Alma Mater Studiorum · Università di Bologna

Scuola di Scienze
Dipartimento di Fisica e Astronomia
Corso di Laurea in Fisica

DEVELOPMENT OF A HETEROGENEOUS CLUSTERING ALGORITHM FOR PARTICLE SHOWER RECONSTRUCTION IN HIGH ENERGY PHYSICS USING THE SYCL ABSTRACTION LAYER

Relatore:
Prof. Francesco Giacomini

Correlatore:
Dott. Felice Pantaleo

Presentata da:
Luca Ferragina

Anno Accademico 2021/2022

Abstract

Nei prossimi anni è atteso un aggiornamento sostanziale di LHC, che prevede di aumentare la luminosità integrata di un fattore 10 rispetto a quella attuale. Tale parametro è proporzionale al numero di collisioni per unità di tempo. Per questo, le risorse computazionali necessarie a tutti i livelli della ricostruzione cresceranno notevolmente. Dunque, la collaborazione CMS ha cominciato già da alcuni anni ad esplorare le possibilità offerte dal calcolo eterogeneo, ovvero la pratica di distribuire la computazione tra CPU e altri acceleratori dedicati, come ad esempio schede grafiche (GPU). Una delle difficoltà di questo approccio è la necessità di scrivere, validare e mantenere codice diverso per ogni dispositivo su cui dovrà essere eseguito. Questa tesi presenta la possibilità di usare SYCL per tradurre codice per la ricostruzione di eventi in modo che sia eseguibile ed efficiente su diversi dispositivi senza modifiche sostanziali. SYCL è un livello di astrazione per il calcolo eterogeneo, che rispetta lo standard ISO C++. Questo studio si concentra sul porting di un algoritmo di clustering dei depositi di energia calorimetrici, CLUE, usando oneAPI: l'implementazione SYCL supportata da Intel. Inizialmente, è stato tradotto l'algoritmo nella sua versione standalone, principalmente per prendere familiarità con SYCL e per la comodità di confronto delle performance con le versioni già esistenti. In questo caso, le prestazioni sono molto simili a quelle di codice CUDA nativo, a parità di hardware. Per validare la fisica, l'algoritmo è stato integrato all'interno di una versione ridotta del framework usato da CMS per la ricostruzione. I risultati fisici sono identici alle altre implementazioni mentre, dal punto di vista delle prestazioni computazionali, in alcuni casi, SYCL produce codice più veloce di altri livelli di astrazione adottati da CMS, presentandosi dunque come una possibilità interessante per il futuro del calcolo eterogeneo nella fisica delle alte energie.

Contents

Introduction

Research in high energy physics focuses on studying particles at the most fundamental level in order to discover how they interact. In order to gain an insight on the physical processes that involve matter at the subatomic scale, it is necessary to accelerate particles and make them collide at very high energies. The Large Hadron Collider (LHC) [1], at CERN (Counseil Européen pour la Recherche Nucléaire) [2] accelerates protons to 14 TeV, collimating them in two parallel beams that collide at specific sites where the particle detectors are located. Four main experiments are hosted at the LHC facility: ALICE [3], LHCb [4], CMS [5] and ATLAS [6], of which the latter two are general purpose experiments looking to improve our knowledge of the Standard Model [7]. When protons collide at such high energies, many other particles are produced and scattered in every direction. These particles fly through the detectors and are stopped at different levels depending on their charge and energy, and can thus be identified. In high energy physics there are mainly two ways to look for new physics: either increase the energy at which particles collide, or observe more collisions to gain insight into the rarest processes. For this reason, LHC periodically undergoes a series of hardware upgrades that mainly aim to address both the aforementioned points. In particular, the LHC is scheduled to receive a major upgrade by 2029, which will deal with both points. Firstly, the energy of the colliding beams will be increased up to the theoretical limit of the machine of 7 TeV per beam, with resulting collisions at 14 TeV in the center of mass. Secondly, the number of collisions observed per unit time (luminosity) will increase by roughly a factor 10. Because of this, detectors have to receive periodic hardware upgrades as well, to improve their sensitivity. As a result of the hardware upgrades, the statistics of each event increase as well as the amount of data to be collected and processed with very strict time requirements. To accomplish such a task, each experiment's software must be regularly updated together with the hardware. In particular, we can distinguish two kinds of algorithms used to process the data coming from the detectors: trigger and reconstruction algorithms. The first kind filters the collisions' data in real time by selecting and saving only events that satisfy particular conditions due to which they might be good candidates for new physics. Reconstruction algorithms are instead used to study these potentially interesting events by reproducing the chain of decays and interactions with the detectors that have occurred after the proton-proton collision.

In this context, the CMS experiment has invested part of its resources to explore different heterogeneous computing solutions. This means that several parts of the software reconstruction can run simultaneously on different types of devices, like CPUs, GPUs or FPGAs. Recently, considerable efforts have been put in order to improve the code portability across different architectures and backends. Since writing multiple versions of the same code for each device would be extremely inefficient, some abstraction layers have been considered. In every case, the main objective is to produce an executable file that can run on many different architectures while maintaining a level of performance as close as possible to the native implementation.
This thesis focuses on one of these abstraction layers, SYCL [8], which is based on the ISO C++ standard [9]. In particular, the porting experience of one CMS clustering algorithm from CUDA [10] code, designed to run exclusively on NVIDIA GPUs, to SYCL, with a focus on the performance and physics analysis, is discussed.
In Chapter 1 the clustering problem in high energy physics is introduced, with a description of the CLUE algorithm, chosen for the porting experience. In Chapter 2 the SYCL standard is presented, along with its various implementations. In particular, more details are given on Data Parallel C++, the chosen implementation to carry out the port. Finally, in Chapter 3 the porting results are shown, as well as the performance measurements.

Chapter 1

Clustering algorithms in high energy physics

1.1 The clustering problem

Cluster analysis is, in general, a non trivial problem. Even the definition of cluster can change wildly depending on the context. Due to these circumstances, many clustering methods have been developed, based on various notions of cluster [11]. Currently, some of the most common clustering algorithms can be divided in the following categories:

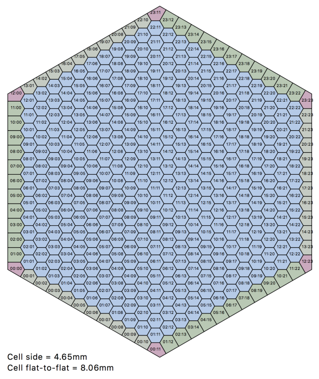
- *Connectivity models*: generally based on distance between points;
- *Partitioning*: such as k-mean algorithms, represent each cluster by optimizing a single function which depends on the distance [12];
- *Density models*: consider dense regions in the data space as clusters;
- *Hierarchical methods*: build clusters following a recurring dendrogram with splitting or merging.

There is no general correct way to produce clusters: each problem requires a specific approach depending on the conditions known a priori, the expected results and also the desired performance.

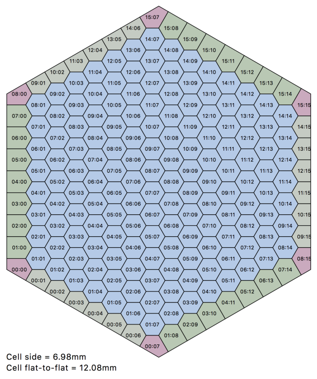
In the upgrade plan for the future revision of LHC, HL-LHC (High Luminosity Large Hadron Collider), calorimeters with high readout granularity have been suggested to observe fine grained images of hadronic and electromagnetic showers [13]. In the case of CMS, the High Granularity calorimeter (HGCAL) will replace the current endcap calorimeters. It consists of several layers and has both an electromagnetic and hadronic compartments.

- The Electromagnetic section (CE-E) is made up of cassettes each containing two layers of sensors and two layers of absorber;
- The Hadronic section (CE-H) contains both silicon sensors and scintillator tiles, as shown in Figure 1.1.

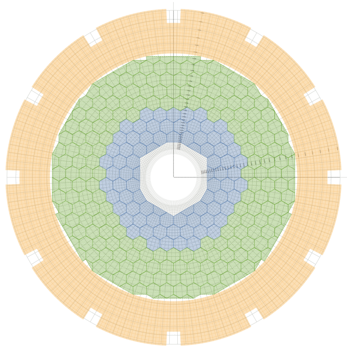
The latter is divided into two sections: the first having a finer sampling than the second. The first few layers of the fine part of the Hadronic section are made of hexagonal silicon sensors only, each having a surface area of 0.5 or 1 cm^2; layers in the last part of the hadronic compartment instead are mixed, with both silicon sensors and scintillator tiles [14].



(a) High density silicon sensors



(b) Low density silicon sensors



(c) Layout of a mixed layer with both silicon and scintillator sensors

Figure 1.1: Schematic representation of the different sensor layouts on HGCAL layers. *Source:*[15, slide 4]

Following a collision, particles that enter the detector interact with its materials and produce showers, whose energy is measured by the sensors on each layer. By clustering the resulting energy deposits (hits) it is possible to reconstruct the shower shape and its main parameters and, after other steps in the reconstruction, identify all of the hits produced by a single particle. Since the calorimeter will have an unprecedentedly high granularity, the most efficient way to cluster the deposits is to group them in bidimensional clusters layer by layer [16] and then connect clusters in subsequent layers with pattern recognition algorithms. In this context, a computational challenge arises, due to the larger amount of data collected in the era of HL-LHC and the limited computational time available at the CMS High Level Trigger (HLT) [17], responsible for the selection of the events of interest. Since event reconstruction must happen at millisecond-level time frames, the algorithm employed needs to be highly efficient and scale well (i.e. linearly or better) as the number of hits increases, in order not to be a bottleneck for the performance. These specific requirements have brought CMS to explore the potential of heterogeneous computing with hardware accelerators such as Graphics Processing Units (GPUs) or FPGAs which can help achieving a higher number of events processed per unit time (throughput) and better energy efficiency.

The input of any clustering algorithm is a set of n points and the output is a set of k clusters which is usually one or two orders of magnitude smaller than n. For clustering in high energy physics, n usually varies from a few thousands to a few millions, while k generally depends on the number of incoming particles as well as the number of layers of the calorimeter. In the case of the HGCAL detector, the average number of hits in a cluster $m = n/k$ can be estimated to be in the order of 10. This leads to the relation between the number of hits n, the number of clusters k and the average number of hits in a cluster m as $n > k \gg m$. Starting from partitioning algorithms, these are not applicable in this case since the number of clusters k is not known a priori. Moving to hierarchical clustering, this method is not suitable as well, since it does not scale well with the number of points as each decision to split or merge needs to scan over many objects or clusters. Density based methods are the most interesting for the HGCAL application, as they are capable of finding clusters of any shape and are efficient for large spacial data collections. However, well-known existing density-based clustering algorithms intrinsically include serial processes that are hard to parallelize. Due to all these reasons, CMS has developed its own clustering algorithm to fit the specific application in the HGCAL detector.

1.2 CLUE standalone

1.2.1 What is CLUE?

Clustering of Energy (CLUE) is “a fast parallel clustering algorithm for high granularity calorimeters in high-energy physics” [18], which was specifically developed to address all of the aforementioned challenges. This algorithm follows a density-based approach with some specific optimizations implemented to give it a greater expression of parallelism. The input data of the algorithm is a series of hits each having its corresponding coordinates (x, y and $layer$) and $energy$. For each point, two key variables are calculated: the local density ρ and the separation δ. The first one represents the energy density in the area of the hit, while the second variable corresponds to the distance from the hit and the nearest hit with higher local density. From these two parameters it is possible to cluster points depending on arbitrary thresholds set on a case-by-case basis.

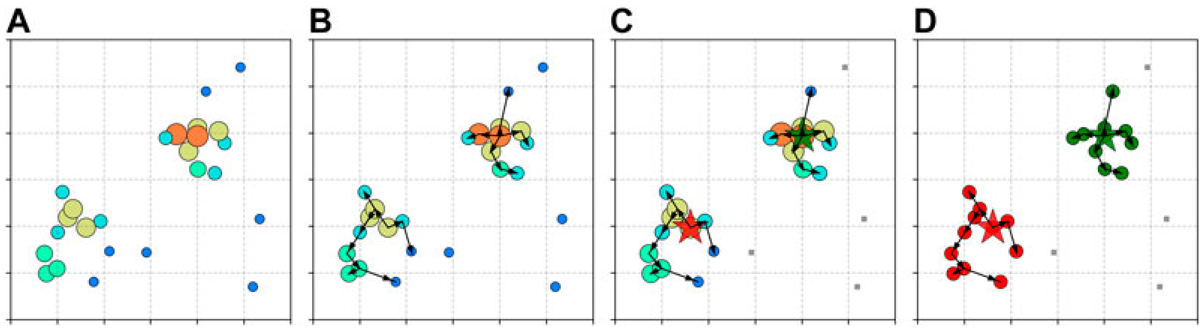


Figure 1.2: Schematic representation of CLUE clustering procedure: from fixed spacial grid to clusters.

Source: [18], figure 2

1.2.2 Clustering procedure

Since CLUE was specifically designed to be used in high-energy calorimetry, where hits are registered on sensor cells whose layout is a multi-layer tessellation, the algorithm's data is indexed with a fixed grid, which divides the space into rectangular bins. This indexing choice allows to fast query the neighborhood of multiple points at the same time to better express parallelism on GPUs and when running on multiple threads.

CLUE requires three parameters: the critical distance, d_c, is the cut-off distance used for the calculation of the local density; the critical density, ρ_c, is the minimum local density to promote a point as a seed or the maximum density to demote it as an outlier; finally the outlier delta factor, *odf*, is a multiplicative constant applied to the critical distance to obtain the maximum distance between a point and its nearest point with higher density over which the point is excluded from the clustering process. These parameters are chosen at runtime and can be tuned by the user based on the clustering application. For example, d_c can be tied to the expected shower size and the granularity of the detectors (possibly differing between silicon-based layers and scintillators), ρ_c can be tuned to maximize the signal to noise ratio and *odf* can be chosen considering the shower separation. In this way, the use of configurable parameters allows CLUE to be more flexible at clustering different events for the specific desired goals of physics.

In Figure 1.2 the clustering procedure of CLUE is shown in a 6×6 grid. First, a fixed grid is constructed and each point is arranged in the bin mapped to its coordinates. Then, CLUE calculates the local density ρ for each point by querying its neighborhood, as shown by the different sized and colored dots (A). Afterwards, each point gets assigned a nearest higher nh, defined as the nearest point with higher local density and their distance δ is calculated (B). Points with ρ higher than the critical density ρ_c and δ greater the critical distance d_c are promoted as seeds, while points with lower density and larger separation ($\delta > d_c \times odf$) are demoted to outliers. All the other points that are neither seeds nor outliers are marked as followers of their respective nearest highers. As a consequence of this logic, each seed has a chain of followers which can be iteratively navigated to build a cluster. At the end of the procedure, only clusters and outliers remain, with the latter points having no followers and not being followers of any other point.

1.2.3 GPU implementation

As previously discussed, CLUE was first developed in order to provide a parallelizable clustering algorithm for high occupancy scenarios in high energy physics, targeting GPUs as hardware accelerators. The main advantage of this class of devices is given by their massive number of physical threads, than can be used to make computations in parallel. The GPU implementation of the CLUE algorithm takes advantage of the multi-threading capabilities of GPUs thanks to the used data structure and the design of each function in the step-by-step procedure. It assigns one GPU thread to each point, for a total of n threads, to create the spacial grid, calculate the local density and separation, promote or demote the point and register points as followers of the corresponding nearest higher. In the last step of the algorithm, each seed is assigned to a single thread, with a total of k threads that build clusters in parallel going through the chain of followers iteratively. Since the results of each step are required in the following ones, it is necessary to synchronize all the threads before moving onto the next computation. This is naturally done by implementing each step into a separate kernel. Data for all of the points is stored in the global device memory as a single structure-of-array (SoA), which contains the points' coordinates, layer numbers and energies.

One key factor to take into account when developing software for GPUs is that, due to the parallel nature of the execution, it can happen that multiple threads try to access and modify the same address in global memory at the same time. For the CLUE application, thread conflicts can happen in three cases:

1. multiple points need to register to the same bin simultaneously;
2. multiple points need to register to the list of seeds simultaneously;
3. multiple points need to register as followers to the same point simultaneously.

Because of this, some atomic operations are required to avoid race condition among threads. When performing an atomic operation, a thread is granted exclusive access to a specific memory location that becomes inaccessible to all other threads until it finishes. The usage of atomic operations naturally causes some serialization among threads in a race. Its impact is negligible in cases 1 and 3, since the bin dimension and the number of followers per point are usually small. By contrast, serialization in case 2 can cause a significant slowdown since the number of seeds k can be large. However, this operation is still faster on GPU memory when compared to the data transfer between host and device; therefore, the total execution time of CLUE does not suffer significantly from this serialization.

1.3 Heterogeneous computing and compatibility layers

1.3.1 The need for heterogeneous computing

The high luminosity upgrade for the LHC, scheduled before the beginning of the next runs from 2029 onwards, will increase the number of collisions per unit time (namely the *luminosity*) by roughly a factor of 10, resulting in an average *pileup* of 200 proton-proton collisions every 25 ns. Such an increase in the number of events per second will pose an incredible challenge for both online and offline software reconstruction of each experiment [19]. This added complexity far exceeds the expected increase in the processing power for conventional CPUs and thus calls for the exploration of novel and different solutions. One such possibility is *heterogeneous computing*, which is already being exploited by industry and high performance computing centers to achieve better efficiency and higher throughput by matching the job to the best possible architecture. Specifically, applying this new paradigm to CMS software reconstruction (CMSSW) and its framework means that part of the work can be offloaded to one or more graphics processing units (GPUs), thus easing the load on the CPUs while simultaneously increasing the total throughput and improving energy efficiency. Tasks offloaded to a GPU are executed in a parallel fashion and, most importantly, asynchronously with respect to the code running on CPU, meaning that the CPU resources freed by offloading work to GPUs can be used for other tasks at the same time. The shift to heterogeneous computing has already proved to be a success for CMS offline reconstruction, where GPU implementations show up to three times increase in performance in some cases [19]. However, adopting heterogeneous computing also poses some challenges, especially in code portability and maintainability. While having the possibility to offload work on different hardware looks promising, it also means that in order to be able to efficiently execute software on heterogeneous hardware, the code should be written and optimized for each specific backend[1] to be targeted. Even considering only the main GPU manufacturers, one would need to take into account three different implementations for each kernel in all of the algorithms, leading to a massive code duplication, whose maintenance represents an insurmountable challenge and a great resource sink.

These reasons led to explore different compatibility layers to ease the transition to heterogeneous computing. A candidate layer must support multiple architectures such

[1]In software engineering, the physical infrastructure or hardware on which the code is executed.

that the programmer can write a single source code without having to sacrifice performance.

1.3.2 The Patatrack group

The adoption of heterogeneous computing in CMSSW has been first proposed by the CMS Patatrack group. The group was founded in 2016 and is composed by a diverse set of people whose main objective is to demonstrate that it is possible to adopt heterogeneous computing in both online and offline reconstruction with benefits both in terms of software and hardware. Firstly, the team developed parallelizable algorithms for calorimetry reconstruction as well as the Pixel Detector[20], with promising results which led to a wider push to support heterogeneous computing in CMMSW. In 2021, the group managed for the first to reconstruct collisions with GPUs[21] offloading $\sim$30% of the HLT processing to GPUs. The number of algorithms that can run on GPUs is planned to grow during subsequent runs with the goal of offloading at least $\sim$50% of the processing during Run-4 of HL-LHC and $\sim$80% in Run-5. In this context, the team is developing new algorithms as well as porting old ones to GPUs.The work presented here is part of Patatrack's research on compatibility layers for heterogeneous computing and development of parallelizable algorithms.

1.3.3 SYCL and other compatibility layers

As previously discussed, the adoption of heterogeneous computing, while improving performance and efficiency, also poses a great challenge in the ability to maintain code and port it to different architectures. Therefore, in recent years the development of portability layers has ramped up significantly as more and more applications require to offload work to different accelerators. Many different solutions are now available, each proposing a different way to handle the problem but all trying to use a single source code to produce a unique executable file able to run on as many different backends as possible. In this context, the CMS collaboration and in particular the High Level Trigger developers have started looking for a suitable compatibility layer to adopt for heterogeneous event reconstruction code. A large fraction of CMSSW algorithms has been successfully ported to CUDA a few years ago, and it is being executed already on both CPUs and NVIDIA GPUs (which handle the more computing-intensive parts). In order to adapt the software reconstruction to more vendors and accelerators, a large fraction of the framework has already been ported to such a compatibility layer and this new version is scheduled to run in production after the LHC Christmas shutdown between 2022 and 2023. The compatibility layer currently in use is alpaka: a header-only abstraction library for accelerator development [22]. Such portability library allows the abstraction of the underling levels of parallelism by mapping some custom-defined classes to different hardware depending on the backend. In general, the work division is very similar to

CUDA's grid-blocks-threads division with the addition of an extra element layer that allows for different mappings on CPUs and GPUs. In this way, the compatibility layer is able to exploit the different parallelization capabilities of each different backend. As previously stated, a large fraction of the software reconstruction has already been ported to alpaka, which thus allows to only develop and maintain one source code while still being able to run the reconstruction on both CPUs and NVIDIA GPUs. Support for other backends is still ongoing: Intel GPUs are not yet supported by alpaka itself, while AMD GPUs are officially supported, but the corresponding backend has not been implemented in the software reconstruction yet.

Although some other compatibility layers, like kokkos [23], have been explored, alpaka is, as of now, the most promising of such layers for the CMS reconstruction. However, SYCL is still a solution being tested for three main reasons:

- SYCL is a royalty-free project, supported by many actors in the tech industry, which bodes well for the future support of the standard;
- As a new compatibility layer is developed, it is always relevant to explore its performance in comparison to native code and other compatibility layers;
- While not all the backends have been implemented yet, SYCL promises to support a variety of devices: from CPUs to GPUs and also Intel's FPGAs.

Furthermore, being a royalty-free project, SYCL can count on a variety of implementations which are explored in more detail in Section 2.2 but, at a base level, this allows for more flexible adaptations of the standard to specific needs.

Chapter 2

SYCL abstraction layer

2.1 Standard

SYCL is an abstraction layer for heterogeneous computing open source end royalty-free, maintained by Khronos Group [24]. At its core, SYCL proposes a more flexible and simple way to write code for multiple devices, thus improving code portability and in general, simplifying the maintenance of code. On a high level, SYCL defines a library that allows programmers to write a single source code to be executed on different devices thanks to the SYCL runtime. A standard SYCL application contains code compliant with the ISO C++ standard which can be roughly divided into host code, executed by the CPU, and device code, enclosed in *kernels*, to be executed by one or more heterogeneous devices. SYCL kernels are basically C++ function objects, so objects of some class that overload the "function call operator" `()`. For this reason, simple kernels are usually implemented as lambdas. Furthermore, device code compilation is more complicated than standard host code compilation, the added complexity is explored in the following sections.

As previously stated, SYCL tries to be as compliant as possible with ISO C++ standards so much that all of the host code can, in theory, be compiled with a standard C++ compiler to produce an executable that can run on the CPU. This is true until any kernel is called or any OpenCL[1] integration is required. Some C++ features are unavailable in SYCL to guarantee the highest degree of portability across most different kinds of devices. A couple of examples are the inability to use function pointers or call virtual functions inside kernels. Furthermore, the whole error handling system is different from standard C++, mainly for the need to throw and catch asynchronous exceptions. Since heterogeneous computing is becoming increasingly more relevant year over year, Khronos

[1]OpenCL™ (Open Computing Language) is an open, royalty-free standard for cross-platform, parallel programming of diverse accelerators found in supercomputers, cloud servers, personal computers, mobile devices, and embedded platforms [25].

is also cooperating with the C++ committee to integrate heterogeneous programming into the standard. Code 2.1 shows the basic syntax of a SYCL application.

```cpp
#include <CL/sycl.hpp>

int main()
{
    // allocate input data on host
    int data[1024];

    // initialize sycl queue. When created, the queue is tied to a device
    auto q = sycl::queue(sycl::default_selector());

    // allocate memory on the device
    auto d_a = sycl::malloc_device<int>(1024, q);

    // copy input data from host to device
    // wait for the copy to finish at the end
    q.memcpy(d_a, &data, 1024 * sizeof(int)).wait();

    // submit task to the device
    q.submit([&](sycl::handler &cgh)
    {
        cgh.parallel_for(sycl::range<1>(N), [=](sycl::id<1> idx)
        {
            // do work on data
        });
    }).wait();

    // copy result data from device back to host
    q.memcpy(&data, d_a, 1024 * sizeof(int)).wait();

    // free device memory
    sycl::free(d_a, q);
}
```

Code 2.1: Sample code of a SYCL application.

Here the most important characteristics are highlighted, with some of them discussed in greater detail in the following sections. First up, the distinction between device code and host code:

- Line 19-25 is device code, every operation inside a sycl::queue::submit() will be executed on the device and the host will only take care of scheduling the work;
- All of the other code will be executed on the host. This includes device selection and every memory operation on it.

This separation is also tied to a different compilation model which will be explained in more detail in Section 2.3. On a general level, the SYCL integration starts by creating a `sycl::queue`, this object is tied to a specific device (that can be a CPU, GPU, or FPGA) and is used to manage its memory and the scheduling of jobs to be executed on it. Memory on the device is allocated through the `sycl::malloc_device` function and freed using `sycl::free`. Note that in both cases a queue has to be provided to get the context and device on which to allocate or free the memory. Finally, note that after each queue operation there is an explicitly `wait` used to synchronize the completion of the kernel. Data dependencies and synchronization are discussed at length in Section 2.4.3. In general, this is not strictly necessary, depending on the goals and structure of the program.

2.2 Implementations

Being an open-source, royalty-free, project means that SYCL has many different implementations which all support different backends. The main SYCL implementations are schematically reported in Figure 2.2.

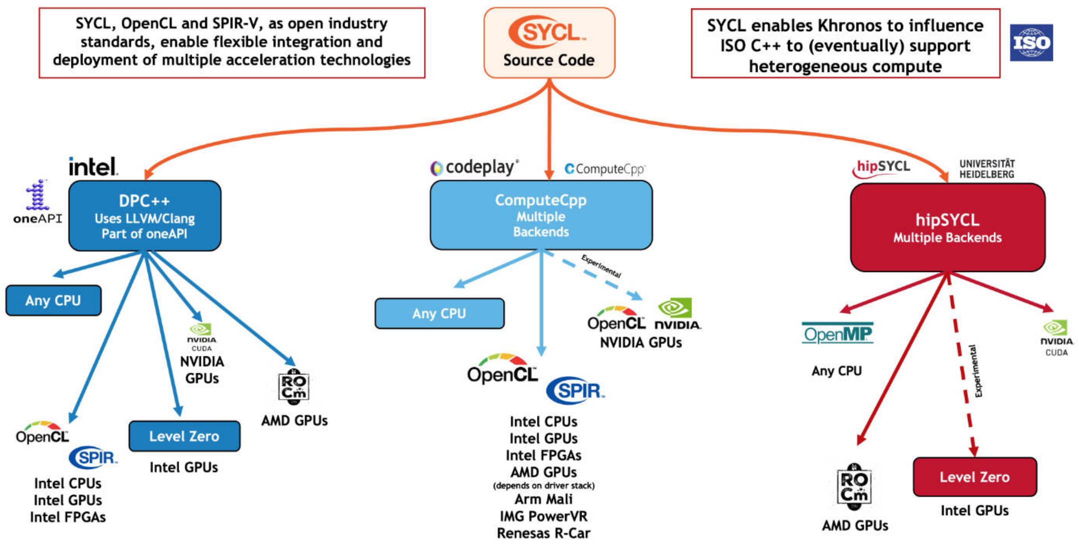


Figure 2.2: Overview of the main SYCL implementations.

Source:[24]

In general, as of SYCL 2020, an implementation is made of four different components, represented in Figure 2.3:

- SYCL interface: a template library that provides the developers with the features of SYCL;
- SYCL runtime: the library that schedules and executes work on both the host and devices. Specifically, it schedules memory operations, launches kernels, and tracks data dependencies;
- Backend interface: point of contact between the SYCL runtime and a specific device. Some examples of backends are OpenCL (used to interface with CPUs), CUDA (used to interface with NVIDIA GPUs), Level Zero (used to interface with Intel GPUs), and HIP/ROCM (used to interface with AMD GPUs). Not all backends are supported in all the implementations.
- SYCL device compiler: C++ compiler used to identify SYCL kernels and compile them down to an Intermediate Representation (IR) to then be linked with code compiled by the host compiler.

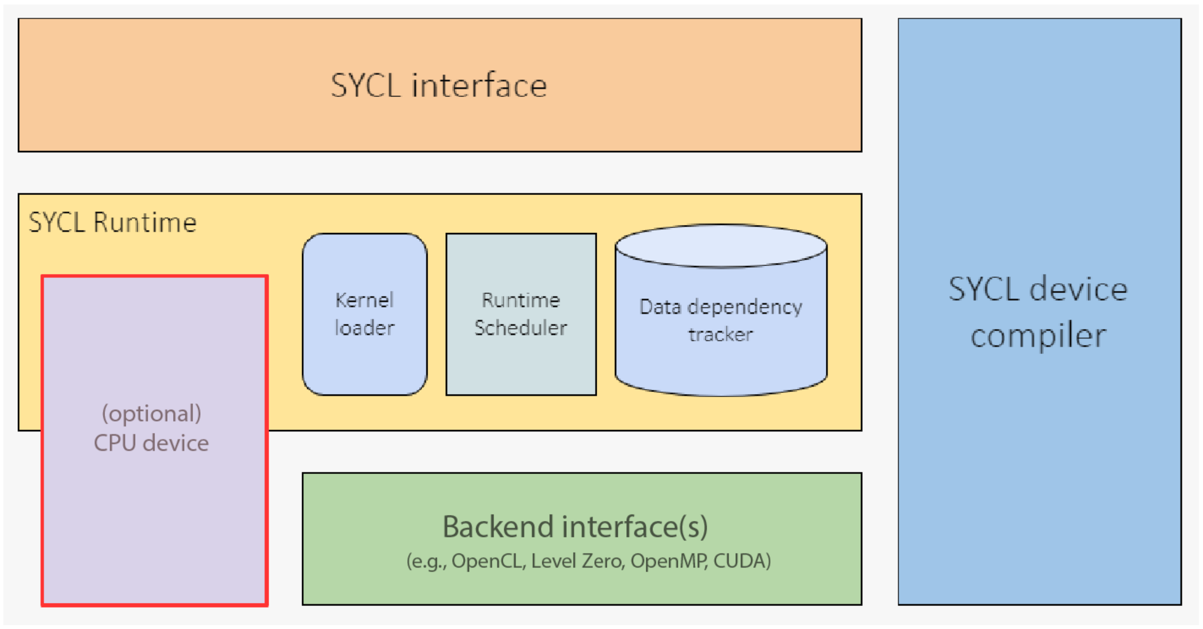


Figure 2.3: Schematic representation of the SYCL implementation backend interface. *Source:*[26, lesson 1 - slide 12]

OneAPI and Data Parallel C++

Data Parallel C++ is the SYCL implementation developed by Intel and integrated into oneAPI, their toolkit for heterogeneous programming [27]. This implementation is compliant with the SYCL 2020 standard and thus supports the Universal Shared Memory model, as described in Section 2.4.1. oneAPI includes a low-level hardware interface known as Level Zero. This backend interfaces directly with the SYCL runtime and is used to execute code on Intel GPUs and FPGAs that support it. As far as other backends are concerned, oneAPI officially supports OpenCL to interface with CPUs and its support for CUDA is under development. Its native support for USM and the auxiliary tools provided, as well as native support for Intel hardware, are what ultimately brought CMS to choose this specific SYCL implementation to explore its capabilities in high-energy physics software reconstruction. A particular highlight is a support tool for developers who are first exploring SYCL capabilities. Included in oneAPI, there is a compatibility tool, Data Parallel Compatibility Tool (DPCT), which can automatically convert CUDA code into SYCL code. While the output from the tool is not always correct and some specific functions or features are not translated entirely, the compatibility tool surely helps to lay the groundwork to move from CUDA code to SYCL code, easing the refining work that needs to be done by the developer. The Data Parallel C++ compiler, `dpcpp`, is regularly forked from the open source LLVM project [28].

2.3 Compilation model

SYCL follows a single-source, multiple-pass compilation model. Since the source code is just one, the kernels are standard C++ function objects[2], generally implemented as lambda expressions. Because host and device codes are different, the code passes through a compiler twice: first through the host compiler to produce a CPU object and then through the device compiler. This second pass on the device code produces a device IR for the specific architecture of the targeted device. Finally, the CPU object is linked with the IR to produce a single executable with code for both the CPU and the device. What was just described is generally known as the Ahead of Time (AoT) compilation model. The SYCL implementation chosen to carry out the porting, oneAPI, also allows compiling only the host code while not targeting any specific device. In this case, the device code will be compiled only at run time using the Just in Time (JiT) compiler. This gives the user more flexibility to run on many different supported devices while not necessarily having access to a compiler or compiled code for a specific device or architecture (as long as both are supported by the implementation).

[2]In C++, a function object is a generic class which implements an overload of the operator function call `()`.

2.4 Execution model

Every SYCL device is divided into multiple Compute Units (CUs) which are in turn split into Processing Elements (PEs). Every kernel computation will be executed on the PEs of the relevant device. When a kernel is launched, a thread hierarchy gets initialized. The main element in a SYCL thread hierarchy is the ND-range, which is a 1, 2, or 3-dimensional grid of work groups, that are equally sized groups of the same dimensionality. These work groups are in turn divided into work items, as shown in Figure 2.4. Diving into the single elements of the hierarchy, in SYCL, the programmer has access to:

- Work item: a single thread within the thread hierarchy;
- Work-group: a 1, 2 or 3-dimensional set of threads (work-items) within the thread hierarchy;
- ND-range: it's the representation of the thread hierarchy used by the SYCL runtime when executing work. It has three components: the global range, the local range, and the number of work groups. The global range is the total number of work items within the thread hierarchy in each dimension; the local range is the number of work items in each work group in the thread hierarchy, and the number of work groups is the total number of work groups in the thread hierarchy, which can be derived from the global and local.

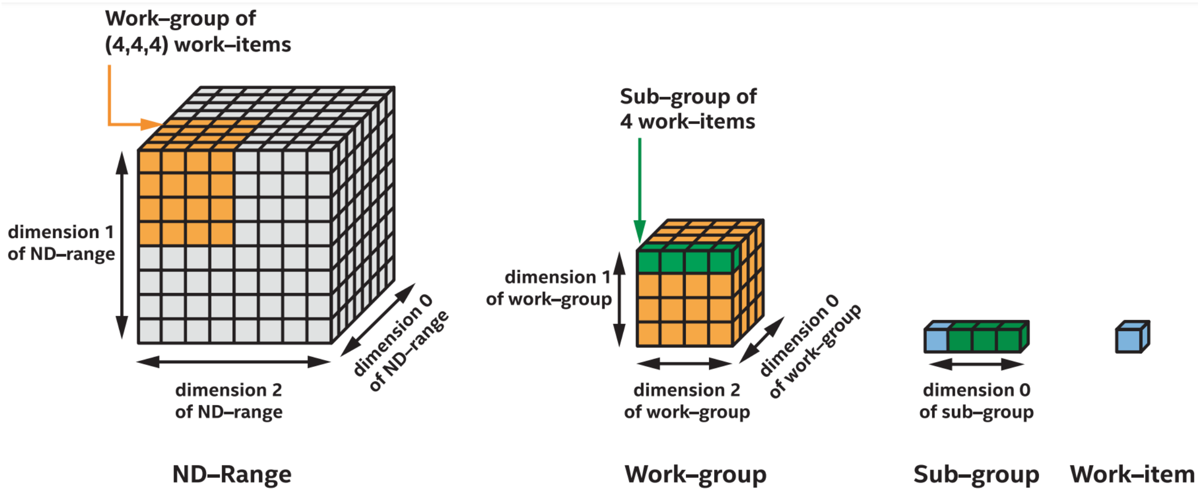


Figure 2.4: Pictorial representation of the SYCL thread hierarchy.

On some modern hardware platforms, there is also the possibility of accessing subsets of the work items in a work-group to execute them with additional scheduling guarantees. These subsets are known as sub-groups.

In SYCL, the work-group size can be left empty, and then implementation can set it to the optimal value according to an internal heuristic. Additionally, SYCL does provide a mechanism to get the preferred work-group size. An example of how to do it is shown in Code 2.5. Typically, choosing a work group size that is a multiple of the preferred one will be enough.

```
auto pWGSM = kernel.get_work_group_info<
sycl::info::kernel_work_group::preferred_work_group_size_multiple>();

queue.submit([&](sycl::handler &cgh){
    cgh.parallel_for(kernel, sycl::range<1>(pWGSM),
                     [=](sycl::id<1> idx){
        /* kernel code */
      });
  });
```

Code 2.5: Query of preferred work-group size for a specific kernel.

2.4.1 Memory management

SYCL is based on the OpenCL memory model but operates at a higher level of abstraction, which means that storage and access memory are separated and treated with different objects: buffers and accessors, respectively. SYCL buffers are, at their core, `std::unique_ptr` wrapped in such a way to make them live only inside the scope they are defined in. The buffer manages memory allocation/copy, while accessors create requirements on the buffers. These requirements can be allocating memory, synchronization between different accessors or data transfer between host and device. Depending on the accessor, data is automatically allocated on the host or on the device. In the more recent SYCL specification, a big push has been made to allow explicitly allocating memory using C-like pointers. This method offers greater flexibility by allowing the programmer to explicitly allocate and copy exactly when they intend to, often reducing memory overhead. In order to allocate using regular pointers, the device has to support Universal Shared Memory (USM). Furthermore, SYCL operates in three different memory spaces:

- Private memory: region of memory allocated per work item and only visible to that work item. Cannot be accessed from the host;
- Local memory: contiguous region of memory allocated per work group and visible to all of the work items in that work-group. This memory is allocated and accessed using an accessor and cannot be accessed from the host;

- Global memory: a memory visible by all of the work groups of the device.

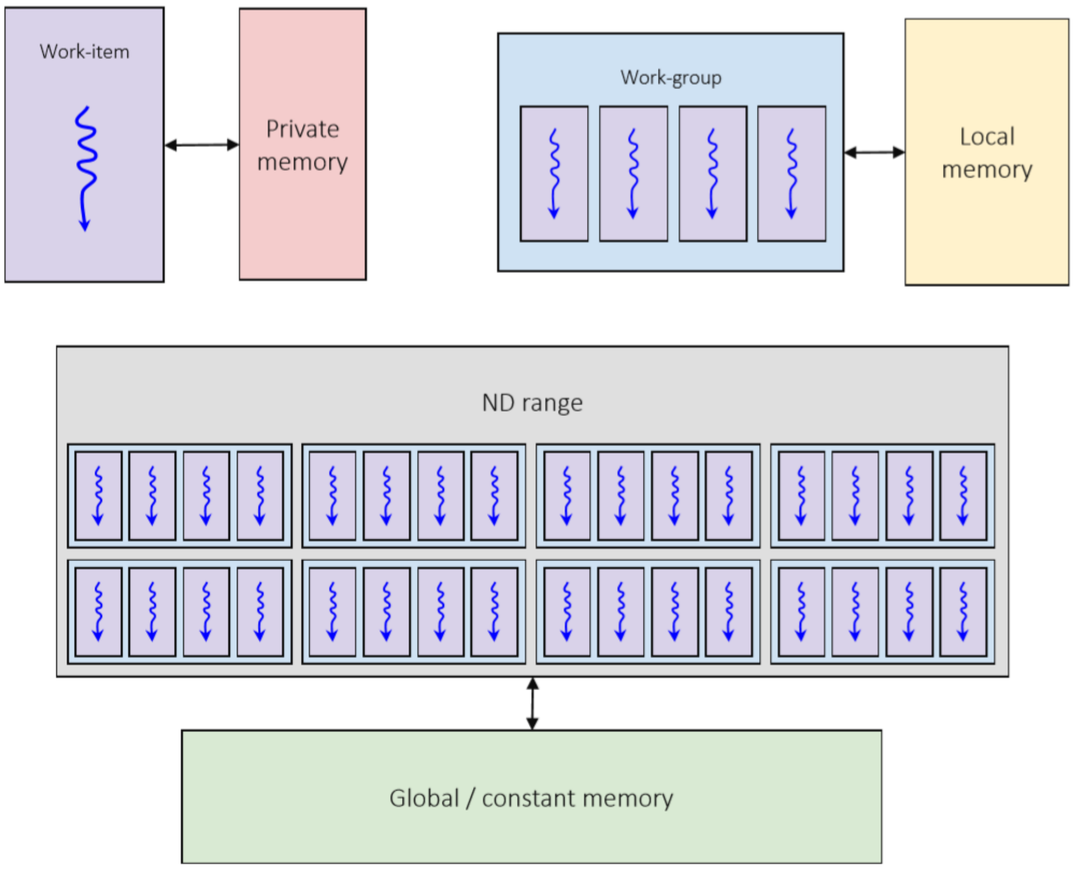


Figure 2.6: SYCL memory layout.
Source: SYCL Academy [26, Lesson 4 - slide 6]

Explicit allocation of memory through pointers has still some limitations like the inability to specify whether the allocation should be in private or local memory and defaults to private. However, this fairly new feature is still been worked on and improvements can be expected in the next SYCL releases.

Memory allocation and transfer

Due to the previously mentioned efficiency reasons, this work has been carried out entirely using the USM model. In this case, memory allocation is done through the `sycl::malloc` function which is divided into three different methods:

- `malloc_host`: allocate memory on the host that is accessible on both the host and the device. These allocations cannot migrate to the device's attached memory so kernels that read from or write to this memory do it remotely, often over a slower bus such as PCI-Express;

- `malloc_device`: device allocations that can be read from or written to by kernels running on a device, but they cannot be directly accessed from code executing on the host. Data must be copied between host and device using the explicit USM memcpy mechanisms;

- `malloc_shared`: like host allocations, shared allocations are accessible on both the host and device, but they are free to migrate between host memory and device-attached memory automatically.

Memory allocated through these methods needs to be managed using specific SYCL functions:

- `sycl::queue::memset(ptr, value, num_bytes)`: Set `num_bytes` of memory allocated using malloc functions to `value`;

- `sycl::queue::memcpy(dest, src, num_bytes)`: Copies `num_bytes` from `src` to `dest`;

- `sycl::free(ptr, queue)`: Frees memory located at `ptr` and previously allocated usign a SYCL malloc function.

Excluding the last method, the other twp are also available as methods of the `sycl::handler` class, so they can be used inside a `queue::submit()` scope as demonstrated in Code 2.7.

```
q.submit([&](sycl::handler &h) {
    // copy hostArray to deviceArray
    h.memcpy(device_array, &host_array[0], N * sizeof(int));
    });
q.wait();
```

Code 2.7: Example of explicit memory copy using a sycl handler.

Note that in order to allocate and deallocate in SYCL it is necessary to have declared a queue first, as to choose the device on which each memory operation should be executed. Furthermore, since copying is asynchronous by default, it is a good practice to always use `sycl::queue::wait()` after any group of copying actions to prevent segmentation faults or unexpected behaviours. Finally, in SYCL it is not necessary to specify the direction of copying (host-device, device-device, device-host) as it is deduced at run time. The usage of the discussed methods is exemplified in Code 2.8.

```
std::vector<float> h_a(N);
auto queue = sycl::queue{sycl::default_selector{}};
auto d_a = sycl::malloc_device<float>(N, queue);
auto d_b = sycl::malloc_device<float>(N, queue);

queue.memset(d_a, 0x00, N * sizeof(float));
queue.memset(d_b, 0x00, N * sizeof(float));

queue.memcpy(d_a, h_a.data(), N * sizeof(float));
queue.memcpy(d_b, d_a, N * sizeof(float));
queue.memcpy(h_a.data(), d_b, N * sizeof(float)).wait();

sycl::free(d_a, queue);
sycl::free(d_b, queue);
```

Code 2.8: Code sample for SYCL memory management methods.

2.4.2 Kernels execution

SYCL kernel functions are called using one of the following invoke API entries (which are methods of SYCL handlers):

- `single_task`: The kernel function is executed exactly once;
- `parallel_for`: The kernel function is executed ND-range times passing thread identification objects as parameters.

Examples for both of these kernels' calls are shown in Code 2.9.

The handler defines the interface to invoke kernels by submitting commands to a queue. A handler can only be constructed by the SYCL runtime and is passed as an argument to the command group function. The command group function is an argument to submit.

The SYCL kernel function invoke API takes a C++ callable object by value which is most often expressed as a lambda. If local memory is required or work-group size is specified manually, then the corresponding `nd_range` object must be used as the first parameter. In turn, the `nd_item` associated with the `nd_range` can be passed inside the kernel and its method for barriers or work-group operation can be used.

One tricky issue found when dealing with SYCL kernels is that, as per SYCL specification [8], variables can be passed inside a kernel (`parallel_for`) only by value, furthermore the capture of *this is not allowed neither implicitly nor explicitly (in general, this would point to host memory which is not accessible on the device). To resolve this issue

```
// execute the kernel function exactly once
auto queue = sycl::queue(sycl::default_selector{})
queue.submit([&](sycl::handler& h)
             {
                 h.single_task([=]{a[0] = 1.0f});
             });

// execute the kernel function nd-range times
auto queue = sycl::queue(sycl::default_selector{})
queue.submit([&](sycl::handler& h)
             {
                h.parallel_for(range, [=](id<1> i) {a[i] = b[i]});
             });
```

Code 2.9: Examples of SYCL kernel calls.

you can simply create local copies of all the variables needed inside the kernel before the `parallel_for` gets called, as shown in Code 2.10.

```
sycl::queue q = sycl::queue(sycl::default_selector{});
q.submit([&](sycl::handler& h)
{
    auto var_kernel = var;
    h.parallel_for(...,[=](...)
    {
        /* use var_kernel here */
    });
});
```

Code 2.10: Example of queue submission using a multidimensional range.

2.4.3 Synchronization

SYCL provides a single synchronization level that is across all of the work items within a single work group using barriers that can be called inside kernels.

- `mem_fence`: inserts a memory fence on global memory access or local memory access across all work items within a work group.

- `barrier`: like the previous one, but it also blocks the execution of each work item within a work group at that point until all of them have reached that point.

In Code 2.11 synchronization between work-items is demonstrated using a snippet from a parallel reduction kernel.

```
sycl::queue q = sycl::queue(sycl::default_selector{});
unsigned int tid = item.get_local_id(0);
unsigned int i = item.get_group(0) * item.get_local_range().get(0)
                   + item.get_local_id(0);

sdata[tid] = input[i];

item.barrier();
```

Code 2.11: Synchronization between work-items within the same work-group. In this example, consider sdata as an array in local memory. This snippet needs to be inside a `parallel_for` scope to have access to the `sycl::item` used for synchronizing.

SYCL does not provide any memory fences or barriers across the entire kernel, only across the work items within a work group.

As previously noted and shown in Code 2.1, memory operations, in particular memory copies, and kernel execution in SYCL are asynchronous by default. This means that by just scheduling a particular operation, it is not guaranteed that it will be executed before any other unless relevant data dependencies are specified by the programmer. To be more specific, SYCL queues are by default not in order, meaning that different kernels (i.e. `queue::submit()`) can and will be executed at the same time whenever possible to improve performance. If the data used by different kernels is not independent, the queue should be initialized as in-order by passing an extra argument in the constructor as shown in Code 2.12.

Using an in-order queue corresponds to adding a `sycl::queue::wait()` after each and every submission. In this way, operations are executed in issue order on the selected device.

In case the code allows for some kernels to be executed in parallel while others have explicit dependencies, it is possible to use a default queue and specify the desired data dependencies using the member function `sycl::handler::depends_on()` which can take an event or an array of events as parameters, as shown in Code 2.13. Note that, in this particular example, the second submit has an explicit dependency on the first, so the queue will wait and synchronize before executing it. However, the last submission can be executed at the same time as the first one since no data dependency is specified.

```cpp
// create a default queue
// kernels will be executed in such a way to
// maximize parallel execution
auto q = sycl::queue(sycl::default_selector());

// create an in-order queue
// kernels will be executed in the order they
// are scheduled
auto q2 = sycl::queue(sycl::default_selector(),
                      sycl::property::queue::in_order());
```

Code 2.12: Creating a queue defaults to an out-of-order one. Kernels can be executed sequentially when the queue is created specifying the in-order property.

```cpp
// initialize an out-of-order queue
auto q = sycl::queue(sycl::default_selector{});
// define an event
auto e = q.submit([&](sycl::handler& h)
                  {
                      /* kernel */
                  }
q.submit([&](sycl::handler& h)
         {
             h.depends_on(e);
             /* kernel */
         });
q.submit([&](sycl::handler& h)
         {
             /* kernel */
         })
```

Code 2.13: In-order kernel execution using explicit data dependencies.

Chapter 3

SYCL implementation of CLUE

3.1 Porting the standalone version of CLUE

3.1.1 Notable changes

As previously discussed, the first step of the porting experience was the translation of CLUE in SYCL. The original code, which can be found on a CMS Patatrack gitlab repository [29], already implements the CPU serial, the CUDA and the multi-threaded CPU versions (the latter using Threading Building Blocks[1]).

Since, as highlighted above, oneAPI provides the developers with a conversion tool from CUDA code to SYCL, the conversion process began by passing the CUDA version of the code through the tool. This produced the first prototype of the SYCL code which needed some modifications in order to run correctly. While the underlying logic and implementations are almost the same, two examples of practical differences between the CUDA implementation and the SYCL one are explored in the following:

- kernel submission;
- use of variables inside kernels as demonstrated in Section 2.4.2

CLUE Kernel submission: CUDA vs SYCL

CLUE is split into five kernels each of which deals with one part of the clustering procedure, as discussed in Section 1.2.2. In the CUDA version, work is divided in a `grid/block/thread` fashion: when a kernel is submitted, the entire problem is covered by the grid size, then it's split into smaller parts each of which is assigned to a block that finally assigns a single task to each of its threads. The work division in SYCL, while

[1]Threading Building Blocks (TBB) is a performance library that allows simplifying the work of adding parallelism to complex applications across different architectures [30].

similar in principle, is quite different as it needs to adapt to a multitude of heterogeneous backends. It can be mapped to the CUDA work division almost completely in case the code is running on a GPU. In particular, the CUDA hierarchy `grid/block/thread` becomes `nd_range/work-group/work-item` in SYCL. It must be pointed out that this mapping concerns only GPUs, while on CPUs the hardware mapping can look quite different depending on the vectorization[2] capabilities of the specific device. In Code 3.1 the two equivalent ways to cover the same problem with the same work division in CUDA and SYCL are shown.

```
// CUDA version
const dim3 blockSize(numThreadsPerBlock, 1, 1);
const dim3 gridSize(numBlocks, 1, 1);

kernel<<<gridSize, blockSize>>>(d_input, d_output);

// SYCL version
auto queue = sycl::queue(sycl::default_selector{});
const sycl::range<3> blockSize(1, 1,numThreadsPerBlock);
const sycl::range<3> gridSize(1, 1, numBlocks);

queue.submit([&](sycl::handler& cgh)
{
  cgh.parallel_for(sycl::nd_range<3>(gridSize * blockSize, blockSize),
                                    [=](sycl::nd_item<3> item)
    {
        kernel(d_input, d_output, item);
    });
});
}
```

Code 3.1: Difference in work division and kernel submission in CUDA and SYCL.

One thing to note in the example above is the difference in the order of dimensions between CUDA `dim3` and `sycl::range`. In fact, the former follows the order (x, y, z), while the latter follows the inverted order (z, y, x). While this might not be an issue in general, specifically when running SYCL code on NVIDIA GPUs, the order of the dimensions becomes relevant as most of those GPUs have limited thread capabilities on the z dimension. For example on the NVIDIA A10 used for testing, the z dimension of

[2] Automatic compiler optimization which processes an operation on multiple pair of operands at the same time. Compilers can generally transform `for` loops in a set of vector operations

`blockSize` is limited to a maximum of 64, while the other dimensions can go up to 1024. Due to these considerations kernels were submitted in the way demonstrated above to keep the mapping of one point per thread in the clustering procedure independently by the hardware that would actually run the code through SYCL.

3.1.2 Physics validation and performance comparison

In order to validate the physics results of the new implementation, a set of synthetic datasets, which had previously been used to validate the physics results of CLUE, was considered. In this case, the outputs produced by the different implementations were compared while using the same input data and parameters. As described in Section 1.2.2, points with a density below the critical one could be demoted to outliers if their δ is larger than the specified threshold. The synthetic datasets used were generated in such a way as to resemble high-occupancy events with conditions similar to what is expected to occur in the proposed high-granularity calorimeter upgrade for CMS. Results were validated in different scenarios, each time increasing the number of hits per event. In particular, a total of 100 layers are input to CLUE simultaneously, with each layer having a fixed number of points with unit energies. In this scenario, the density characterizes clusters whose energy has a Gaussian distribution with standard deviation σ set to 3 *cm*. The 5% of the points is generated as noise distributed uniformly over the layers. To test CLUE's performance scaling, the number of points in each layer is increased from 1,000 to 10,000 in 10 equaling steps so that the total number of points per event in the test ranges between 10^5 and 10^6. Figure 3.2 shows the performance scaling of the different CPU implementations. The results of these comparisons are obtained as the average value of the execution time across ten runs of the algorithm with the same parameters. Note that SYCL seems to provide a massive advantage compared to native serial code when running on the same hardware. This is mostly due to its native support for multi-threading to oneTBB, the oneAPI implementations of Threading Building Blocks. While the serial code is naturally limited to only use one CPU thread, the SYCL version can offload work to all of the available threads. The other TBB implementation shown in the plot is run by using the corresponding backend available with Alpaka and, at the time of the data taking, some unpredicted thread locks decreased performance significantly according to the implementation of atomic operations in the portability library. This performance issue should be solved in later releases to bring the performance more in line with the one demonstrated by SYCL. However, the performance of the TBB implementation scales linearly with the number of points per layer as expected by the logic of the algorithm.

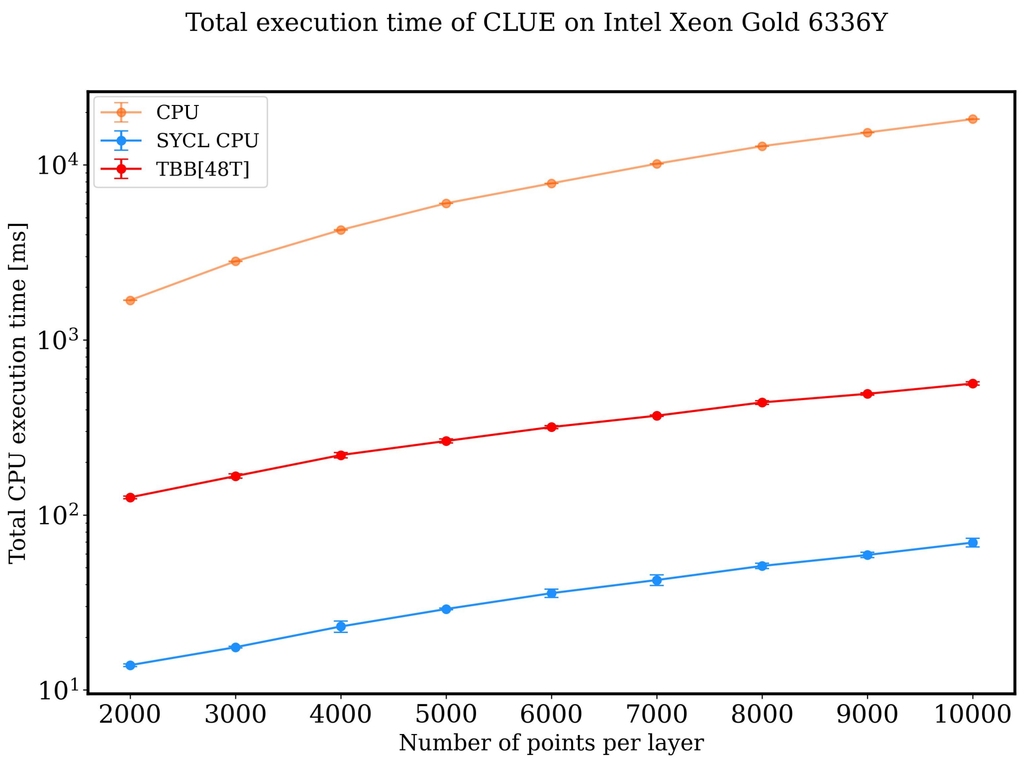


Figure 3.2: Performance comparison of CLUE's different CPU implementations (lower execution time is better). Note that the y-axis is in log scale, showing the massive improvement that SYCL has over the native serial code. This advantage is mostly given by the native support for multi-threading in SYCL applications.

The largest speedup factor in execution time is achieved by using GPUs. The main aim of the porting was to show that it is possible to write a single source code to compile and run on a multitude of devices. In Figure 3.3 it is shown how the total execution time of CLUE running on the same GPU is similar when using SYCL with the CUDA backend and native CUDA code. SYCL code is completely unchanged compared to the CPU version shown above, thus achieving at least one of the objectives of using portability layers in heterogeneous computing scenarios. The only difference between this executable and the one used for the CPU benchmark is the compiler that produced it. In fact, oneAPI's compiler, `dpcpp`, does not yet support the CUDA backend natively and is still in development; therefore, an open source fork of `dpcpp`, `LLVM`, was used to compile the project and test it on NVIDIA hardware. The results look extremely promising, especially considering that this backend is not officially supported yet.

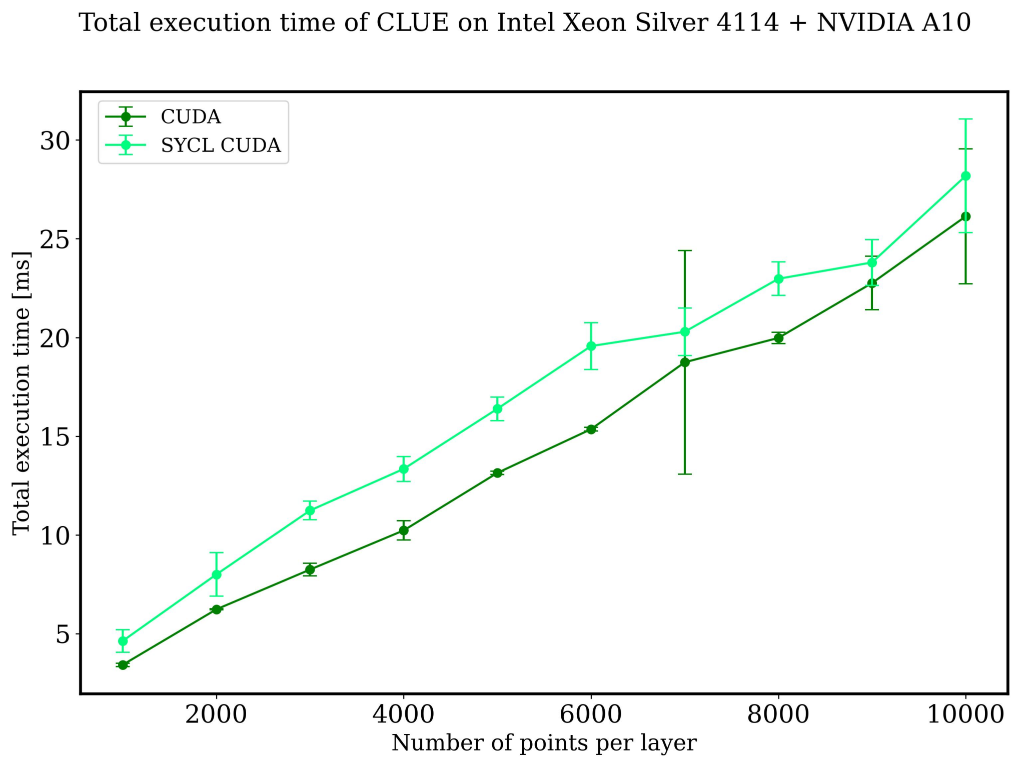


Figure 3.3: Performance comparison between native CUDA code and SYCL code running through the CUDA backend (lower execution time is better).

Note that, in general, using GPUs allows to improve the performance by roughly a factor 3, when compared to the multi-threaded execution on CPU, and even more when considering the other implementations.

3.2 Integrating CLUE in a CMSSW-like framework

The porting of the standalone version of CLUE was carried out mainly to experiment with SYCL and become familiar with its functioning. Once the task was completed, actual physics evaluation and performance measurements had to be carried out in a framework resembling the one used in CMSSW. Although this required significant changes and adaptations as well as actual developments on the framework itself, better results were obtained in terms of both raw performance and physics reconstruction.

3.2.1 An introduction to the framework

Similarly to what has previously been done by the Patatrack CMS group for the standalone pixel track and vertex reconstruction [20], an approximation of CMSSW framework [31] was used to process data and schedule the execution of CLUE, from now on referred to as heterogeneous CLUE.

The main goal of the framework is to facilitate the deployment of software and test both already-consolidated and new algorithms in a realistic testbed before including them in the official software of the CMS experiment. The core of the execution is the Event Data Model (EDM) which is centered around the concept of *Event*. Topologically, an Event is a C++ object which contains all the raw data and information related to simultaneous collisions whose average number is known as pileup and which form a single particle collision event. In the specific case of CLUE, each Event contains the 2D coordinates, layer, and energy of every hit produced by a collision event. The Event is used to pass data from one module to the next and it represents the only way to access or modify data. Some modules might require additional information to process events that are provided through the *Event Setup* module. Furthermore, the framework is modular, meaning that each step of the reconstruction is represented by a separate plugin that can be plugged into the main run. Plugins are compiled in shared libraries and the execution must be configured to schedule and run the desired plugins. Although the framework includes six different types of modules, only the following ones are used with the CLUE algorithm:

- Source: Produces one or more Events by reading data from an input file (csv, binary, ROOT...), gives the Event status information, and can add data on host memory.
- EDProducer: Producer modules read data from an Event, produce an outcome and put it back into the Event.
- OutputModule: Reads data from the Event and stores the output to external media.

In this context, it is possible to offload part of the workload to accelerators when using one of the implementations of the framework and modules that take advantage of a compatibility layer such as Alpaka or SYCL.

Figure 3.4 shows heterogeneous CLUE's workflow step by step. Modules are grouped by color and the arrows indicate the order of execution. The red module is an under development integration of the 3D clustering module used in CMSSW reconstruction. Each producer module can have an associated ESProducer, used to add relevant information to the Event through the Event Setup, like the parameters needed for clustering.

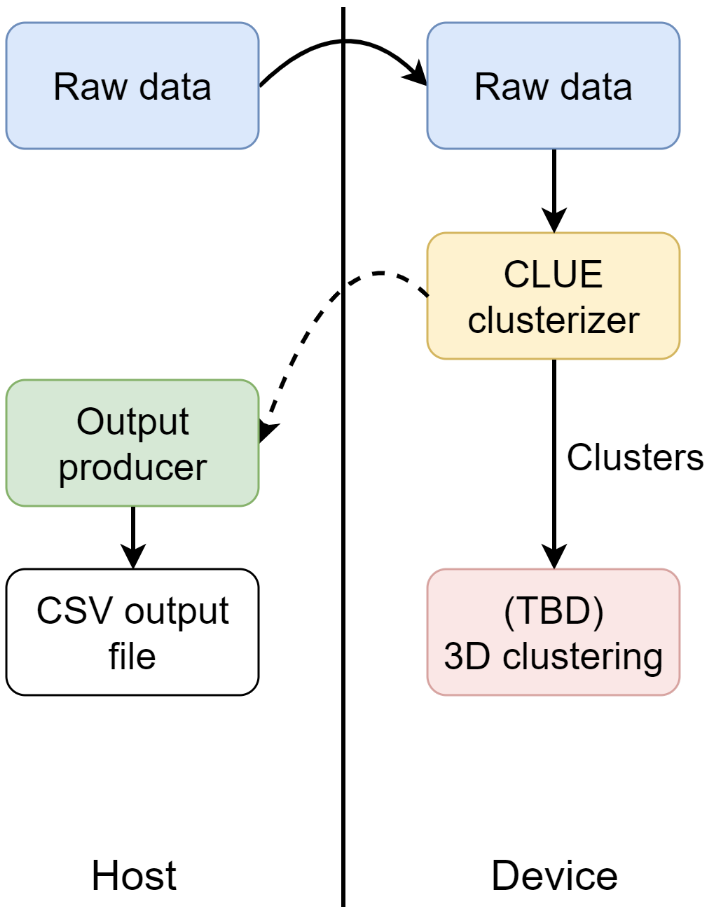


Figure 3.4: Heterogeneous CLUE workflow. Blue modules are the results of Source, the yellow module is the single EDProducer, while the green module is the OutputModule which (optionally) produces a csv output file per event at the end.

Another fundamental aspect of the framework is its integration with Threading Building Blocks to take advantage of multiple threads and process more events at the same time. This last aspect, in particular, is carried out by creating multiple `EDM streams`. Each of them executes the entire workflow (except for the Source, reads data from all the events only once per run) on a different event and the framework allows the creation and execution of multiple streams at the same time. This means that on a more capable device, like a multi-threaded CPU or a GPU, multiple events can be analyzed at the same time, thus greatly increasing the total throughput achieved by the algorithm. CLUE hence acts as a somewhat simple introduction to the framework's inner workings and provides a benchmarking platform for the different implementations. It is important to note that detaching modules from CMSSW and integrating them into standalone versions of the framework allows to experiment new algorithms, data structures and memory management methods as well as to test performance without having to interface them with the full reconstruction of the CMS detector. Such idea of a testbed has led to the creation to the standalone pixel tracking module as well as this CLUE implementation.

3.2.2 Changes and improvements from the standalone version

Integrating CLUE in the framework allows taking advantage of more efficient scheduling, and finer control of the number of threads and concurrent streams used for the execution while also providing better performance measuring capabilities. However, in order to fully take advantage of the framework, significant changes had to be made to the core of its SYCL implementation.

Starting from the most fundamental aspect, some parts of the framework, ported from CUDA, needed to be optimized for SYCL's inner workings, while some others had to be rewritten entirely. Since SYCL is designed to be compatible with multiple backends, its logic for device selection cannot match the one implemented in CUDA and is rather similar to alpaka. In the current version of the code, the user can select which device to offload work to by a simple command line parameter `--device` followed by the type of device wanted or its backend. The device selector will take care of finding all of the compatible devices and offload work to the selected one by creating an in-order queue on that device per `EDM stream` to schedule memory operations and kernel execution.

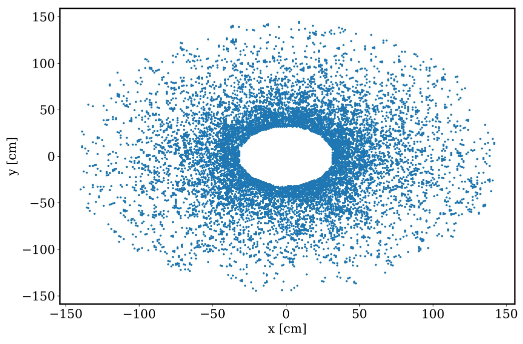
When offloading work to an accelerator, the most costly operation in terms of overhead is device memory allocation. That is why the framework includes a module for caching the allocations, which had to be entirely rewritten for SYCL. This module ensures that the least amount of allocations is performed and that memory that has been freed can be reused without needing another allocation. Using CLUE as an example, the algorithm needs 13 allocations per run for input variables, results, and some inner variables used to make calculations. Using the caching allocator, when running with a single stream, CLUE will allocate 13 blocks of the required size in the device memory, then reuse those blocks for each event and finally free the device memory at the end of the execution. At the beginning of the execution one caching allocator is constructed on each available device. Device memory is accessed through custom-defined unique pointers which interface directly with the caching allocator on the specific device. This approach has two main benefits:

- Costly device memory allocations are reduced to a minimum number by reusing memory whenever possible;
- All of the cached memory is freed at the end of the execution, thus allowing no memory leaks.

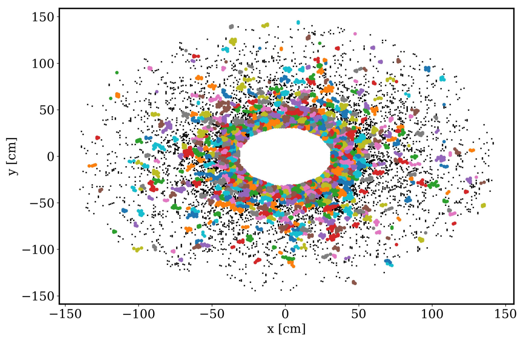
From the algorithm's point of view, these changes mainly reflect in what modules of the workflow are allowed to allocate device memory. Since a single `sycl::queue` is produced per event and gets propagated by the framework and considering that the queue is needed for all of the memory operations and kernels scheduling, the algorithm needed some optimizations to perform queue-bound operations only when one is obtainable from the `EDM event`.

3.3 Results

Heterogeneous CLUE, was tested with a simulated dataset obtained by running the reconstruction of a $t\bar{t}$ event with 200 pileup on a recent version of CMMSW and dumping the information of hits to a binary file. The conditions chosen are the most similar to the ones that would be seen in the proposed HGCAL upgrade [14].
In Figure 3.5 and Figure 3.6 are shown the clustering results of CLUE on layer 5 of the simulated dataset. In particular, Figure 3.5 shows hits registered in the entire detector, while Figure 3.6 focuses on a smaller window to better show the clusters built by CLUE.

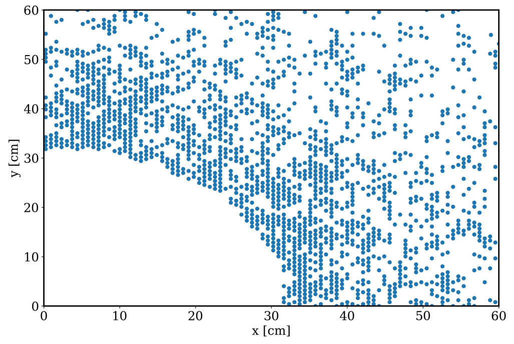


(a) Input data for layer 5.

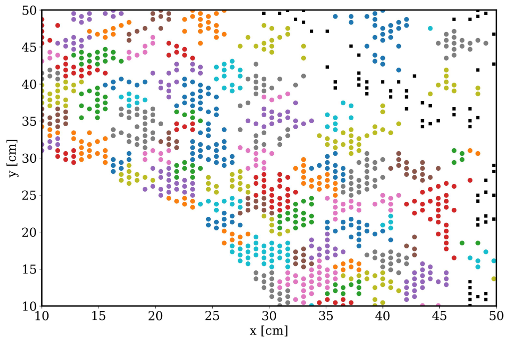


(b) Clusters produced by CLUE.

Figure 3.5: Example of CLUE clustering on layer 5 of the simulated dataset dumped from CMSSW. Each cluster is characterized by a different color (with some repetitions due to the large number of clusters) while outliers are represented as black squares.



(a) Input data for a small window of the detector.



(b) Clusters produced by CLUE.

Figure 3.6: Clustering results of CLUE on a small detector window. Each color corresponds to a different cluster. Outliers are represented as black squares.

3.3.1 Physics performance

Heterogeneous CLUE's performance has been evaluated on both CPUs and GPUs with different backends using Alpaka, SYCL, and native code whenever possible. In particular, the following hardware was used to benchmark the algorithm:

- Intel(R) Xeon(R) Silver 4114 CPU 2.20GHz with driver OpenCL [2022.14.7.0.30_160000];
- NVIDIA Tesla T4 with driver CUDA [11.6];
- Unreleased Intel GPU;

Figure 3.7 shows the computing performance of each implementation of heterogeneous CLUE on CPU, scaling with the number of threads. As seen in the standalone version, SYCL implementation looks to be able to take better advantage of multi-threading on CPU showing a 21-57 % performance increase when compared to Alpaka and 14-102% improvement with respect to the native serial implementation. The measurements are obtained as the average value of 10 consecutive runs on 1000 events. The exact results are shown in Table 3.1. One particular case that needs to be clarified is the single-thread execution. Both Alpaka and the native implementations show less than half the performance of SYCL in this case. This happens because SYCL is actually using two threads in this particular instance by relying on the multi-threading capabilities of the CPU and dedicating one thread to the framework execution and one to the algorithm itself. This is confirmed by observing that the throughput remains almost exactly the same when using two threads.

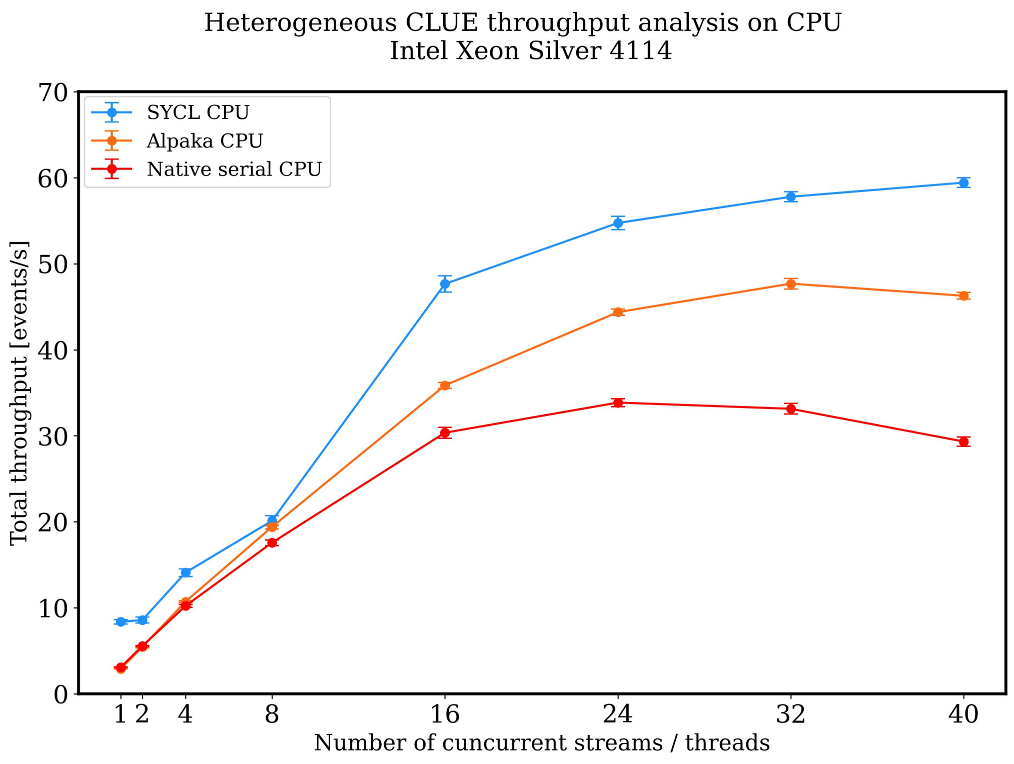


Figure 3.7: Performance comparison of the different CPU implementations of heterogeneous CLUE (higher throughput is better).

Total throughput on Intel Xeon Silver 4114 (ev/s)			
Threads	Alpaka	Native serial	SYCL
1	2.894 ± 0.014	3.09 ± 0.05	8.4 ± 0.3
2	5.44 ± 0.12	5.55 ± 0.05	8.6 ± 0.3
4	10.72 ± 0.09	10.22 ± 0.15	14.1 ± 0.4
8	19.4 ± 0.2	17.6 ± 0.3	20.1 ± 0.6
16	35.9 ± 0.4	30.4 ± 0.6	47.7 ± 0.9
24	44.4 ± 0.4	33.9 ± 0.5	54.7 ± 0.8
32	47.7 ± 0.6	33.1 ± 0.6	57.8 ± 0.6
40	46.2 ± 0.4	29.3 ± 0.5	59.4 ± 0.6

Table 3.1: Detailed throughput analysis for the CPU implementations of heterogeneous CLUE.

For the GPU implementations, the throughput increases from 3 to 10 times depending on the comparisons. Figure 3.8 shows the comparisons between the two different compatibility layers discussed, executing logically identical code on the same hardware.

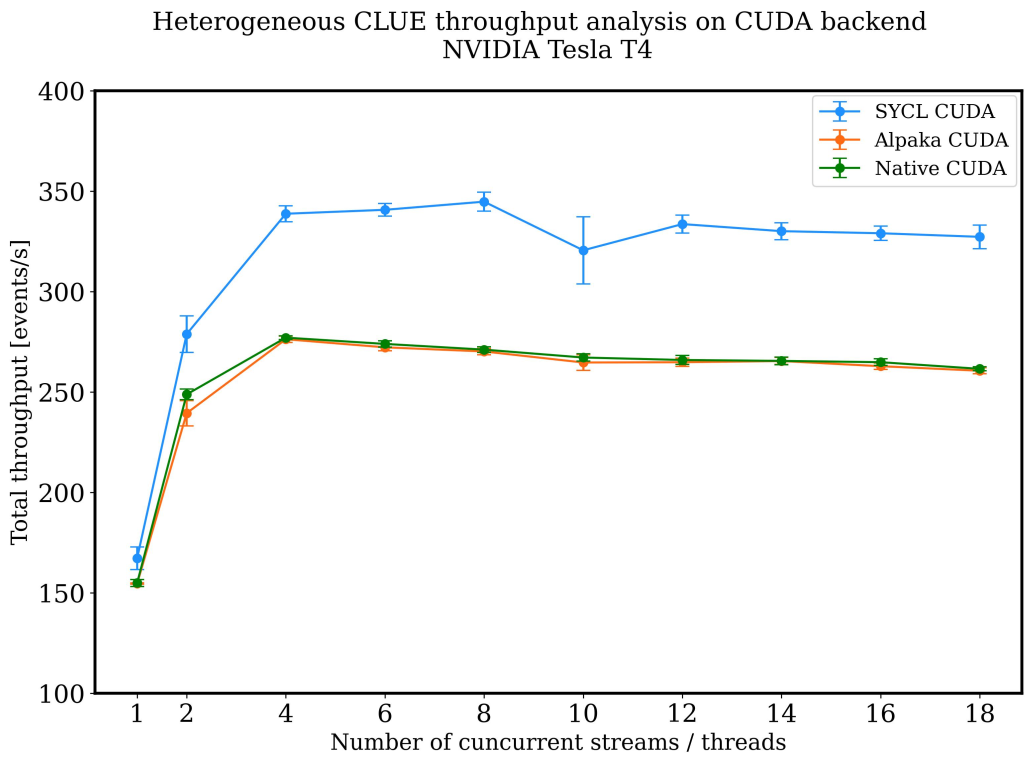


Figure 3.8: Comparison of SYCL and Alpaka heterogeneous CLUE implementations when running on the CUDA backend (higher throughput is better).

The difference in performance is harder to explain in this case, but on a general level it seems that, at least for this particular workload and hardware, SYCL has better memory management capabilities than Alpaka and CUDA, thus improving speed when transferring data, allocating or setting memory, and synchronizing streams. One thing to note is that an increasing number of streams can lead to performance degradation and instability, especially on SYCL, which shows its best performance when using 8 threads/streams. A detailed breakdown of the data is presented in Table 3.2.

Total throughput on NVIDIA Tesla T4 (ev/s)			
Streams	CUDA	Alpaka	SYCL
1	154.9 ± 1.7	154.6 ± 0.3	167 ± 6
2	249 ± 3	239 ± 6	279 ± 9
4	277.1 ± 0.9	277 ± 2	339 ± 4
6	273.9 ± 1.5	272 ± 2	341 ± 3
8	271.1 ± 1.4	270 ± 2	345 ± 5
10	267.2 ± 1.8	265 ± 4	321 ± 17
12	266 ± 2	265 ± 2	334 ± 4
14	265.5 ± 1.9	265 ± 2	330 ± 4
16	264.8 ± 1,7	262 ± 2	329 ± 4
18	261.5 ± 1.0	261 ± 2	327 ± 6

Table 3.2: Detailed throughput analysis for the CUDA implementations of heterogeneous CLUE.

Finally, throughput analysis on the unreleased Intel GPU cannot be disclosed due to the pre-alpha state of the hardware, which is still under NDA. However, testing on this hardware has been successful in showing that a single SYCL source code can run on CPUs, Intel GPUs, and NVIDIA GPUs. This demonstrates the potential of SYCL as a compatibility layer to ease code maintainability across different backends.

3.3.2 Porting considerations

As discussed, multiple adjustments had to be made in order to transition from CUDA to SYCL. In this context, the compatibility tool developed by Intel has provided help in more than one occasion, but its output cannot yet be trusted in all situations, especially when dealing with more complex projects. In particular, some of the areas in which the tool provides little to no help are:

- Porting of more complex projects made up of multiple compilation units;
- Error management;
- Atomic operations.

Regarding the second point, the error management system employed by SYCL is based on throwing and catching exceptions, similar to what is done in plain C++, while CUDA implements its own error management system. Because of this, it is up to the programmer to add the relevant checks to make sure that the SYCL code can handle eventual errors. As far as atomic operations[3] are concerned, SYCL implementation

[3]In programming, an atomic operation guarantees exclusive access to a particular memory address to the thread that is executing the operation itself.

is still pretty much undocumented and lacks some fundamental CUDA methods. Re-implementing needed methods is possible, but must be done on a case-by-case basis and completely by hand.

In general, expressing parallelism through SYCL is more difficult than doing the same through CUDA, mostly due to the heterogeneous nature of the SYCL code. The creation of `nd_ranges` with the desired dimensions to cover any specific problem is extremely verbose and the methods to obtain the computing capabilities of the device in use are not always helpful in creating an optimized work division. This problem becomes especially evident in complex code that makes extensive use of CUDA warps[4] which are inherently more difficult to port to other backends. One possible solution is given by utilizing sub-groups in SYCL, though the sub-group dimension is hardware-dependent and needs to be known at compile time in the kernel invocation.

To sum up the porting experience, the compatibility tool proves itself useful in easing the first steps when getting to know SYCL but fails on more complex projects. Everything that is not translated automatically can be ported by hand even if the documentation is not fully developed yet, so the porting involves a lot of tinkering and experimenting to get the desired results. However, the results obtained in the end look promising, but SYCL is not ready for large-scale adoption at this point: both for the ever-changing and evolving nature of the relatively young SYCL standard and the absence of a complete compiler for all the supported backends.

[4]In CUDA, a warp is a collection of 32 threads that execute the same instruction.

Conclusions and future work

The Large Hadron Collider is scheduled to receive a massive hardware upgrade in the next few years. This has the main objective to push the boundaries of high energy physics research by increasing the number of observed collisions per second by a factor of 10 in order to observe rare events and decays more often. Together with the collider, also the experiments' detectors will have to undergo significant upgrades, not only to keep up with the massively increased data rates, but also to renew components and modules deteriorated by the highly radioactive environment where the detectors are. The increased sensibility of the detectors must be accompanied by corresponding software upgrades to process and analyze the collected data with very strict time constraints. This has led CMS to explore the possibilities offered by heterogeneous computing, which would allow to offload part of the computing work to different accelerators, like GPUs or FPGAs, while increasing performance and efficiency leaving the experiment' hardware budget virtually unchanged. This approach would naturally lead to writing a different source code per implementation, thus making the job of maintaining and updating the code impossible, especially on a large scale such that of CMS reconstruction software. However, a solution to this problem comes from compatibility layers. These are code abstraction layers that has the single aim of providing the ability to write a single source code which can than be compiled to produce an executable file that can run on a wide variety of devices and backends. CMS has already identified some of such compatibility layers that might offer promising results in high energy physics: Alpaka and SYCL. Throughout this work, the performance of SYCL has been explored on a particular reconstruction algorithm, CLUE, already in use in CMS reconstruction workflow. In particular, the SYCL implementation backed by Intel, was used to implement first a standalone version of the algorithm and then integrate the same algorithm in a CMSSW-like framework allowing to measure performance in comparison with native implementations and Alpaka: another compatibility layer already used by CMS. The results obtained in this context are extremely promising, showing the ability to write a single source code and produce an executable able to run on CPUs, Intel GPUs and NVIDIA GPUs with good performance results.

However, there still are some key issues in the SYCL implementation used:

- While it is possible to compile for the CUDA backend, it is not yet officially supported by the compiler included in oneAPI. The use of the open source fork of the compiler, LLVM, is fine for testing implementations but lacks the optimization passes and stability guarantees coming from an official implementation which is scheduled for early 2023;

- In general, the documentation is still lacking and often forces the programmer to implement standard methods by hand;
- It might be possible to improve performance by using device-specific parameters, but at the same time this would decrease the effective portability of the code.

Optimization in general is still ongoing, especially when it comes to memory operations and synchronization between the host and the device. In order to further test the performance offered by SYCL, a porting of the aforementioned Patatrack standalone pixel tracking module (Pixeltrack Standalone) [20] is being developed. By porting a more complex application, many more limits of the SYCL standard and its oneAPI are being discovered and reported to Intel, which is assisting the porting. Once Pixeltrack standalone will have been fully ported to SYCL and integrated with the improvements made to the SYCL implementation of the CMS framework, it will be possible to obtain more significant performance measurements and comparisons with the other implementations. The results' comparison will surely be one of the deciding factors to choose which portability layer to rely on during Run4 of HL-LHC.

Acknowledgements

At the end of this three-year journey there are so many people that had a role in making it better and more worthwhile that I think I'm going to struggle to list them all, but I'll try my best. Starting from the most relevant for this work, I want to thank my supervisor, Francesco Giacomini not only for the guidance while writing this thesis, but for allowing me to have this great experience in the first place and helping me navigate towards my master's. Next up, the group I've been working with for the past few months, Patatrack, has been incredibly great in introducing me to high energy physics reconstruction. A particular thank you to Felice Pantaleo and Andrea Bocci for their invaluable guidance and desire to share their knowledge with me. As for the others, I want to mention Wahid and Tony for always providing helpful suggestions from positions I can only look up to. I also want to thank Aurora, Nikos and JJ for being the best students-colleagues I could have wished for in this amazing experience at CERN. Moving closer to home, I feel like I need to start with my family. There are probably a thousand reasons to thank all of them for supporting during the last 22 years. However, I can think of three reasons in particular to thank my father and it has a lot to do with old stories and bedtime. As for my mother, she has simply always been there, through thick and thin and, no matter how hard I tried to stay miserable, has always managed to get to me and cheer me up. As per my brother, he's always been a point of reference, someone to look up to and one of my biggest motivations to move forward even when I didn't feel like I could do it. Moving a bit further away, I cannot forget my uncle (yes, yellow and red are still great colours, even in Switzerland), my aunt (and her amazing late dog) and my grandmother who can never get enough of telling me how much she believes in me and what I do. There are some more people who have been extremely important in the path that led me here and I want to take some time to thank them as well. Starting from the physics-related one, this thesis would probably have never been made if it weren't for professor Accorsi and his completely unique way of showing me the incredibly interesting world of physics which is hiding right behind a curtain made of math. As per the other one, I can positively say that nothing I've written until now would have been the same if it weren't for the extremely inspiring work of professor Guastamacchia who instilled in me the love for the English language. It would be impossible for me not to thank two of the people who know me best, so thank you Manuel and Nello (see, I've used the short

version, just like in my contact list) for all of the memories and the Fridays and, well, everything else, you probably know it better than me at this point. Finally, there are at least 23 reasons for which I am deeply grateful to you, Francesco, but I'll keep things simple just for this time. Thank you for always being there, especially when I'm not at my best, to show me how to take life one small step at a time and for always seeing the best part of me.

Bibliography

[1] The Large Hadron Collider - CERN. [Online]. Available: https://home.cern/science/accelerators/large-hadron-collider

[2] CERN. [Online]. Available: https://home.cern/

[3] ALICE Collaboration. [Online]. Available: https://alice-collaboration.web.cern.ch/

[4] LHCb Collaboration. [Online]. Available: https://lhcb.web.cern.ch/

[5] CMS - CERN. [Online]. Available: https://home.cern/science/experiments/cms

[6] Atlas Experiment. [Online]. Available: https://atlas.cern/

[7] The Standard Model - CERN. [Online]. Available: https://home.cern/science/physics/standard-model

[8] SYCL 2020 specification - The Khronos Group. [Online]. Available: https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html

[9] The ISO C++ Standard. [Online]. Available: https://isocpp.org/std/the-standard

[10] CUDA. [Online]. Available: https://developer.nvidia.com/cuda-zone

[11] V. Estivill-Castro, "Why so many clustering algorithms: A position paper," *SIGKDD Explor. Newsl.*, vol. 4, no. 1, p. 65–75, jun 2002. [Online]. Available: https://doi.org/10.1145/568574.568575

[12] S. Lloyd, "Least squares quantization in pcm," *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, 1982.

[13] The CALICE Collaboration, "Calorimetry for lepton collider experiments - calice results and activities," 2012. [Online]. Available: https://arxiv.org/abs/1212.5127

[14] A.-M. Magnan, "HGCAL: a high-granularity calorimeter for the endcaps of CMS at HL-LHC," *Journal of Instrumentation*, vol. 12, no. 01, pp. C01 042–C01 042, jan 2017. [Online]. Available: https://doi.org/10.1088/1748-0221/12/01/c01042

[15] M. Rovere. Clustering and reconstruction in HGCAL. [Online]. Available: https://indico.cern.ch/event/949440/contributions/3988881/attachments/2092853/3516903/20200828_FCC_TICL_MR.pdf

[16] Z. Chen, C. Lange, E. Meschi, E. Scott, and C. Seez, "Offline reconstruction algorithms for the cms high granularity calorimeter for hl-lhc," in *2017 IEEE Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC)*, 2017, pp. 1–4.

[17] V. Gori, "The CMS high level trigger," *International Journal of Modern Physics: Conference Series*, vol. 31, p. 1460297, jan 2014. [Online]. Available: https://doi.org/10.1142%2Fs201019451460297x

[18] M. Rovere, Z. Chen, A. Di Pilato, F. Pantaleo, and C. Seez, "Clue: A fast parallel clustering algorithm for high granularity calorimeters in high-energy physics," *Frontiers in Big Data*, vol. 3, 2020. [Online]. Available: https://www.frontiersin.org/article/10.3389/fdata.2020.591315

[19] A. Bocci, V. Innocente, M. Kortelainen, F. Pantaleo, and M. Rovere, "Heterogeneous reconstruction of tracks and primary vertices with the cms pixel tracker," *Frontiers in Big Data*, vol. 3, 2020. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fdata.2020.601728

[20] Patatrack group. Standalone Patatrack pixel tracking. [Online]. Available: https://github.com/cms-patatrack/pixeltrack-standalone#readme

[21] A. Bocci. First collisions reconstructed with GPUs at CMS. [Online]. Available: https://cms.cern/news/first-collisions-reconstructed-gpus-cms

[22] alpaka - Abstraction Library for Parallel Kernel Acceleration. [Online]. Available: https://github.com/alpaka-group/alpaka#readme

[23] Kokkos core libraries. [Online]. Available: https://github.com/kokkos/kokkos#readme

[24] The Khronos Group. SYCL. [Online]. Available: https://www.khronos.org/sycl/

[25] OpenCL overview - The Khronos Group. [Online]. Available: https://www.khronos.org/opencl/

[26] Codeplay. SYCL Academy. [Online]. Available: https://github.com/codeplaysoftware/syclacademy

[27] oneAPI specification. [Online]. Available: https://www.oneapi.io/spec/

[28] Intel corporation. Intel project for LLVM. [Online]. Available: https://github.com/intel/llvm

[29] CLUE gitlab repository. [Online]. Available: https://gitlab.cern.ch/kalos/clue

[30] Intel corporation. Threading Building Blocks. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html

[31] A. Bocci. CMSSW Application Framework. [Online]. Available: https://twiki.cern.ch/twiki/bin/view/CMSPublic/WorkBookCMSSWFramework