on NVIDIA and AMD hardware, comparing results to the original CUDA ones. In addition to it, I analyze code differences from a performance perspective. The code is compiled with *DPC++* and *hipSYCL* because both compilers work with the available GPUs, and it allows analyzing better the current situation of SYCL as a standard. The results of the studies show how it is possible to convert all the existing code from CUDA to SYCL and only in a few cases use some workaround to convey the current limitation of the standard. It also demonstrates that thanks to its rapid evolution SYCL is catching up quickly with proprietary code and it enables researchers to bring their applications to new hardware. The outcome obtained by SYCL code performance-wise is very encouraging for the future because, as I demonstrate, are already comparable with CUDA ones. Concluding that thanks to the current performance level reached SYCL, as a standard, is ready to be considered usable in a performance-oriented environment. Moreover, the great flexibility and portability of the code across vendors, without the necessity to learn a new language, make it a very intriguing framework for the supercomputers of the near future.

Keywords: SYCL, Heterogeneous Systems, GPGPUs, DPC++, HPC, CUDA, hipSYCL, molecular docking

Abstract in lingua italiana

L'obiettivo della mia tesi è verificare e convalidare la portabilità e le performance che si possono ottenere in ambito GPGPU computing utilizzando SYCL, un nuovo standard in C++ per programmare Heterogeneous Processors. Per raggiungere il mio obbiettivo, sono partito da un'applicazione di docking molecolare, scritta in CUDA e sviluppata al Politecnico di Milano. Ho convertito tutto il codice in SYCL, assicurandomi che il nuovo codice producesse gli stessi risultati dell'originale e che fosse il più simile possibile a quest'ultimo. Durante il porting ho analizzato le diverse possibilità e limitazione che un framework non proprietario, SYCL, ha in confronto a del codice nativo per un hardware specifico come CUDA. Le performance sono state testate su GPU NVIDIA e AMD, comparando i risultati con quelli ottenuti da CUDA, in quanto è lo standard per l'applicazione. Inoltre ho analizzato le differenze nelle performance ottenute, andando a ricercare cosa nel codice potrebbe causarle. Il codice SYCL è compilato utilizzando entrambi i compilatori disponibili che permettono di eseguire il codice sull'hardware prescelto, ovvero *DPC++* e *hipSYCL*. In questo modo, ho la possibilità di capire meglio la situazione attuale e le possibilità di SYCL senza che sia vincolato dallo sviluppo di un singolo compilatore. I risultati dello studio mostrano come sia possibile convertire tutto il codice preesistente da CUDA a SYCL, in modo che solo in pochi casi sia necessario trovare altre strategie per ovviare le limitazioni correnti del nuovo standard. Anche che grazie alla sia rapida evoluzione SYCL sta recuperando velocemente la distanza dagli standard proprietari ed inoltre permette di eseguire il codice su nuovo hardware dove prima non era possibile. I risultati delle performance ottenuti con SYCL sono molto incoraggianti per il futuro, perchè, come ho dimostrato, sono già comparabili a quelli di CUDA. Concludo che grazie alle performance attuali raggiunte SYCL come standard, è pronto per essere considerato utilizzabile in ambienti dove la performance è fondamentale. Inoltre la grande flessibilità e portabilità del codice su hardware di diversi provider, lo rendono un framework molto interessante per i supercomputer del prossimo futuro.

Parole chiave: SYCL, Heterogenous Systems, GPGPU, DPC++, HPC, CUDA, hip-SYLC, docking molecolare

Contents

Introduction

For almost four decades in computer science, there was the belief that about every two years the number of transistors in an Integrated Circuit(IC) would have doubled. This idea is known as Moore's law. This idea led computer engineering development, but in the last twenty years it has been proved to be wrong. Better, the technology reached its maturity, and the physical improvement is not that high anymore. The physical reasons for this outcome are multiple. First of all, the size of the transistors is still shrinking but not that fast, and every year it's closer to the physical limit for the current needed for its function. It is impossible to power all the area and the transistors available in the IC, which leads to having some parts of the chip turned on and parts turned off. The necessity of keeping some part of the chip turned off is driven by the power consumption and by thermal design, which implies that the temperature reached by the IC must be below a given threshold. This phenomenon is called *dark silicon*.

In these same decades, as a society, our needs for more powerful PCs and CPUs have been kept increasing, so engineers came up with different ideas to cope with it. The first idea to mitigate the *dark silicon* effect is to add multiple cores of the same design in the same chip. The different cores communicate through shared memory, but are capable of handling different loads, distributing temperature and powers, so it's not needed to have all the silicon powered all the time altogether. Giving birth to physical parallelism.
The other way is provided by Heterogeneous Systems that are combinations of different type of components in the same system. The different parts can be on the same die, as in the case of *System On a Chip*(SoC) or can communicate through different buses on a bigger system. Examples of the most famous Heterogeneous Systems are the SoC present in every phone and recently computers, thanks to ARM, and the most common combo of CPU and GPU in personal computers.
The implementation of Heterogeneous Systems does not only solve the problem of increasing the performance for simple tasks but allows scientists to use very powerful systems to perform the computation for their research, as it was done in the origin of the computer era. The necessity to perform more difficult research and forecast always require more powerful technology.

Nowadays, the ability to build and exploit big and complex computing systems has a fundamental role for every country to push their research, these kinds of machines are called *supercomputers*. To have a better understanding of the current situation about it, from 1993, a project called **TOP500** ranks them in order of computational power. Reading these ranking it's important to notice that now all the most powerful systems in the world implement some form of Heterogeneous System, from the *ARM* architecture of the first one to other seven systems, out of the first ten, that use a lot of GPUs.[1]

The development of the architectures and hardware for *supercomputers* leads the development of hardware, and there's a huge competition between companies to provide the best possible solution. Shortly, there will be many different supercomputers that utilize GPUs from different vendors, besides all those that are equipped with NVIDIA hardware that can be viewed in the top500 list. By the end of the year, there will be new supercomputers like LUMI[2], a machine located in Finland that use AMD MI250X, and in the U.S.A. there will be *Frontier*, that also has AMD MI250X GPUs and *El Capitan* that will have the next AMD GPUs [3]. Moreover, also *Intel* will start to produce its discrete graphics card, and there is a supercomputer ready to exploit their hardware named *Aurora*[4] still based in the U.S. So in the future, there will be a wide diversification in hardware that lead to new languages and newer necessities.

The importance and the influence of GPUs come from their architecture differences from CPUs. It is way easier than the former and allows to put more cores on a small dice. This possibility does not come for free, in fact, many complex operations are way slower on them than on CPU, but the presence of more cores enables different algorithms and massive parallelization speed up some classes of computation, especially in those fields of research that need to analyze or to use a large amount of data. The implementation of these algorithms, the discovery of new fields where it is possible to apply this knowledge and hardware, are grouped in the field of *High-Performance Computing*(HPC). Examples of areas of science that benefits a lot from supercomputer and GPUs computations are many. The most known nowadays is for sure Machine Learning and Artificial intelligence, where the ability to process a lot of data in a very short time is essential. Other examples are more niche and close to the research environment. For example, *Computational Fluid Dynamics* a branch of fluid mechanics that merges numerical calculus and data structure to analyze how fluid flows, where the need to process complex problems exploits very well the ability to split the problem into many parts and analyze them. The last example is the field of drug discovery and molecular simulation because supercomputers and parallelization enable the possibility to test many different combinations of chemical elements.

The arrival of new competitors in the GPU field brings another problem to the industry, in fact, now every company that produces hardware releases also the toolkit/language to exploit it at its best. The standard for all the supercomputers is *CUDA toolkit* by NVIDIA because it was the only hardware manufacturer. Anyway, today there are different alternative programming languages to exploit all the GPUs available. OpenCL wanted to be the portable version of CUDA, following almost the same philosophy, and other standards chose to create directive-based languages such as OpenACC. The newest and most promising language emerging is SYCL, which is a C++ framework that allows bringing modern standard C++ code to hardware accelerators independently from the hardware on which it will run on, enabling the same application to exploit all the future available GPUs.

In this work, I want to analyze the current state of SYCL in all possible forms and verify its usability in the HPC environment. I want to test the portability of the code on GPUs from different vendors, taking into account not only this aspect but also the performance that this new standard can achieve compared to **NVIDIA** proprietary language. In particular, I ported a *molecular docking* application, [5] developed here at Polytechnic of Milan, from CUDA to SYCL. I analyzed the code differences in the conversion and the current state of portability to different hardware. The performance analysis compares what distinct SYCL compilers are capable of doing on the same codebase, but also they behave on different hardware and if it matches up with CUDA.

The structure of the work is composed as follows: chapter 1 is an explanation of the current state of programming language for Heterogeneous Systems, in particular in the HPC environment, so focused on GPUs implementation. The pros and cons of every model, and the state of the art of performance analysis on GPUs, which is a core aspect. Chapter 2 shows how I tackled the porting of the CUDA algorithm to ***SYCL***, all the design choices, the problems encountered, and the various performance obtained with different compilers on **NVIDIA** and **AMD**. The last chapter is an overview of future work, improvement, and tests possible on the new implementation presented and within the ***SYCL*** framework in general.

1 | State Of The Art

This chapter introduces the current state of heterogeneous programming in the HPC field with particular care of the languages available for this task. The chapter can be split into three parts, the first one introduces languages and standards that are the most commonly used or the most promising ones in the field: *CUDA, OpenACC, OpenCL and SYCL*. For each of them, it tells a few of their stories, how they work, advantages and limits of their approach and implementation. The second part makes a more in-depth comparison of the languages, giving a detailed analysis of three aspects that are the core differences and focal point of decision in favor of one of these languages. The three categories on which I focus are: programmability, how easy it is to write code in it, portability, on which hardware can the code be run and performance, which standard/language give the best performance or if the performance obtained are comparable in different tasks. To conclude it provides a brief overview of the molecular docking application used as case study, to better understand the structure and reference of the description of the work.

1.1. History of Heterogeneous System

The idea of exploiting Heterogeneous Systems to solve some intensive computing problems was born with the realization that the *Moore's Law* was slowly showing its limits. So researchers started looking for a method to bring particular computation-intensive tasks on different types of processors. Before these researches, GPUs were used only to process the graphics of the software and video games, functionality from which they take the name *Graphics Processor Unit*. At the beginning of this evolution, in the late '90s, there was no ad-hoc language to separate code compilation and execution on different types of processors, so writing this kind of code was extremely hard, but researchers noticed immediately the performance advantages of it. The first great turn in this environment is the invention of the **Brook** language by *Ian Buck* during his Ph.D. at Stanford. *Brook* is the first programming model that provides a simpler way to exploit GPUs shaders to make other types of computations[6].

After completing his Ph.D., Buck went to ***NVIDIA*** to continue his research and He created **CUDA** the ***NVIDIA*'s proprietary** language, that now is the standard for HPC computation on their GPUs. It has been having the best performances and the top computers in the top500 rank have NVIDIA's GPUs. The limits in portability do not have any effect yet and the limits of the language are overcame by developers.

This hegemony will probably change soon, since also **AMD** and **Intel**, the two NVIDIA's competitors on consumer electronics, are entering the HPC domain with their machines, powered by their CPUs and, more important, with their own GPUs. AMD already has three big contracts: *LUMI* [2], a European machine, *Frontier* and *El Capitain* in the USA [3]. Intel has already one contract to provide GPUs for *Aurora* [4] still in the USA. For these reasons, there is the need for new languages and paradigms that allows close or equal performance to CUDA, but with other characteristics.

1.2. The CUDA Programming Language

1.2.1. History

CUDA was born in 2007 starting from the idea that **Brook** introduces. With it, NVIDIA wanted to expand the possibility of programming GPUs not only for graphics and shaders but to general computing. They wanted to enable all developers to program GPUs as they did on CPUs, without caring about all the extra work of synchronizations and memory issues that were present before. NVIDIA developers' idea and goal were to have control of hardware in a specific and precise way, that **Brook**, since it was just an academic project, could not achieve [7].

To ease the spread of GPUs programming, they had many options. They could invent a new parallel language or add some extension to **OpenGL**. Since it is a company they followed customers' desire, and customers do not want to learn a new programming language or an API. Since C is the fastest language on CPUs, they decided to make the same on GPUs. Right now, CUDA supports many C++ features to make the language far more usable in a modern environment. It supports also FORTRAN, which is another language used in HPC, especially by mathematicians.

Now **CUDA** is not only a language but a complete toolkit provided by **NVIDIA**, that everyone can download from their site [8].

1.2.2. Functionality and Integration

CUDA has been thought from the ground up to be the perfect integration of software on NVIDIA's hardware. This special integration is a fundamental advantage, allowing to create software constructs ad-hoc for the hardware providing better programmability and performance. The design of CUDA and GPUs architectures lead all the other language to be as they are today. A deeper look into how it works is useful to understand further explanations.

CUDA and GPUs parallelism

The integration of the two is so tight that CUDA is an abstraction of the actual GPUs. It divide the code into *host*, which is the CPU, and *device*, which is any GPUs available selected. There is no better words to express how CUDA behaves other than those on NVIDIA's blog[9]: Code running on GPUs *[...] is executed K times in parallel by K different CUDA threads, as opposed to only one time like regular C/C++ functions. [...] A group of threads is called a CUDA block. CUDA blocks are grouped into a grid. A kernel is executed as a grid of blocks of threads. Each CUDA block is executed by one streaming multiprocessor (SM) and cannot be migrated to other SMs in GPU (except during preemption, debugging, or CUDA dynamic parallelism). One SM can run several concurrent CUDA blocks depending on the resources needed by CUDA blocks. Each kernel is executed on one device and CUDA supports running multiple kernels on a device at one time.* Following figure 1.1 is taken from the same article and shows the correlation between hardware and software.

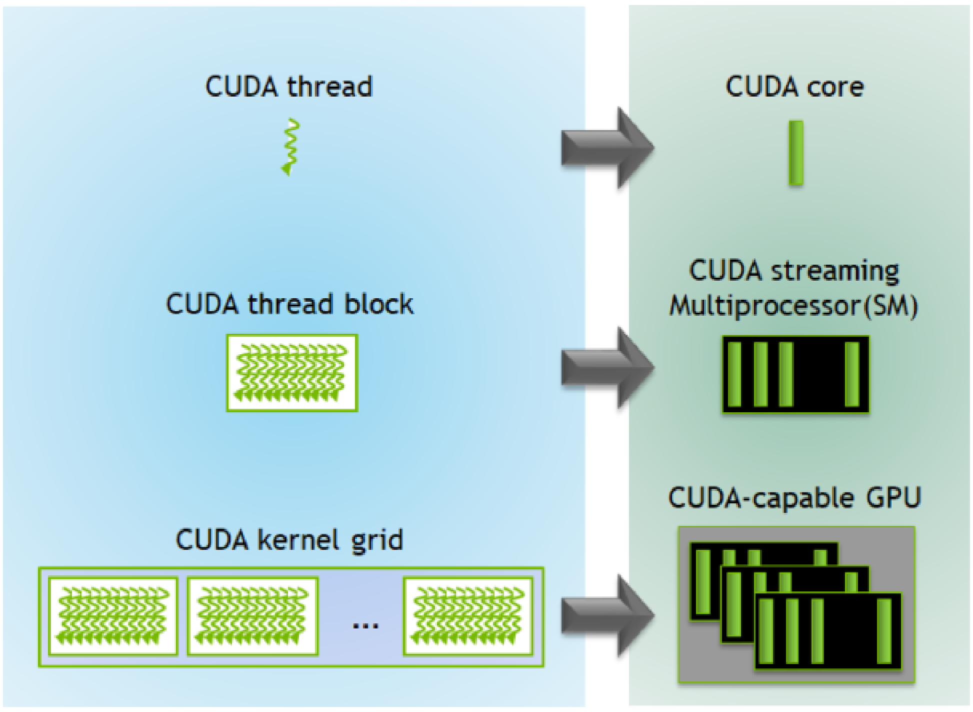


Figure 1.1: Kernel execution on GPU

Hardware

NVIDIA started from the necessity that emerged by trying to port general computation on GPUs and introduced many technologies [10]. Before 2006, GPUs have only a texture memory and each operation could read and write only one address in memory. There was no memory available to share data among works and the structure of the computation was different. NVIDIA introduced the idea of **Thread Programming**, where the developer can explicitly call a function to run on GPU and allocate several threads to make the work, so now it can run millions of instructions at the same time on different cores. With these new features, came the support to other data types and more difficult hardware implementation that allowed complex constructs such as loop and branching and the relative optimization at compile time. One of the first new technology introduced was the possibility to have a *global memory* where the program can write without being restricted to a given type or position in memory and that all the process of writing is handled with pointers. Related to memory, NVIDIA introduced also the *shared memory*, which is a particular type of memory that all the threads in the same block can use to store computation results and shared data among them.

The above hardware introductions are just a few examples of how much work NVIDIA put in the research in this field and all of them represent the first idea developed. Nowadays, NVIDIA's GPUs architecture is far more complex and every hardware news was followed by some software command to exploit it for general computation, a couple of examples are cooperative groups and thread warp.

Software

In the original plan of NVIDIA, *CUDA* should have been the natural evolution of C for GPUs. It shares many constructs of the C language to speed up the adoption of the language as a standard for GPU computing. To a C programmer, it is very easy to understand since it shares the same memory model used for C on CPUs. For example, allocations are done through `malloc`, copies between different memory addresses follow the same logic as C, using pointers to memory, and it does not provide a garbage collector. The difference between the two is that with CUDA, the developer must also take care of the memory on the accelerator and not only on the host CPU.

To keep the language simple and as close as possible to the thread model of C, the developer assigns which part of the code should run on the accelerator by declaring it in a particular function that will be called using triangular brackets after the function call. In those brackets, the developer writes code's properties, such as how to parallelize it and the local memory necessary. These functions above are called kernels, developers have in their hands how to design the code and use it. Compilers only take care of compiling the code for the correct machine, using some particular *declaration specifier* added by NVIDIA. It possible to select and address the very particular thread or block or grid that is currently executing using some constructs, e.g, `threadId.x, threadId.y or blockId.x` and different aspect of the execution with similar keywords. Nowadays, CUDA is much more similar to C++ than to C, even if the way memory is handled is still the same.

1.2.3. Advantages and Limits

This proprietary approach to heterogeneous programming has some advantages and some limits. The most important feature is that everything is designed, tuned, and optimized by the same company. This choice can guarantee a great integration between hardware and software, which leads to better performance and control over what users can implement and how everything interacts. Another good point is that there is only one source of the *CUDA Toolkit* and everything it provides will work as intended and eventual bugs are fixed by the single-source provider.

Limits of a closed source technology and language are many and are related more to the philosophy of development and what it implies. It cannot guarantee code portability to another platform, which can be a limit in the future if NVIDIA GPUs will not be top-level hardware. Integrating new functionalities in the language is harder since not anyone has access to the implementation. Another barrier is more related to the language itself. The idea of developing a *C language for GPUs* brings with it all the flaws that the original C language has, e.g. memory control and all the low-level feature that make it a little bit outdated nowadays.

1.3. openACC

OpenACC[11] is a standard designed for port code execution on different kinds of accelerators, in particular, it is used for GPGPU computation. It was designed by Cray, CAPS, NVIDIA, and PGI et al. as a solution for one of the shortcomings of CUDA. They wanted to lower the difficulty of coding for those systems to expand the possibility of porting code on heterogeneous systems. It was released as an open standard capable of being implemented by different companies, foundations, and universities. Besides NVIDIA, now different organizations are working on making OpenACC works e.g., Cray, a subsidiary of Hewlett Packard Enterprise, and the Oak Ridge National Laboratory.

The structure and the implementation of **OpenACC** are completely different from CUDA ones. It uses *compiler directives* to let the programmer point out which part of the code should run on the *accelerators*, but the let the compiler decides how to do it. OpenACC is similar to OpenMP but targetting GPUs because it came out OpenMP was only targeting multi-threading on CPUs so there was the need for something similar for GPUs.

1.3.1. Advantages and Limits

OpenACC has some technical and practical characteristics that make it unique and differentiate it from CUDA, as always there are advantages and limits.

Advantages

- **Single source code** The compiler directives are added where needed, but the total structure of the code is not changed. There is no need to write a new file as in CUDA. This aspect simplifies a lot the introduction of parallelism.
- **High level programming language**. The fact that it's a compiler directives language, abstracts a lot of technicalities and lets programmers focus only on the parallelization and data locality. It supports C++ as well as FORTRAN, which is a

very well-known language by researchers in HPC, lowering the barrier to accelerate specific applications.

- **Low learning curve**. Since it's a compiler directive approach, it doesn't require the same knowledge as CUDA does. It lowers the barrier from *common* programmers to access accelerators because it takes away many design features since it is high level. For example, differently from CUDA, using OpenACC programmers don't take care about memory allocation and where resources must be before being used by the accelerators or the CPU, compilers take care of it. It's possible to suggest where data should be to help the compiler, and it's possible to avoid useless memory copying, but it's not necessary at all. This particular aspect lowers the knowledge barrier needed to exploit GPGPUs computing.
- **Rapid Development**. All the characteristics above lead to a fast development even by an inexperienced developer.
- **Interoperability**. The code written using OpenACC can be used along with CUDA code, OpenMP, and MPI.
- **Portability**. This is the main advantage over CUDA. If exists a compiler for any given hardware the OpenACC program can run on it. This enables the possibility to exploit also GPUs from other manufacturers than NVIDIA, such as AMD. This is possible since OpenACC is an open standard, not owned by a company that decides directions and implementations, everyone can contribute to it.

Limits

The characteristics expressed above try to overcome some **CUDA** shortcomings, but it is not perfect. Every aspect described in the previous section introduces some problems that stop OpenACC from being the standard of heterogeneous high-performance computing instead of CUDA.

The portability factor, which is the most appealing, suffers from fragmentation. There are many compilers under development that support it on paper, but each of them has different features implemented. This brings difficulty to developers that need to adapt the code to the compiler. Moreover, since many companies and institutions are working on it, some of them may fail, be acquired, or stop working on it. Any of these events cause problems to the implementation and reliability of porting some code to OpenACC. The only available and well-implemented compiler, that supports OpenACC, is provided by NVIDIA, so it is useful only on their GPUs. The situation highlights another problem, OpenACC was born with the idea of being used on many different platforms, but this is not true yet since NVIDIA acquired PGI that had the leading OpenACC compiler [12]. The best hope to spread OpenACC is given by GCC, that it's implementing it, but it is coming slowly added to the stack. Right now GCC supports OpenACC, but the user must recompile it with all the necessary flags, and it uses LLVM to enable the code generation on AMD GPUs. It will take some time before seeing it available to everyone.

The ease of use for which it is designed contributes make it usable by anyone with little explanation, but the same feature that make it also less customizable and less flexible and less performance oriented. This is not a problem for simple application or to show the power of GPGPUs computing, but in order to exploit all the computer power from extremely powerful and expensive machines is not enough. In high performance environment where a lot of specific work is required and where many expert researchers are available CUDA is still preferred but in other cases OpenACC is very useful, e.g., on SUMMIT [13], an important US supercomputer, most of the code is written in OpenACC. Moreover the idea of using `pragmas` as language feature implementation seems easy at first, if the instruction to give is simple, but more complex directives that could lead to speed up in the code are not so trivial and learn them is completely different than learn a language specification of API as in CUDA or other languages.

1.4. OpenCL

OpenCL [14] is an open standard for cross-platform parallel programming of different types of accelerators, from supercomputers to FPGA. It is managed by the *Khronos Group* that decides and publishes the standard of development for the technology and it guarantees that companies' implementations of OpenCL are conformed to the standard, giving its certification.

The *OpenCL* standard emerged, when the technologies that may need a particular code implementation started to rise. At first the development was led by *Apple* in 2008 and later the *Khronos group* takes the role of coordinate the forces of many companies that use and publish their support. As the image 1.2 shows, OpenCL is used by many companies in different fields and today is a valid alternative to CUDA in GPGPU programming , even if it presents a very different approach and more important it is not the effort of a single company that controls hardware and software.

OpenCL is not bounded to a single hardware, so how it exploits parallelism is different from *CUDA* and it does not influence the hardware evolution, but the language and the feature that it offers are similar to the NVIDIA's technology and based on common request

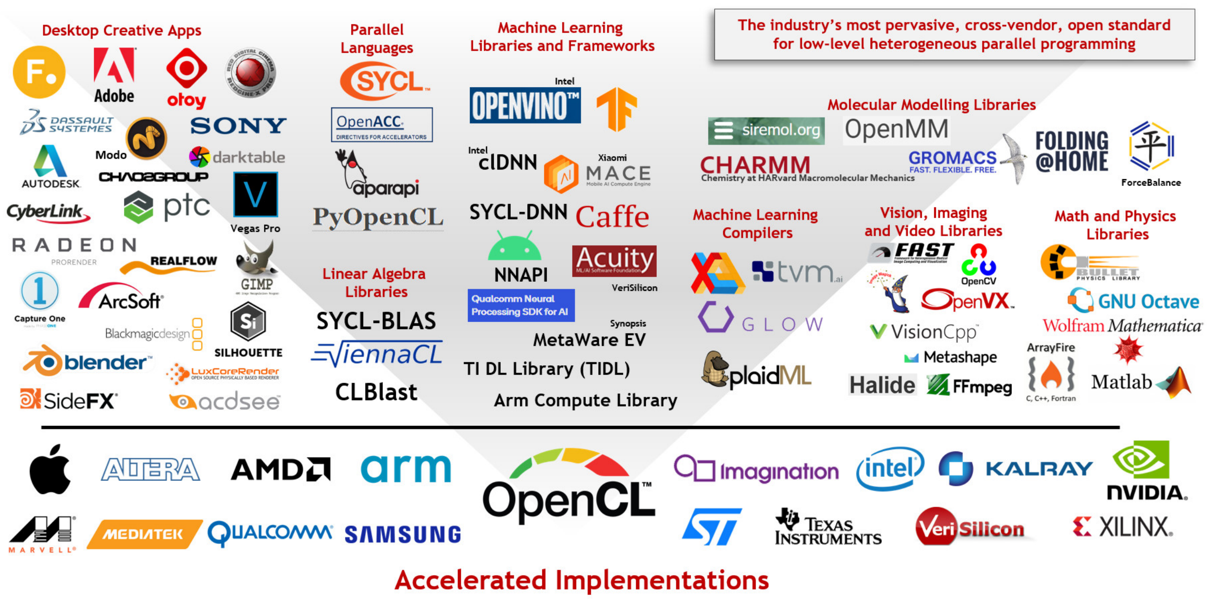


Figure 1.2: OpenCL users and major contributors

of the companies involved in the *Khronos Group* directive group.

1.4.1. How it works

It is based on C language [15], in particular on C99, that is the latest standard for the C language, but it's also a very low level language, where developers need to take care of every details by them self, and it misses higher level abstraction that other languages provides, for this reason since *OpenCL 2.0* it allows to write code in C++. In order to understand which part of the code runs on the host CPU and which on GPUs, developers have to write the code for the accelerators in particular functions, named kernels, that will be compiled and send to it. This methodology is shared with CUDA and allows a fine grained division of the workload that it is entirely managed by programmers. To accomplish it, OpenCL offers two technical API, the first one is a *platform API layer* that runs on the host CPU and it is used first to enable a program to discover what parallel processors or compute devices are available in a system, select which one to use and adapt the code for the accelerator. The second API is the *Runtime API* that enables the compilation of the kernel for the correct accelerators and handle also the communication between host and accelerator, generally provided by the company that build the hardware on which will run the OpenCL code.

A characteristic point of OpenCL is the possibility of providing the binary with two different compilations method: *online* or *offline* compilation. The latter is the most common one and newest. It allows to compile all the code in to binary format and distribute it, but since the framework has portability in mind, in order to be portable to different machines, without recompile it, in this case it has to have compiled all the architecture that developers want to support. In reality, in order to provide portability and performance, the GPU code is not compiled to binary, but is left in a intermediate representation called **SPIR-V**, which is lately set up for the appropriate accelerator. The former allows the code to be entirely portable, because it lets the kernels written in high-level code and compile them at runtime, when necessary, on the target machines, using the Runtime API. This method leaves more room for portability, without using any particular trick, but affects performance and the kernels' code is left inside the binary.

1.4.2. Advantages and Limits

The decision of the standard leads to design choices that have various impacts, some are advantages and some are limits.

The advantage that stands out for OpenCL is **portability**. The core idea of OpenCL is to be extremely portable and to guarantee good performance on a large amount of different hardware. This goal is hard to achieve, but luckily many companies took part in the standard. So effectively OpenCL can give important speed up in the computation on different kind of devices, which make it a good option for many companies when they have to ship software.

The limitations of the standard and the approach, especially in the HPC environment, are much easier to spot. The first limit is related to having multiple standards available simultaneously, standards are always available as long as someone uses them. This is not a problem per se because the more choices the better, but different standards implement different features, and if companies don't support the same version the code that can run on some devices may not work on others. For example, for years in the GPGPUs environment, NVIDIA decided to freeze its support of OpenCL on *OpenCL 1.2*, if you want to use it on NVIDIA's GPUs, the code must be compliant with the runtime version they provide. This decision stopped the evolution of it even if AMD's GPU support *OpenCL 2.0+*, because the code may lose some specification and would not be portable anymore, losing in this way the best feature of the language. Recently NVIDIA announced the support for OpenCL 3.0, which brings them in front of AMD as OpenCL support.

A feature that turned out to be a limit is the choice to use C99 as standard for the kernel language. This idea was good at the beginning because it lets developers use a language that they already know but, since the times changed, it doesn't allow the evolution of the standard to a higher-level language, like C++, which gives more flexibility to the code. C++ has been implemented since the standard 2.2, but it is not the standard for everything yet, due to fragmentation problems.

The last limitation that has been addressed lately, but still not completely solved, is the online/offline compilation. Providing online compilation is better for portability but bad for performance and for critical applications, where you cannot distribute the code inside the binary. The offline compilation addresses it partially with the creation of *SPIR-V* but it is not fully compatible with everything, e.g. it is not compatible with NVIDIA's GPUs, and besides that, it exists two versions of it that are not compatible with each other.

The idea of OpenCL was very promising, a group that managed the standard, where many companies cooperate, is the future, but poorly designed choices and adoption stopped it from achieving great results on HPC while in embedded system is less a concern.

1.5. SYCL

The newest standard that is entering the scene is managed, sponsored, and spread by the *Khronos Group* as OpenCL. It was born as its substitute but takes the heredity of OpenCL for the concept as heterogeneous language capable of running everywhere and having the same performance as native developed languages.
SYCL is a standard and is capable of producing code for every type of heterogeneous system, from the FPGA to GPGPUs computing, using the same syntax and the same compiler. As the new proposed standard is supported by different companies but, since it is only a standard, different companies developed their own compiler, while others decide to join forces and develop one compiler. Generally speaking, all compilers are based on LLVM Clang, so as host they support every feature that LLVM supports.

The current standard developed is the second iteration of it and it's called *SYCL2020*, that comes after *SYCL 1.2*. The main difference between the two and the greatest revolution is that the latter takes entirely the heredity of *OpenCL* and all the accelerators programming is based on it, so it needs *OpenCL* runtimes installed on the machine, while the newest one is entirely based on the code compiled, that is handled differently upon which is the target architecture and upon the implementation of the compiler. Since **SYCL** is just a standard it's difficult to explain how it works, due to the fact that each compiler and company decides how to implement it, the current situation (late 2021) is well represented by figure 1.3, of all those compilers in the image, only three are useful and related to the GPGPUs programming environment: ***DPC++***,developed by *Intel* as open source project, ***hipSYCL***,developed by the *University of Hamburg*, and ***ComputeCpp***,

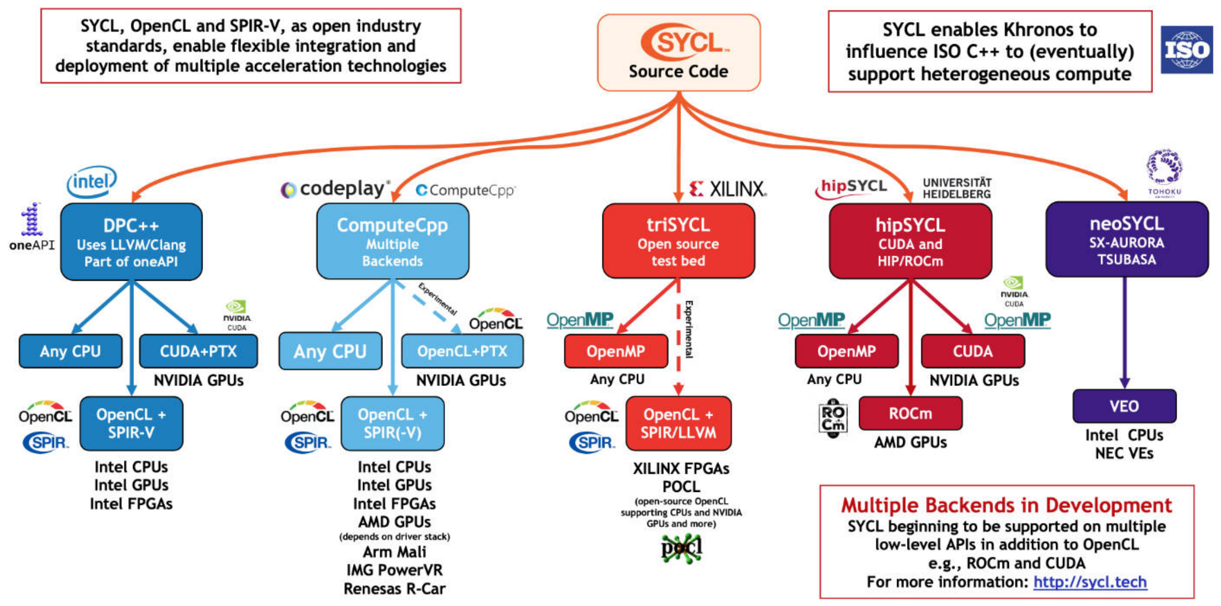

SYCL, OpenCL and SPIR-V, as open industry standards, enable flexible integration and deployment of multiple acceleration technologies
SYCL Source Code
SYCL enables Khronos to influence ISO C++ to (eventually) support heterogeneous compute
DPC++ Uses LLVM/Clang Part of oneAPI
Any CPU
CUDA+PTX NVIDIA GPUs
OpenCL + SPIR-V Intel CPUs Intel GPUs Intel FPGAs
ComputeCpp Multiple Backends
Any CPU
OpenCL+PTX NVIDIA GPUs
OpenCL + SPIR(-V) Intel CPUs Intel GPUs Intel FPGAs AMD GPUs (depends on driver stack) Arm Mali IMG PowerVR Renesas R-Car
triSYCL Open source test bed
OpenMP Any CPU
OpenCL + SPIR/LLVM XILINX FPGAs POCL (open-source OpenCL supporting CPUs and NVIDIA GPUs and more)
Experimental
hipSYCL CUDA and HIP/ROCm
OpenMP Any CPU
CUDA NVIDIA GPUs
ROCm AMD GPUs
neoSYCL SX-AURORA TSUBASA
VEO Intel CPUs NEC VEs
Multiple Backends in Development
SYCL beginning to be supported on multiple low-level APIs in addition to OpenCL e.g., ROCm and CUDA
For more information: http://sycl.tech

Figure 1.3: SYCL compilers available implementations

developed by *Codeplay*.
Referred to figure 1.3 there are some changes, *Codeplay* joined the force with *Intel*, trying to port SYCL performance on NVIDIA's GPU to the same level of CUDA, moreover *DPC++* started to support *AMD*'s GPUs using their **HIP** stack and exploiting *SPIR-V* as Intermidiate Representation. All those efforts are made in *DPC++* by **Intel**, because it is the base language for their HPC platform based on their hardware and because they aim to merge their sycl compiler in the clang standard compiler.

Giving a unique description of how these compilers work is not possible, but the general approach and idea implemented by *SYCL* are the same. It wants to reduce the barriers, from a programming language perspective, of the effort needed to write code for heterogeneous platforms and increase the portability of the code, the same two areas for which *OpenCL* was born. The main difference from OpenCL is the total independence of SYCL from companies. It does not need any extra code provided by them but it can utilize the toolkit/compiler they already supply.

The actual implementation and differentiation of the code for the accelerators are handled and highlighted by the kernel as in *CUDA* and *OpenCL*. It can be done in two different simple ways: writing directly a lambda function or writing a class functor then implementing the function. The accelerators are selected at runtime, using any of those available on the machine. Compilation and execution are independent, and the framework will operate in the best way possible on the given hardware. In any case, which GPU is connected to the machine must be known at compile-time to create the correct assembly but it is not necessary to know it when the code is written.

The code produced by a **SYCL** compiler is indistinguishable from code written and compiled by companies' products, thus proprietary performance analysis tools can be used without any effort. It is a great advantage because it gives developers a variety of tools to check the code, and it is not dependent on the standard, which would make the process of adoption slower.

1.5.1. Advantages and Limits

As for *OpenCL*, the decisions taken by the standard committee have advantages and downsides. The general idea of improving *OpenCL* producing *SYCL* reduced a lot of the errors made in that standard, giving an edge to SYCL. First of all, the choice of C++ as the default language helps a lot the adoption, because it's one of the most used languages and it is a common point with **CUDA**, so developers do not need to learn a new language or paradigm of programming. Also, C++ developers from other fields can use it freely because it follows the C++17 standards.

In many of its implementation, SYCL uses different compilers to achieve its goals. The CPU part is based on clang, while the accelerators' part is implementation-dependent. So far, all the compilers developed exploit proprietary compilers for the NVIDIA hardware, while *DPC++* and *hipSYCL* have distinct philosophies for other hardware. This combination is used to obtain a high level of performance and tool integration with the preexistence toolkit in any case. The possibility of utilizing proprietary toolkits lets developers profile the code. *Profiling* is the activity of looking at a very low level what the code is doing and how it performs in specific parts of the computation. It is crucial when trying to extract every performance possible. In this way, developers do not need to learn new tools but can use the ones they already know, reducing the learning curve and increasing the adoption.

One of the limits of *SYCL* emerged even in its early stages, is fragmentation. This idea of one standard implemented in various ways leads companies to create their product for their hardware. The risk is to have a world of *SYCL-like languages* that are similar to each other but not completely compatible. For this reason, the idea of *Intel* to upstream their implementation to LLVM should encourage other companies to collaborate to it. A limit that comes from the standard is that, as it is right now, some functionality may be in the standard but may not be implemented in the chosen compiler or for the chosen hardware. It limits choices for the developer, forcing the adoption of a more mature environment which is useless if the desired outcome is portability. Having one hardware mature and not the other would lean towards NVIDIA and consequently use CUDA as a toolkit/programming language.

1.6. State of The Art Language Comparison

This section gives a deeper look into the differences and peculiarities of the standards and language mentioned in Section 1.1. Some of the following comparisons were already done in the *Advantages and Limits* section 1.5.1 of each one of them, but now I dug further in three characteristics:*programmability, portability, and performance*

1.6.1. Programmability

Programmability may have different meanings in different fields, in this specific case, we mean the characteristics of the language that make it easier for a developer to write efficient code, enter the HPC world, and exploit the power of the hardware. Many of the languages presented in the previous sections are similar in the idea but different in the implementation. To make it easier to understand, each of the following sections presents one language with a simple vector addition as an example. That is illustrated with comments on the language involved. The section finishes an overall comparison.

CUDA

The CUDA code must be split by developers deciding what runs on GPUs and CPUs. Kernels for the accelerator are highlighted by the special attribute `global` before the function definition. They are launched by giving it some (from two to four) parameters that tune the execution and resource allocation. To allocate or initiate variables and to call functions, CUDA implements the same strategy and structures of C/C++ but to differentiate it uses the prefix `cuda` to all its specific libraries. For example, the three vectors are initialized on the GPU with `cudaMalloc` as it does with a common `malloc`. `cudaMemcpy` is used to copy data from the host to GPU. As promised, CUDA looks very similar to C/C++ code to an expert developer. Moreover, the code is very flexible and lets developers implement whatever they want and exploit hardware in any way possible.

```
// CUDA kernel. Each thread takes care of one element of c
__global__ void vecAdd(double *a, double *b, double *c, int n) {
    // Get our global thread ID
    int id = blockIdx.x*blockDim.x+threadIdx.x;
    // Make sure we do not go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
```

```
}

int main( int argc, char* argv[] ) {
    // Size of vectors
    int n = 100000;
    // Host input vectors
    double *h_a;
    double *h_b;
    //Host output vector
    double *h_c;
    // Device input vectors
    double *d_a;
    double *d_b;
    //Device output vector
    double *d_c;
    // Size, in bytes, of each vector
    size_t bytes = n*sizeof(double);
    // Allocate memory for each vector on host
    h_a = (double*)malloc(bytes);
    h_b = (double*)malloc(bytes);
    h_c = (double*)malloc(bytes);
    // Allocate memory for each vector on GPU
    cudaMalloc(&d_a, bytes);
    cudaMalloc(&d_b, bytes);
    cudaMalloc(&d_c, bytes);
    int i;
    // Initialize vectors on host
    for( i = 0; i < n; i++ ) {
        h_a[i] = sin(i)*sin(i);
        h_b[i] = cos(i)*cos(i);
    }
    // Copy host vectors to device
    cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);
    int blockSize, gridSize;
    // Number of threads in each thread block
    blockSize = 1024;
    // Number of thread blocks in grid
    gridSize = (int)ceil((float)n/blockSize);
    // Execute the kernel
    vecAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);
    // Copy array back to host
    cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost );
    // Sum up vector c and print result divided by n, this should equal
    1 within error
    double sum = 0;
    for(i=0; i<n; i++)
        sum += h_c[i];
    printf("final result: %f\n", sum/n);
    // Release device memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    // Release host memory
    free(h_a);
    free(h_b);
    free(h_c);
    return 0;
}
```

Listing 1.1: example of vector add in CUDA

The idea of programmability is different from other areas of computer science, but the usage of C/C++ common function call and the ability to easy handling memory make CUDA a good programming language for HPC. Since posting all the code may be redundant, I removed from Listing 1.1 include statements that are useless in the example.

OpenACC

OpenACC has a completely different approach to the problem. It is straightforward C++ code (in listing 1.2 is C) with the addition of compiler instruction that will be automatically identified and implemented. Those instructions are called with the `#pragma` syntax before the code that developers want to run on the accelerator. The rest of the code doesn't change in any way from C/C++.

```
int main( int argc, char* argv[] ) {
    // Size of vectors
    int n = 10000;
    // Input vectors
    double *restrict a;
    double *restrict b;
    // Output vector
    double *restrict c;
    // Size, in bytes, of each vector
    size_t bytes = n*sizeof(double);
    // Allocate memory for each vector
    a = (double*)malloc(bytes);
    b = (double*)malloc(bytes);
    c = (double*)malloc(bytes);
    // Initialize content of input vectors, vector a[i] = sin(i)^2 vector b[i] = cos(i)^2
    int i;
    for(i=0; i<n; i++) {
        a[i] = sin(i)*sin(i);
        b[i] = cos(i)*cos(i);
    }
    // sum component wise and save result into vector c
    #pragma acc kernels copyin(a[0:n],b[0:n]), copyout(c[0:n])
    for(i=0; i<n; i++) {
        c[i] = a[i] + b[i];
    }
    // Sum up vector c and print result divided by n, this should equal 1 within error
    double sum = 0.0;
    for(i=0; i<n; i++) {
        sum += c[i];
    }
    sum = sum/n;
    printf("final result: %f\n", sum);
    // Release memory
    free(a);
    free(b);
    free(c);
    return 0;
}
```

Listing 1.2: example of vector add in OpenACC

This approach seems magnificent but even this little snippet of code shows two major drawbacks. Firstly, developers need a special compiler to obtain the expected result. Utilizing different compilers may lead to different results. The second point is that the code written is not flexible to developers' needs because there is no way to express parallelism with finer control than what common C/C++ can offer. So adapting algorithms and invocations based on the hardware differences is not possible. A noticeable improvement and benefit of OpenACC are that areas of the code can be ported with almost no effort by any C/C++ programmers, without further studies or adaptation for simpler sections. Although, there may need a different approach than pragma syntax in those cases where the code to parallelize is too complicated.

OpenCL

OpenCL has a third approach to the matter, as it can be noticed by the following code 1.3, it is more verbose than others. This characteristic is due to following the C99 guidelines and the flexibility necessary to target different architectures, not only GPUs. Another big difference is in the first lines of the example. The kernel code is not written as a function, but it is written as a long string. The kernel will not be called but read as a file and then enqueued. The choice of putting kernel code as a string in the same file as the main function is arbitrary, it can be written in a separate file as OpenCL code and then read from the function. The code still present common traits to figure 1.1 and figure 1.2 related to the initialization and management of the host part, in common to the C/C++ heredity.

```
// OpenCL kernel. Each work item takes care of one element of c
const char *kernelSource =                                          "\n" \
"#pragma OPENCL EXTENSION cl_khr_fp64 : enable                       \n" \
"__kernel void vecAdd(  __global double *a,                          \n" \
"                       __global double *b,                          \n" \
"                       __global double *c,                          \n" \
"                       const unsigned int n)                       \n" \
"{                                                                  \n" \
"    //Get our global thread ID                                     \n" \
"    int id = get_global_id(0);                                     \n" \
"                                                                   \n" \
"    //Make sure we do not go out of bounds                         \n" \
"    if (id < n)                                                    \n" \
"        c[id] = a[id] + b[id];                                     \n" \
"}                                                                  \n" \
                                                                   "\n" ;
int main( int argc, char* argv[] ){
    // Length of vectors
    unsigned int n = 100000;
    // Host input vectors
    double *h_a;
    double *h_b;
    // Host output vector
    double *h_c;
    // Device input buffers
    cl_mem d_a;
    cl_mem d_b;
    // Device output buffer
    cl_mem d_c;

    cl_platform_id cpPlatform;        // OpenCL platform
    cl_device_id device_id;           // device ID
    cl_context context;               // context
    cl_command_queue queue;           // command queue
    cl_program program;               // program
    cl_kernel kernel;                 // kernel

    // Size, in bytes, of each vector
    size_t bytes = n*sizeof(double);
    // Allocate memory for each vector on host
    h_a = (double*)malloc(bytes);
    h_b = (double*)malloc(bytes);
    h_c = (double*)malloc(bytes);
    // Initialize vectors on host
    int i;
    for( i = 0; i < n; i++ ) {
        h_a[i] = sinf(i)*sinf(i);
        h_b[i] = cosf(i)*cosf(i);
    }
    size_t globalSize, localSize;
    cl_int err;
    // Number of work items in each local work group
    localSize = 64;
    // Number of total work items - localSize must be devisor
    globalSize = ceil(n/(float)localSize)*localSize;
    // Bind to platform
    err = clGetPlatformIDs(1, &cpPlatform, NULL);
    // Get ID for the device
    err = clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &device_id,
   NULL);
    // Create a context
    context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
    // Create a command queue
    queue = clCreateCommandQueue(context, device_id, 0, &err);
    // Create the compute program from the source buffer
    program = clCreateProgramWithSource(context, 1,
                            (const char **) & kernelSource, NULL, &err);
    // Build the program executable
    clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
    // Create the compute kernel in the program we wish to run
    kernel = clCreateKernel(program, "vecAdd", &err);
    // Create the input and output arrays in device memory for our
   calculation
    d_a = clCreateBuffer(context, CL_MEM_READ_ONLY, bytes, NULL, NULL);
```

```c
    d_b = clCreateBuffer(context, CL_MEM_READ_ONLY, bytes, NULL, NULL);
    d_c = clCreateBuffer(context, CL_MEM_WRITE_ONLY, bytes, NULL, NULL);
    // Write our data set into the input array in device memory
    err = clEnqueueWriteBuffer(queue, d_a, CL_TRUE, 0,
                                   bytes, h_a, 0, NULL, NULL);
    err |= clEnqueueWriteBuffer(queue, d_b, CL_TRUE, 0,
                                   bytes, h_b, 0, NULL, NULL);
    // Set the arguments to our compute kernel
    err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_a);
    err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_b);
    err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &d_c);
    err |= clSetKernelArg(kernel, 3, sizeof(unsigned int), &n);
    // Execute the kernel over the entire range of the data set
    err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &globalSize, &
localSize,
                                                              0, NULL,
NULL);
    // Wait for the command queue to get serviced before reading back
results
    clFinish(queue);
    // Read the results from the device
    clEnqueueReadBuffer(queue, d_c, CL_TRUE, 0,
                                bytes, h_c, 0, NULL, NULL );
    //Sum up vector c and print result divided by n, this should equal 1
    within error
    double sum = 0;
    for(i=0; i<n; i++)
        sum += h_c[i];
    printf("final result: %f\n", sum/n);
    // release OpenCL resources
    clReleaseMemObject(d_a);
    clReleaseMemObject(d_b);
    clReleaseMemObject(d_c);
    clReleaseProgram(program);
    clReleaseKernel(kernel);
    clReleaseCommandQueue(queue);
    clReleaseContext(context);
    //release host memory
    free(h_a);
    free(h_b);
    free(h_c);
    return 0;
}
```

Listing 1.3: example of vector add in OpenCL

All the functions specific to OpenCL are prefixed by `cl`, but differently from *CUDA*, there are many functions that are completely different from C and unknown to C/C++ developers that have never written code using OpenCL. Moreover, the verbosity is noticeable in the complete example, which has about 60 lines of code more than *CUDA* and it is the simplest possible with no particular implementation or algorithm. A reason for verbosity is also that in OpenCL programmers need to take care of the creation of the context in which the code will run, the device, all the possible errors, and kernel argument. The learning curve is much steeper than CUDA or OpenACC.

SYCL

SYCL has the same approach CUDA had 15 years ago with C, it tries to have a structure of the code as close as possible to modern C++. The idea of the *Khronos Group* is to make it as easy as possible for C++ developers to write code following the SYCL standard. Right now, SYCL2020 follows the C++17 standard, and for any C++ developer learning it is effortless.

The following code is concise and the part related to the GPU code is pure C++. It uses buffers to handle the vector's lifespan on GPU and accessors to access it. The kernel code starts at line 31 and finishes at line 36. It is very concise, easy to write, and flexible. There's no memory handling by programmers, and it reduces complexity. All the memory handling in listing 1.4 is related to the code written in C, which could be replaced with a modern container in pure modern C++ style, but I want to keep it consistent within all the examples.

```
1  int main(int argc, char* argv[]) {
2    int n = 100000;
3    // initiliaze host vectors
4    double* h_a;
5    double* h_b;
6    double* h_c;
7    size_t bytes = n*sizeof(double);
8    h_a = (double*)malloc(bytes);
9    h_b = (double*)malloc(bytes);
10   h_c = (double*)malloc(bytes);
11   for(int i = 0; i < n; ++i) {
12     h_a[i] = sin(i)*sin(i);
13     h_b[i] = cos(i)*cos(i);
14   }
15   { //open scope for buffers
```

```
  //initialize buffers from host memory
  cl::sycl::buffer<double, 1> buff_a(h_a, n);
  cl::sycl::buffer<double, 1> buff_b(h_b, n);
  cl::sycl::buffer<double, 1> buff_c(h_c, n);
  //set up range for parallel code on GPU
  cl::sycl::nd_range<1> thread_range{static_cast<int>(n*1024),1024};
  //select available GPU to run code on
  cl::sycl::queue q(cl::sycl::gpu_selector{} );
  //submit code that will run on GPU and set up memory access
  q.submit([&] (cl::sycl::handler& h){
   //set up accessor for GPU access
   auto acc_a = buff_a.get_access(h);
   auto acc_b = buff_b.get_access(h);
   auto acc_c = buff_c.get_access(h, cl::sycl::write_only,cl::sycl::no_init);
   //kernel code
   h.parallel_for(thread_range ,[=](cl::sycl::nd_item<1> it){
    int i = it.get_local_id(0)+it.get_group().get_id(0)*it.get_local_range(0);
    if(i < n)
     acc_c[i] = acc_a[i] + acc_b[i];
   });
  }).wait();
 }// close the buffer
 //free host memory
 free(h_a);
 free(h_b);
 free(h_c);
 return 0;
}
```

Listing 1.4: example of vector add in SYCL

The way listing 1.4 is written is not the only way possible in SYCL. The kernel can be in a different file or within a class. The buffer/accessor memory handling can be substituted with the use of *usm*(Unified Shared Memory) writing code even more similar to *CUDA*, even though it forces developers to renounce to some modern C++ features because it consists in managing memory using pointers as it is in CUDA. Moreover, the way the accelerator is selected in listing 1.4 targets only GPUs, but it is possible to target also FPGA and CPUs, with the same code. Given its characteristics, nowadays SYCL is probably the easiest way C++ programmers can access heterogeneous computer programming exploiting all the flexibility, customization of the code, and computing power available by using GPUs.

Final Comparison on Programmability

Overall all the solutions proposed have pros and cons and their particularities. It is evident that OpenACC is the most concise one for small and intuitive code to be parallelized, CUDA and SYCL follow the same idea of how to divide the code between platforms and how to write flexible and reusable code. SYCL is closer to standard C++ and offers higher abstraction to the programmer that may simplify and speed up the learning curve. For what concern OpenCL code(listing 1.3), it is easy to say that it seems way more verbose and less intuitive than the others. It comes from old paradigms and requires a lot of effort and attention from the programmer to be correct.

1.6.2. Portability

The second fundamental characteristic is **Portability**. With this term, I mean the capabilities/possibility of a language to target different hardware and processors. This feature is going to be a decisive characteristic since, in recent years, there has been a huge investment in high-level GPUs and accelerators by the industry. So starting from now, having the possibility to write the code ones(or with a unique language) and run it on whatever hardware will be a key feature.

The time in which a language was designed and which problems were targeted has a great effect on the current portability.
CUDA was the first language designed, furthermore, it is created and shaped by **NVIDIA** for their GPUs. It shows how they see the future of GPGPUs computing and what they can make hardware-wise, so there is no space for portability, it runs only on NVIDIA's GPUs or chip.

The first version of *OpenACC* still comes from **NVIDIA**, but with a different idea in mind. They wanted to have an easier gateway for heterogeneous programming for developers and open this standard also for other manufacturers such as AMD. But the evolution of the support for the standard is slow both for NVIDIA and AMD. Currently exist many different compilers that support it on NVIDIA GPUs, and the ROCm compiler by AMD should support it. The biggest effort to bring a common compiler for both comes from GNU/GCC even though it's evolving slowly. Right now, the version of it that supports OpenACC is not distributed but programmers should recompile GCC with the appropriate flags to have it. In any case, it has better support than CUDA for multiple hardware vendors.

The other two standards have a completely different story, starting from the oldest:

OpenCL. It was born with the idea of being the portable standard of the future, and it respected that promise. OpenCL does not only works on GPUs but with the correct runtime library, it works also on CPUs letting developers choose where to run the code and how to optimize it. The problem with it comes from fragmentation, as stated in section 1.2. Every company decides to support up to a certain version of it, if programmers are not careful and use functions not covered by all the targeted hardware, the code would not be portable anymore. An example related to HPC is NVIDIA's decision to stop support at standard 1.2 until last month while AMD supported also version 2.0+, so some functions may not work on NVIDIA, breaking applications. Recently NVIDIA declared support for OpenCL 3.0 and the situation is flipped, even though standard 3.0 is closer to 2.2, rather than 1.2, so developers should get more portability and performance from this decision. This example is only in HPC because when targetting embedded systems the situation is even worst.

SYCL is the last standard that emerged, and it is possible to consider it the evolution and best version of *OpenCL*, these considerations are evident even in portability. The portability of code written in SYCL is extremely dependent on which compiler will be used as image 1.3 shows very well. Different companies support different hardware. I will make some points on the two compilers more related to Heterogeneous computation on GPUs. The first one is *DPC++* open source project led by **Intel** and joined **Codeplay** last year. They aim to upstream their code for compiling SYCL to the original LLVM/-clang. Their last version of it, *SYCL 1.2*, was entirely based on OpenCL, so the same portability is guaranteed as long as the runtimes libraries are installed on the systems. The current version, *SYCL2020* now allows different backends to enable the same level of portability with better performance, e.g., on NVIDIA's GPUs it uses NVPTX backend, on AMD's GPUs it uses ROCm while, where the proprietary backend is not available, it still uses OpenCL. The second option in this environment is *hipSYCL* open source project, developed by Aksel Alpay and Vincent Heuveline at the University of Hamburg, which has a completely different approach. It exploits compilers from different vendors to make the code portable.

It is important to mention that the portability in SYCL doesn't stop at GPUs compilation, but it can be expanded also to FPGA, making it more future-proof. There already are other compilers that try to bring SYCL code on FPGA, like trySYCL from *Xilinx*, that have recently been thinking about joining forces with DPC++, and DPC++ itself has basic support for Intel's FPGA.

The following table sums up portability for different standards. SYCL column doesn't consider any particular compiler implementation, but the standard in general, as it is done for OpenACC.

	CUDA	OpenACC	OpenCL	SYCL
x86_64	-	-	✓	✓
ARM	-	-	✓	✓
NVIDIA's GPUs	✓	✓	✓	✓
AMD's GPUs	-	✓	✓	✓
Intel's GPUs	-	-	-	✓
FPGA	-	-	-	✓

Table 1.1: Portability of code in different standards

In the coming years, the United States will open four new supercomputers for research purposes, and three of them don't have NVIDIA's GPUs but Intel's or AMD's ones. Specifically Intel will build *Aurora* [4] and AMD will build *Frontier* and *El Capitain* [3]. This means that the importance of portability of code is a fundamental aspect for them and all world of HPC. Even European Union is starting some new supercomputers based on AMD's GPUs, like Lumi [2] which will be the European more powerful supercomputer. Given these facts, the table 1.1 shows eloquently the situation and how SYCL is supposed to be the standard of the future.

1.6.3. Performance

The last aspect to take into account is the *Performances*. This is even more important when choosing and analyzing a language/framework in the HPC world. Currently, there aren't any works that provide a complete comparison of all the considered frameworks. So I illustrate the situation by providing different views that allow me to make an extensive comparison of them. The scarcity of papers is mainly due to the very early stage of *SYCL*'s life. This problem is evident, particularly for what concerns its performance on GPUs. It is more difficult to have a complete overview since every implementation has slightly different results.

In comparing the frameworks, I start by looking at how CUDA and *OpenCL* compare to each other, then I see the two NVIDIA's standards compare to each other and for last I show the current state of the art of SYCL running on GPUs, primarily against the same implementation in CUDA, but also with comparisons with *OpenCL* and *OpenACC*.

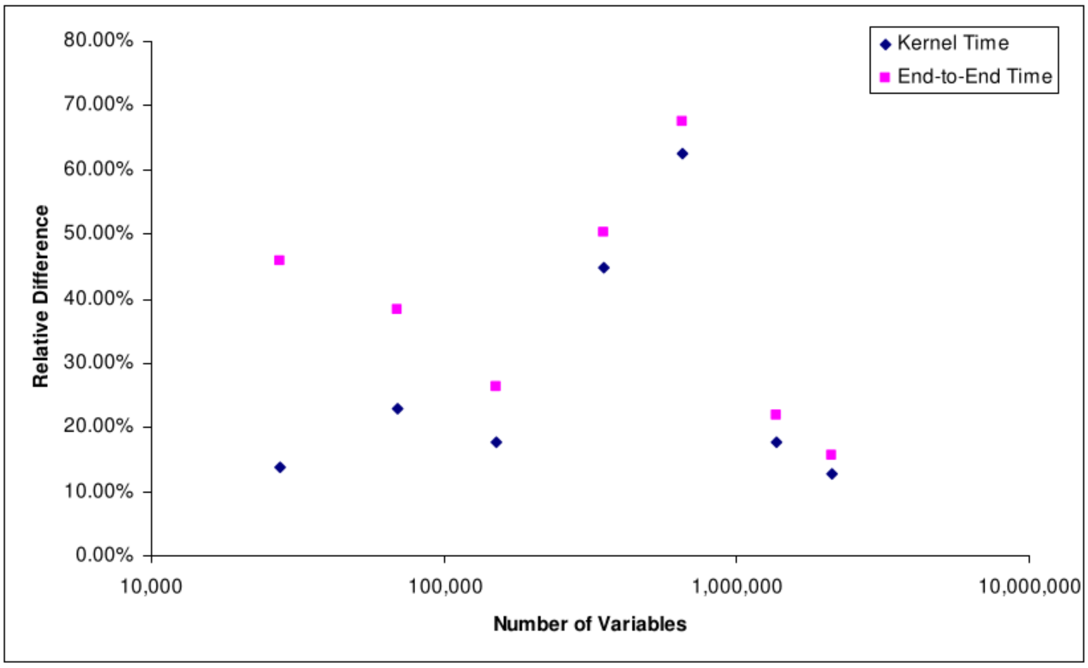


Figure 1.4: Relative difference in running time between CUDA and OpenCL, image from [16]

Starting the performance comparisons with the two great contenders of the HPC code in C/C++ derivatives, the idea is to verify if the performances achieved with *portable OpenCL code* are similar to the CUDA ones. This work was done by Kamran Karimi et al.[16] in 2010, which is quite some time ago, but it is still a valid starting point. The application analyzed is a real-world application they used the *CUDA* code available and port the rest to *OpenCL* in the following way: *"For this paper we optimized the kernel's memory access patterns. We then ported the CUDA kernel to OpenCL, a process which, with NVIDIA development tools, required minimal code changes in the kernel itself, as explained below. Other related code, for example to detect and setup the GPU or to copy data to and from the GPU, needed to be re-written for OpenCL"*. In the attempt of providing a comparison only of the GPU kernel code and avoiding differences introduced by CPU optimizations of different compilers, authors reduce CPU code execution. Even if the code running were the same in both languages, the differences between the two are noticeable(figure 1.4) as the authors acknowledge *For all problem sizes, both the kernel and the end-to-end times show considerable difference in favor of CUDA. The OpenCL kernel's performance is between about 13% and 63% slower, and the end-to-end time is between about 16% and 67% slower.*

The huge difference in terms of performances cannot be disregarded, even if for some particular problem it is not relevant and ten years passed, this kind of result is to be taken into account if decides to go with OpenCL. Luckily for OpenCL, things are getting better as it is possible to observe from the 2017 paper by Suejb Memeti et al. [17]. The work carried on is slightly different, instead of using a real-world application, researchers used a series of benchmarks from the HPC world that emulates common computing in various applications such as fluid and molecular dynamics, medical imaging, and data mining. Results vary a lot as they say: *"The performance and energy consumption behavior of OpenCL and CUDA are application dependent. For Rodinia benchmark suite, for some applications OpenCL performs better, however there are several applications where CUDA performs better."* and a few lines after *"Results show that the OpenCL implementation of seven out of 12 applications [...] perform better than their corresponding CUDA implementations."*, which is the opposite results of the previously cited works [16].

Generally speaking, the way the code is written impacts a lot of the performances, the code used in benchmarks tends to be easier to write and does not use custom operations that could give an advantage to CUDA. In the first paper [16], authors tried not to rewrite the code, but to use translating mechanism from CUDA to OpenCL, while the benchmarks in the second paper [17] are written directly for OpenCL. More often than not, benchmarks' kernels are smaller and less complicated than real-world applications. The result is that compiler optimizations and code optimizations are less effective when benchmarking them. Nonetheless, OpenCL is faster than CUDA in the last case.

In [17], authors not only compare CUDA and OpenCL, but they compare also OpenCL with *OpenACC*(Figure 1.5). The work on this part is not as complete as the other comparison and consists of only three applications. They concluded that: *"OpenCL performs better than OpenACC [...] We may observe that OpenCL implementation of LBM and MRI-Q is significantly faster than the corresponding OpenACC implementation, whereas for the implementation of Stencil the results are comparable."* So, as it seems from this work, even if *OpenCL* code is more difficult to implement, it gives better results than OpenACC and comparable results with CUDA.

Comparing performance between OpenCL and NVIDIA's frameworks is not enough, but it is also important to compare the two proprietary ideas. This work is presented by Mikhail Khalilov and Alexey Timoveev in their recent paper *Performance analysis of CUDA, OpenACC and OpenMP programming models on TESLA V100 GPU* [18]. Here they show various performances evaluations using three different benchmark applications and evaluation parameters. Starting with memory bandwidth analysis using the *Babel Stream* benchmark [19], followed by single precision matrix multiplication, and conclude with two real-world applications.

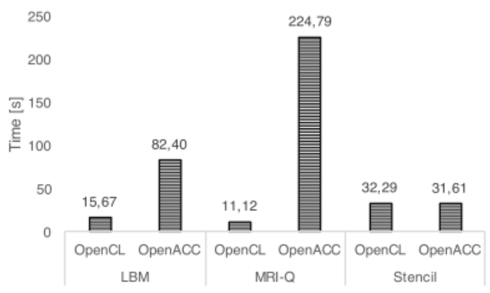
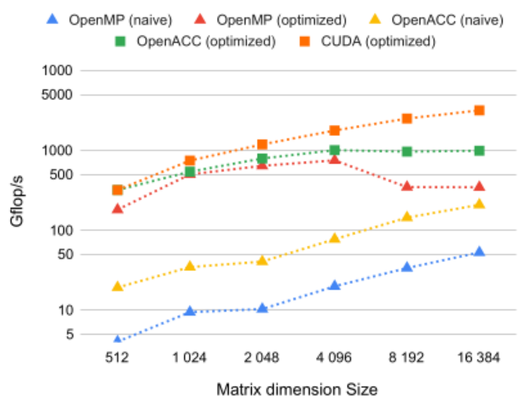


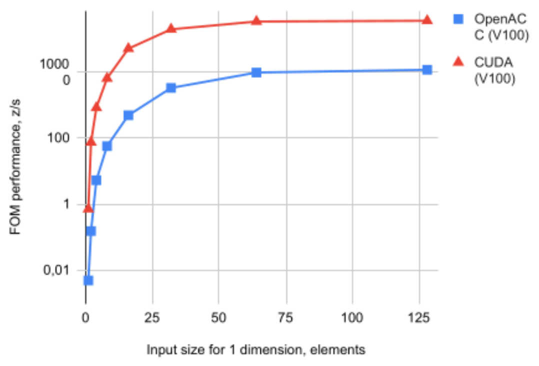
Figure 1.5: Time comparison between OpenCL and OpenACC

In the bandwidth case, OpenACC performs almost at CUDA's level having on average only 3-4% loss in performance, while for the dot-product OpenACC suffers a huge performance loss, performing at 15% of what is capable CUDA. The second test on `sgemm` (single precision general matrix multiplication) has the same result in almost any case. CUDA brings better performance in any matrix dimension tested, even against optimizes OpenACC code, which is au-pair only at the smaller matrix size. The last evaluation is the most reliable since it is done on a real-world application. Even in this case, **CUDA** brings way better performance respect **OpenACC** that doesn't seem to offer any other value on the table, besides *programmability*.

For synthesis I leave the following graphs [1.6b, 1.6a, 1.7] taken from the work cited above [18].



(a) Gflops matrix multiplication



(b) Comparison of OpenACC and CUDA performance on LULESH application

Figure 1.6: CUDA vs OpenACC GFlops operation

Right now, the only missing comparison is among well-established frameworks and the emerging standard: **SYCL**. Since the standard implementation started to emerge only

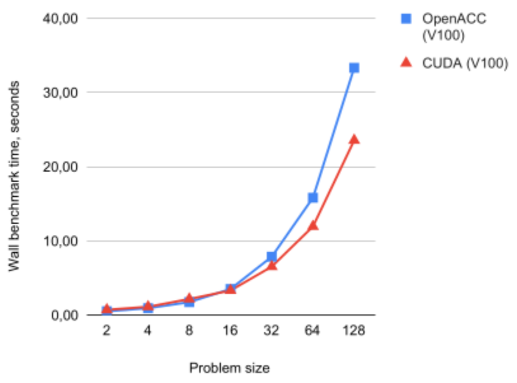


Figure 1.7: Time comparison between CUDA and OpenACC on Cloverelaf application

a couple of years ago finding articles that analyze the current situation is quite difficult for various reasons. In the first place, SYCL is continuously changing and rapidly improving due to its short life only during the last year it has been changing a lot, going from *SYCL1.2* to *SYCL2020*. Another reason is that the multiplicity of compilers and implementations make it difficult to have a complete overview. Successful and usable implementations on GPUs came up lately, so all the work is very early stage. For the sake of this work, I focus only on the latest possible implementation on GPGPUs computing, without considering all the implementation for CPUs.

A recent work by Istvan Z. Reguly [20] tries to do a full comparison of SYCL against CUDA and OpenACC on NVIDIA GPUs. The author used three different algorithms to test performance and two different sizes of data sets and evaluated performance efficiency in the three cases for a large variety of compilers and frameworks. The first two algorithms are pretty similar, while the third one is more complicated. All of them can be implemented in different ways, but for the sake of the study, the author chose the best performance for each implementation. The two data sets are made to evaluate different situations, and the second one reflects a much more complex scenario. In this paper, there are many compilers and frameworks used, and I cite only those related to this case: *CUDA NVCC, CUDA clang++, OpenACC PGI, SYCL ComputeCpp, and SYCL hipSYCL*. All the tests are executed on two GPUs an NVIDIA V100 and an NVIDIA P100.

As the data shows in 1.8, the performances in CUDA are not related to the compiler used, and in some cases, like *algorithm 1 Problem two*, the open-source compiler clang performs better than NVCC. It's possible to notice how **OpenACC** it is a valid alternative for simple algorithm and simple test cases, even though it loses a lot in terms of performance when the code gets more complicated. Very interesting are the results obtained by SYCL.

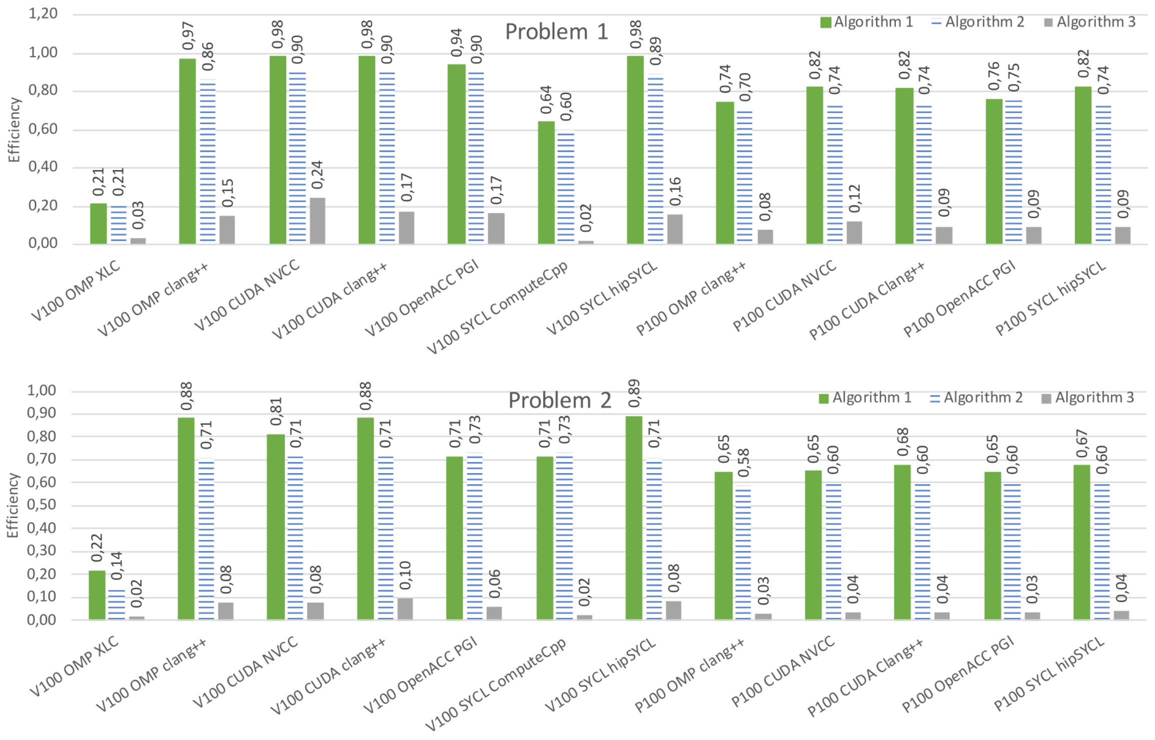


Figure 1.8: Performance efficiency CUDA, OpenACC and SYCL

The *ComputeCpp* implementation is far from being competitive using the first data set. It recovers a little bit on the second one, being closer to OpenACC, while the performances obtained by *hipSYCL* are another story. It is competitive with CUDA and clang in all the tests implemented. It is fair to say that the support of ComputeCpp for the CUDA backend was in the early stages of development, and it was recently dropped. All the effort for that joined the development of *DPC++*. Moreover, the two SYCL implementations have a completely different approach.

Given the good results achieved, it is interesting to have a better look at the performance comparison between CUDA and SYCL on GPUs. Until the last few months, the only reliable SYCL implementation for GPUs was *hipSYCL*. Brian Homerding and John Tramm provide an insightful overview of it [21]. Their work focuses on conventional benchmarks evaluating the performance in many different micro-benchmark. They state that: *"The results show that several kernels have a performance difference above 20%, while most fall within a narrow range."*. These results are interesting and show the strengths of the hipSYCL compiler. They didn't stop to that benchmark and test also with the *N-Body* simulation. Results obtained are clear (Figure 1.10a). SYCL loses the 16% percentage respect CUDA, which may not seem a lot on small kernels, but it can become a huge

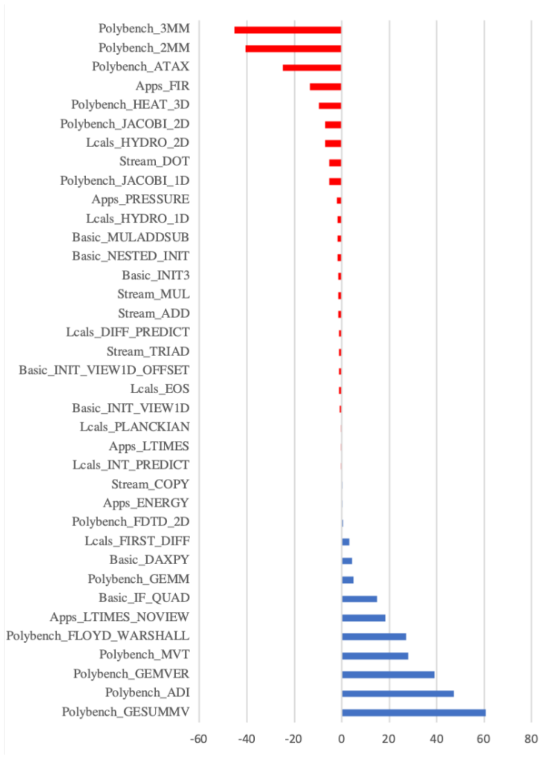
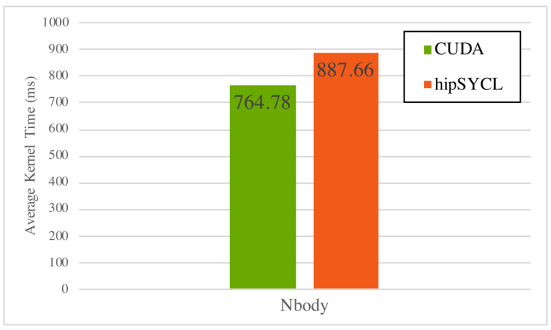


Figure 1.9: percentage speedup hipSYCL relative to CUDA

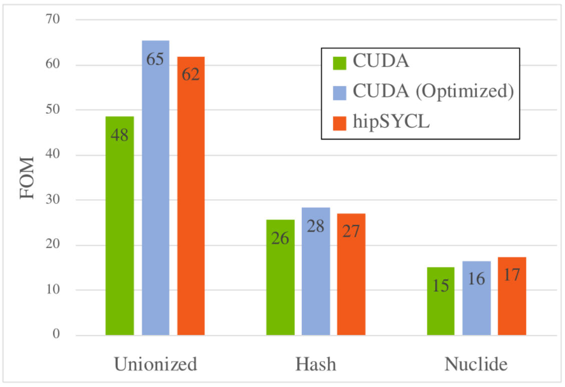
difference if the kernel runs a few times.

It is also interesting their last benchmark(figure 1.10b), which shows a difference in performance in favor of hipSYCL against a general version of CUDA. They define the code written for the last test as follows: *"general, portable optimizations in mind, but do not use programming model specific or vendor specific intrinsics, so as to make for an even playing field."* This test shows how it is possible to obtain great performance in SYCL without writing low-level specific code, and even after some code optimizations in CUDA, the performance is very close.

To conclude the overview on the current performance comparison among standards, it is necessary to look at how SYCL compares to OpenCL at the current state of implementation. It is relevant because SYCL can be considered the heir of OpenCL. Tom Deakin and Simon McIntosh-Smith [22] from Bristol University made an in-depth comparison on different machines not only against OpenCL but also against vendor-specific languages, from CPU to GPUs. In this case, I am interested only in the results on GPUs. They run benchmarks on Intel, NVIDIA, and AMD's GPUs. The state and evolution of SYCL are still not complete since the paper is from 2020, but they used all the available implementations and took into account the best result. For NVIDIA's GPUs the possibilities

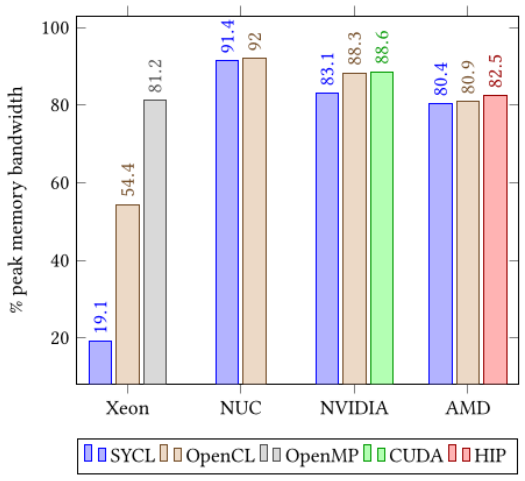


(a) Performance of N-body kernel on V100 GPU (lower is better)

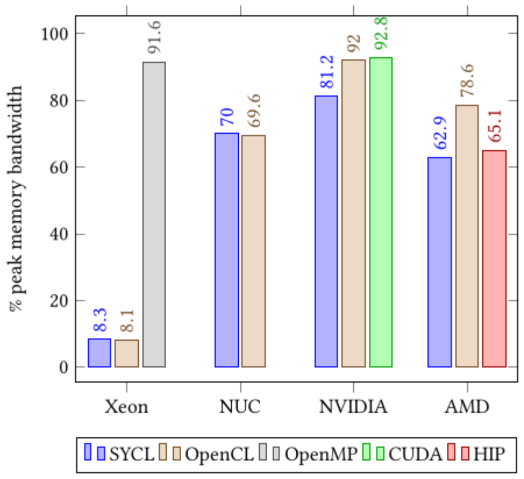


(b) hipSYCL vs CUDA non specific code performance

were *ComputeCpp* or *hipSYCL*. The Intel's NUC GPU was supported by *ComputeCpp* or *Intel/LLVM*(commonly referred to as *DPC++*). Their works compare SYCL also on Xeon CPU, but I will not consider the result for this work. They run three different benchmarks: the BabelStream, one of the most commonly used in memory bandwidth performance evaluation, Heat, and CloverLeaf. The first shows how SYCL behaves on the same hardware against OpenCL, but also against vendor-specific language. Results depend a lot on the kernel considered. For some kernels SYCL is a little further from native language respects OpenCL implementation, while for others is at the same level as vendor-specific languages and as OpenCL(Figure 1.10a and 1.10b).

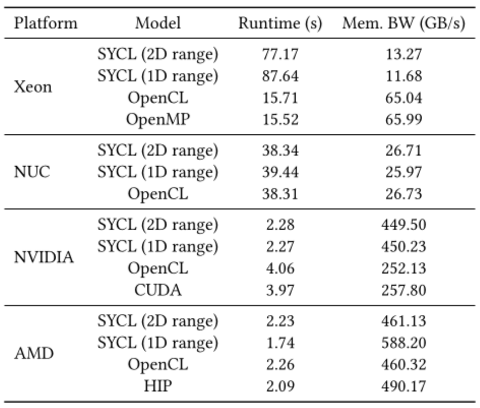


(a) BabelStream Copy results



(b) BabelStream Dot results

Figure 1.10: SYCL, OpenCL and native language comparison



Platform	Model	Runtime (s)	Mem. BW (GB/s)
Xeon	SYCL (2D range)	77.17	13.27
	SYCL (1D range)	87.64	11.68
	OpenCL	15.71	65.04
	OpenMP	15.52	65.99
NUC	SYCL (2D range)	38.34	26.71
	SYCL (1D range)	39.44	25.97
	OpenCL	38.31	26.73
NVIDIA	SYCL (2D range)	2.28	449.50
	SYCL (1D range)	2.27	450.23
	OpenCL	4.06	252.13
	CUDA	3.97	257.80
AMD	SYCL (2D range)	2.23	461.13
	SYCL (1D range)	1.74	588.20
	OpenCL	2.26	460.32
	HIP	2.09	490.17

Figure 1.11: Heat average runtime and bandwidth results

The most compelling results come from the second and the third benchmark run because the kernels involved are more probing of the capabilities of the language in a real-world scenario. The *Heat* benchmark reveals a whole different view because if the SYCL implementation is written correctly, it outperforms OpenCL and vendor-specific languages without changing the code underline. The graph shows two-column for SYCL due to a missing implementation of a feature on *hipSYCL*, caused by the relatively young age of the compiler.

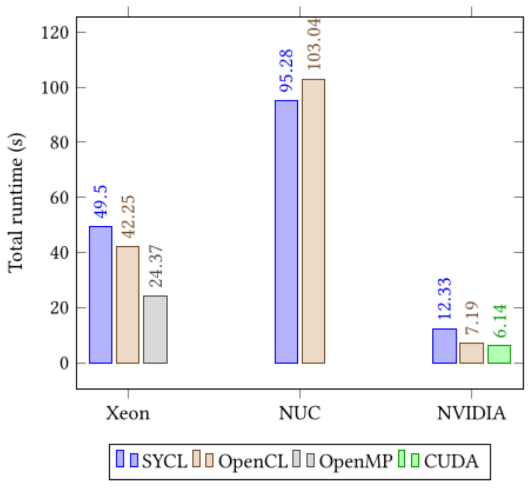


Figure 1.12: CloverLeaf results SYCL vs vendor specific language and OpenCL

I report the table 1.11 of the results, where it is clear the values of the comparison and how memory bandwidth is related to execution in this kind of kernel. On the other hand, the last benchmark proposed shows (Fig. 1.12) different shortcomings of SYCL at that stage. There is no comparison on AMD since none of the available implementation seemed to run on that hardware and the performance achieved on NVIDIA's hardware is quite far not only from *CUDA* but also *OpenCL* and since this last benchmark is one of the closest to real-world application it is a little bit discouraging. Even though SYCL is still very competitive on Intel's hardware, which can be a good forecast of what it will be able to do when their discrete GPUs will be available to the public.

1.7. Ligen: the case study molecular docking application

This specific application was developed at Politecnico di Milano [23] and based on *Dompè* owned ligen procedure [24]. In my case, I consider only the geometrical docking algorithm that is used to filter out incompatible *ligands*. This part is composed by many different kernels of different sizes. At first it sets up all the *ligands* for the computation and then proceeds with the real algorithm. Figure 1.13, taken from [23], shows an overview of steps and corresponding kernels of the algorithm.

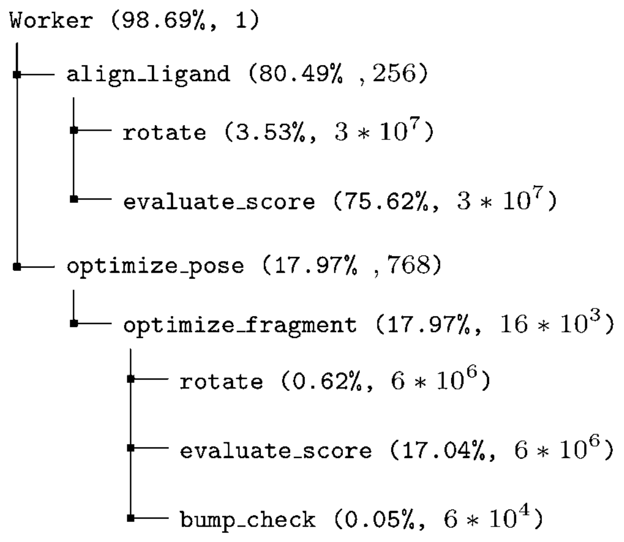

Worker (98.69%, 1)
align_ligand (80.49% , 256)
rotate (3.53%, $3 * 10^7$)
evaluate_score (75.62%, $3 * 10^7$)
optimize_pose (17.97% , 768)
optimize_fragment (17.97%, $16 * 10^3$)
rotate (0.62%, $6 * 10^6$)
evaluate_score (17.04%, $6 * 10^6$)
bump_check (0.05%, $6 * 10^4$)

Figure 1.13: Application algorithm phases

2 | Contribution

In my work,I wanted to test the feasibility of converting CUDA code into the new SYCL standard and having it run on different platforms. Achieving performances that, for a large-scale and performance-critical application, are close enough to the original one on various hardware. These tests and considerations are required to assess if SYCL is mature enough to be a valid alternative for real-world applications. In this chapter, I show my work and my evaluations in porting a CUDA optimize application to SYCL. I show how a porting from *CUDA* to *SYCL* can be done. I introduce the novelties in SYCL standard respect *CUDA* and which design decisions an engineer has to make when performing such action. To give a more comprehensive overview, I describe the work done in converting a complex *molecular docking* application to SYCL, originally written at Politecnico di Milano [23] and based on *Dompè* owned ligen procedure [24]. The characteristic high-throughput and compute time of this software make it a good candidate to evaluate pure GPUs performances since every slow-down is easy to detect and analyze.

This chapter is split into two main sections. The first one describes carefully all the design and code decisions taken, dividing the work into different phases. Each one of them regards different kernels with distinct characteristics and challenges. Every sub-section analyzes in-depth the differences among the original CUDA code, the SYCL code on NVIDIA's GPU, and the SYCL code on AMD's GPUs. The second section describes the performance obtained with the new code, and besides simple code comparison, I performed an in-depth analysis of the kernel's performance. The performance evaluation includes not only CUDA versus SYCL on NVIDIA GPUs but also what kind of performance is achievable on AMD's machines. The analysis concludes with considerations on portability and performance of the code written in SYCL. All the tests were made using all the available SYCL implementations, that exploit GPGPUs computations accordingly to the available hardware.

2.1. Design choices porting the code

Every time a new language/framework wants to be widely used, it needs to provide a clear and simple way to rewrite all the existing code from the most spread language to its own, in order to be a possible new standard for the industry. In SYCL specific case, there are four different decisions necessary when starting to use it for a new project. First of all, engineers must chose which compiler use among all the possible implementation. In case they work on the translation of CUDA code, they have to decide how to perform it. Since SYCL offers two ways to handle memory, developers have to choose which one fits best their work. Lastly, particular aspect of the code can be focus more on portability on different types of accelerators, e.g., GPUs or FPGA, or target one type specific of accelerators trying to achieve the best performance possible.

2.1.1. Choose a compiler

SYCL is a standard defined language, so there exists different compiler implementations. Therefore, the first choice engineers need to take is to decide which compiler to utilize for the project. The problem arises since there are many different implementations of the standard, and they are slightly different from each other. The only ones that target GPUs are *hipSYCL* and *OneAPI* (also known as DPC++) so the choice is simple. At the moment I am writing, both implementations support NVIDIA and AMD's hardware, that are the only currently available, while only oneAPI can target Intel future GPUs. Moreover, the support for AMD's GPUs from DPC++ is in early stage and not advertised yet as fully functional, while a great effort has been made in porting CUDA through ptx, the low-level parallel thread execution virtual machine and instruction set architecture (ISA) used by NVIDIA GPUs. In addition, versions of OneAPI distributed by Intel don't provide the possibility of compiling code for any hardware not owned by them. Users must download and compile the open-source version, *DPC++*, that they aim to upstream to LLVM. HipSYCL only provides one open-source version that needs to be compiled by the user, but with a completely different approach, it supports natively AMD and NVIDIA's GPUs. In my case, I decided not to discard any available implementations, since I want to give a complete comparison of the code and performance portability on both implementations. So everything I will describe will be valid for both implementations if not stated otherwise.

2.1.2. How to perform the porting from CUDA

Obviously, the SYCL standard from the Khronos Group doesn't cover how to perform the porting from CUDA code, even though it is a necessity in the industry. Since there are many implementations in CUDA, developers of SYCL want to bring all the available code to this new standard to show the achievable performance. For this reason, the effort is significant, and everyone is writing some tools or their guide to facilitate the transition. So far, the two most useful guide/tools are the one from **Codeplay Software** and **Intel**. The former provides a very detailed translation guide from CUDA terms to SYCL ones. The latter provides a complete and updated wiki, but also a proprietary tool, named **dpct**, that converts the CUDA code into fully functional OneAPI code. The advantages of a complete transcription of code from one framework to another are evident. It reduces the time to convert tedious and obvious code from one to another without losing the correctness of the implementation. The Intel tool is very performant and can reduce the porting time. It provides different ways to do it and lets developers port even structured projects. It writes the code following different styles and can leave the original code commented. As usual automatic code translation is not perfect. In this case, the code transliterated is a very simple one while more complex constructs are not converted, and the tool leaves a comment to the developers, letting them know that they have to rewrite it.

The main reason for this behavior is that some complex constructs in CUDA are not so easy to convert in SYCL, or they don't exist yet. The counter effect of this porting is that inexperienced engineers found very complex code to convert without having general knowledge of how SYCL works. Another more relevant counter effect is that, when it performs the porting, it converts all the calls to a different SYCL header which is not standard and is distributed only by the official Intel OneAPI package. This is not bad per se, but this would produce SYCL code that is not portable on different hardware besides the Intel one, and that is not possible to compile with other not proprietary compilers. This effect goes against the basic philosophy of the standard, the idea of having code portable everywhere and not being locked to a single hardware provider or company.

During my work, I used and tested both approaches. I started trying *dpct*, but I stumbled upon the drawbacks above, so I decided to let it convert the code but do not keep the result of the tool in my code and pick only parts of the code that were usable everywhere. In conclusion, the only parts helpful were those that select which thread, block ids, or size to use or memory allocation on different hardware. So it was not very useful, since after a short time coding using SYCL these details are known. Moreover, in complex CUDA project structures, the automatic tools were unable to identify the best way to rewrite the code in a performance-oriented way. So much of the code was not translated or in a way that needed to be rewritten. For the rest of the work, I used *Codeplay Software and Intel* documentations but also the Khronos's specification.

2.1.3. Unified Shared Memory vs Buffers and Accessors

The main difference between CUDA and SYCL is how the memory is handled by developers. In CUDA, it still follows the old *C-like* allocations type where it is the developer, using pointers, that decides where and how much memory utilize. It is a very performance-oriented style that requires effort from coders that have to know where they want the data and how much memory use. For this reason, it is also much more error-prone. In the old specification (SYCL1.2), the only way to allocate and handle memory was the new safe way provided by modern C++, specifically C++17. Developers have to follow the RAII (resource acquisition is initialization) design pattern when they want to move data from host to accelerator. This method brings away from humans the possibility of errors in allocating memory. It handles everything that regards memory, letting the compiler decides how to manage memory and trying to have the best performance. Moreover, it was possible to obtain a better scheduler that could give a lower end-to-end computation time by letting the compiler handles data dependencies. This new more modern approach is still available and encouraged when writing SYCL code. Starting with DPC++ and following Intel push, SYCL's new standard (SYCL2020) decides to provide also the old pointer style to handle memory. It gives more freedom to HPC developers to do also some *tricks* that allow obtaining better performance in ways that automatic managers and compilers are not yet capable of producing. In addition, code written in CUDA is straightforward to convert in SYCL if using the same paradigm of memory handling.

In an application that already exists and where the memory structure and management has already been done porting CUDA code to SYCL is extremely easier when using pointers rather than buffers and accessors, so I decided to keep this programming style. The other reason is that in the code used for testing the SYCL performance and portability, the allocation of memory is done in the constructor, away from its first use. So allocating the memory, without keeping the RAII design pattern, performs a lot better rather than letting compilers figure out the best moment to move or to allocate data on the accelerator. Moreover, sometimes data do not have to be moved from host to GPU or vice versa, and pointers make it easier to understand. Since the data layout has been already established during the conversion of the code, there is no need for a new modern approach in this case. A second reason is that SYCL is designed to run the same code on many different hardware but in my particular case this is not a necessity. Algorithms and code are designed to run on GPUs, in particular on NVIDIA GPUs, so bringing it to CPUs or FPGA is not a concern for this type of application. If designing new code that needs to exploit different types of hardware, using buffers and accessors would be the right choice because it frees developers from thinking about lower memory handling.

2.1.4. Portability or extreme Performance oriented

Implementing code following the SYCL standards means that there is the idea that the code must run on different machines with different hardware. Engineers need to decide if write code that runs smoothly on everything or tune each version of the code to the hardware on which it runs. Starting from CUDA-designed code, there is already a strict requirement on compatibility and required performance to obtain since the new code base must achieve nearly the same execution time as the legacy one. In doing it, developers could disregard portability. The other option is to write general code that runs on AMD and NVIDIA GPUs, and in the future on Intel's ones, but not considering differences in architectures like the dimensions of the warp/sub_group/wavefront (three different ways to call the same thing, the minimum execution group). This decision is linked to lower-level calls like shuffle, group voting, or specific reductions, which are usually taken into account when tuning the application for specific hardware.

My work focuses particularly on the performance aspect too. Understanding how much performance is lost if deciding to go for full-portability, how close is SYCL to native frameworks, and how much effort is necessary for having a full performance-oriented version of the code on different platforms. At the beginning, I migrated the code having in mind only NVIDIA's GPUs, and I tried to exploit all the hardware and low-level specification by design because the code-base to convert is very specific. Having only one version running only on one platform loses the SYCL's characteristic of being multi-platform, I tested the same code on AMD GPUs. I also have a completely different version of the code developed at University of Salerno inside the European LIGATE project[25]. This version is more portable and does not consider the underline hardware and I used it to complete the performance comparison. Besides having the code run on NVIDIA with performance close to the native CUDA, the challenge is to have the same code that runs even on AMD GPUs with performance near to native CUDA and that it is the same or as similar as possible to the SYCL code.

2.2. Kernel Porting

To produce a study on porting the code to this new standard, I need to do concrete work and analyze the results. The code on which I focused is a molecular docking algorithm that has been developed here at Polytechnic of Milan, implementing with CUDA the work of Renxiao Wang et al. [26]. I followed as close as possible the starting code-base without considering porting to other hardware. I made further considerations on running and adapting the code where necessary afterward.

This section is divided almost into the same part as the work is. The first one describes the porting compares how the code runs and behaves with the framework and the problems related to differences in hardware. The second section finally makes performance comparisons and analyses of the problems related to SYCL. In the following section, there are not some small and simple kernels that do not have any challenges in their portable implementation because their porting is just a matter of changing the calls for the different frameworks. There will be some reference to them in the performance section 3.

2.2.1. Initialize memory

One of the most complicated points to handle when writing high performance-oriented application is how to initialize memory and how to reduce the latency between the usage and the loading of the data. In the CUDA application, a great speed up is obtained by splitting the allocation of memory, one of the most expensive operations, between the constructor of the class and the start of the computing operations. In SYCL, especially if following the more modern concept of SYCL/C++, this aspect is a little more difficult to achieve for a couple of reasons. The first reason to notice is that SYCL needs to create a `queue` before doing any operation on the selected accelerator. So there is the need to select the hardware and then create the queue. These two operations are very costly. If possible, the two calls need to be done asynchronously and long before the needed start of the computation. The second reason is that the memory model of SYCL based on buffers and accessors makes it more demanding to decide when setting up memory because the design choice of RAII deallocates the buffers when they go out of scope so it is hard to split memory handling and computation, even though it has its benefits.

To allocate memory in the constructor as it is in CUDA, I created as many queue as necessary in the constructor and then hold a reference to them in an *unordered_map*. In this way, the queue is not deallocated after it goes out of scope, hence it is possible to reuse it afterwards. This workaround is necessary since queues may be compared to the idea of *CUDA streams* but developers can use *streams* in any part of the code without any further work while this is not possible using SYCL queues. Using this approach with multi threading is possible to have one queue for each thread.

A second difference between the two frameworks is the result of the SYCL immaturity in its implementations. In the reference code, researchers used texture memory, a specific memory used for 3D memory mapping, capable of optimizing memory reading of this type. The SYCL standard document foresees this possibility and calls it *image* but the implementation in compilers is not ready and not usable for 3D memory mapping. On paper, it will be possible to have it at a certain point, and it could be used on 2D mapping right now but that is not useful in my case. So I spread the 3D memory in a single array of the three sizes in sequence as it is done in CPU programming. The impact of my solution is not clear. This may have cost some time when deciding if the address that I want to read exists or not because in texture memory this is done by the underline command and does not create branches, which are necessary in my case. In depth comparison is in 3.5

2.2.2. Transfering data with USM

Moving data across host and GPU is critical in all heterogeneous applications. Reducing the amount of data movement and precisely times how and when transfer data have a huge impact on the performance of the computation. The functions currently available in SYCL are much less refined than the counterpart in CUDA, especially when dealing with pitched memory. In the target application, particular data need to be moved a couple of times. They are allocated in the constructor using pitched memory in CUDA and corresponding aligned memory in SYCL. When sending data up and down from GPUs, CUDA lets you use pitched memory copy, such that there is no waste in the memory bandwidth and time used to complete the operations. SYCL doesn't provide such feature to maintain consistency with the allocation and between computations. Developers have to transfer the original amount of data allocated in the constructor, otherwise, the following computations get the wrong data.

I illustrate how this affects the application and provide meaningful examples. Generally, in the constructor, the application doesn't know yet the size of the molecules it is going to analyze. It allocates in memory the maximum empirical known size of the molecules, which requires a little bit more memory than needed for smaller molecules but guarantees to avoid memory overflow, the same is done in CUDA.

```
cudaMemcpy2DAsync(source,l.num_atoms*sizeof(atom_type) , destination,
    pitch_atoms, num_atoms*sizeof(atom_type),all_data_necessary,
    cudaMemcpyDeviceToHost,stream);
```

Listing 2.1: transfer data across devices CUDA

```
q_gpu.wait();
q_gpu.memcpy(destination, source, max_number_atoms*sizeof(atom_type)*
    all_data_necessary);
```

Listing 2.2: transfer data across devices SYCL

When the application run sizes are known. After some computations, the application needs to bring data down to the host to sort them and then resend them on the GPU to conclude the operations. CUDA allows the pitched copy of the data without losing anything and improving efficiency. SYCL cannot do it, and this influence also the following code that has to deal with useless numbers, that I remove when sending the new data back to the GPU.

Following examples explicate better the difference, Listing 2.3 shows how in SYCL this little memory difference has an impact also in the code that follows, which implement unnecessary loop of operation and how those data are removed when being sent back to the GPU. For comparison Listing 2.4 is the same code but in the CUDA version.

```
for (int i = 0; i < ligand::max_atoms; i++) {
      new_location = static_cast<output_ligand::fp_type>(
    h_atom_positionx[pose_id*ligand::max_atoms+i]);
}
/.../
q_gpu.memcpy(destination, source, num_atoms*sizeof(atom_type)*
    all_data_necessary);
```

Listing 2.3: dealing with SYCL memory transfer

```
for (int i = 0; i < l.num_atoms; i++) {
     new_location = static_cast<output_ligand::fp_type>(h_atom_positionx
    [pose_id*l.num_atoms+i]);
}
/.../
cudaMemcpy2DAsync(destination, pitch_atoms, source, num_atoms*sizeof(
    atom_type), num_atoms*sizeof(atom_type), all_data_necessary,
    cudaMemcpyHostToDevice,stream);
```

Listing 2.4: memory transfer in CUDA

2.2.3. Simple and small kernels

In general, small kernels didn't require any particular effort but simply modifying the calling structure to SYCL functions. These simpler kernel implementations work flawlessly on any NVIDIA GPU. Special attention was necessary only for one of those kernels, the one that takes care of moving the atoms of the molecules to the center before performing the rotation.

Targetting NVIDIA GPUs

The reason why this kernel needs attention is due to how it is implemented in CUDA. It uses shuffles in warps to perform critical operations, convert this operation is straightforward on SYCL, as shown in 2.5and 2.6, only if the underlined hardware will be the same.

The following code shows how the difference between the two versions is just in the framework used, which change the available functions, both codes are very easy to understand. On CUDA, first, it computes a reduction on the warp exploiting the shuffle operation, which is indispensable to obtain better performance on GPUs, and then broadcast the result of the first thread of the warp, the only one to have the correct result. This last operation is a consequence of how the CUDA reduction is written by researchers since it is not provided by the library.
On SYCL, thanks to the implementation of SYCL2020, developers don't have to write their reductions because they are already provided and already use the technology to work on any accelerators or group selected. So if using GPUs and running in a warp, it uses shuffles by default and broadcasts the result to all the threads in the same warp, so the code is much more concise and better optimized.

```
//reduce sum.
avg_x = reduce_sum_tile_shfl<T,32>(tile, avg_x);
avg_y = reduce_sum_tile_shfl<T,32>(tile, avg_y);
avg_z = reduce_sum_tile_shfl<T,32>(tile, avg_z);
//broadcast to all tids
avg_x = tile.shfl(avg_x,0);
avg_y = tile.shfl(avg_y,0);
avg_z = tile.shfl(avg_z,0);
```

Listing 2.5: CUDA example

```
//reduce sum.
avg_x = cl::sycl::reduce_over_group(tile, avg_x, std::plus<>())
```

```
avg_y = cl::sycl::reduce_over_group(tile, avg_y, std::plus<>())
avg_z = cl::sycl::reduce_over_group(tile, avg_z, std::plus<>());
```

Listing 2.6: SYCL porting example on NVIDIA

From Targetting NVIDIA GPUs to AMD GPUs

So far, I discussed how to write the code taking into account only NVIDIA GPUs because that is the first thing available when converting the code. I was already testing the code with the two possible compilers but only on that hardware. Luckily, I was able to test the code on an AMD platform, which highlights some difficulties to get the code running on it, and it shows how much is portable highly-specific code. These sections, one for each class of kernels, present which changes are necessary to have highly customized code in SYCL when the underlining hardware changes. Converting the code on AMD wasn't easy because it requires understanding how the AMD GPUs hardware is different from NVIDIA's one and how to exploit it in the best way possible without changing the code too much. This work is not strictly necessary when writing SYCL code because it can be written in a more portable way that doesn't use low-level implementation, such as warps and shuffles. I wanted to extract the best performance possible and keep the code as close as possible to CUDA. The analysis of the effect of my choice on performance is in section 3.

In the kernels considered, changing the dimensions of the group was not possible without major refactoring, so I opted to add an *if-else* statement to cope with different warp sizes. The effect is visible in Listing 2.7. This change is effective in getting the correct results, and it also shows how is possible not to be bound to some hardware vendors, with minor tweaks.

```
if( tile.get_local_id() < 32 )  {
        avg_x = reduce_sum_tile_shfl<T,32>(tile, avg_x, 32);
        avg_y = reduce_sum_tile_shfl<T,32>(tile, avg_y, 32);
        avg_z = reduce_sum_tile_shfl<T,32>(tile, avg_z, 32);
        //broadcast to all tids from 0 to 31
        avg_x = cl::sycl::group_broadcast(tile, avg_x, 0);
        avg_y = cl::sycl::group_broadcast(tile, avg_y, 0);
        avg_z = cl::sycl::group_broadcast(tile, avg_z, 0);
    }
    else if (tile.get_local_id() >= 32 ){
        avg_x = reduce_sum_tile_shfl<T,32>(tile, avg_x, 32);
        avg_y = reduce_sum_tile_shfl<T,32>(tile, avg_y, 32);
        avg_z = reduce_sum_tile_shfl<T,32>(tile, avg_z, 32);
        //broadcast to all tids from 32 to 63
        avg_x = cl::sycl::group_broadcast(tile, avg_x, 32);
        avg_y = cl::sycl::group_broadcast(tile, avg_y, 32);
        avg_z = cl::sycl::group_broadcast(tile, avg_z, 32);
    }
```

Listing 2.7: SYCL porting example on AMD

The adjustments here are needed since the sub_group size is 64 and not 32, and there is the need to shuffle the correct result and use the reduction on the correct amount of data. It is done by taking the same reduction implementation from the CUDA code and rewriting it, giving as a parameter how big is the warp size. The other noticeable change is the split between threads based on the `id` in the warp. Such change lets every thread exchanges data as it does on NVIDIA, but then broadcast to every thread the correct result. It is not possible to use the given function as above in Listing 2.6, because it uses by default all the `sub_group` size, and it is not what I wanted here. Hence the code is more complex but very similar to the CUDA code provided. This workaround is necessary if developers want to use shuffle and group algorithms to exploit the particular feature available on GPUs. In other cases, they may not be necessary. Deep performance analysis can be found in section 3.

2.2.4. The Alignment Phase: a Compute Intensive kernel

In this section, I analyze the conversion of the first big kernel of the application, which takes care of the alignment phase of the algorithm. From the software point of view, this is the most compute-intensive kernel. It performs the largest number of computations and has the biggest block/group size. Due to the compiler optimization possibility provided by distinct compilers, both elements are central in the analysis of the performance. These two characteristics impact a lot the porting thanks to the technology used and available, and the results in how the code is executed are very different.

In this kernel and many others, there is a workaround to cope with the missing texture API in SYCL, so I show how I did to keep having the correct results and how it adds divergences in the code. In telling about this work, I split the section into two parts. The first one is about porting the code from CUDA to SYCL but maintaining the same hardware. The second part is about the modification necessary to make the software run on AMD's GPUs but still have the correct results. Thanks to the availability of NVIDIA's profiling software with code compiled with SYCL, I was able to show also some limitations and differences in compilers that make the porting a little bit more laborious, but that explains better performances (see 3).

Targetting NVIDIA GPUs

This kernel presents many specific characteristics that make it harder to translate, and it also needs a deeper knowledge of SYCL. The first element of difference from CUDA is how is allocated shared memory among threads in the same block. In CUDA, it is a kernel's characteristic that developers set when launching it. It is allocated inside the specific kernel, and its dimension is specified as size in byte in the part of code that precedes the call (see 2.8).

```
//shared memmory size
size_type shmem_size = how_many_items * sizeof(item);
//launch kernel
kernel_name<<<grid_size, block_size, shmem_size, stream>>>(args);
```

Listing 2.8: CUDA shared memory allocation

In SYCL, the *shared memory* is called *local_memory* and, as it has been implemented until now, it can be allocated only as a special accessor. So in this case, the usage and knowledge of accessors are necessary to developers, even if the chosen method of handling memory is *usm* as I did. The quantity of memory to allocate is decided inside the part of code that handles the kernel, not before, and its size of it is selected by developers giving the actual number of elements to store, the class of the item, and the size of the shared memory(2.9).
Moreover, different implementations offer slightly different function calls, the example here is that *hipSYCL* offers one more possibility than *DPC++*, the local_memory can be allocated calling a particular class(2.10). This is an element of difference between the two implementations that can produce code that cannot be compiled in DPC++, while the general accessors work with both.

```
::sycl::accessor<item_class, 1, cl::sycl::access::mode::read_write,cl::
    sycl::access::target::local> local_memory(number_of_item,handler);
```

Listing 2.9: SYCL shared memory allocation

```
::sycl::local_accessor<item_class, dimension> local_memory(
    number_of_item, handler);
```

Listing 2.10: hipSYCL shared memory allocation

How local memory is allocated is not a problem in the porting. The peculiarity in this kernel comes up when the developer decided to reuse local memory for a variable to not waste local memory. This behavior is common when local memory is not enough inside the kernel implementation. Technically, this action can be performed in SYCL, and it is possible in *DPC++*, as I show in Listing 2.11, but in *hipSYCL* this function is not yet implemented, so it is not possible to reuse local_memory differently. Luckily for me, in my case, the application doesn't actually use all the memory, so I allocated two different arrays when the compiler selected is *hipSYCL*.

```
//CUDA memory cast
(new_type)* shmem_new = (new_type*)shmem_old;
//SYCL memory cast
::sycl::local_ptr<new_type> shmem_new{cl::sycl::local_ptr<void>(
    shmem_old)};
```

Listing 2.11: example of local/shared memory cast

The second point of difficulty in this kernel is when the code checks if a given result of the computation is inside the memory mapped in the texture. The problem arises from the missing texture memory handling in SYCL. In CUDA, the texture memory checks by default if the memory reading is correct, and in case it is not, it returns a given value by *clamping to the edge* of the memory. In SYCL, there is not this possibility yet. I use simple flattened vector/memory pointers, so I have to check if the results of the computation are indeed inside the memory. This workaround adds verbosity, complexity, and divergence to the code. It has not a lot of sense to describe it, while the following code is much clearer(Listing 2.12).

```
//setup sizes for better readability of the code
::sycl::int3 sizes(size_x, size_y, size_z);
::sycl::int2 strides(size_x, size_x*size_y);
//setup variable to handle the result, increase code readability
::sycl::int3 coord(idx_x,idx_y,idx_z);
//if coordinates are correct, take the value from memory
value = is_valid(coord,sizes) ? score_acc[coord.x() + coord.y()*strides.
    x()+ coord.z()*strides.y()] : OSC ;
```

Listing 2.12: handling memory check in SYCL without texture memory availability

In Listing 2.12 I set up two variables that I need to validate the results for the following computation. These variables are not necessary but having them makes the code much more readable. For the same reason, I wrap the coordinate result in a single variable and call the *is_valid* function that simply checks if the values are inside the size of the memory, if so it returns the value in memory, otherwise it returns the value that is at the edge of memory, taken from CUDA and CPU implementation.

For comparison, I show also the CUDA implementation of this particular case in Listing

2.13. CUDA is simpler and much easier to read and write because in handling texture memory all the steps written in SYCL are provided by the library function.

```
value = (tex3D<return_type>(score_texture, idx_x, idx_y, idx_z));
```

Listing 2.13: CUDA texture reading

This workaround is applied across the application, and I do not cite it in other kernels. Measure the impact of it, it is not simple, but I try to acknowledge it and analyze the differences in the section regarding performances 3. In any case, this is removed in future releases of the code in CUDA.

The last part of this kernel takes the results of all the warp inside a group and performs a reduction, using only the first warp. This is possible in some readable and easy but powerful code in CUDA, exploiting high-level libraries and construct provided by NVIDIA, i.e., coalesced group. These operations have an impact on how the memory is set up, on which thread reads and uses which data, on the reduction code, and how to select which threads need to run reduction operations. CUDA offers an API to let run some code only by selected and active threads, without creating divergence and without setting up the memory to make the reduction without errors.

```
cg::coalesced_group active = cg::coalesced_threads();
value_1 = shmem_asint[tid];
value_2 = shmem_asint[tid+32];
active.sync();
reduction_operation_on_warp(warp, value_1, value_2, &return_values);
active.sync();
```

Listing 2.14: CUDA cooperative groups handling last reduction

In Listing 2.14 I highlight the two important functions *cg::coalesced_ threads* that take only the active thread in that particular branch and the sync called on *active* that synchronize only those threads, performing reductions only to that. These high-level functions in SYCL are missing. Since the code for these functions is used to read from memory and setup variables to the last reduction, I exploited the neutral term of addition to let the reduction performs on all the threads and have the correct result. To have that, I added divergence in the code and some operations that are not strictly needed. The main problem is given by the fact that I could not synchronize only on active threads, which forced me to set up every thread and memory read, otherwise, the code would block and wait on kernels that would never arrive at the synchronization barrier.

```
value_1 = (tidz < 32 ) ? shmem_asint[tid] : static_cast<X>(0);
value_1 = (tidz < 32 ) ? shmem_asint[tid+32] : static_cast<Y>(0);
```

```
::sycl::group_barrier(it.get_group());
///final reduction: only use first warp
reduction_operation_on_warp(tile, value_1, value_2, &return_values);
tile.barrier();
```

Listing 2.15: CUDA cooperative groups handling last reduction

In Listing 2.15 I created divergence when cleaning the value of wrong results and set it to zero or the correct value from memory. Synchronization is necessary, otherwise, some data may not be taken in time before starting the reduction. Overall, once understood the problem with synchronization and realized that SYCL doesn't offer yet a way to select on which threads operate, writing, understanding the code, and getting the results wanted is straightforward.

The impact of these changes is minimum in the readability and portability of the code, it is still simple to write code in SYCL and have it running on NVIDIA GPUs, even though developers need to make some effort to understand problems and work-around. For what concerns performance effects and comparisons see section 3.

From Targetting NVIDIA GPUs to AMD GPUs

By porting this kernel to AMD, I had to face the difference in the size of the warps, which is very recurrent in this application. Its size can be chosen by SYCL at runtime, using a function call. The point is that setting up sizes at runtime doesn't take into account the physical limits of the GPU. This kernel has been thought and coded to exploit groups/blocks of size 32x32 to compute a certain amount of rotations inside the block. This size is the physical limit of 1024 threads per group that both GPUs allow. Having a group of 32 threads is also much more convenient in CUDA because it fits the size of the warp. On AMD this assumption fall and the consequent advantages are not there anymore. Moreover, to get the correct result it needs some changes.

```
if(tile.get_local_id() < 32)
    reduce_max_tile_shfl_pos<32, X, int>(warp, value_1, value_2, &
   return_values);
else if( tile.get_local_id() >= 32){
    reduce_max_tile_shfl_pos<32, X , int>(warp, value\_1, value_2, &
   return_values_2);
}

if (tile.get_local_id() == 0 ) {
    local_memory[tidz / 32] =       return_values.value_1;
    local_memory[tidz / 32 + 32] = return_values.value_2;
}
else if ( tile.get_local_id() == 32 ) {
    local_memory[tidz / 32] =      return_values_2.value_1;
    local_memory[tidz / 32 + 32] = return_values_2.value_2;
}
```

Listing 2.16: work around to make kernel works on amd platform

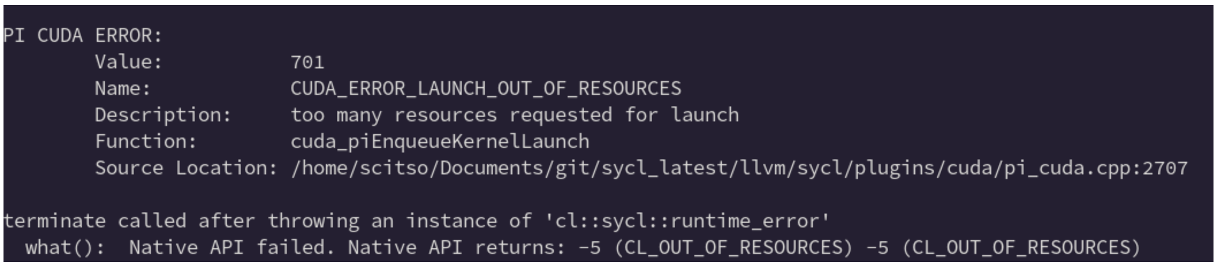
In this case, all the reductions consider the size of warp of 32, while AMD's hardware has a size of 64. To solve this problem there are different possibilities. Firstly, it is possible to rewrite the kernel considering the new hardware and optimize for it which is probably the best solution but it takes much more effort, and it is a different approach from the one I chose. The second possibility, which I tried, is to change the size of the block. I tried to let select the size at runtime but it gives a runtime error since 64x32 threads don't fit the hardware. It was possible to change to 64x16 the size of the block and modify the code a little bit. It would take some time to rethink the algorithm and split the code in two. In the end, I decided to do almost the same thing I did on the previous kernel (2.2.3), rewrite or reuse reduction functions and split the code into two *if-else* branches that allow me to do computations on the correct amount of data. This operation is also needed when the first thread of the warp stores values in memory. In this case, both the first thread and the thirty-second have to do it.

In this process, the size of the memory allocated doesn't have to change, and the code is almost identical to the original. Developers can select one of the two implementations setting a flag to the compiler.

Deep Difference Discovered: from float to double

This all application supports computation performed in float and double-precision. For hardware-related reasons, I started to work with single-precision checking feasibility and performance. Once I completed all the applications, I was given access to much better GPUs with computation in double-precision hardware accelerated, which gives much better performance and let check the result by comparing them with the original CPU output in a more precise way. At this point, I switched to using doubles to profile the code and to collect performance, but the application breaks in an unexpectable way if I use the DPC++ compiler it returns an error at runtime `CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES` (see Figure 2.1) both on *gaming* GPU and on *compute dedicated* GPU.

The native code worked and also the SYCL code works if I use *hipSYCL* as a compiler, the problem was related to DPC++, and I had no idea of what was happening. The key



```
PI CUDA ERROR:
        Value:           701
        Name:            CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES
        Description:     too many resources requested for launch
        Function:        cuda_piEnqueueKernelLaunch
        Source Location: /home/scitso/Documents/git/sycl_latest/llvm/sycl/plugins/cuda/pi_cuda.cpp:2707

terminate called after throwing an instance of 'cl::sycl::runtime_error'
  what():  Native API failed. Native API returns: -5 (CL_OUT_OF_RESOURCES) -5 (CL_OUT_OF_RESOURCES)
```

Figure 2.1: runtime error with double precision enabled

to cracking this problem was to utilize all the NVIDIA tools that work also with the code written and compiled in SYCL, reducing the group size and running the code it works, so I used **NVIDIA NSight Compute** to analyze at a fine grain the code.
From this analysis emerges that CUDA and DPC++ compile the code in two very different ways. The former takes into account the number of registers used and the number of threads, so it is capable of not exceeding the number of registers used in the application and scheduling operations accordingly. The latter has a different approach, more similar to how it should work on the CPU. It tries to load as much data as possible in order not to stop the computation and wait if some data is not yet in the register while executing. At the moment I am writing, I don't know if it inherits this implementation by LLVM or is SYCL specific or if it is possible to do otherwise and perform a different type of optimization.

These kinds of errors are difficult to find because code running on GPU doesn't return specific and informative errors. They are also difficult to correct because one option would be to change the implementation. In this case, changing the implementation would make it compiler bounded. It doesn't make sense in any case and much less in a framework like SYCL that wants to be very portable and independent from compiler or hardware.

Luckily in a presentation given by *Codeplay Software*, the speaker shows a series of flags useful when porting the code to CUDA. In this way using `-Xcuda-ptaxas '-maxrregcount=64'` flag, I can limit the number of registers used. I set the limit of registers to the maximum number possible in my case, which is also the hardware limit for a 32x32 thread block. Now the application runs correctly also with *DPC++*. The compiler applies this limit to all the kernels compiled so it may have some effect on the performance of other kernels. Further analysis in section 3.

Interesting that two different SYCL compilers behave differently with the same code and give different errors. It may be an opportunity if developers want to use a particular feature not implemented in one of them but it could be a problem because it makes the code compiler dependent, which creates competition inside the framework and favors native languages. After all, engineers don't have to worry if the code written would work or not. This problem with registers doesn't emerge on AMD GPUs. I couldn't make the same analysis or check the number of registers used because the native suite of software on AMD is much smaller and less effective than the NVIDIA one. This is related to the fact that NVIDIA is the industry standard and their software has been around for much more time.

2.2.5. Optimization and Final score. Divergent kernels

This section analyzes the porting to SYCL of three different kernels that are characteristics of three phases of the molecular docking application. They are also interesting to analyze because in each of them there is some peculiarity in the porting from CUDA to SYCL or in the porting to AMD hardware. Moreover, each of them is characterized by many different branches that bring divergence in the execution.

Targetting NVIDIA GPUs

The first kernel, the optimization phase, has the same problem of reduction and memory reading presented in section 2.2.4 but, at the end of it, there is one last reduction that depends on the dimension of the grid along the y-axis. It is implemented in CUDA with a switch construct based (see Listing 2.17) on the size of the y-axis that takes that as a limit and performs the reduction using only the active threads. This selection is not possible in SYCL because it cannot select which threads of the warp are active and take part in the execution, but all of them take part in it. It is possible to use *if-else* branches and select only the desired kernels but it is not possible to synchronize inside that part of the code, otherwise, the execution stalls. The only solution is to let all the threads enter the reduction. Instead of using the library reduction provided by SYCL, use a custom reduction that takes into consideration only data from those threads that respect the required size. In this way, even synchronizing, there is no problem of stalling the execution since all threads reach the synchronization point (see 2.18). As above, this method lets some threads run without any reason, it is not the best solution possible and entails some wasted resources, but it allows the code to run correctly.

```
if (tid < y_dimension)
  {
  switch (y_dimension){
    case 2:
      {
        final_reduction_optimize_wrapper<W,2>::func(args);
        break;
      }
    case 4:
      {
        final_reduction_optimize_wrapper<W,4>::func(args);
        break;
      }
    case 8:
      {
        final_reduction_optimize_wrapper<W,8>::func(args);
        break;
      }
    case 16:
      {
        final_reduction_optimize_wrapper<W,16>::func(args);
        break;
      }
    case 32:
      {
        final_reduction_optimize_wrapper<W,32>::func(args);
        break;
      }
    }
```

Listing 2.17: cuda switch for last reduction in optimize_kernel

```
if (tid < item.get_local_range().get(1)){
      int y_dimension = item.get_local_range().get(1);
      final_reduction_optimize_wrapper<W,32>::func(args);
}
```

Listing 2.18: sycl last reduction in optimize_kernel

The second kernel takes care of checking if the molecule computed so far respects some characteristics to be correct also from a chemical point of view. At a certain point, it requires sorting the results obtained. Sorting elements on GPUs is different from CPUs and can be implemented in different ways. In the original CUDA version, researchers decided to use one of the algorithms available in the examples provided by NVIDIA and adapt it to the application. This operation can be done in SYCL by changing the algorithm for sorting or by taking the same algorithm and porting it to SYCL. To be consistent with the methodology used in this work, I opted to use the same algorithm and port it to SYCL is straightforward. Even though they were thought with CUDA in mind, it applies the same logic to the code without any problem. At the end of this kernel, the original implementation used a smaller partition of the warp to perform the last reduction and to check the result of all included kernels. As explained above for the other kernel, this is not possible in SYCL. Instead, I set up the last reduction and the control of the result to be done on the all warp considered, as possible to see in 2.18.

The last kernel checks other another chemical property that is important to have to obtain correct results. It was a very simple kernel to convert from CUDA to SYCL. It doesn't have any challenge besides the number of branches and group functions to call,e.g., shuffles and broadcasts. Luckily all those functions are available on all compilers. This kernel deserves to be mentioned for performance-related value and for its characteristics in the AMD conversion which I talk about in the next section, while performance comparisons are in section 3.

From Targetting NVIDIA GPUs to AMD GPUs

These kernels have in common the way blocks and grids are decided to run them most efficiently. Therefore every block size is based on a multiple of thirty-two to exploit warp size and its functions. This setting is not a problem converting it to SYCL if the underlining hardware will be NVIDIA's. When writing SYCL and running it on different hardware, it is compulsory to think about it and adapt the original code to the new situation.

When converting these threads, there were specific problems that weren't predictable. First of all, in all of them, there is a massive usage of group algorithms that coordinate the work inside the warp. These functions, e.g., any_of_group, all_of_group, and broadcast were not implemented in DPC++on AMD hardware. This problem was not solvable by me, the only way was to open a bug report in the GitHub repository, and they fixed it in a couple of weeks. Those functions were already implemented in hipSYCL but that compiler has another relevant problem for me. It does not allow printing from GPU kernels on this hardware. So I preferred to implement it in DPC++ because it makes it is easier to debug. Resolved this problem, I focused on solving the reduction and warp problem.

For the third kernel, the simpler one, there weren't any issues with registers and size of the blocks, and it makes it trivial to write the kernel to another platform. I was able to convert a 32 block size to 64. The kernel runs perfectly and better because reductions corrected themselves, and the code exploits the hardware in the best way. This tuning is not done by hand by changing parameters inside the code but by letting the application tune itself at runtime using SYCL specific functions available on every compiler. An example is shown by the first line of 2.20, where the queue is the way to address the accelerator in SYCL and collect info using a specific function.

```
kernel_3<<<grid,32,0,stream>>>(args)
```

Listing 2.19: CUDA third kernel launch

```
auto warp_range = queue.get_device().get_info<cl::sycl::info::device::
    sub_group_size>()[0];
::sycl::nd_range<3>kernel_3_range{{1,1,grid_size*warp_range},{1,1,
    warp_range}};
kernel_3(kernel_range, args...)
```

Listing 2.20: sycl kernel_3 launch

The *optimize_kernel* was the most difficult one to bring to AMD and have it run precisely. The dimensions of the block per se weren't a problem because they were well below the maximum allowed. The reason is that this kernel used a large number of registers. Having a smaller number of threads leave some spare registers that can be used without any runtime overloading, which lead to mistake in the computation. In the beginning, I didn't notice the problem, but then I realized that changing only one dimension wasn't enough to fix the reductions and the results. Understood that the size of the block was the problem, I made a couple of tests. The first is the one that requires less work and lets code as close to the first implementation as possible. Since it wasn't effective, I changed the approach and rewrote the smallest part possible of the kernel.

In the first attempt, I tried to change how the blocks are allocated along the dimensions without changing the overall dimensions. The x-axis was composed of 32 threads to comply with the warp size I tried to bring it to 64 threads to fix the reduction but halved the size of the y axis. This correction seems to work, and it works in other cases, but here the y-dimension of the blocks is used to retrieve data from memory. If missing one half on one size, it corrupts the kernel results. Spotting the errors and attributing them to the implementation is not straightforward since it misses only some molecules. The second approach required rewriting part of the kernel to comply with the necessity of dealing with all the data. I changed the size of the x-axis to 64 to accommodate the warp size and divided the other useful dimension. Then, I set up the code inside the kernel in a loop along the variable controlled by the y-thread index. In this way, the code changes slightly because I added an outer loop, but it still considers all the data correctly. I left out from the loop the closing reduction since it needs all the correct data already computed from every block, and it needs to be performed only once.

This approach gives the correct results, but it shows also the limits of portability. I consider it a limit because I needed to change the code to follow the hardware at a low-level otherwise, I could not get the correct result maintaining the same abstraction as in SYCL version that runs on NVIDIA's hardware. It could be achieved without so many changes, writing code that doesn't use such low-level features. In this case, the only way to get the correct size of thread group, the correct kernel implementation and best performance is to split the code and choose the right one using preprocessing macros. These considerations show other two things. SYCL allows developers to choose not to exploit low-level code and have an application running everywhere or exploit the hardware with low-level code but still have the necessity to write two different applications that get compiled accordingly to the machine. The last point is still a great leap forward because researchers need to know only one framework and not one framework for each hardware, but even if the framework is the same, they have to know the underlining architecture.

2.2.6. Non portable kernels and their work-around

The last kernel of the application is the sum of many different operations, and it is implemented in CUDA using *dynamic programming*. This technique allows one kernel running on GPU to launch other kernels running on the same GPU but with different grid and block sizes. It guarantees developers to make computation and synchronization among them as if they were launched in the standard way. It is a complex approach that exploits hardware and software both specific to NVIDIA. This feature is not available in SYCL so porting this kernel, as it was thought on CUDA, is not possible.

Before explaining how it is possible to complete the porting of the application, I want to point out that this technique is going to be removed from the application in future versions of the toolkit. Moreover, code written with this technique is complex to profile neither NVIDIA tools do it. The performance comparison of the two will not be possible but since it is part of the application, it was necessary to write code for it to complete the molecular docking.

Targetting NVIDIA GPUs

This kernel is the only one that is entirely different from the original code. The way the latter work is the following. It launches several kernels in the same block, one for each permutation developers are considering. Then each of them launches kernels that effectively make the computations necessary. The first block coordinates the work and collects the results. Besides how the algorithm is designed, it is important to state that in these kernels many specific API calls and functions are implementations dependent. I had to identify them understand and convert or remove them because they are unnecessary with the new algorithm.

Original code is constructed by the outer kernel that uses the block-Id to select on which point to of the memory extract data, in a way that every block operates on a different part. Each thread extrapolates the data then the first thread of each block launches another kernel that takes care of the different parts of this complex algorithm. This process of launching kernels is repeated for every specific sub-algorithm. The general kernel and all the other threads in each block are used when there is the need to make reductions. So after the computation, there is a complex way to perform synchronization to have the correct data on which they operate.

I converted it to another parallel mechanism and algorithm using two dimensions and spread the work using dimension-y to select the data offset in memory and the x-axis dimension to select and manipulate data for computation. The structure of SYCL lets developers easily break big kernels into smaller ones. Every small kernel corresponds to one of those kernels that in CUDA are launched from GPUs because I can select the correct data with the y-dimension. In this way, I can set block and grid size to optimize each kernel execution. These changes forced me to add some checks on the computation results and the value read from memory by each thread. If the value read is wrong the computation is stopped to accelerate some phases of the application. In the original application, this process is covered in the first outer kernel, in which the first thread coordinates all the others in the same block.

Changing the way kernels are written is not the only challenge that these kernels brought. In the implementation, many different and very specific features need to be converted in SYCL even after the change of paradigm. The main change and peculiarity is how synchronization is applied between kernels and how SYCL implements atomic functions with respect to CUDA. For what concern synchronization, the original code is very intricate, because it has to wait and synchronize all the different blocks and grid that comes from the secondary kernel launched. Example 2.21 shows how it is done in CUDA. Luckily, this measure is not necessary for SYCL, and the end of the kernel sets the sync point before proceedings in different computations.

```
__syncthreads();
if (threadIdx.x == 0) {
  cudaDeviceSynchronize();
    const auto err = ::cudaGetLastError();
    if (err != ::cudaSuccess) {
    printf(__FILE__ ":" LIGEN_XXTOSTRING(__LINE__) ": ERROR on pose %d:
   subgrid launch failed, marking pose as invalid\n", poseIdx);
    }
    isValid = isValid && (err == ::cudaSuccess);
}
```

Listing 2.21: CUDA dynamic programmin synchronization mechanism

Related to synchronization and algorithms, another difference is that the original implementation needs one last reduction which takes the values from all the sub_blocks and extracts the last sum. Even in this case, this is not necessary in SYCL because the reductions are already completed by each group of the kernel. It simplifies a lot the implementation and reduces the memory access in the final part of the application.

The way atomics is implemented is completely different between the two frameworks. In CUDA programmers use a function that will implement atomicity by itself, and they do not have to take care of memory or declare the memory in any particular way. In SYCL, it is necessary to declare inside the kernel on which part of memory developers want to perform atomic operations. This style gives better control and can lead to better performance because not all the memory accesses or all the operations need to be atomic. On the other hand, it adds overhead and complexity to the code and requires extra knowledge from the developers. (Examples 2.22 and 2.23).

```
if (tile.thread_rank() == 0) {
#if defined(__CUDA_ARCH__) && __CUDA_ARCH__ < 600
#warning No atomicAdd_block available for this SM arch, falling back to
   grid-wise atomic
    ::atomicAdd(out, tile_sum);
#else
    ::atomicAdd_block(out, tile_sum);
#endif
  }
```

Listing 2.22: CUDA atomic sum example

```
cl::sycl::atomic_ref<Dist,
        cl::sycl::memory_order::relaxed,
        cl::sycl::memory_scope::work_group,
        cl::sycl::access::address_space::global_space> atomic_score(
   score[m_idx]);
/.../
atomic_score += value;
```

Listing 2.23: SYCL atomic sum and memory declaration example

The example 2.23 is particularly related to DPC++, because when the project started *hipSYCL* and *DPC++* had different namespaces for their implementation of atomics, now they both have the same namespace. This was due to the fact that in DPC++ often some new features are considered external to the standard and belong to a different namespace than *standard* SYCL. The example 2.22 has a series of compile-time preprocessing specifications because the function name changed during the years. In this way, the code should work on all the hardware available. The relevant part of it is the function called `atomicAdd()` that probably entails all the extra checks and work that in SYCL are done by the developer.

From Targetting NVIDIA GPUs to AMD GPUs

Differently from other kernels, this one required almost no modification to run smoothly on different hardware, probably because it was thought from the ground up with SYCL in mind. The selection of the size of block and grid was done to cope with CUDA warp size. This specific operation can be done at runtime and select the correct one independently from the underlining hardware while the grid size was a multiple of both 32 and 64. So the division into groups is straightforward, and the optimization level is the same.

The only limit and difficulty that emerged were in the `is_nan` function that was not implemented in *hipSYCL*, which forced me to find a work-around in the implementation to make it work correctly as in DPC++ or CUDA.

2.3. Overview on portability

In the idea of the SYCL standard, *Kronos Group* stresses a lot the portability aspect meaning some code that can be compiled and run on many different devices and hopefully get native performance out of it. This is a very difficult task that also OpenCL tried to achieve. Since I could test the application with different compilers and on different architectures, I spotted differences in SYCL implementations that emerge given all the variable elements tested and thanks to the tool provided by NVIDIA and AMD I could analyze them deeper. This section contains the peculiarities, difficulties, and considerations about SYCL implementations presenting them separated by compilers or hardware differences.

2.3.1. Portability and code differences between DPC++ and hipSYCL

The two implementations on which I based my job are the LLVM/Intel, known as DPC++, and hipSYCL. I started using them on 06/2021. Besides comparing the differences between them now, I highlight the problems of the previous release and the huge work done by both teams towards having a shared and universal SYCL framework. The term *release* is to be taken lightly. Here I am referring to commits in the GitHub repository of the two projects, that are in continuous and rapid development, so it is useless to base the work on a single release number.

The first problem that occurs when testing an application on both compilers is that *hipSYCL* doesn't compile with *BOOST Library* if the version used is above 1.68. This is not a major problem for many, but since that particular library is one of the most common and spread, for other projects that use its new functionality, this problem may be a deal-breaker forcing them not to use *hipSYCL*.

Another difference that still occurs between the two is how they let you select the id of the running thread. In particular, what they let you do in a conditional statement comparing thread-id to an integer. The radix of the function call is the same but how compilers can understand and compare it to an integer is slightly different, as shown by Listing 2.24. *hipSYCL* requires an extra function call to get the integer of the id before comparing it, otherwise, it breaks at compile-time while *DPC++* doesn't.

```
::sycl::nd_item<size> it;
//hipSYCL
if(it.get_local_id().get_id(dimension) == integer)
//DPCPP
if(it.get_local_id(dimension) == integer)
```

Listing 2.24: comparison of the two sycl compilers in thread id conditions

I don't think this is a deal-breaking problem because, in the more precise way of hipSYCL, even DPC++ works. Nonetheless, I think that engineers would like to know it before writing an entire application. Discovering it afterward leads them to refactor every code of this kind if they want to change SYCL implementation.

Some functions are implemented within one project but not in the other. Simple example is *sycl::is_nan()* function, that I used in one of the kernel. DPC++ has it, but hipSYCL didn't. In hipSYCL also wasn't possible to use the function provided by the standard library. This little detail required me to search how to substitute it in a way that provides the same result. I found a work-around for it, which required some time, that now I can consider wasted, since in a very recent commit *hipSYCL* fixed it.

The last problem related to different implementation was fixed towards the end of my work, but, as in the previous case, I had already put in the work to cope with it. The naming scheme of some of the most important and useful functions available in SYCL2020 was different, and the code wouldn't compile.

```
//dpcpp
::sycl::reduction_over_group(args);
::sycl::any_of_group(args);
::sycl::all_of_group(args);
//hipSYCL
::sycl::group_reduction(args);
::sycl::group_any_of(args);
::sycl::group_all_of(args);
```

Listing 2.25: example of different function naming, but with same meaning

In Listing 2.25, I show some of them. Each of them is related to reductions or group algorithms very important in GPUs computing and programming, so the point was or rewrite them or use the correct function for the correct compiler. The difference is small and all the naming is understandable but changing them is a waste of effort. Luckily, hipSYCL's names are correct now and respect the guideline of the standard as the others.

From a developing environment point of view, both compilers support CUDA, but the version they are bound to is different. In my experience, **DPC++** supports any version of the CUDA toolkit starting from 10.1. The compiler works, and the application gives the correct results equal to those of legacy applications in CUDA or on CPU. Even if the documentation states that it is tested only on some particular GPUs with a given version of the toolkit. On the other hand, **hipSYCL** supports only version 10.1 of the toolkit, but it compiles even if the version provided is different. The application compiles and runs with CUDA toolkit 11.4 as a pointed reference, but the results are all wrong. I don't know what is related to this bug, but I hope it will be fixed soon if it is not already done. In any case, this is not a deal-breaker, but once again, researchers need to be aware of it, and **hipSYCL** documentation states very clearly this limit.

2.3.2. Across Hardware Vendors

Going from one company's hardware to another is not painless as it will be in the future plan for SYCL. Right now, the development upon them is in a different state. Since the NVIDIA hardware is more spread and used in the industry, it has more advanced support in SYCL, in particular in the Intel-developed LLVM, because it has the support of other companies focused solely on it.

These differences are not visible on every function or the common interaction with AMD's GPUs, but when I tested the porting to that hardware. I had some missing functions that completely broke the application. In my particular case, the functions that were not available were those directed to group algorithms implemented on `sub_groups`. All of them used on `sub_group` broke at compile-time, and there was no way to work around it besides rewriting them. They worked on single blocks because the way they are implemented is different. To solve this problem, I had to signal it on the GitHub repository by opening a bug report. Luckily, in this case, the community and the companies were very responsive and implemented them in a couple of weeks. The problem is solvable, but it stopped the work for that time.

Interesting to notice that using *hipSYCL* there was no problem with this function, pointing out that the two approaches of the compilers impact how the application works and what is the state of development of the two. HipSYCL approach seems to be a little be more complete on new and different hardware, but it is not free of limitations. Developing on it has a shortcoming that I could not avoid reporting. While working on GPUs is important to have the possibility of *printing* values from the kernels to check what is happening at runtime without using a debugger that could result in a bigger effort than required. At the current state *hipSYCL* doesn't offer this feature if the underline hardware is AMD, so printing from kernels is not possible. The only way to do it is to enable it as *unstable feature* when compiling it, but this could not work. Developing using solely hipSYCL as a compiler has this limit even if it provided all the functions I needed. Contrary to above, *DPC++* offers this feature without any problem.

It is important to highlight that NVIDIA's GPUs and AMD's GPUs have a tiny difference that becomes big when tuning applications around it. The size of their warp/wavefront. When the application was coded and generally when an application is designed with NVIDIA hardware in mind, it is frequent and correct to customize some kernels to exploit its warp size. Using warp-related functions and size has a huge impact on performance because the code is tight to the hardware and is optimized by the compiler. Moreover, warp functions use hardware *tricks* that reduce the memory latency and speed up the computation. When changing the hardware, given that all the functions work and are available, even if any SYCL compiler allows developers to run the code on the new platform, the results may be wrong because the size of the wavefront is different. This problem is not solvable by changing the functions but only by some workaround in the kernel or by rewriting the kernel and selecting the correct kernel with a preprocessing macro. It is in the hand of the researchers to choose which option is the best in each case because each of them has different impacts on performance and time to code. The only complete way around it is to write code platform agnostic that works on SYCL, but it is not sure that it would respect the expected performance on the hardware.

3 | Experiment and Results

This chapter describes the experiment taken to verify performances obtained by the new SYCL code in different cases and how it compares with native CUDA code. It is divided into different sections. The first one illustrates the setup for the experiment: machines and OS used. It also informs about the general methodology for testing. The second section describes the performance achieved by each sub-group of kernels split into categories accordingly to compute complexity. Each of these sections describes respectively the performance on NVIDIA and AMD GPUs. The third section focuses on the comparison of the code I developed against a more portable version of the application developed at the University of Salerno, inside the European project LIGATE[25]. It demonstrates the various performance achievable. The last section is a small overview of the memory management performance comparing SYCL and native CUDA applications.

3.1. Experiment Setup

SYCL wants to be a real alternative to CUDA and all the other frameworks available and used in the industry. Providing only a good portable and easy to write language is not enough. It has to provide also the same level of performance, or at least it has to be close enough to those achieved by the native code. This section assesses what level of performance can be achieved with it, and makes some comments on why and where there could be possible flaws. I split the comparison into different parts. The kernels are split according to the time that they take on CUDA. They are broken down into small kernels, those I do not even mention in section 2.2, and the other more important ones. The comparison of performances is also split by platform. Firstly, I compare SYCL on NVIDIA's GPU against CUDA then on AMD's GPU. The results of SYCL are distinguished by compiler: DPC++ and hipSYCL. To provide a more intuitive way to understand why I analyze and split performance comparison in this way, the plot 3.1 is a good reference. It shows the total *runtime* of every kernel, in this way it is possible to understand the importance of some small kernels that are launched inside a loop, i.e., *set_ optimization* and *eval_ optimization*. In addition to it, table 3.1 also indicates the

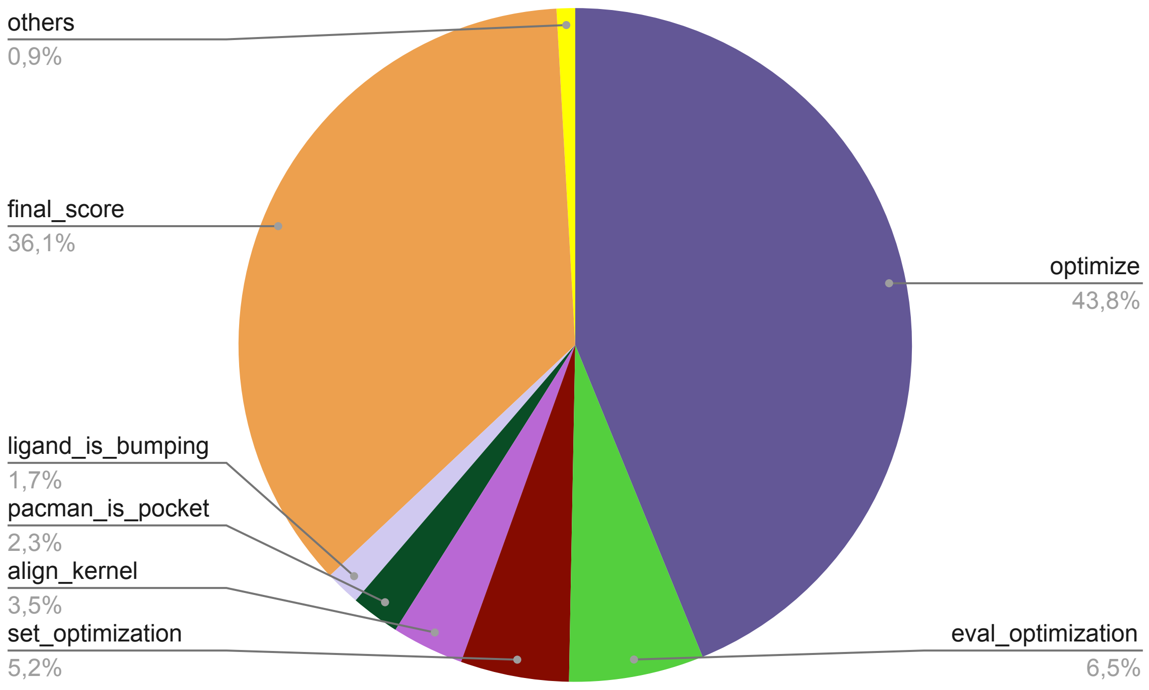


Figure 3.1: CUDA percentage of time spent in each kernel

different percentages of total time spent on each kernel by the implementations. The smallest kernels are grouped in the *others* section due to their trivial impact on the total running time of the application.

	CUDA	**DPC++**	**hipSYCL**
optimize	43,83	63,24	59,43
eval optimization	6,47	7,16	4,92
set optimization	5,21	6,64	4,41
align kernel	3,46	5,09	4,55
pacman is pocket	2,34	2,79	2,29
ligand is bumping	1,69	2,73	1,44
final score	36,12	1,53	22,11
others	0,89	3,6	0,85

Table 3.1: Difference in time spent in each kernel by different implementations

Methodology

How to profile the application is different and platform-dependent. On NVIDIA's GPUs, I could leverage the tools provided by the *CUDA toolkit* also on the application produced by SYCL with every compiler. On AMD's GPUs, the choice of tools is much more restricted, but also, in this case, SYCL code can be analyzed by exploiting existing tools that are part of the ROCm suite. Being able to use legacy tools is a critical point of this standard since it does not provide any additional tool and profile HPC applications is critical to achieve the best possible performance.

To get valuable data the application runs ten times and I considered the average results. Since the processing time depends on the size and the type of the target molecule, I managed to run the code on a variety of molecules. However, for simplicity the results here are only on the Raloxifene one. It is one of the biggest and more testing ones. The times presented for DPC++ are divided in two because, after testing the application, reading and searching deeply in the GitHub repository and documentation, I found that it has some problems with proper inlining of function calls. This problem is fixed by adding `-fgpu-inline-threshold=n` to the compiler flags, with n big enough, in my case equal to one thousand. In table 3.1, the numbers related to DPC++ are those obtained using this flag. This detail is very fundamental as tables and plots demonstrate, that changes drastically the performance results.

Machines and Compiler used

To test properly this application, I was able to use two different machines provided by Politecnico di Milano and by CINECA (see table 3.2). The former lets me use a server running Ubuntu 20.04LTS and equipped with an AMD Ryzen 5900X as CPU and two NVIDIA A100 as GPUs. It has been used to test all the SYCL applications and to have a solid starting point and comparable performance of the native code.

The latter provided a machine having RHEL 4.3 as operating system with two AMD EPYC 1600 processor and four AMD MI100 GPUs. Even if the number of available GPUs was different, it doesn't affect the outcome since the application is tuned to use only one GPU at a time. All available compilers can produce code for every configuration used, even though it is not stated clearly in their documentation.

	NVIDIA	AMD
CPU	AMD Ryzen 5900X	AMD EPYC 1600
GPU	NVIDIA A100	AMD MI100
Operating System	Ubuntu 20.04 LTS	RHEL 4.3

Table 3.2: Machine used for testing

I used three different compiler. For CUDA, I used the *nvcc* version 11.4 that comes with corresponding toolkit. On AMD as base for SYCL I used ROCM 4.2. For SYCL, I used two different compiler both on NVIDIA and AMD: INTEL/LLVM, that is commonly referred to as *DPC++*, and *hipSYCL*. For those last two compiler is also important point out which commit were used. On AMD platform I used the commit of the 08/02/2022 for hipSYCL and for DPC++I used the first commit available with the functions I needed guaranteeing the code porting.

The compilation for the available hardware needs different flags that depend upon the platform:

```
//NVIDIA DPC++ compiler flags
-O3 -march=native -mtune=native -fsycl -fsycl-unnamed-lambda -fsycl-
    targets=nvptx64-nvidia-cuda -Xsycl-target-backend '--cuda-gpu-arch=
    sm_75' -Xcuda-ptxas "--marxrregcount=64" -fgpu-inline-threshold=1000
//NVIDIA hipSYCL compiler flags
-O3 -DNDEBUG --hipsycl-gpu-arch=sm_75 --hipsycl-targets=cuda:sm_75
```

Listing 3.1: SYCL NVIDIA specific compiler flags

```
//AMD DPC++ compiler flags
-O3 -march=native -mtune=native -fsycl -fsycl-unnamed-lambda --fsycl-
    targets=amdgcn-amd-amdhsa -Xsycl-target-backend --offload-arch=gfx908
     -fgpu-inline-threshold=1000
//AMD hipSYCL compiler flags
-O3 -DNDEBUG --hipsycl-gpu-arch=gfx908 --hipsycl-targets=hip:gfx908
```

Listing 3.2: SYCL AMD specific compiler flags

The flags are different from each other and depend on the platform. The only other relevant difference is inside the example related to NVIDIA, where hipSYCL supports only architectures up to sm_75 because it doesn't support the latest CUDA toolkit yet.

3.2. Small kernels: SYCL vs CUDA

The first category of kernels that I am going to analyze is the one included in *others* in plot 3.1. This kernels are small and trivial to convert from CUDA to SYCL.

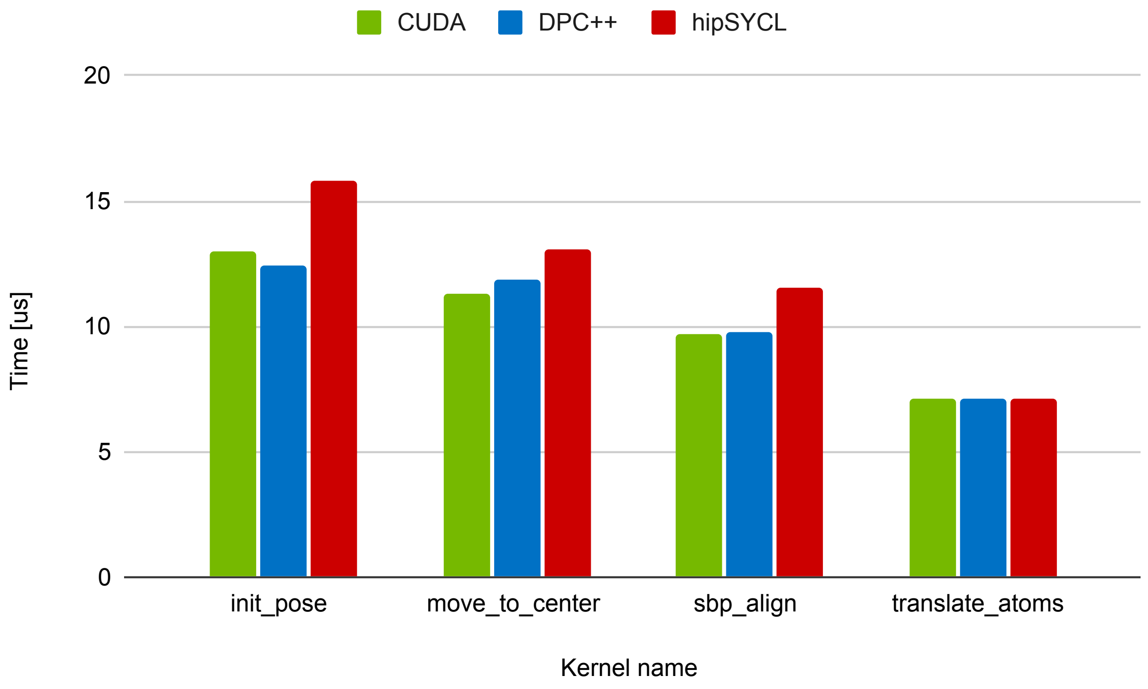


Figure 3.2: SYCLvsCUDA performance comparison on small kernels

As plot 3.2 shows, performances achieved on the NVIDIA A100 by SYCL are very close and sometimes better than native ones, even though there is a slight difference between DPC++ and hipSYCL. It is noticeable that DPC++ performs better than hipSYCL in almost every kernel. The only exception is *translate atoms* which is the simpler kernel of the group.

Focusing on the CUDA code, the runtimes achieved by *DPC++* are very encouraging, and we can consider them to be equals.

The shown results are aligned with those obtained by other studies and presented in section 1.6.3. It is not a surprise that small kernels behave very similarly to benchmark test code, even though these times are a little bit more probing than trivial code used for benchmarks.

The performance times on AMD for these kernels are not included here because the available time on the CINECA machine was limited and I couldn't complete the study. I focused to run complete test with different compiler to analyze kernels that have a bigger contribution to the overall time.

3.3. Main Time Consuming Kernels

In plot 3.1, we can identify five different kernels that account for the 70% of total computation time. In this section, I will address the performance achieved in these kernels. I start evaluating them in single precision and then comparing the results in double precision. Kernels are divided into two sub-groups to provide a better view of the graphics and a more relevant analysis. Starting with their comparison with CUDA on NVIDIA hardware and then comparing them on AMD hardware.

3.3.1. SYCL vs CUDA on NVIDIA GPUs

The first group to analyze is composed of three kernels: align_kernel, pacman_is_pocket, and ligand_is_bumping. The first one is the most compute-intensive, the second one implies multiple algorithms inside the same kernel, and the last one is a fairly simple but big and diverse kernel. All of them take care of important phases of the application related to specific chemical properties to respect.

The 3.3 plot is very important for SYCL performance, particularly DPC++, because it shows how close SYCL is to CUDA. The *ligand_is_bumping* kernel shows that all the versions of SYCL are competitive with native code. It is important to notice that this is the only case in which hipSYCL provides better performance than everyone else. There are two distinctive characteristics to point out about the other kernels. In the align_kernel and pacman_is_pocket, *hipSYCL* results are too far to be competitive against CUDA, but also it is too far also from *DPC++* in both version tested. This result is very disappointing because it shows the limits of the converted code. hipSYCL is not capable of optimizing complex and compute-intensive code on GPU while using single-precision numbers.

On the other hand, the results are very encouraging for *DPC++* when using single precision. In all three cases, it shows that it is equal to or better than CUDA, even closer to it if developers enable better inlining of the code. That flag shows already its effect in `pacman_is_pocket` kernel, in which the run time is cut by a 1/3. Showing that DPC++ is an alternative to CUDA even when calling particular functions or algorithms inside the kernel code.

The original application was however using double-precision to get more precise results from a chemical point of view. It also enables the possibility to test better if the results are

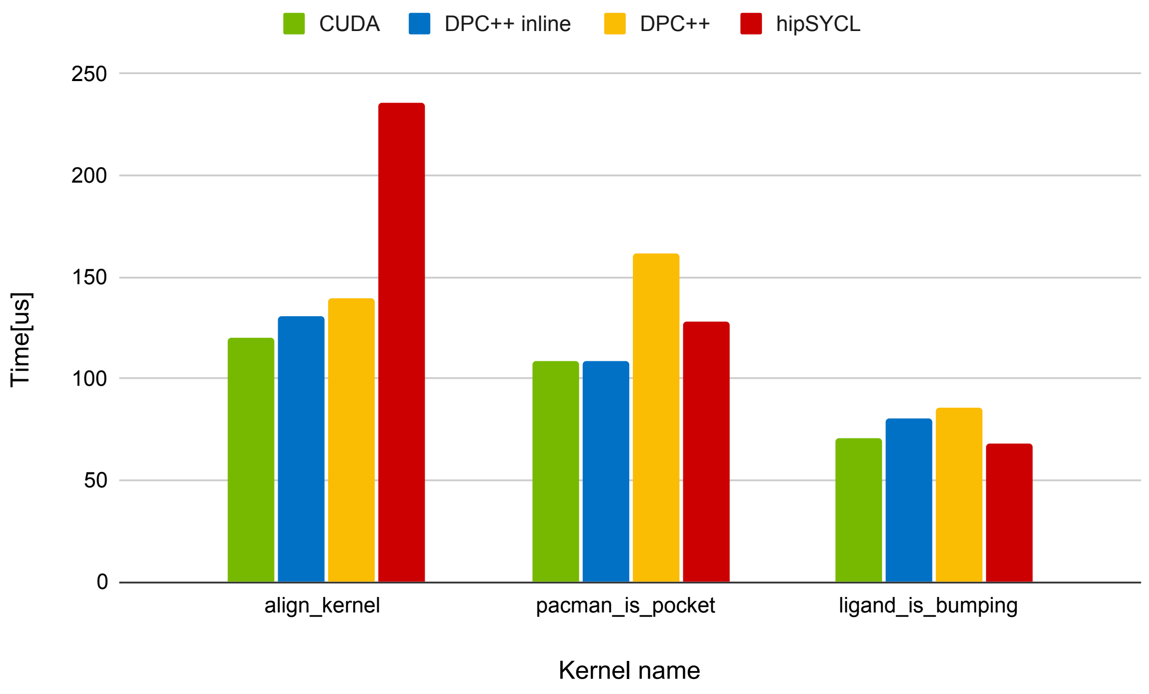


Figure 3.3: SYCLvsCUDA performance comparison on three big kernels using single precision

equal to those obtained on the CPU. Plot 3.4 illustrates what this design choice implies at the performance level.

Before talking about run time, it is compulsory to highlight the workaround needed to make DPC++ run with double precision: the number of registers allowed for each group is blocked at 64. Otherwise, it goes out of resources because the compiler cannot schedule the code in other ways. This workaround is not necessary for hipSYCL.
Changing the precision of the computation lets us discover that CUDA and hipSYCL can obtain almost the same level of time spent in each kernel. Therefore switching from one precision to another has no particular effect, and the analysis above is still valid. On the other hand, for DPC++ the results were quite astonishing and strange before discovering the inlining problem. The drop in performance was not acceptable because the time of execution of these kernels is too far from CUDA and hipSYCL. It also doesn't make sense if compared with what it can achieve in single precision. In particular, if looking at the *align_kernel*. Force a better inline to the DPC++ compiler brings the result to another level and shows the real quality and performance portability achievable with SYCL, as it is clear from 3.4.

Giving a deeper look at the numbers, the *ligand_is_ bumping* kernel is the one where

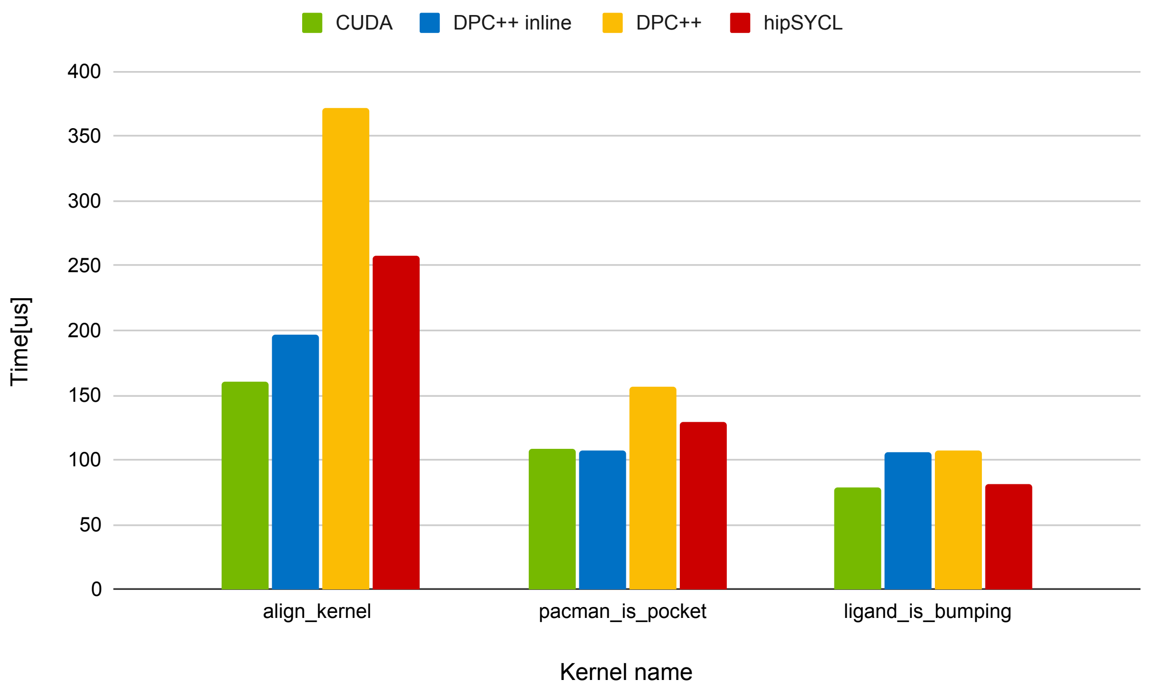


Figure 3.4: SYCLvsCUDA performance comparison on three big kernels using double precision

hipSYCL still has the best result, and it is also the one where DPC++ has the largest loss with respect to single-precision. It may be caused by the limited usage of registers necessary to make the application available. Using NVIDIA Nsight Compute, I was able to analyze the number of registers used by it and, if let the compiler schedule operation as it prefers, it would use 72 registers for this kernel. It would speed it up and make it closer to native code results.

The inlining implementation has a huge effect on the other two kernels. For *pacman_ is_ pocket*, it brings its timing lower than CUDA as it was for the single-precision case. It is linked to how it is written the kernel and how it uses external functions for these reasons better inlining benefits the most this part. For the *align_ kernel*, the new performances are astonishing. A simple flag to force the compiler to do something that should be common brought the performance down by almost 50%. It is the best SYCL implementation, it is very close to CUDA even if with this type of precision, it still loses about 20% of time compared to it. The nature of this difference is given by how the compiler schedules the operations. Using NVIDIA Nsight Compute it is possible to view how the assembly operations are ordered, and the two codes use very different optimizations. The SYCL code loads all the possible data from memory to register way before it is necessary, with the idea that memory loading during computation would make the computation unit wait for data, which is the most common and correct way of working on CPU. NVIDIA compiler instead loads some data and starts the computation using a lower number of registers more effectively, allowing the code to run faster, but also without causing problems with hardware limitation as SYCL does in the *DPC++* case.

The last three kernels are equal to 55.5% of the GPU computing time. I analyze these kernels altogether because they are all part of the same chemical phase of the algorithm, and they are launched from inside a loop. This particular condition may affect performances in some cases, as I will show in Figure 3.7.

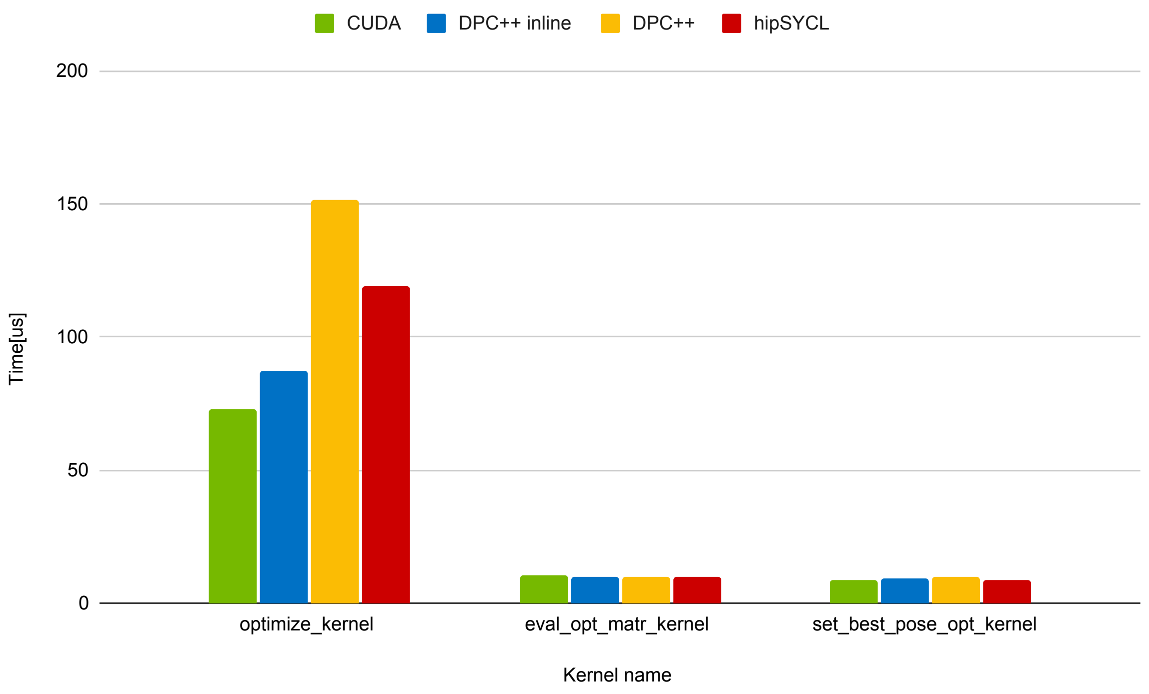


Figure 3.5: SYCLvsCUDA performance comparison on three big kernels in a loop using double precision

In this case, the differences between single-precision and double-precision are important if looking at the numbers, but it doesn't change anything in terms of relations among the frameworks. Figure 3.5 shows the results for double-precision, and it can be taken to account even in the analysis relative to single precision. The plot illustrates the time for a single run of the kernels, as said above, this code is executed multiple times. For this reason, the total time of execution is much more relevant than the single run, and tiny differences in time may add up to become substantial. In this case, it is possible to retrieve differences between the frameworks also from a single run.

The two smallest kernels, once again, favor SYCL performance against native code on every compiler, even if the code is not optimized. *DPC++* and *hipSYCL* have no problem at all with them but the situation is different in the *optimize_kernel*. In this case, the presence of a lot of different branches causes a lot of problems for hipSYCL, which is not capable of producing code that is comparable with CUDA reaching a slow down of about 60% on a single kernel. This loss is not acceptable in a high-performance environment. It is even less acceptable if the kernel is inside a loop the performance degradation is too much. On a different level is the DPC++ code that can provide very solid results compared to CUDA, once overcame the difficulties in inlining the code. It is not yet at native-code level, it loses 15/20% on a single run, but in this case, it is about 10/15us, a more than acceptable loss.

The differences in many kernels are various. In some of them, DPC++ performs better than CUDA. In others, DPC++ is at the same level, and in three particular cases, DPC++ is far from CUDA times, attesting about a 20% loss when using double-precision, while in single-precision the difference is even less. Table 3.3 allows understanding better which kernel and which time I refer to, considering the double-precision execution times.

	CUDA	**DPC++**
align kernel	160,543 us	196,13 us
optimize kernel	72,703 us	87,11 us
ligand is bumping	78,304 us	105,38 us

Table 3.3: Time of kernels that show the most difference from native code in double precision

Investigating those kernels is possible to have a better understanding of which particular part of the code or characteristic is the more difficult for SYCL to compile and, probably, which are its current limits. These three kernels have common traits. For example, in the first two functions in the table 3.3, the CUDA implementation uses the texture memory, which was removed during the porting, and both conclude their operation with a final reduction heavily optimized in CUDA, but that is not possible to optimize in the same way in SYCL. Another common trait of these kernels is the use of specific low-level functions that use group algorithms and that are provided by the framework. The last similarity is the number of branches included in these kernels. In particular for the last two examples, every time there is a branch in kernel code, it can be a bottleneck because it is an issue for GPU architecture, and it is up to the compiler to handle them carefully.

In my analysis, I consider that the usage of texture memory does not have a particular impact on the results, because it is not used in future implementations. On the other hand, the possibility of using a more granular specification of threads inside the warp, utilizing a bitmask for selection, is the key component that causes a lot of that time loss. The usage of reductions and group algorithms doesn't seem to have any impact on the other kernels. So it is safe to assume that doesn't affect these. On the branching factor, I can assume that is one of the most impactful because the two kernels that show the biggest loss has many branches.

NVIDIA tools allows better analysis, in which has emerged differences that are not in the hand of developers. The *align_kernel* performance is impacted by the limitation of registers that DPC++ doesn't handle as well as *nvcc*. These kinds of effects depend on the compiler architects. The last kernel is the one that has the worst time lost when changing from single to double-precision, besides having the worst time comparison to CUDA in any case. The particularly bad result, in this case, is due to the necessity of the *align_kernel* to limit the number of registers which impacts this kernel a lot more. The register's limitation impacts a lot of the way the SYCL compiler handles branches since both kernels would like to use 72 registers instead of 64, but this is not possible at this moment.

For reference, it is interesting to give a look at the numbers of these kernels in single precision to better understand the damage of register limitation.

	CUDA	**DPC++**
align kernel	120,13 us	130,56 us
optimize kernel	62,84 us	72,03 us
ligand is bumping	71,1 us	80,74 us

Table 3.4: Time of kernels that show the most difference from native code in single precision

The difference in time goes from 10us in single precision to 36us in double precision for the first kernel and from 9us to 27us for the last one. Besides register limitation there aren't any other changes.

3.3.2. SYCL on AMD GPUs

The tests on AMD hardware were more difficult and limited in time, for two main reasons. First of all, the tool available was more limited than NVIDIA ones, and it wasn't properly set on the machine. I had trouble figuring out how to set it and use it with DPC++and hipSYCL.
Moreover, the porting of the application required some time for software limitations and to adapt it in general. I follow the same structure of the previous section, but this time I provide performance related to double precision and only about the six more time-consuming kernels because the time of the others is similar to those represented in Figure 3.2.

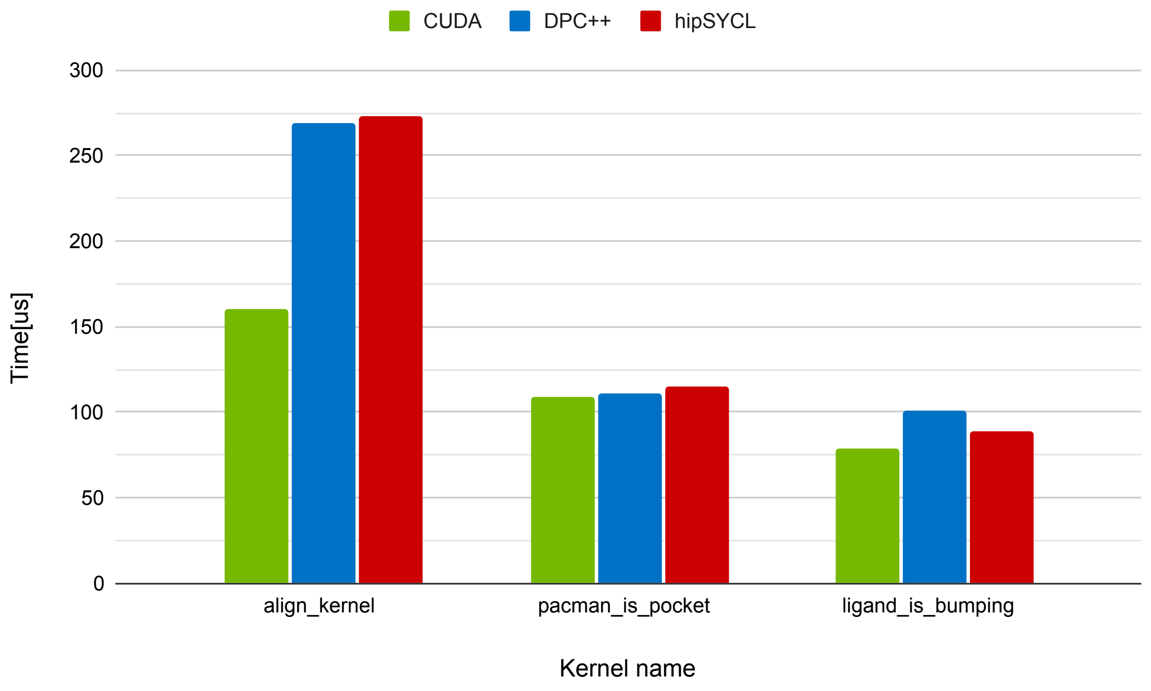


Figure 3.6: SYCL performance comparison of three kernels using double precision on AMD MI100 against CUDA related performance on NVIDIA A100

The results obtained in the first kernels illustrated by Figure 3.6 have contrasting results. On one side the kernel in the center and the one on the right present a really good outcomes compared to the CUDA performance and it allows to think about switching from one platform to another. This level of performance is obtained in that code that didn't require a lot of effort to compile and run correctly on the new hardware. This is an interesting point because if the code is thought from the beginning taking into account

the underlying hardware or being adaptable to the hardware without particular attention or modification, it performs at the same level as native code. In particular, on AMD *hipSYCL* seems closer to CUDA that *DPC++*.
On the other hand, there is a massive loss of performance in the first kernel which is the most compute-intensive one. The real difference in runtime between the first one and the other two is not in the nature of the kernel but is in the discrepancies between the SYCL and the native code. It was designed to exploit at the hardware level the way NVIDIA's GPUs work. During the porting, I had to make some adjustments in form of many branches and different memory accesses. The changes let the code work and provide the correct computations but also affect the required runtime. This is very clear given the numbers displayed in Figure3.6. My opinion is that during the porting this type of kernel should be re-designed for AMD hardware and maybe also for NVIDIA to reduce the general gap from native code because losing about 68% of performance is not acceptable. This decision, however, would limit the single code run everywhere opportunity provided by SYCL.

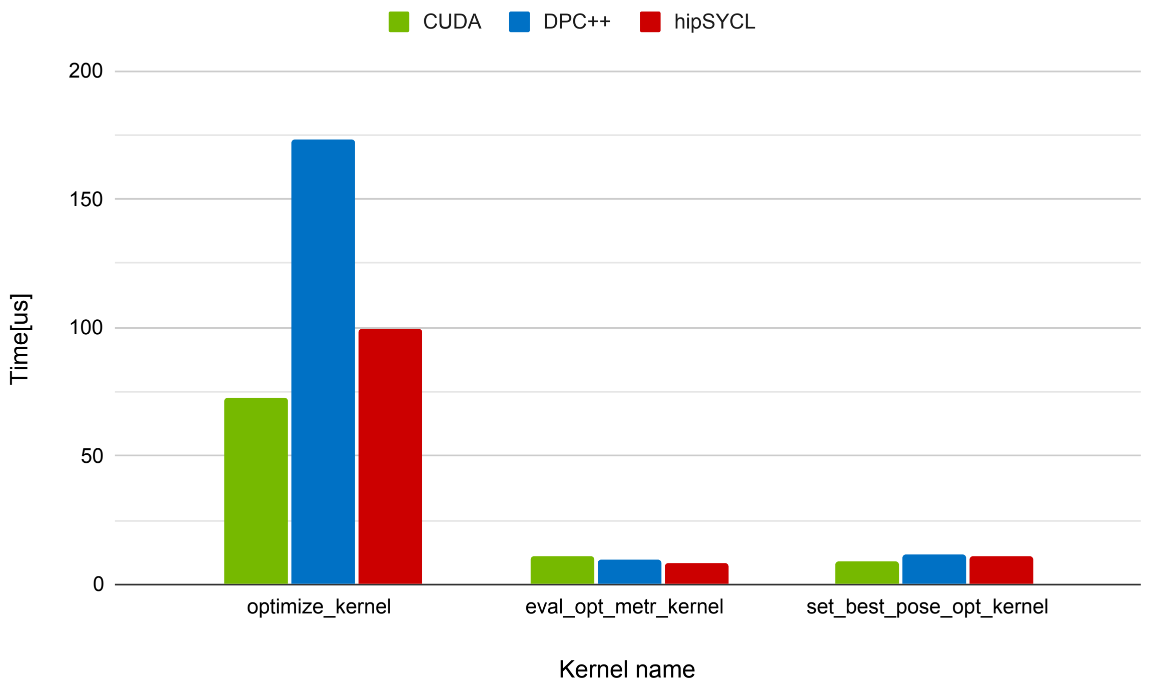


Figure 3.7: SYCL performance comparison of three kernels in a loop using double precision on AMD MI100 against CUDA related performance on NVIDIA A100

The second part of kernels maintains the same line as the first with one more peculiarity. The smallest kernels keep providing good reasons to use SYCL. The performance is the same as native code on CUDA, which means it is the same as SYCL on NVIDIA hardware with the same code. As the benchmark shows performances and code are portable within this situation. The *optimize_kernel* shows all the limits of this code and the porting of this application. Looking at the column relatives to DPC++is incredible to see how much execution time is necessary to respect CUDA and hipSYCL. To have the application run correctly, the changes in the code necessary were a lot and modified how the kernel is composed by adding another inner loop in it. This expansion is not well interpreted by DPC++, and the compiler provides code that is not in line with the expectation. In this case, hipSYCL produces code that is closer to the performance of native code on CUDA. Probably, kernels launched inside a loop have a bigger effect on the code compiled by DPC++ than hipSYCL because they use different intermediate representations.

The numbers achieved on AMD in the two more complex kernel are probably due to some code that needed to be changed during the porting, and that is not the best way to port code from one platform to another. When the code is complex and designed with only one platform in mind, it needs to be re-engineered, while simple operations do not suffer from hardware differences.

The application was tested against CUDA code to verify if the SYCL version may have a good performance on AMD that could lead researchers to change platform. Writing SYCL code that runs very well on NVIDIA hardware but not having the possibility of changing the machine is useless, it is better to keep using CUDA. It is not necessary to test it against ROCm native code because this is not the starting point of many research centers, as in this case. The idea is to bring the CUDA native molecule docking application to other hardware and exploit its performance.

3.4. Portability vs Performance

In the design choices made at the beginning of the work, I decided to implement a SYCL version of the code as close as possible to the underlining hardware with the hope of achieving a similar level of performance. As already stated in section 2.1.4, this is not the only viable way. Thanks to the European Project **LIGATE** [25], I was able to test the same application written in a portable way by the University of Salerno. I refer to it as *portable version* because of the design of choice of having a version that not only runs on GPUs but also CPUs exploiting the same SYCL code. In addition to it, this version of the code resulted to be also more portable to different vendor hardware in the switch from NVIDIA to AMD GPUs. The first noticeable shortcoming of it is the possibility to be compiled only using *hipSYCL* and not with *dpc++*, that in the above section resulted in being the most performant of the two but it may be related to the porting of the code and not to the compilers. The machines used for testing are the same mentioned in section 3.1. In the plots, I will show the differences among kernels concerning the other two SYCL versions, the best DPC++ version and the hipSYCL one, but also I will always leave CUDA native code as a comparison for the required times.

Different from the previous sections, I present results obtained only with double precision since those are the most relevant. I split the kernels into those that have performances closer to the others and kernels that are far from being considered at the same level with an analysis of what may cause such differences.

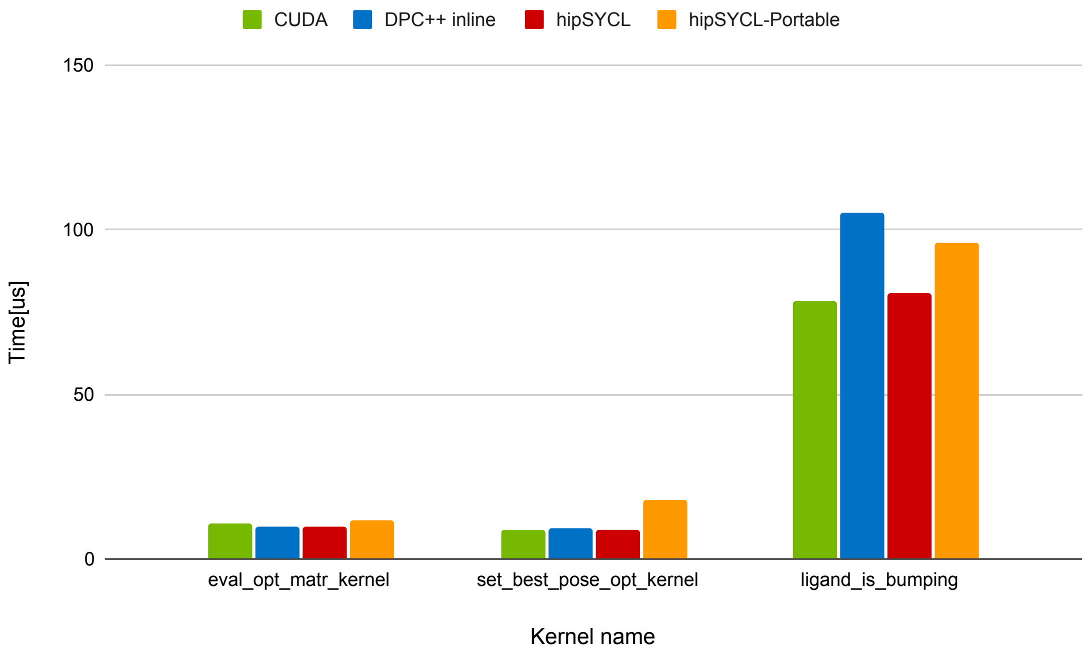


Figure 3.8: Comparison of SYCL performance with different approach on NVIDIA A100, selected kernels with close results

Figure 3.8 presents those kernels that don't suffer significant performance loss if written in a hardware-agnostic way, but all of them lose if compared with other versions. The only case of better performance is achieved in *ligand_ is_ bumping* against DPC++, that in this kernel suffers from the enforced register limitation. In any case, against the same compiler(hipSYCL) with different code, this version is behind. It seems that on small kernels and in those kernels with a good amount of branching and group function the results are close enough to the other version. In this case, the difference can be considered bearable because the code can run on many platforms.

On the other side in 3.9 the outcome presented is completely different. The performances

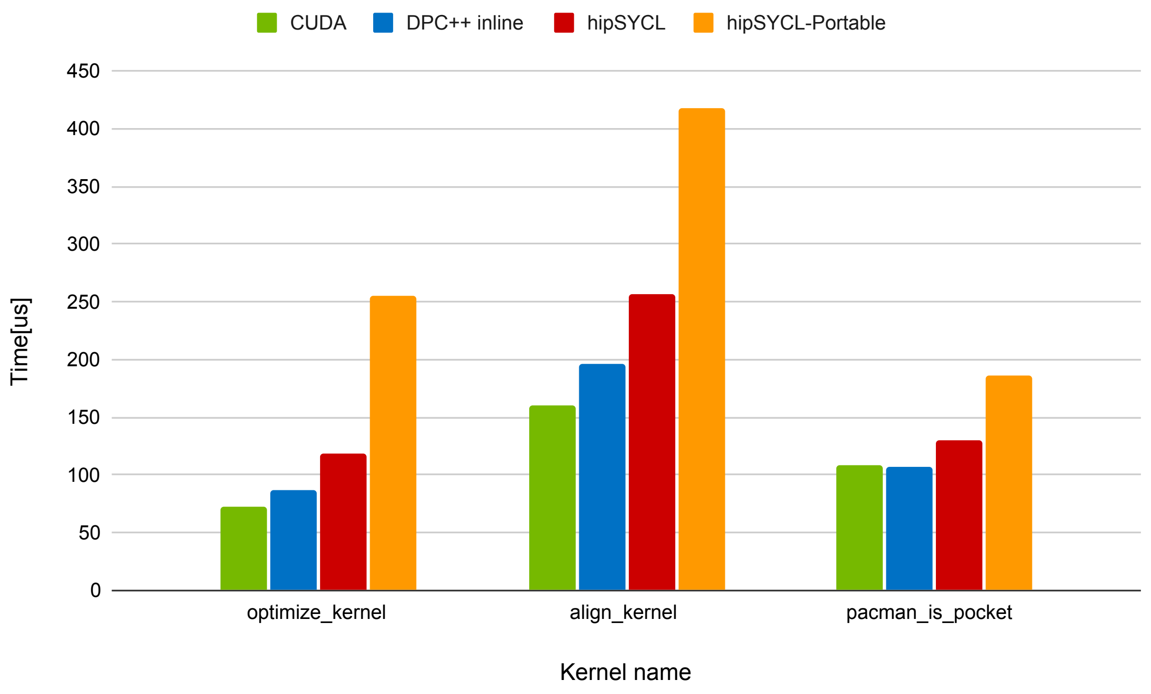


Figure 3.9: Comparison of SYCL performance with different approach on NVIDIA A100, selected kernels with very bad results

achieved in this case are too far from anything that can be considered fine in an HPC scenario. The loss in *align_kernel* is close to 160% of the original CUDA application time and 112% of the DPC++ time. The situation is the same on the average time of *optimization kernel* with a time achieved of about 2.5x against CUDA and about 1.9x against DPC++ times, versus the same compiler, it performs respectively 60% slower and loses 1.19x of performance. These results are not acceptable and denote how targeting specific hardware brings many improvements that are not obtainable if the code is hardware agnostic.

3.4.1. Portable SYCL on AMD GPUs

Portability allows the same code to run on AMD hardware without any particular effort besides those related to the immaturity of the compiler on this platform. The performance is compared in the same way as the previous section on the NVIDIA machine.

The first kernels compared are those that produce the most encouraging results. In fact, in figure 3.10 the times are similar to those obtained by SYCL on this kind of hardware, even though still behind hipSYCL on the same hardware in every case. On small kernels, the

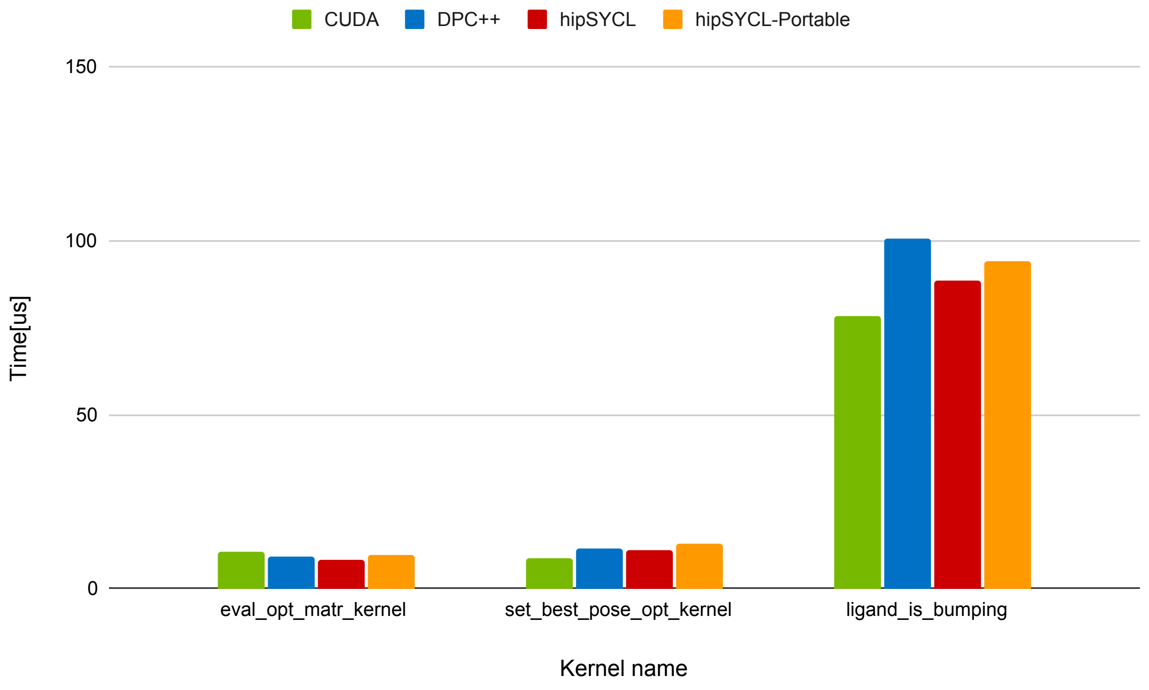


Figure 3.10: Comparison of SYCL performance with different approach on AMD MI100, selected kernels with results close to NVIDIA

situation doesn't change and I can confirm the results obtained by standard benchmarks. Moving on to bigger and more complex kernels, as those presented in 3.11, run time goes up unpredictably. The code is slower, much slower, than the counterpart more optimized for GPU, and not even close to the level of native code on NVIDIA.

In particular in Figure 3.11 is incredibly out of scale the result for *pacman_is_pocket* is almost three times worst than CUDA and SYCL since the other implementation on AMD is very close to CUDA.

The reason for these huge performance drops can be found in how the group algorithms are called and implemented. In SYCL, it is possible to utilize group-related algorithms such as reduction and voting mechanism on group/block of the GPU and not strictly on warp/sub_groups. In this way, the code can be compiled and be effective on different platforms. In these four kernels, not using the warp mechanism has this impact, and it changes a lot the results. The way these functions are implemented is hidden from the developers. It is up to them to look deeper into the code and understand better the situation to avoid such pitiful performance.

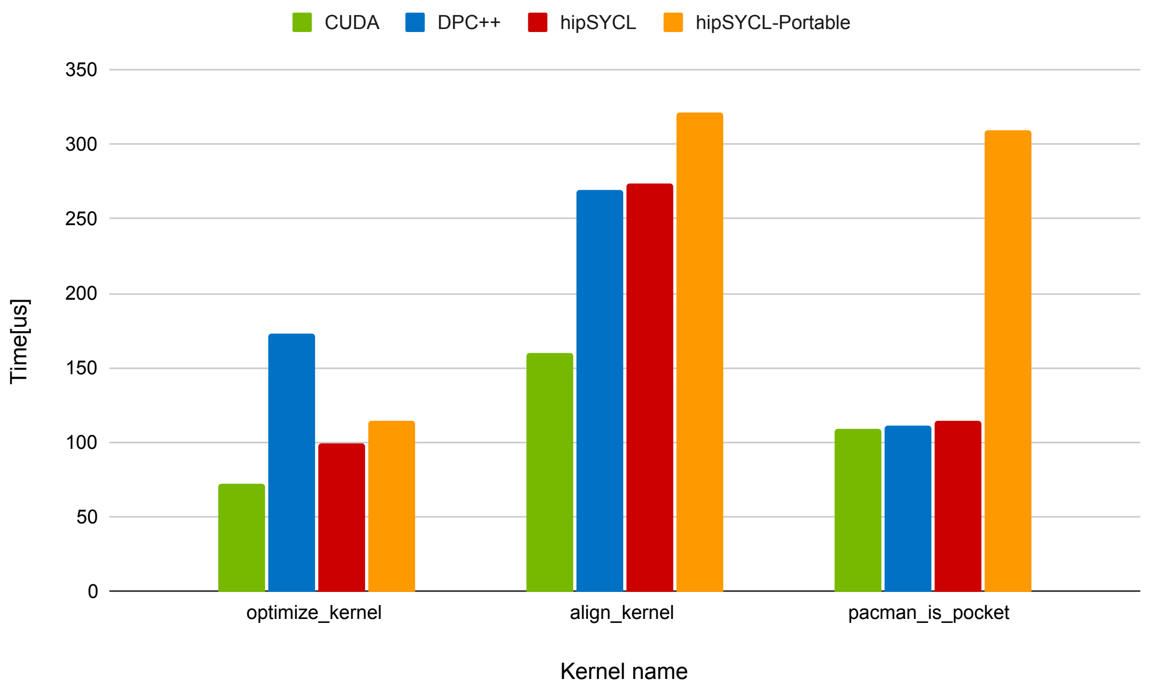
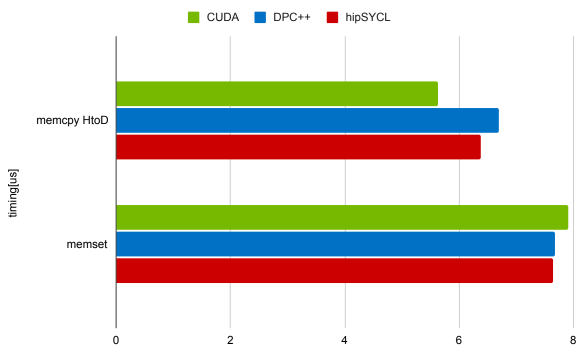


Figure 3.11: Comparison of SYCL performance with different approach on AMD MI100, selected kernels with very bad results

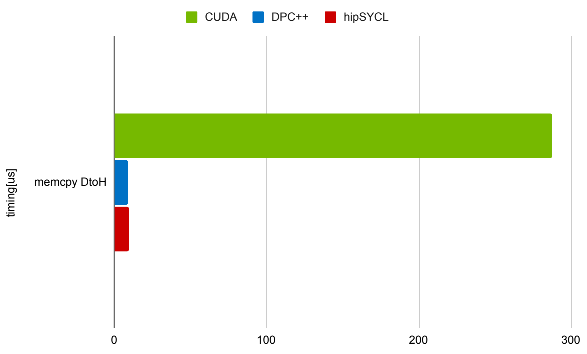
3.5. Memory Bandwidth comparison

To conclude the comparison and the measurement of SYCL performance it is necessary to look at memory bandwidth and throughput.
The test methodology is the same described above and the results are taken with the NVIDIA tools. Unluckily, I wasn't able to test and retrieve these data on CINECA's machine, so I cannot make a comparison with AMD results.
On the NVIDIA platform, it's possible to appreciate the work done by SYCL developers. In Figures 3.12a and 3.12b there are the memory operations that characterize the transfers and the settings necessary to move data across heterogeneous systems. These operations are key to obtaining good performance during the computation because their work is to set up memory with the wanted data before the computation takes place, such that the compute unit doesn't have to wait. In SYCL, the time reached is at worst close to CUDA one, in fact, in two out of three operations it has better results. In host-to-device copy, CUDA has better performance but it gets absolutely the worst in device-to-host movements, where SYCL is the best by far. This last example could be false by the way the last scoring algorithm is implemented in the two different frameworks.
These results are very encouraging for the future of SYCL because both compilers pro-

vide similar performance, so the goodness of the result is not related to some particular expertise, but to the framework itself.



(a) SYCL vs CUDA time for memory transfer device to host



(b) SYCL vs CUDA time for memory transfer

Figure 3.12: SYCL vs CUDA time of memory operations

Another point were memory allocations is very important is the initial set up. To check if setting up the memory and if the work-around on the texture memory and different methods of implementations have a meaningful impact on the total time, I measured the time taken by the constructors to allocate memory because there aren't kernels to profile. I wrapped the constructor with timers using the `chrono::high_resolution_clock` and compared the results of a series of ten runs of the algorithm both for CUDA and SYCL. I run ten different tests on single molecules and files with multiple molecules.

Figure 3.13 shows the difference in the average time of the test done. Times on the left are in microseconds and the standard deviation for SYCL. In this case, *DPC++*, is 7153.53 us and for *CUDA* is 3709.25 us. As the plots show there is no significant difference in the two constructors. SYCL achieves performances that are a little bit better than CUDA but has a higher standard deviation.

Overall I can say that the changes made in the porting of the constructor are not relevant in this phase, and the missing implementation of *image* is not a problem when considering memory performance. SYCL can achieve even better results than CUDA in this particular case, and its *usm* implementation is very competitive even if at an early stage. To be fair, the usage of texture memory and its characteristics have been removed from the next version of the code because it wasn't necessary, and doesn't guarantee better memory performance.

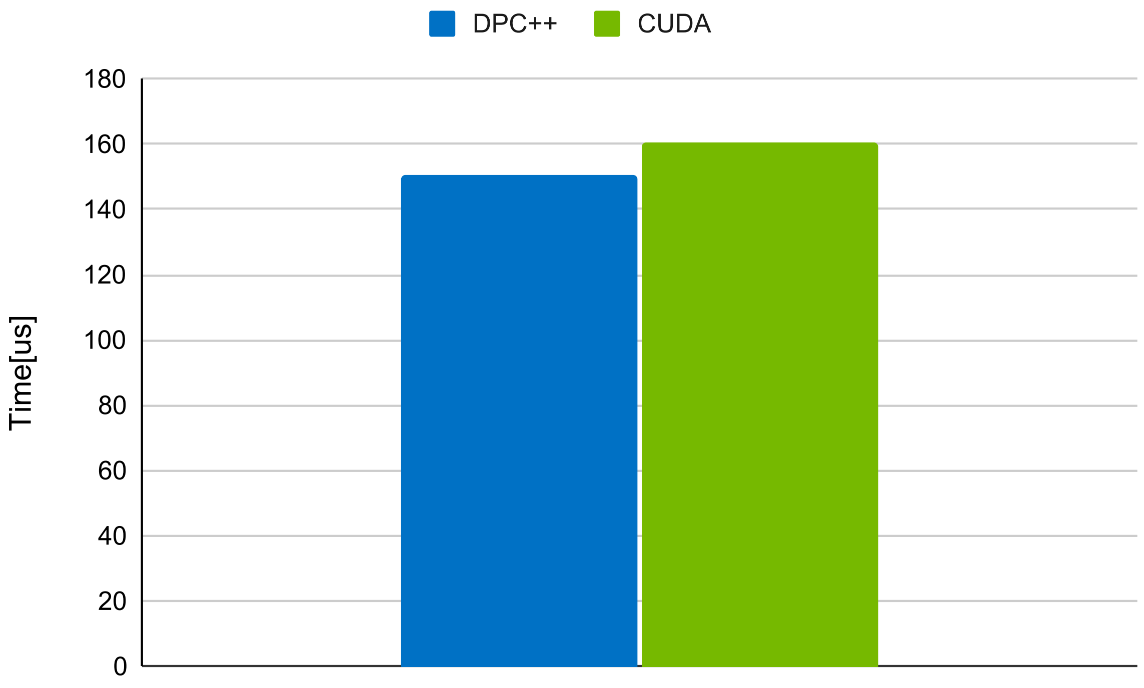


Figure 3.13: Time comparison of setting up memory

4 | Conclusions and future developments

The research presented aims to understand whether the SYCL framework can really empower engineers to write portable code in one single language across different hardware/accelerators and if it can reach the same performance as proprietary native languages. The analysis was done by converting one *molecular docking* application from CUDA to SYCL and studying portability and performance on NVIDIA and AMD GPUs.
This chapter summarizes the results obtained with my work, dividing the consideration between portability and performance. To conclude, it points out some future development, and possible work for the continuation of my research and for SYCL in general.

4.1. Observations

The following considerations must take into account that SYCL is at early stage of development in all its implementations. Results obtained are evaluated and referred to with this specific vision.

4.1.1. Portability

In general, for what concerns the portability of applications from CUDA to SYCL, I am confident in saying that it is a reality that can be safely explored by researchers. In almost any case, the code can be converted, and only rarely it requires a work-around such as those I presented in section 2.2 and in particular for some kernels such as in section 2.2.4, but for all the others it is straightforward. More importantly, if the technologies are not yet implemented in SYCL often are not necessary and it leads to rewriting the code in a better way as it happened for section 2.2.6.
Nonetheless SYCL is a major step forward to portability, my application can run on NVIDIA and AMD hardware with minor tweaks (see 3.4), that in my opinion will always be necessary, because the hardware is different and performance are extracted only if the

engineers know it, but the possibility of write the code in one single language,compile it and have it available on different hardware is a major step forward.
In the spectrum of portable language, *OpenCL* is the only one that can offers the same level of flexibility as CUDA, but now SYCL overcomes many of OpenCL problem, and besides its problem linked to the early staged of the evolution, it is by far easier to program in it. To conclude, it is clear from my work that SYCL is ready to be use as high level language for heterogeneous systems, substituting OpenCL without causing any problem related to programmability and portability going very close to common general modern C++.

4.1.2. Performance

A big part of my work was to assess if SYCL is capable of providing performances that are at least comparable with those of native code without considering which SYCL implementation is used. At the beginning of my study, it didn't seem possible since the first results obtained analyzing kernels weren't close to the CUDA ones. During this time happened a couple of things that changed the outcome.

The first one, and probably the most important, is that compilers improved quickly. The work that companies and the community have been putting into it is very encouraging for future possibilities and adoptions of this new framework. Secondly, I started to understand better the SYCL library and how GPUs hardware works, it brought my code closer to the native one and it allowed me to exploit better the SYCL library and the hardware. Understanding how to exploit the hardware available and discovering important compiler flags such as `-fgpu-inline-threshold` brought my code to a completely new level. The performances shown in section 3 are the results of these evolutions in the compiler and my knowledge. It shows how the SYCL kernels are very close to the CUDA ones especially if compiled with DPC++ with the best inlining possible, so I can safely say that SYCL is ready to be used for heterogeneous systems, not only for its great portability and programmability but also for the performance that is can bring to the software. Results obtained don't come for free, but developers must be willing to study the hardware and adapt algorithms to the new platform because otherwise, the outcome is terrible.

Section 3 shows also that complete portability of performance is not here yet. In fact on two kernels, the times obtained by DPC++ on AMD hardware are not close to those on NVIDIA, and even if hipSYCL is promising, it cannot be the only compiler that takes seriously AMD hardware. Luckily for many kernels, it is possible to obtain the same performance with the same code. Moreover, considering the traction that AMD hardware is having, I am sure that in the next months DPC++ will provide better compilation for AMD GPUs. On the positive side, if researchers redesign the code to be more adapted to AMD GPUs architecture, the performance gap can be closed more easily.

To summarize, SYCL is ready to be used in HPC environment and allows exploiting GPUs as native code. It is neither perfect nor magical, and maximum performances are achieved if developers take care of tuning the code to the hardware. Right now, I think that the best way to use it is to write common kernels for the easiest one, and for the more complex task use *preprocessing macros* to select the best kernel for the architectures on which the software will be deployed. In this way, the same application can run on different supercomputers without issues and with comparable performance.

4.2. Future Work

The possibility of further work on this topic and this thesis are many. The first one to explore is to bring the new version of this application from CUDA. It has been completely redesigned, and the native code can achieve even better performance. Thanks to its new characteristics, it seems to be easier to bring to SYCL. It is the first thing to do because further work on the same code base would be useless. During the porting from CUDA, I would focus on trying to write code and algorithms that are easy to implement on AMD machines and that this other hardware can exploit properly, or at least two comparable versions that obtain similar runtime. For sure, there is the necessity to use SYCL on AMD GPUs and provide feedback to the developer of DPC++ and hipSYCL to let them improve compilers but also to gain confidence with that hardware.

Another possibility to expand the work is to test the application on the new and upcoming *Intel GPUs* that are not yet available to the general public, so I couldn't use them. The opportunity to use SYCL and especially *OneAPI* would be fantastic, and it would show the real possibility of this language since OneAPI is the toolkit provided by Intel. *DPC++* is only a little part of the complete suite of software that Intel makes, which comprehends a full framework similar to the NVIDIA ones. It would be an occasion to compare the toolkit of two big companies. Writing an application that runs on hardware that natively supports SYCL would also be a great way to test the language, and understand what kind of performance this hardware will be capable of. Bringing the new codebase to this hardware will achieve maximum portability and independence from hardware vendors.
All the above work would be even more important if supported by many innovations and integration by compilers engineers that are working on SYCL. With the hope that idiomatic differences among SYCL implementations will disappear as it seems to me after

these months of work.

Bibliography

[1] top500. Top500, 2020. URL `https://www.top500.org/lists/top500/2021/11/`.

[2] LUMI official sites. Lumi system architecture. URL `https://www.lumi-supercomputer.eu/lumis-full-system-architecture-revealed/`.

[3] AMD. Amd powering exascale. URL `https://www.amd.com/en/products/exascale-era`.

[4] Aurora intel sites. Aurora webpage. URL `https://www.intel.com/content/www/us/en/high-performance-computing/supercomputing/exascale-computing.html`.

[5] Renxiao Wang et al. **SCORE**: A new empirical method for estimating the binding affinity of a protein-ligand complex. 1998. URL `https://www.researchgate.net/publication/ating_the_Binding_Affinity_of_a_Protein-Ligand_Complex`.

[6] Ian Buck. brookgpu website, 2003. URL `https://graphics.stanford.edu/projects/brookgpu/talks.html`.

[7] IanBuck. The history of cuda, 2008. URL `https://www.youtube.com/watch?v=JkvqWe1ZT2w`.

[8] NVIDIA. Cuda toolkit. URL `https://developer.nvidia.com/cuda-toolkit`.

[9] Pradeep Gupta. Cuda refresher: The cuda programming model, 2020. URL `https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/`.

[10] Ian Buck. Gpu computing: Past, present and future, 2011. URL `https://www.youtube.com/watch?v=30X_tqGGgfQ`.

[11] OpenACC foundation site. URL `https://www.openacc.org/`.

[12] Wikipedia. Openacc. URL `https://en.wikipedia.org/wiki/OpenACC#Compiler_support`.

[13] Summit supercomputer website. URL `https://www.olcf.ornl.gov/summit/`.

[14] Khronos Group. Opencl official site, . URL `https://www.khronos.org/opencl/`.

[15] Khronos Group. Opencl official github, . URL `https://github.com/KhronosGroup/OpenCL-Guide`.

[16] Kamran Karimi et al. A performance comparison of cuda and opencl. 2010. URL `https://arxiv.org/abs/1005.2581v3`.

[17] Suejb Memeti et al. Benchmarking opencl, openacc, openmp, and cuda: programming productivity, performance, and energy consumption. 2017. doi: https://doi.org/10.1145/3110355.3110356.

[18] Mikhail Khalilov and Alexey Timoveev. Performance analysis of cuda, openacc and openmp programming models on tesla v100 gpu. *J. Phys.: Conf. Ser. 1740 012056*, 2021. URL `https://doi.org/10.1088/1742-6596/1740/1/012056`.

[19] Deakin T., Price J., Martineau M., and McIntosh-Smith S. Evaluating attainable memory bandwidth of parallel programming models via babelstream. 2017. doi: 10.1504/IJCSE.2017.10011352.

[20] István Z. Reguly. Performance portability of multi-material kernels. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 26–35, 2019. doi: 10.1109/P3HPC49587.2019.00008.

[21] Brian Homerding and John Tramm. Evaluating the performance of the hipsycl toolchain for hpc kernels on nvidia v100 gpus. In *Proceedings of the International Workshop on OpenCL*, IWOCL '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375313. doi: 10.1145/3388333.3388660. URL `https://doi.org/10.1145/3388333.3388660`.

[22] Tom Deakin and Simon McIntosh-Smith. Evaluating the performance of hpc-style sycl applications. In *Proceedings of the International Workshop on OpenCL*, IWOCL '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375313. doi: 10.1145/3388333.3388643. URL `https://doi.org/10.1145/3388333.3388643`.

[23] Emanuele Vitali, Davide Gadioli, Gianluca Palermo, Andrea Beccari, Carlo Cavazzoni, and Cristina Silvano. Exploiting openmp and openacc to accelerate a geometric approach to molecular docking in heterogeneous hpc nodes. *The Journal of Supercomputing*, 75, 07 2019. doi: 10.1007/s11227-019-02875-w.

[24] Renxiao Wang, Liang Liu, Luhua Lai, and Youqi Tang. Score: A new empirical

method for estimating the binding affinity of a protein-ligand complex. *Molecular modeling annual*, 4:379–394, 1998.

[25] Ligate eu project, 2021. URL `https://www.ligateproject.eu/`.

[26] Renxiao Wang, Liang Liu, Luhua Lai, and Youqi Tang. Score: A new empirical method for estimating the binding affinity of a protein-ligand complex. *Molecular modeling annual*, 4:379–394, 1998.

List of Figures

List of Tables