Trabajo de Fin de Grado

Introduction to High Performance Computing with SYCL

Adriano dos Santos Moreira

La Laguna, 11 de julio de 2024

D. **Francisco de Sande González**, profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor y D. **Alberto Cabrera Pérez**, doctor en Informática por la Universidad de La Laguna e Ingeniero de Software en Codeplay Software Ltd. como co-tutor.

C E R T I F I C A N

Que el presente trabajo de Fin de Grado titulado:

"*Introduction to High Performance Computing with SYCL*"

ha sido realizado bajo su dirección por D. **Adriano dos Santos Moreira**.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente memoria del Trabajo en La Laguna a 11 de julio de 2024.

Agradecimientos

Por toda la asistencia que me ha prestado, agradezco a mi tutor Francisco de Sande, por orientarme en el trabajo de fin de grado con sus años de experiencia y criterio distinguido y exigente. Del mismo modo, agradezco sus años de docencia, que lo alaban por lograr transmitir su carácter riguroso a las materias que imparte.

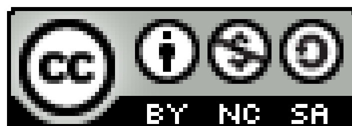
De igual forma, gracias a mi co-tutor Alberto Cabrera y a Codeplay Software Ltd. por colaborar en la elaboración de este trabajo, su conocimiento profundo de SYCL ha sido de gran utilidad. No solo por saber sobre la plataforma, sino también por enseñarme particularidades y apuntes importantes sobre la programación paralela.

Asimismo, agradezco a Wooptix por su contribución en el apartado de procesamiento de imágenes de este trabajo.

Por otra parte, agradezco enormemente a mi familia, mi pareja y mis amigos más cercanos por haber sido un soporte clave durante este periodo de mi vida. Gracias por tener ilusión con cada logro mío y darme ánimos para alcanzar más metas. En especial doy gracias a mis padres por la educación que me han dado y por enseñarme los valores que me hacen querer seguir adelante.

Este trabajo ha sido financiado por el Ministerio de Ciencia e Innovación a través de los proyectos PID2019-107228RBI00, AEI/10.13039/501100011033, PDC2022-134013I00 y TED2021-131019B-I00.

Licencia



© Esta obra está bajo una licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 4.0 Internacional.

Resumen

El objetivo de este trabajo es desarrollar experimentos con SYCL, una plataforma de abstracción centrada en el paralelismo, como una forma de introducción a la computación de altas prestaciones. La relevancia de esta plataforma reside en que define un protocolo uniforme para la ejecución paralela, de manera que el código sea portable entre varios proveedores y plataformas. De esta forma, SYCL actúa como una capa de abstracción que reduce la dependencia entre el código y la plataforma física de ejecución. La experimentación con SYCL incluye benchmarks comparativos con CUDA y ejecución en serial para ver la competencia práctica de SYCL. Por otro lado, examinamos SYCL en el contexto de un proyecto real en la industria, desarrollando un programa de procesamiento de imágenes.

El objetivo de este trabajo ha sido introducir al estudiante en el ámbito de la computación de altas prestaciones, HPC por sus siglas en inglés. Para ello se ha utilizado como lenguaje vehicular SYCL. SYCL es un modelo de programación de alto nivel que se ha desarrollado con el objetivo de mejorar la productividad de la programación en entornos de computación heterogénea. Es asimismo un estándar promovido por Khronos Group que fue anunciado en 2014. Se trata de un lenguaje específico de dominio de código fuente único basado en C++.

Sistemas de computación heterogénea son aquellos que utilizan diferentes tipos de núcleos de cómputo como CPUs, GPUs, ASICs, FPGAs o NPUs. Al asignar diferentes cargas de trabajo a procesadores diseñados para fines específicos o procesamiento especializado, se mejora tanto el rendimiento como la eficiencia energética. El enfoque tradicional a la hora de programar este tipo de sistemas obligaba a desarrollar códigos específicos para los diferentes tipos de aceleradores hardware presentes en el sistema. Una de las grandes fortalezas de SYCL es que el programador desarrolla un único código fuente escrito en un lenguaje que es una extensión de C++ y ese mismo código es traducido por el sistema SYCL para su ejecución en los diferentes aceleradores presentes en la plataforma de cómputo. Así pues, SYCL actúa como una capa de abstracción que reduce la dependencia entre el código y la plataforma física de ejecución.

En este Trabajo Fin de Grado, además de la formación en los conceptos y técnicas necesarias para el uso de la plataforma, se ha incluído la evaluación de diferentes benchmarks que comparan las prestaciones de diferentes códigos programados tanto usando SYCL como CUDA, el estándar de facto en la programación de GPUs de Nvidia.

Asimismo se ha utilizado SYCL para programar una aplicación de proceso de imágenes para acreditar el beneficio del uso de esta plataforma frente al enfoque tradicional para este tipo de aplicaciones de cómputo intensivo.

Palabras clave: Paralelismo, SYCL, Acelerador, Altas Prestaciones, Kernel, Benchmark, Speed-up, Portabilidad.

Abstract

The objective of this work is to develop experiments with SYCL, an abstraction platform focused on parallelism, as a way of introduction to high performance computing. The relevance of this platform lies in the fact that it defines a uniform protocol for parallel execution, so that the code is portable across multiple vendors and platforms. In this way, SYCL acts as an abstraction layer that reduces the dependency between the code and the physical execution platform. Experimentation with SYCL includes comparative benchmarks with CUDA and serial execution to see the practical competence of SYCL. On the other hand, we examine SYCL in the context of a real industry project, developing an image processing program.

The objective of this work has been to introduce the student to the field of high performance computing (HPC). For this purpose, SYCL has been used as the vehicular language. SYCL is a high-level programming model that has been developed with the aim of improving the productivity of programming in heterogeneous computing environments. It is also a standard promoted by Khronos Group that was announced in 2014. It is a single source code domain-specific language based on C++.

Heterogeneous computing systems are those that use different types of computing cores such as CPUs, GPUs, ASICs, FPGAs or NPUs. By assigning different workloads to processors designed for specific purposes or specialized processing, both performance and energy efficiency are improved. The traditional approach to programming such systems required the development of specific code for different types of hardware accelerators present in the system. One of the great strengths of SYCL is that the programmer develops a single source code written in a language that is an extension of C++. language that is an extension of C++ and that same code is translated by the SYCL system for execution on the different accelerators present in the platform. Thus, SYCL acts as an abstraction layer that reduces the dependency between the code and the physical execution platform.

In this Final Degree Project, in addition to the training in the concepts and techniques necessary for the use of the platform, it has been included the evaluation of different benchmarks that compare the performance of different codes programmed using SYCL as well as codes programmed using both SYCL and CUDA, the de facto standard in Nvidia GPU programming.

SYCL has also been used to program an image processing application to demonstrate the benefit of using this platform versus the traditional approach for this type of computationally intensive applications.

Keywords: Parallelism, SYCL, Accelerator, High Performance, Kernel, Benchmark, Speed-up, Portability.

Contents

Introduction **1**

1 Goals **5**

2 Related Work **7**
2.1 OpenMP . . . 7
2.2 HPL . . . 8
2.3 CUDA . . . 10
2.4 OpenCL . . . 12
2.5 Directive-based Languages for Accelerators . . . 14
2.5.1 OpenMPC - OpenMP Extended for CUDA . . . 14
2.5.2 hiCUDA . . . 15
2.5.3 PGI Accelerator Model . . . 15
2.5.4 OpenACC . . . 16

3 SYCL **17**
3.1 What is SYCL? . . . 17
3.2 The Queue . . . 19
3.2.1 Task Graph . . . 19
3.2.2 Device Selection . . . 20
3.2.3 Errors and Exceptions . . . 21
3.3 Buffer/accessor Model . . . 22
3.4 Unified Shared Memory Model . . . 24
3.5 Work Submission . . . 25
3.5.1 Memory Operations . . . 26
3.5.2 Basic Kernels . . . 27
3.5.3 NDRange . . . 28

4 Benchmark Comparisons **29**
4.1 Execution Platform . . . 29
4.2 Mandelbrot set . . . 30

4.3 Floyd–Warshall algorithm 33
4.4 Molecular dynamics 35
4.5 Backpropagation 36

5 An Industry Case Study: Image Processing with SYCL 39
5.1 The erosion operation 39
5.2 Supporting Code 41
5.3 Erosion Solution Development 42
5.3.1 Serial Implementation 42
5.3.2 SYCL Implementation 44
5.4 Results 49

6 Conclusiones y Líneas de Trabajo Futuras 53

7 Conclusions and Future Lines of Work 55

8 Budget 57

Bibliography 60

List of Figures

1	Trends of microprocessors.	2
2	SYCL implementations.	4
2.1	CUDA memory hierarchy.	11
2.2	OpenCL platform model.	13
2.3	OpenCL NDRange.	14
3.1	Dependency graph example. *Image From Data Parallel C++* [1]	20
3.2	Dissection of an NDRange. *From Data Parallel C++* [1].	28
4.1	Mandelbrot benchmark. Results for SYCL, CUDA and serial executions.	31
4.2	Mandelbrot benchmark. Results for SYCL and CUDA executions.	32
4.3	Mandelbrot benchmark. Speed-up graph for SYCL and CUDA executions.	32
4.4	Floyd–Warshall algorithm. Results for SYCL, CUDA and serial executions.	33
4.5	Floyd–Warshall algorithm benchmark graph for SYCL and CUDA executions.	34
4.6	Floyd–Warshall algorithm speed-up graph for SYCL and CUDA executions.	35
4.7	MD benchmark. Results for SYCL and CUDA executions.	36
4.8	Backpropagation benchmark. Results for SYCL and CUDA executions.	37
5.1	Simplified erosion operation.	40
5.2	Conceptual UML class diagram for the erosion application.	41
5.3	Original FOC sample image.	49
5.4	Zoomed in FOC image. Original (left) and eroded (right).	50
5.5	Original Wooptix sample image.	51
5.6	Zoomed in Wooptix sample image. Original (left) and eroded (right).	52

Preface

High Performance Computing (HPC) [2] is a practice that utilizes powerful processors in parallel to process big data and solve complex problems at incredibly high speeds. These systems operate at rates that are significantly faster than those of regular systems. Supercomputers, which incorporate millions of processors, have traditionally been the norm for HPC. Currently, the fastest supercomputer is Frontier, located in the United States, with a processing speed of 1.102 exaflops (quintillion floating point operations per second).

HPC enables organizations to gain a competitive advantage by revealing new information that advances human knowledge. It is used for tasks such as DNA sequencing, automating stock trading, as well as running AI algorithms and simulations. An important example regarding the latter case is autonomous driving.

All these processes analyze massive amounts of streaming data from IoT sensors, radar systems, and GPS in real time to make split-second decisions. Higher performance in computer science is one of the key factors behind the evolution of hardware and software. There exists a necessity for faster, more efficient calculus and data management. The way we are able to fulfil this need has evolved with time. For a significant number of decades, performance was increased by the upgrade in frequency of single-threaded CPUs, as seen on Figure 1. But there was a moment where an increase in frequency did not justify the also scaling power consumption.

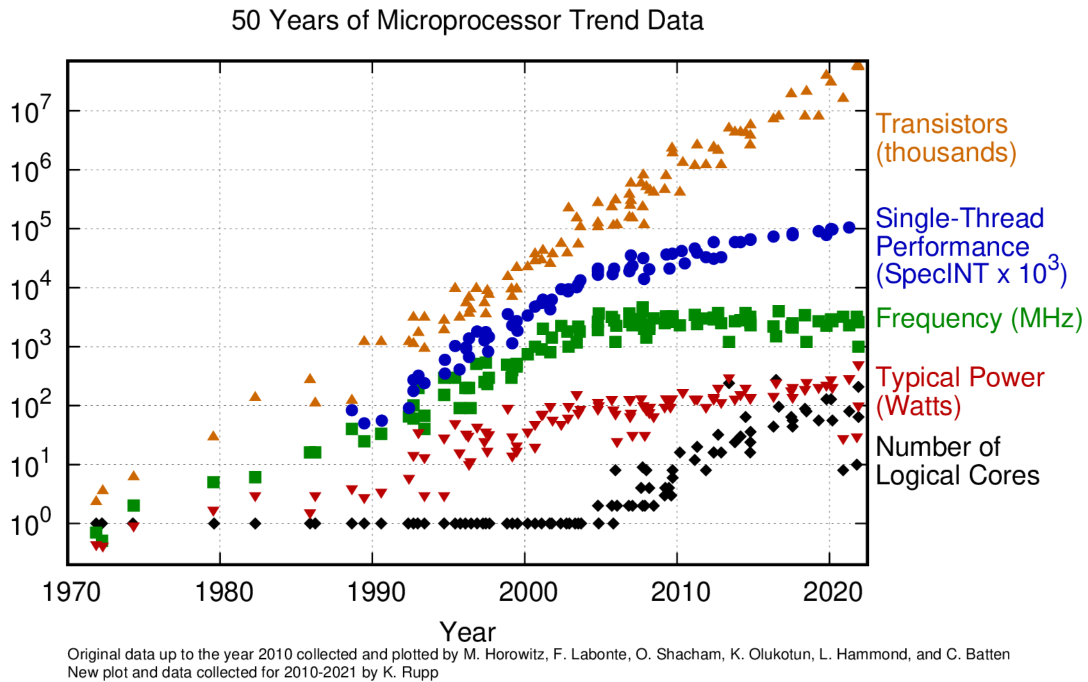


Figure 1: Trends of microprocessors.

This particular moment (around 2006) is known as "Hit the *Power Wall*" [3]. At that time, the performance improvements of uniprocessors had come to an end due to power constraints. This was a clear indicator that other factors have to be modified in order to gain an increase in performance. The next step is to invest on parallel architectures.

On the other hand, in the scientific and technological field, HPC is a great tool that helps researchers. It allows the simulation of complex environments and systems (useful in physics, biology, chemistry, etc.) and the performance of intensive calculations of long duration in conventional computers, among other applications. Due to its great time efficiency and the variety of areas of knowledge in which HPC can be used, the potential for the study of parallel computing and its application in high performance computing is clear.

There has been interest in parallel computation for a long time. In fact, parallel computers existed before 1980 (more in-depth on page 14 of *"Parallel Computing Works!"* [4]), but greater achievements on the field were accomplished later on. Nowadays, there is a wide variety of devices for different kinds of computation objectives (CPU, FPGA, GPU, ASIC, etc.). It has become an heterogeneous world full of distinct architectures. As a matter of fact, it has been claimed that

we are currently in *"A New Golden Age for Computer Architecture"* [5].

The reasons behind the different specializations is to optimize for a particular kind of task. This entails a more effective use of memory bandwidth and increased performance from the deliberate elimination of unnecessary accuracy, among other advantages.

With the vast amount of available devices comes the also varied collection of APIs and other utilities regarding the use of both hardware and software. Some relevant abstraction mechanisms for parallel work include CUDA[1], OpenMP[2], OpenCL[3] and specially SYCL [6], but keep in mind that this is nowhere near the total amount of tools that provide this service.

This situation is quite problematic because this heterogeneous world requires heterogeneous solutions and tools. This means that, in order to produce code that is targeted towards multiple back-ends or devices, there needs to be an abstraction that permits to do so, allowing multiple tools to work together. Moreover, regarding HPC, both hardware and software abstraction utilities may allow for an easier way of expressing scalability, portability and versatility. This begs for a tool that provides a more generalistic yet time and performance efficacious approach. In response to this situation comes SYCL.

SYCL is a single source, high-level, standard C++ programming model, that can target a range of heterogeneous platforms. There are many back-end implementations that support SYCL (Fig. 2). We are going to focus on the Intel DPC++ branch[4].

---

[1] CUDA Library of Resources `https://developer.nvidia.com/cuda-zone`

[2] OpenMP API specification `https://www.openmp.org`

[3] OpenCL Main Page `https://www.khronos.org/opencl/`

[4] Intel oneAPI DPC++ compiler `https://github.com/intel/llvm`

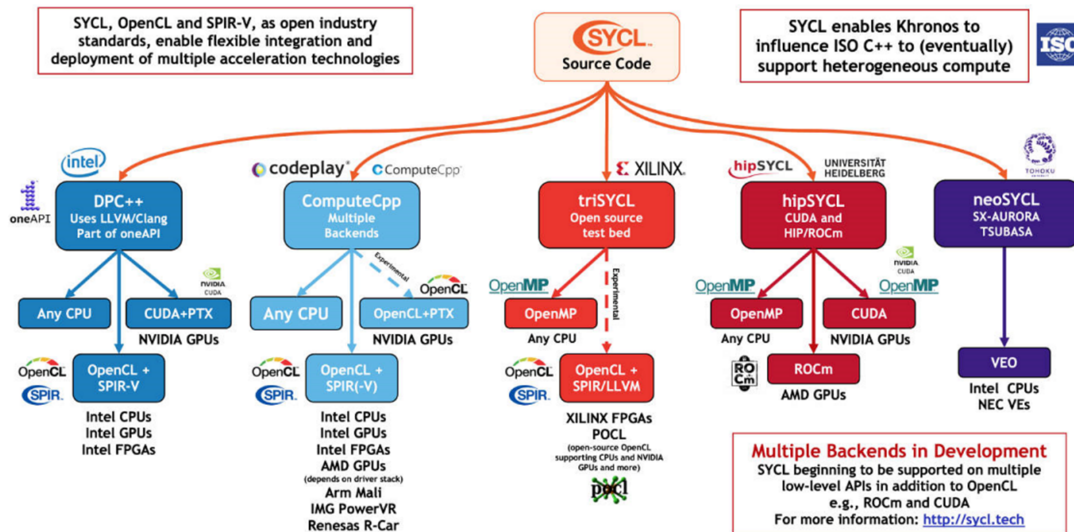


Figure 2: SYCL implementations.

This work aims to perform a deep exploration of SYCL and its capabilities, as well as producing benchmark tests with other platforms such as CUDA. Another topic of interest is the analysis and implementation of different data management models. As a given, this study will be covering these tasks from a HPC perspective when possible.

All the code examples written by the student for this work follow the principles of Martin's Clean Code book [7] and conform to the Google's Style Guide[5], while code excerpts from other works will remain essentially untouched. The reason behind the decision to write code using Google's Style is completely arbitrary, the importance of choosing a style relies on being consistent with its use: *"The last thing we want to do is add more complexity to the source code by writing it in a jumble of different individual styles." - Robert C. Martin.*

Every piece of code written by the student for this project is publicly available on the GitHub repository dedicated to this work [8]. Also note that every listing has a link to its original source and in the case of the student's code, there will be a direct link to the corresponding file in the work's repository, labeled as *"See on GitHub"*.

[5] Google Style Guides `https://google.github.io/styleguide/`

Chapter 1

Goals

This document summarizes the research and development work carried out by the student in the achievement of his Final Degree Project (*Trabajo de Fin de Grado*, TFG), which will conclude his studies for the degree *Grado en Ingeniería Informática* at the *Escuela Superior de Ingeniería y Tecnología* at the Universidad of La Laguna (ULL).

This project has the following main goals:

1. A first objective has been for the student to acquire basic knowledge on the topic of High Performance Computing. Achieving this objective requires an effort on the part of a student of the Degree in Computer Science at the ULL who has to familiarise himself with diverse concepts that are not studied in the syllabus of this degree. Concepts such as performance, acceleration, latency, data parallelism, portability, accelerators, etc. have had to be studied on their own since they are not studied in the necessary depth in the syllabus, and others have also required effort on the part of the student to be understood with greater precision.

2. On the one hand, the aim is to broaden the knowledge of the parallel programming model using *SYCL* [6] and the development of applications for this scheme in the context of *HPC* [2].

3. Another objective of this work is to investigate and deepen in the techniques and technologies related to *Data Parallel C++* [1] present today.

4. At the same time, the student is expected to acquire knowledge about the different implementations of the parallel programming model as well as the necessary tools and techniques to optimize its performance efficiently.

5. Along with the previous point, the student should also elaborate a comparative

study between different parallel programming implementations, focusing on practical and execution related differences.

6. Finally, after the corresponding research and information gathering, the student is expected to apply the acquired knowledge to develop some functional implementation that meets the proposed needs.

Chapter 2

Related Work

Given the growing relevance of heterogeneous distributed memory systems and the large development effort they pose nowadays, the research community has come up with a number of interesting proposals to facilitate their usage [9, 10, 11, 12, 13]. Although this work mainly covers the SYCL platform applied to HPC, it is also important to have an overview of the various approaches for parallel computing and related technologies. There has been several developments in this field such as the ones cited above, although we are not going to discuss all of them.

This chapter is intended to make us aware of the general sense regarding the basic logic behind parallel oriented APIs and tools. Inherently from one abstraction to another, we will see similarities within their core, revealing crucial mechanisms of parallel reasoning as well as establishing a connection between abstract procedures and physical parallel-oriented devices.

2.1 OpenMP

OpenMP[1] is the predominant API used to manage shared-memory parallelism used in scientific applications [14]. It allows for more efficient load balancing for multithreaded tasks. This abstraction is also designed to be a flexible standard, so it becomes easy to implement on different platforms. In essence, OpenMP extends C/C++ and Fortran with compiler directives and runtime functions that allow for a high level of parallelism expressiveness [15]. It is composed of the following basic ideas:

- Control structure: Reduced amount of control structures, enough for most parallel applications.
- The data environment: Environment context for each process.

[1] The OpenMP API specification for parallel programming `https://www.openmp.org`

- Synchronization:
  - **Explicit** synchronization using interprocess communication, which is slow.
  - **Implicit** synchronization present when starting and finishing parallel and control constructs.
  - OpenMP also offers different **tools for synchronization**, depending on the specific action and/or conditions, which are usually more time efficient than explicit synchronizations.
- The runtime library: A miscellaneous set of mechanisms to tune an application, such as dynamically changing the number of processes used to execute parallel regions.

Listing 2.1 presents a simple OpenMP application which runs a parallel for loop written in C++.

```
void simple(int n, float *a, float *b) {
  int i;
#pragma omp parallel for
  for (i=1; i<n; i++) /* i is private by default */
    b[i] = (a[i] + a[i-1]) / 2.0;
}
```

Listing 2.1: Consecutive pairs average on OpenMP. *Original source*.

The purpose of the function is to calculate the average of each pair of consecutive elements in the a array and store the result in the corresponding position of the b array. The OpenMP directive `#pragma omp parallel for` is used to parallelize the loop. As noted in the example code, the loop counter i is implicitly private in OpenMP, meaning that each thread gets its own private copy of i.

2.2 HPL

The HPL (Heterogeneous Programming Library) [16] is a framework built on top of OpenCL that enables the exploitation of heterogeneous computing in C++. Within HPL, the primary application operates on the host, and the code segments executed in OpenCL consist of kernel functions. These functions can be written either directly in C++ using the HPL embedded language or in native OpenCL C.

The main functionality of this library comes with its `Array` class, which encapsulates the data to be manipulated inside of the kernels, indicating both the data type and the number of dimensions of the array. On the other hand,

scalar types should also be encapsulated within their HPL equivalent (`Float`, `Int`, etc.). Listings from this section illustrate these ideas.

```
using namespace HPL;

// SAXPY kernel in which thread idx computes y[idx]
void saxpy(Array<float,1> y, Array<float,1> x, Float alpha) {
  y[idx] = alpha * x[idx] + y[idx];
}

int main(int argc, char **argv) {
  Array<float, 1> x(1000), y(1000);
  float alpha;
  // The vectors x and y are filled in with data (not shown)
  // Run SAXPY on an accelerator, or the CPU if no OpenCL capable accelerator
     is found
  eval(saxpy)(y, x, alpha);
}
```

Listing 2.2: SAXPY on HPL (HPL embedded language). *Original source*.

Listing 2.2 displays how to perform a SAXPY (single precision A X plus Y) operation using the HPL embedded language. To create a kernel we simply write a regular C++ function and use the data types provided by the library as the argument types. To invoke the kernel, we call the `eval()` function with the kernel name as the argument followed by another call with the arguments for the execution of the kernel, as shown in Listing 2.2

Since this platform is based on OpenCL, there is a method to execute already coded OpenCL kernels. Listing 2.3 exemplifies this case. The key difference from the former example remains on the procedure for the kernel call. Just before the `eval()` call, we associate the OpenCL version of the kernel (specified in the `saxpy_kernel` string) with the HPL function handle via the `nativeHandle()` function, which arguments are a function whose arguments define an equivalent version of the kernel written as specified for the HPL framework (additional details regarding the kernel behaviour can be specified), the name of the kernel function within the code and the actual OpenCL code.

```
using namespace HPL;

// String with the OpenCL C kernel for SAXPY
const char* saxpy_kernel =
  "__kernel void saxpy(__global float *y, __global float *x, float alpha) {\n \
     int i = get_global_id(0);                                             \n \
     y[i] = alpha * x[i] + y[i];                                           \n \
  }";

// Function whose arguments define the kernel for HPL
void saxpy_handle(InOut< Array<float,1> > y, In< Array<float,1> > x, Float
    alpha) { }

int main(int argc, char **argv) {
  Array<float, 1> x(1000), y(1000);
  float alpha;
  // The vectors x and y are filled in with data (not shown)
  // Associate the kernel string with the HPL function handle
  nativeHandle(saxpy_handle, "saxpy", saxpy_kernel);
  eval(saxpy_handle)(y, x, alpha);
}
```

Listing 2.3: SAXPY on HPL (OpenCL kernel). *Original source*.

2.3 CUDA

As presented in [17], CUDA[2] is a programming model and a parallel computing platform developed by NVIDIA, aimed to be used in general purpose GPUs for their CUDA-enabled GPU devices. It was originally launched in 2007 and one of its main goals is to allow for the creation of scalable programs within the parallel computing paradigm. To accomplish this, CUDA offers a simple yet powerful abstraction based on three key points: a hierarchy of thread groups, shared memories and barrier synchronization.

[2] CUDA C++ documentation https://docs.nvidia.com/cuda

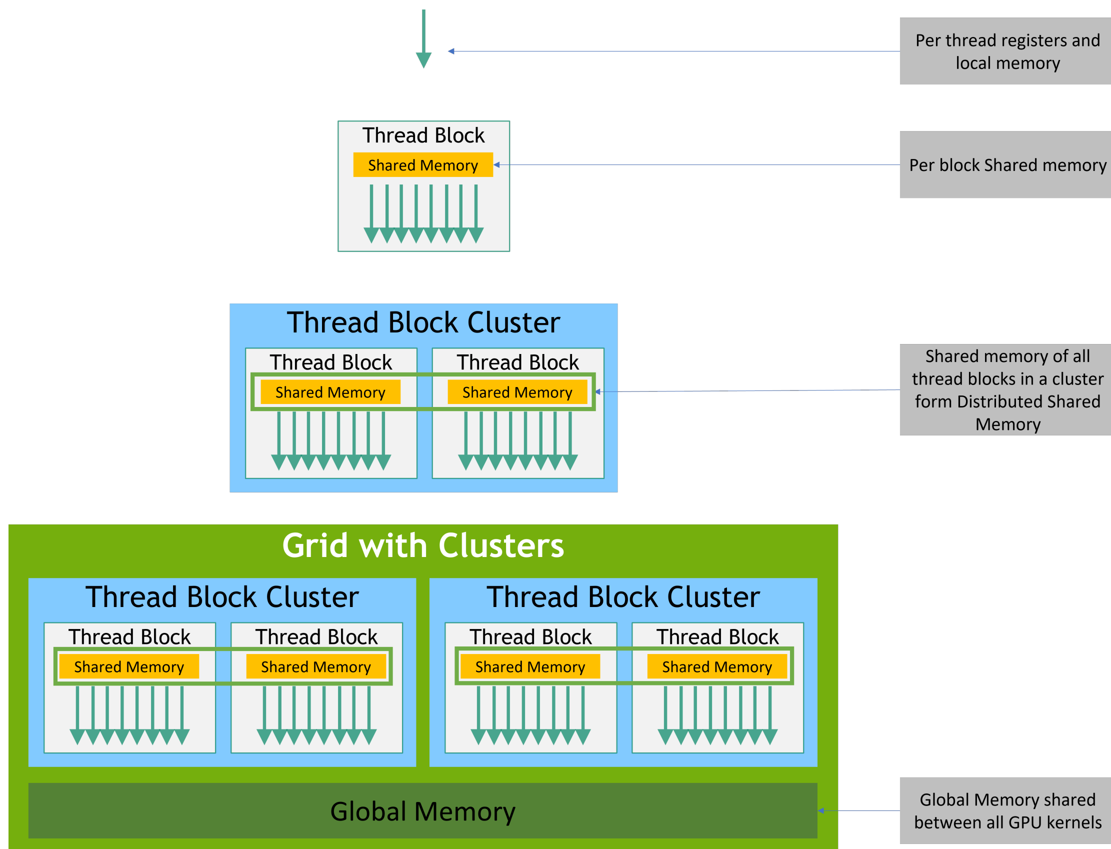


Figure 2.1: CUDA memory hierarchy.

With the CUDA platform, problems can be subdivided in blocks of threads within a grid and subproblems can be solved in parallel cooperation within a block. The memory system in place for all of these pieces works as shown in Figure 2.1 The division in multiple independent pieces allows for scheduling on an undetermined number of GPU processors, thus enabling hardware scalability. This platform is used broadly to solve real world problems regarding physics, biology and data mining, among others.

The CUDA platform is available on a wide variety of languages, primarily focusing on C/C++ and Fortran. To display the basic usage and expressions for CUDA, we present in Listing 2.4 an example program written in C++.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C) {
  int i = threadIdx.x;
  C[i] = A[i] + B[i];
}

int main() {
  ...
  VecAdd<<<1, N>>>(A, B, C); // Kernel invocation with N threads
  ...
}
```

Listing 2.4: Vector add on CUDA. *Original source*.

First of all, we have a declaration of a function starting with the `__global__` keyword, which specifies the following function as a kernel, which can only be executed on the device. Inside of it there is the actual kernel code, in which we can highlight the use of `threadIdx`, a 3-dimensional vector which identifies the working thread.

Similarly, we have `blockIdx` that provides a unique identification for thread blocks. On the other hand, inside of the main function there is a kernel invocation. This asynchronous call comes with execution configuration denoted by `<<<1, N>>>`, in which the first parameter specifies the number of blocks and the second specifies the number of threads per block. Both of which can be simple integers or `dim3` values to indicate these are multidimensional entities.

2.4 OpenCL

OpenCL is an open standard designed for general-purpose parallel programming on multi-core architectures [16, 18].

It addresses a wide range of applications and acts as an efficient, low-level programming interface. The goal of OpenCL is to establish itself as the foundation platform for a parallel computing ecosystem. OpenCL aims to be a tool to produce portable yet efficient code. There is a clear division of ideas that comprehend the standard, which is composed of the models: Platform, Memory, Execution and Programming. These are explained in further detail in the reference above. Following, we have an overview of the OpenCL functionality.

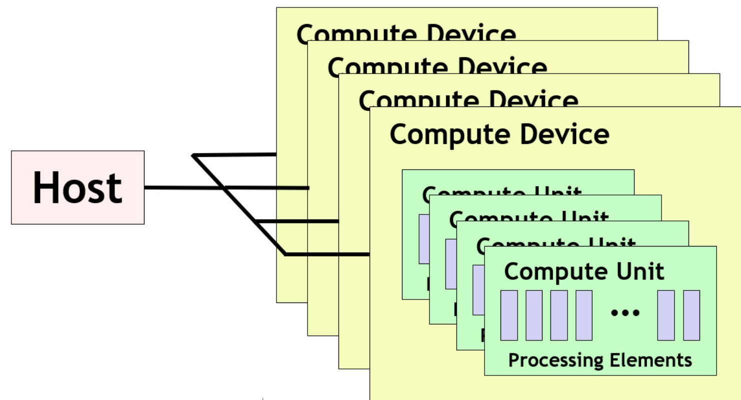


Figure 2.2: OpenCL platform model.

As we can see in Figure 2.2, an OpenCL host is connected to one or more OpenCL devices, which in turn are composed of one or more compute units (CU). Such units are subdivided in one or more processing elements (PEs). The PEs are the ones in charge of executing actual code.

The way an OpenCL application operates is by running the host program on the host platform, which is in charge of defining the kernel contexts and managing kernel executions, whilst submitting commands from it to execute kernels on the devices.

Similarly to CUDA, kernels are scheduled to run under a defined index space, meaning that each kernel instance has a specific identification, which determines how the kernel will execute.

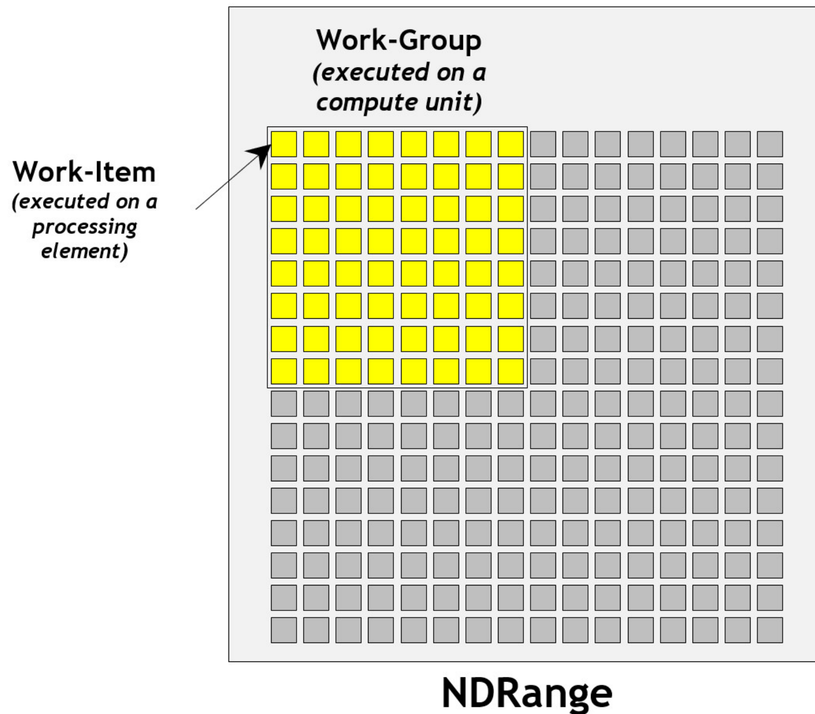


Figure 2.3: OpenCL NDRange.

A PE is responsible for the execution of a work-item, which is encompassed within a work-group, as we can see in Figure 2.3. This coarse-grained composition allows for the execution of different strategies when using the index space. Furthermore, the idea of a N-dimensional index space (NDRange), where N is one, two or three, gives users high flexibility to refine their works. This abstraction shapes the indexes used in the work-items, affected by the dimensionality and work group divisions as well.

2.5 Directive-based Languages for Accelerators

In [19], the authors present the most relevant approaches that have been used to leverage heterogeneous architectures using languages enhanced with the use of specific directives.

We will now review the most significant of these approaches.

2.5.1 OpenMPC - OpenMP Extended for CUDA

OpenMPC [20] is an abstraction of the CUDA programming model built on OpenMP. OpenMPC performs a translation from OpenMP to CUDA with the

addition of special directives and variables, which are used to make CUDA-specific optimizations. There is an extensive set of clauses and environment variables. Examples of some of these clauses are:

- `maxnumofblocks(N)`: Specifies the maximum number of thread blocks for a kernel.
- `threadblocksize(N)`: Specifies the thread block size for a kernel.

The translation from OpenMPC to CUDA starts by analyzing the code and passing the result to the OpenMP to CUDA translator, which performs the actual translation aided by the optimization information obtained from the previous step.

2.5.2 hiCUDA

hiCUDA [21] is a high-level abstraction layer built on top of CUDA. It provides an easy to use interface that solves mechanical tasks for the development of CUDA programs. An important highlight that motivates the use of hiCUDA is the fact that the process of migration from existing code to CUDA may be challenging, as evidenced by the need for programmers to manually handle intricate tasks such as managing data transfers between host and GPU memories, and optimizing GPU memory utilization. These tedious tasks are alleviated by the automated work that hiCUDA can offer.

This tool can extract kernels from their original source and decide how to allocate the work threads and blocks according to user defined configuration clauses. This is accomplished using special hiCUDA directives, which would then be translated to actual CUDA code. The authors have developed a prototype of this idea which does exactly that. Is also important to note that the same source files of a hiCUDA project can be used to create both sequential and GPU versions of the code.

2.5.3 PGI Accelerator Model

The PGI Accelerator Model [22] was created by The Portland Group for the Fortran and C languages. It is essentially a set of directives designed to guide the compiler in creating kernels and regions of code that can be offloaded to an accelerator device. This model allows for portability across multiple operating systems, various accelerators and types of host CPUs.

The main functionality is provided by the `acc region` directive, which specifies a region within the code that contains a parallel loop kernel. Most of the directives provided by this model are optional, and are mostly used to improve performance based on compilation hints. On the other hand, this abstraction offers implicit

mechanisms such as data flow and array region analysis to determine when data transfers between host and device should occur, as well as accelerator startup and shutdown, among other uses.

2.5.4 OpenACC

Considering the OpenACC standard, the ULL GCAP[3] research group (*High Performance Computing Group*) developed their own version of the compiler, called `accULL` [18], standing as one of the few available implementations of the OpenACC standard. OpenACC shares an important piece of the high-level features present in the PGI Accelerator Model, it is based on directives that indicate regions of code that could be run on an accelerator device. This abstraction frees the developer from writing device specific code details, allowing them to focus on other tasks. The main *pragma* regions supported by OpenACC are the following:

- `data`: Specifies data regions.
- `kernels`: Groups of loop nests that can be executed on the devices.
- `parallel`: Similar to `kernels` but allows for better control over the code.

There are other annotation mechanisms to further tailor the execution and behaviour of OpenACC such as clauses that can reduce memory transfers like `copy_in` or `copy_out`.

[3] ULL GCAP https://portalciencia.ull.es/grupos/6369/detalle

Chapter 3

SYCL

In this chapter we will take a deep dive into the mechanisms and abstractions that the SYCL platform offers.

We chose *Data Parallel C++* [1] as the baseline book to learn about SYCL. It is a very recently published reference for this platform (first edition published in November 2020, second in October 2023) that also serves as an introduction to parallel programming, as it introduces the very basic concepts and builds on them in a beginner friendly manner. On the other hand, SYCL Academy[1] fills the need for a more practical approach for an introduction to SYCL, offering a 20 lectures long tutorial, containing both lessons and exercises.

Based on the study of the mentioned sources, in the following sections we will cover the foundational ideas and procedures contained in the SYCL platform.

Nevertheless, this chapter does not aim to provide an exhaustive explanation of SYCL and its programming API. For a deeper understanding, the interested reader is referred to the above references, also including the SYCL technical specification [23].

3.1 What is SYCL?

SYCL is a parallel focused abstraction layer created for C++. Although the idea of a mechanism that provides tools for parallelism existed for a long time, the unique trait of SYCL is that it attempts to define a uniform protocol for parallel execution. In such a way, it allows for portable programmability across multiple vendors and platforms. Thus, bringing to existence a powerful tool which fulfils the following statement from *Parallelizing the Standard Algorithms Library* [24]:

"...standard and broadly-accessible functionality should be constructed to bridge

[1] SYCL Academy `https://github.com/codeplaysoftware/syclacademy`

the gap between the abundant parallelism implicit in many applications and the concurrent resources of the target architecture...”

This is a notable effort since these features were desired for almost a decade by the time SYCL 2020 was launched.

It is worth mentioning that SYCL has some similarities to OpenCL, sharing concepts and terminology like NDRanges and command queues (called just queues in SYCL). In fact, when SYCL was created it was an abstraction layer only for OpenCL, and later it was expanded to support multiple back-ends in SYCL 2020.

Having stated what SYCL is, we will review a simple SYCL program that performs a scalar addition so we can grasp how a minimal SYCL operates. In the first line of Listing 3.1 we have the inclusion of the SYCL header file.

```
1  #include <sycl/sycl.hpp>
2  #include <iostream>
3
4  int main() {
5    int summand_a{1}, summand_b{2}, result{0};
6    sycl::queue queue;
7    { // Buffer scope
8      sycl::buffer buffer_a(&summand_a, sycl::range(1));
9      sycl::buffer buffer_b(&summand_b, sycl::range(1));
10     sycl::buffer buffer_result(&result, sycl::range(1));
11
12     queue.submit([&](sycl::handler& handler) {
13       sycl::accessor in_a{buffer_a, handler}
14       sycl::accessor in_b{buffer_b, handler}
15       sycl::accessor out_result{buffer_result, handler}
16
17       handler.single_task([=]() {
18         out[0] = in_a[0] + in_b[0];
19       });
20     }).wait();
21   }
22   std::cout << result << std::endl;
23 }
```

Listing 3.1: Scalar add example using the buffer/accessor model. *See on Github*.

This code performs the sum of `summand_a` and `summand_b`, writing the result in the `result` variable. On line 6 we have the key object of any SYCL program, the queue. It allows for communication between host and device and execute various operations. Lines 8-10 define buffers to manage the previously stated data.

Then we have a call to `submit` in line 12, which schedules a *command group* to be executed. Which is, to simplify, a set of arbitrary code and a kernel execution call. This is the first step to get actual kernel code executing on a device.

Inside this command group there are three *accessors*, which grant access to

the buffers within the kernel. On lines 17-19 there is the kernel definition and invocation.

In the next sections of this chapter we will cover the details and unexplained elements of this example.

3.2 The Queue

Queues[2] are the main piece of action in SYCL. They allow the host program to communicate with an underlying device or devices. An instance of a queue is connected to one device and can execute different operations concerning the device itself and between host and device. Note that multiple queues can be bound to the same device but one queue is only bound to a single device. Another point to mention is that the SYCL host program can be executed on any type of physical device as long as it supports C++17, although is often going to be executed on a CPU.

3.2.1 Task Graph

SYCL organizes the work to be performed using a dependency graph.[3] Each node of the graph represents a unit of work, and the edges between them symbolize either custom or automatically set dependencies, as the example shown in Figure 3.1.

[2] https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html#sec:interface.queue.class

[3] https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html#sec:command-groups-exec-order

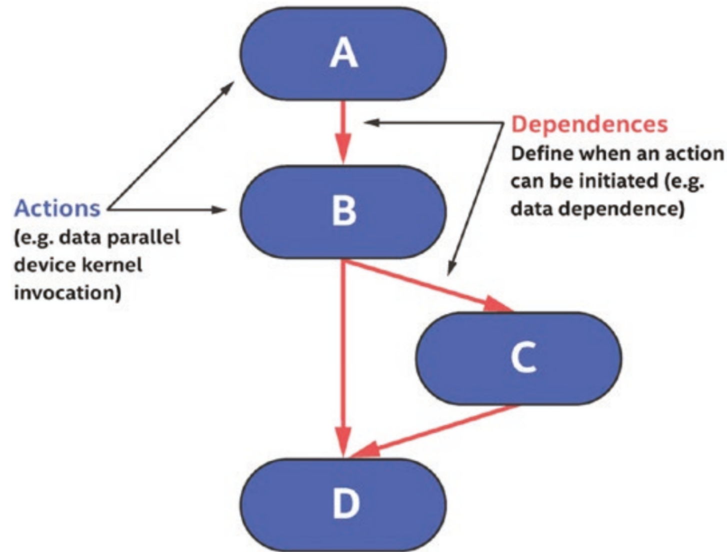


Figure 3.1: Dependency graph example. *Image From Data Parallel C++* [1]

With this structure, it is easy to determine which actions come next, as well as ensuring that the work is safe to execute. SYCL automatically sets dependencies when working with the buffer/accessor model, as it provides the runtime with information regarding the use of the data present in the buffers. On the other hand, in the Unified Shared Memory (USM) model, which is based on memory pointers, there is automatic data dependency management only when using certain types of pointers. It is also possible to create dependencies manually.

3.2.2 Device Selection

When creating a queue, a device is assigned to it. If nothing is specified, the runtime will select a device without taking into consideration the program needs, which can result in a device selection lacking the features the code requires. SYCL will always guarantee that at least one device is available.

There are methods[4] to choose a specific device or with specific characteristics. The functions `cpu_selector_v` (as seen on Listing 3.2) and `gpu_selector_v`, among others, are built-in selectors that can be passed as a parameter to the queue constructor to request for a certain type of device. If device selection fails, the selector throws a `runtime_error` exception.

[4] https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html#sec:device-selector

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

int main() {
  sycl::queue queue{sycl::cpu_selector_v};
  std::cout << "Device: "
            << queue.get_device().get_info<sycl::info::device::name>()
            << std::endl;
}
```

Listing 3.2: Using a CPU selector. *See on Github*.

A more specific approach to device selection can be done by stating *device aspects* using the `aspect_selector` function, which tries to find a device that meets the declared aspects.

There is a list of standard aspects that can be requested, some of them include: `gpu`, `host_debuggable` and `usm_device_allocations`.

There is an even more precise method to select a device. Similarly to the built-in selectors, we can create a custom callable object or function that gives a score to each device. It is an arbitrary procedure, so any technique can be used to calculate the score. A good approach may be to use the `get_info()` function template to retrieve data related to the device and calculate a score based on it.

3.2.3 Errors and Exceptions

When an error occurs in SYCL, it is handled through an exception.[5] There are two types of errors:

- **Asynchronous**, which result in exceptions thrown by the SYCL scheduler and may happen on a device or when trying to launch work on a device.
- **Synchronous** that occur when an error condition can be identified when the host program executes an operation.

[5] https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html#sec:interface.queue.errors

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

int main() {
  auto AsyncHandler = [](sycl::exception_list exceptions) {
    for (auto& exception: exceptions) {
      std::rethrow_exception(exception);
    }
  };
  try {
    sycl::queue queue{AsyncHandler};
    // Potentially exception-prone code
    // ...
  } catch (sycl::exception& e) {
    std::cerr << "SYCL exception has occurred.\n"
              << e.what() << std::endl;
  }
}
```

Listing 3.3: Synchronous and asynchronous error handling procedures. *See on Github*.

The way synchronous and asynchronous errors are handled differ. A synchronous error can be caught within the host program in a similar way to the example of Listing 3.3, where the try-catch block handles the situation when something goes wrong.

On the other hand, asynchronous errors are passed on to an *asynchronous handler*, a function that is called at specific points in the code. This function can be created manually to customize its behaviour and we can find an example in line 6 of Listing 3.3. It can be arbitrarily called using `queue::throw_asynchronous()` (or other similar methods) and is automatically called when a queue or context is destroyed.

3.3 Buffer/accessor Model

Buffers are abstractions that represent an object or collection of objects.[6] The type of the objects they manage can vary from C++ scalar types, SYCL vectors, structures and user-defined types that comply with the notion of being *device copyable*, which we will not go into in detail here.

By themselves, buffers do not hold the data, they simply represent it. This model is based on the interaction with the buffers and stating the actions to

[6] https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html#subsec:buffers

be performed on them.[7] With this information, the runtime schedules all the necessary data transactions to perform the task.

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

int main() {
  int summand_a{1}, summand_b{2}, result{0};
  sycl::queue queue;
  { // Buffer scope
    sycl::buffer buffer_a(&summand_a, sycl::range(1));
    sycl::buffer buffer_b(&summand_b, sycl::range(1));
    sycl::buffer buffer_result(&result, sycl::range(1));

    queue.submit([&](sycl::handler& handler) {
      sycl::accessor in_a{buffer_a, handler}
      sycl::accessor in_b{buffer_b, handler}
      sycl::accessor out_result{buffer_result, handler}

      handler.single_task([=]() {
        out[0] = in_a[0] + in_b[0];
      });
    }).wait();
  }
  std::cout << result << std::endl;
}
```

Listing 3.4: Scalar add example using the buffer/accessor model. *See on Github*.

We will take the first SYCL program presented at the beginning of the chapter to explain the usage of the buffer/accessor model. In lines 8-10 of Listing 3.4, we define the buffers by passing a reference to the original variable and a `range` indicating how many items does the variable hold (if it was an array, its size would match the range).

To use the buffers we use `accessors`, located in lines 13-15. To be created, they just need the buffer that is being accessed and the handler, as the data type of the buffers is deduced. Then we can use the accessors as if they were the original variables.

SYCL provides data consistency and is in charge of moving the information to the places it is used when using the buffer/accessor model. For further optimization, we can communicate to the runtime what we are planning to do with the data. To do this, we can add a third parameter to the definition of the accessors, stating the type of operation that is going to take place, the *access mode*. In the case of the example, it would make sense to mark `in_a` and `in_b` as

[7] https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html#subsec:accessors

`sycl::read_only` and the `out_r` buffer as `sycl::write_only`. This example was not written in this way to prioritise readability, as this is the first contact with the model.

Finally, we need to address line 7, which references the buffer's scope. Getting a buffer out of scope (destroying it) is one of the simplest ways of copying the data managed by the buffer back to the original variable. After line 21, the scope of the buffers ends, resulting in a copy from wherever the most recent version of the information is to its original source. This is one of a handful of methods that exists to retrieve information from a buffer, which include forcing an update of the original variables without destroying the buffer and using a `host_accessor` to access the data from the host.

3.4 Unified Shared Memory Model

USM[8] is a pointer-based model that leverages devices that support a unified virtual address space, meaning that a host memory pointer created using USM serves as a valid pointer address in the device. There are three types of memory allocation:

- `device`: Allocation happens in device memory. Can be directly accessed from the device but has to be explicitly copied to the host to be accessed from there.
- `host`: Host allocated memory that can also be accessed from the device without an explicit copy. To retrieve this data from the device, it is streamed over a bus rather than copied.
- `shared`: Similarly to a `host` allocation, the information can be accessed from both the host and device. The difference is that data can migrate back and forth between host and device.

[8] https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html#sec:usm

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

int main() {
  const size_t kDataSize{1024};
  sycl::queue queue;
  double* summand_a{sycl::malloc_shared<double>(kDataSize, queue)};
  double* summand_b{sycl::malloc_shared<double>(kDataSize, queue)};
  double* result{sycl::malloc_shared<double>(kDataSize, queue)};
  auto initialization_task = queue.parallel_for(kDataSize, [=](sycl::id<1>
      index){
    summand_a[index] = static_cast<double>(index);
    summand_b[index] = static_cast<double>(index);
    result[index] = 0.0f;
  });

  queue.submit([&](sycl::handler& handler){
    handler.depends_on(initialization_task);
    handler.parallel_for(kDataSize, [=](sycl::id<1> index){
      result[index] = summand_a[index] + summand_b[index];
    });
  }).wait();
  sycl::free(summand_a, queue);
  sycl::free(summand_b, queue);
  sycl::free(result, queue);
}
```

Listing 3.5: Multiple scalar additions using the USM model. *See on Github*.

The USM model offers implicit data movement using `host` or `shared` allocations, while also giving the possibility to use explicit data movement through `device` allocations. On Listing 3.5 we can see a parallel scalar addition using `shared` memory. We observe that memory access is pretty straightforward, just as if we were using regular C++ arrays. Although this is simple, behind the scenes there are additional schedules in place to deliver the information. In the last lines of Listing 3.5, we free the memory using SYCL's `free` function.

3.5 Work Submission

A host program can schedule memory management tasks as well as kernel execution tasks.[9] These operations will always be encapsulated within a command group, which is then submitted to the queue.

The submitted work will be organized into the task graph and executed when

[9] https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html#_queue_interface

possible (after meeting the dependency requirements) in an arbitrary order or even may be executed in parallel. This is the behaviour of *out-of-order* queues and the default behaviour of queues. It is possible to create an *in-order* queue by passing the `in_order` queue property to the queue constructor, which will execute tasks one at a time.

Command Group

A command group (CG) is a lambda expression or function object [25] that defines all the necessary elements and details of a task. It takes a SYCL *handler* reference as an argument, which is used to configure the CG and has methods to perform task actions. There are two categories of code within a CG:

- **Host code**: This is arbitrary code that runs on the host, which executes immediately upon submitting the CG. It is used to define buffer accessors and other node dependency configuration, such as `depends_on()` calls to manually set up dependencies. The runtime then uses all the information supplied to determine the relationships between the tasks on the task graph and places the new task where it belongs with the corresponding edges.
- **Action**: Can be either a kernel to execute on the device or an explicit memory operation. It is not required to write an action, but there is a limit to one action per CG.

Although the host code section of a CG can hold any arbitrary host code, it is recommended to **only** write the necessary code to set up the dependencies.

Also note that `depends_on()` calls take an event or events as arguments. Those events represent already submitted tasks and can be obtained from the return object of a submit call.

3.5.1 Memory Operations

Memory transaction operations can vary depending on the context of our code. For example, the `memcopy()` function allows for explicit data movement, which can be used in combination with the explicit version of the USM model to handle memory ourselves. Other memory related functionalities include:

- `memset()`: Fills a memory region with the same unsigned char value.
- `fill()`: Fills a memory region with the same arbitrary object.
- `update_host()`: Updates the memory object referred by an accessor in host to its latest version.

3.5.2 Basic Kernels

There are two types of kernels that can be submitted to the dependency graph. The simplest of them is `single_task()`[10], which executes a single instance of a device function. On the other hand, we can launch multiple instances of device code using `parallel_for()`, which can be executed with different combinations of work sizes.

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

int main() {
  const size_t kDataSize{1024};
  sycl::queue queue;
  double* summand_a{sycl::malloc_shared<double>(kDataSize, queue)};
  double* summand_b{sycl::malloc_shared<double>(kDataSize, queue)};
  double* result{sycl::malloc_shared<double>(kDataSize, queue)};
  auto initialization_task = queue.parallel_for(kDataSize, [=](sycl::id<1>
      index){
    summand_a[index] = static_cast<double>(index);
    summand_b[index] = static_cast<double>(index);
    result[index] = 0.0f;
  });

  queue.submit([&](sycl::handler& handler){
    handler.depends_on(initialization_task);
    handler.parallel_for(kDataSize, [=](sycl::id<1> index){
      result[index] = summand_a[index] + summand_b[index];
    });
  }).wait();
  sycl::free(summand_a, queue);
  sycl::free(summand_b, queue);
  sycl::free(result, queue);
}
```

Listing 3.6: Multiple scalar additions using `parallel_for()`. *See on Github*.

The scalar addition presented earlier serves as a fitting example. In Listing 3.6, we present an addition performed for every element of the summand arrays.

[10] https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html#_single_task_invoke

3.5.3 NDRange

A more granular method of defining a kernel is by using an NDRange, which offers an execution space that can be broken down into different groups, as we can observe in Figure 3.2.

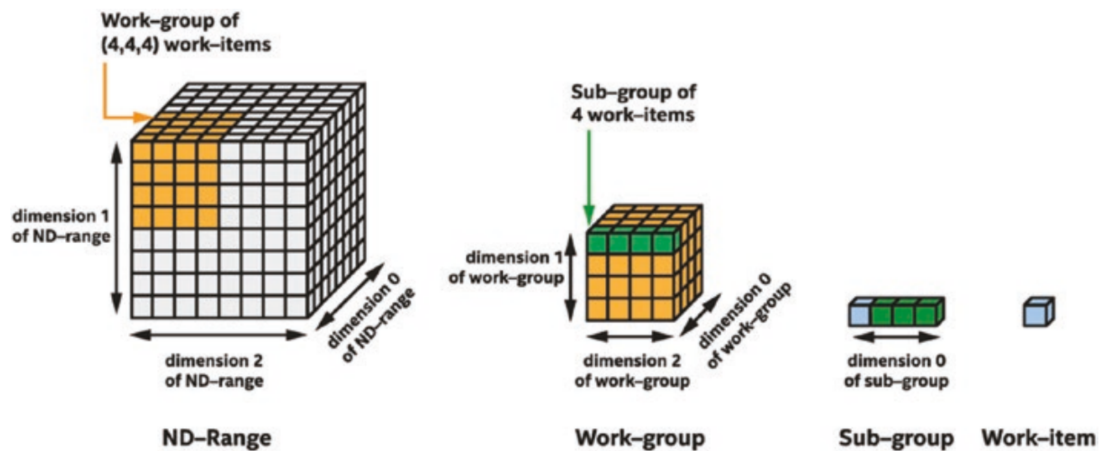


Figure 3.2: Dissection of an NDRange. *From Data Parallel C++* [1].

This coarse-grained structure enables the application of various strategies within the index space, allowing for the implementation of work-group oriented algorithm designs. These include strategies like memory tiling, which take advantage of work-group synchronization procedures.

Chapter 4

Benchmark Comparisons

In this chapter, we will run and compare various benchmarks using SYCL, CUDA and sequential CPU executions. For this purpose, we will use *HeCBench*[1], an assortment of benchmarks specialised in heterogeneous computation. HecBench offers over 400 different benchmarks written in HIP, OpenMP, CUDA and SYCL.

The objective of this study is to examine the practical differences between SYCL, CUDA, and serial executions in terms of performance. Consequently, we will only provide a brief description of the algorithms and concepts used in this chapter. For the benchmarks presented we will not go into the details of the SYCL parallelisation of the code. For details of the code in both CUDA and SYCL, the interested reader is referred to the source code available in the HeCBench repository for these algorithms. In Chapter 5 we will present computational results accompanied by a comparison of SYCL and CUDA implementations for other algorithms.

Since this work serves as an introduction to heterogeneous computation and we aim to present a simplified benchmark report, we will only experiment with problem sizes in order to execute the benchmarks, with the rest of the parameters remaining at their default values.

Lastly, the graphs created for these benchmarks were plotted using Python and MatplotLib[2] among other Python utilities.

4.1 Execution Platform

The experiments shown in this chapter have been executed on the *Verode* platform, a computing infrastructure belonging to the High Performance Computing Group

[1] HeCBench `https://github.com/zjin-lcf/HeCBench/`

[2] MatplotLib `https://matplotlib.org`

(*Grupo de Computación de Altas Prestaciones* or *GCAP*)[3] of the Universidad de La Laguna.

Verode is equipped with two Intel® Xeon® CPU Gold 6230N processors, with 20 cores each, for a total of 40 cores using shared memory, this platform also provides an NVIDIA Tesla V100 GPU, especially designed for HPC and data science purposes.

Regarding the software specifications:

- *Verode* runs under the Debian GNU/Linux 11 (bullseye) operating system.
- SYCL programs were compiled using:
  - `clang++` version 18.0.0.
  - `gcc` version 12.3.0 for the GCC toolchain.
- CUDA programs were compiled using:
  - `nvcc` version 12.0.
  - `gcc` version 6.5.0 as the host compiler.

4.2 Mandelbrot set

The Mandelbrot set is a mathematical concept that was first described by Benoit Mandelbrot in 1980. It is an infinite and infinitely complex fractal shape that emerges from a simple equation involving complex numbers.

For this specific benchmark, we had to make a small change in the code to be able to modify the size of the Mandelbrot set region. As a result, this benchmark offers two parameters to experiment with:

- `repeat`: How many times the algorithm is repeated, then averaged over all times. Using a fixed amount of 1000.
- `size`: Side length of the Mandelbrot set region. It always calculates a square area. Using sizes from 1000 to 45000.

Figure 4.1 shows the results for SYCL, CUDA and serial executions.

[3] ULL GCAP `https://portalciencia.ull.es/grupos/6369/detalle`

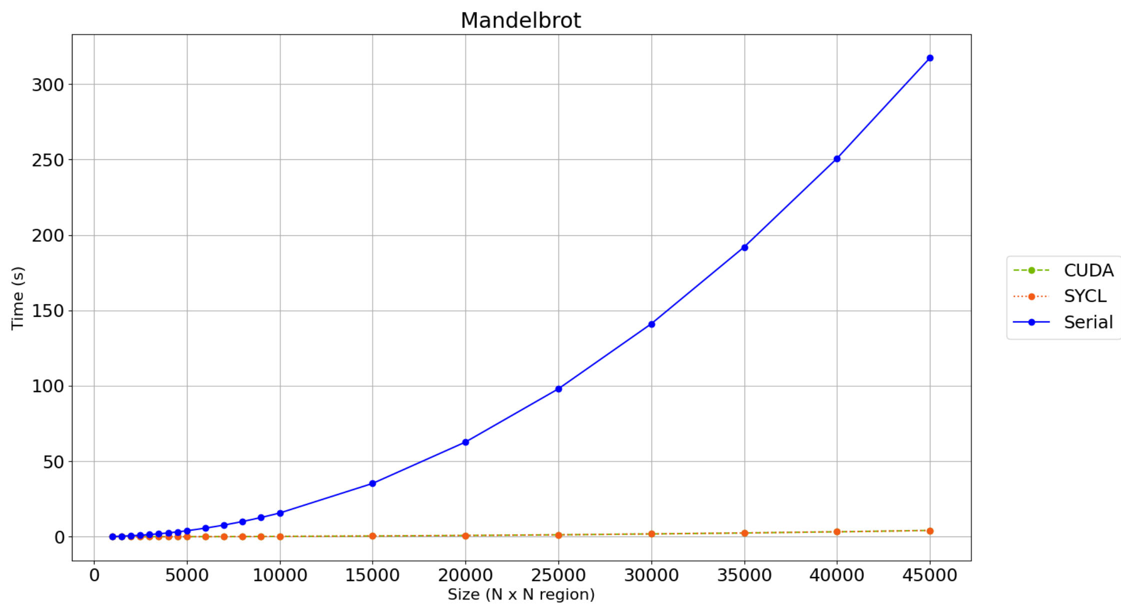


Figure 4.1: Mandelbrot benchmark. Results for SYCL, CUDA and serial executions.

Naturally, we can see in Figure 4.1 that CUDA and SYCL times are significantly better than serial times, reaching a gigantic time gap of over 300 seconds at size 45000.

On the other hand, in Figure 4.2 we can barely notice any differences when comparing the execution times of SYCL and CUDA, which speaks in favour of SYCL since it offers a higher abstraction and justifies any potential performance loss.

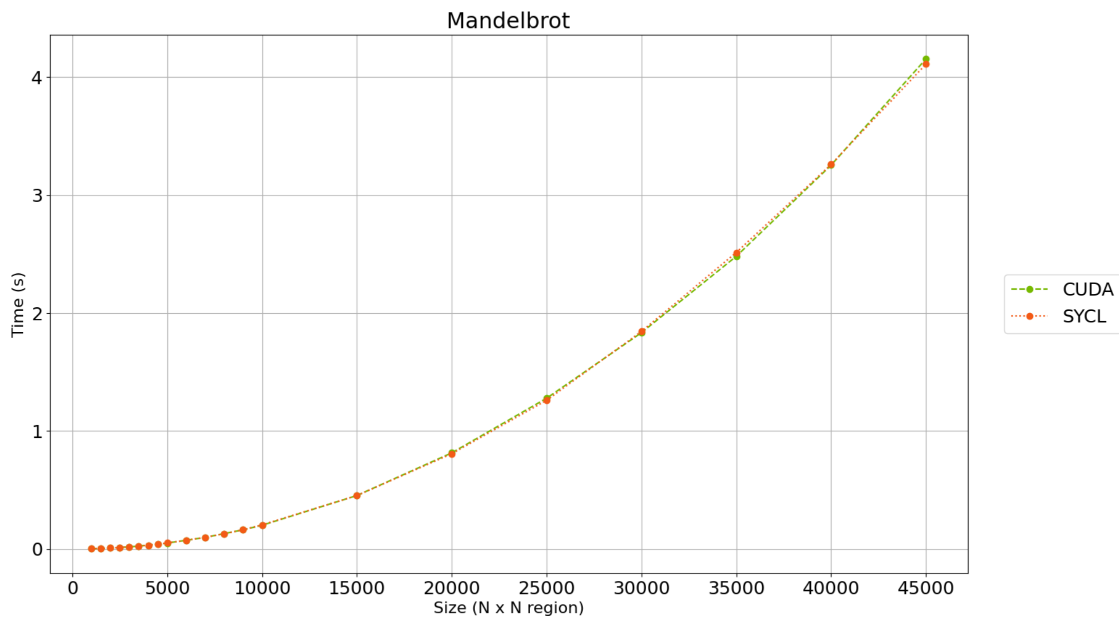


Figure 4.2: Mandelbrot benchmark. Results for SYCL and CUDA executions.

In Figure 4.3, it is observed that the speed up achieved decreases slightly as the size of the problem increases. Again, we can see that both SYCL and CUDA results are fairly similar, although they seem to fluctuate a little.

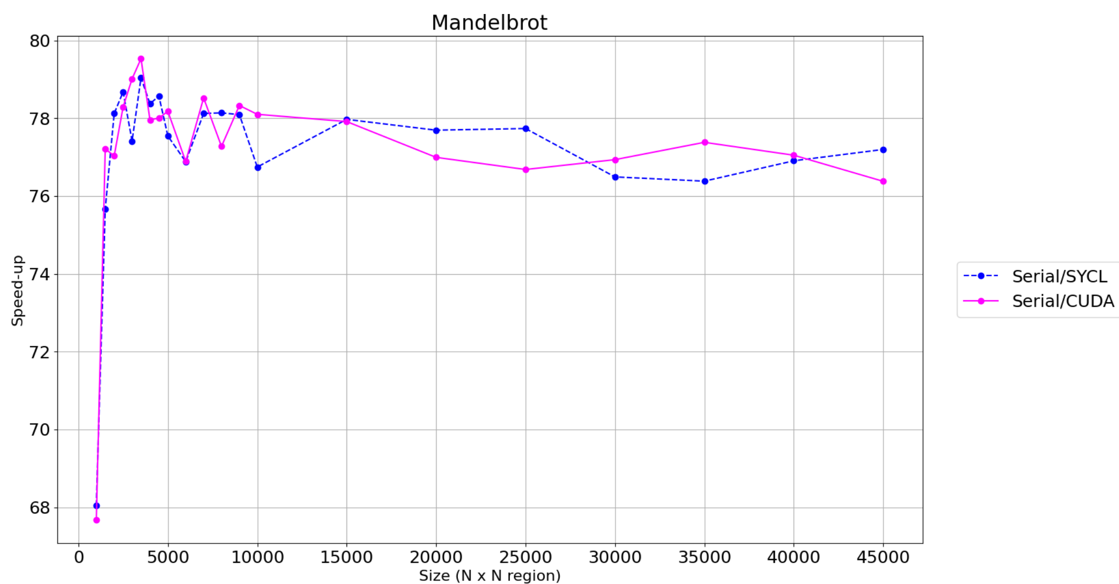


Figure 4.3: Mandelbrot benchmark. Speed-up graph for SYCL and CUDA executions.

4.3 Floyd–Warshall algorithm

The Floyd-Warshall algorithm is a method used to find the shortest paths in a weighted graph, where the weight of each edge represents the distance between two vertices. Unlike other shortest path algorithms like Dijkstra's algorithm, Floyd-Warshall works for graphs with negative edge weights, as long as there are no negative cycles.

These are the parameters we can work with:

- `iterations`: How many times the algorithm is repeated, then averaged over all times. Using a fixed amount of 100.
- `block size`: Block size used in the NDRange. Using a fixed amount of 16.
- `number of nodes`: Amount of nodes in the weighted graph. Using 500 to 10000 nodes.

Figure 4.4 presents the resulting graph for SYCL, CUDA and serial executions.

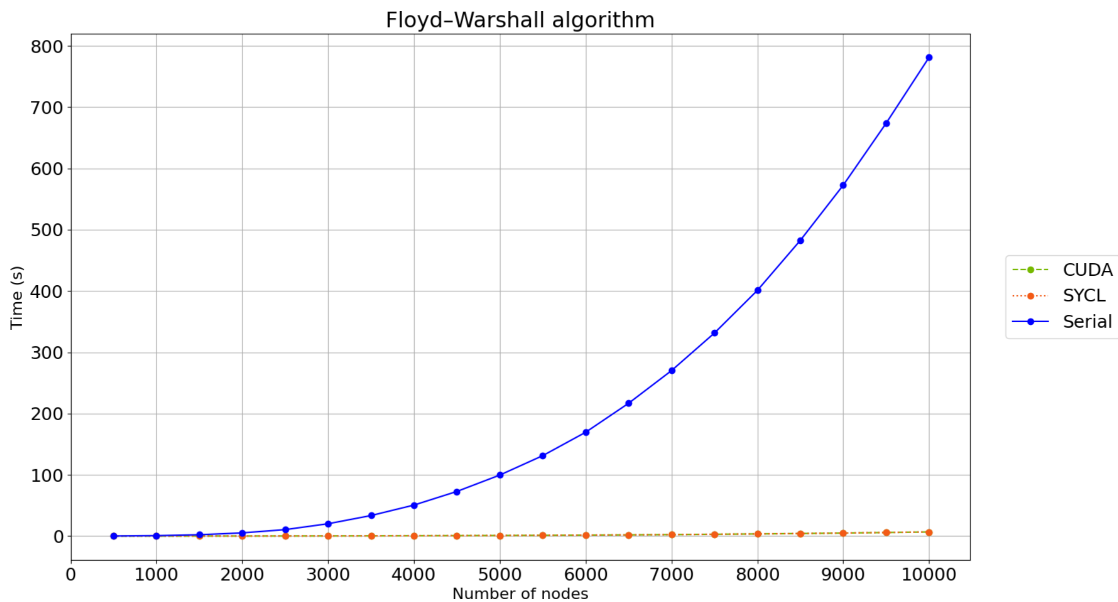


Figure 4.4: Floyd–Warshall algorithm. Results for SYCL, CUDA and serial executions.

The resulting graph in Figure 4.4 clearly demonstrates that serial execution is significantly slower than both SYCL and CUDA implementations as the problem size increases.

A closer examination of SYCL and CUDA graphs (Figure 4.5) reveals that most execution instances exhibit a high degree of similarity in terms of execution time.

However, the last three problem sizes show the greatest differences, although these are not too significant.

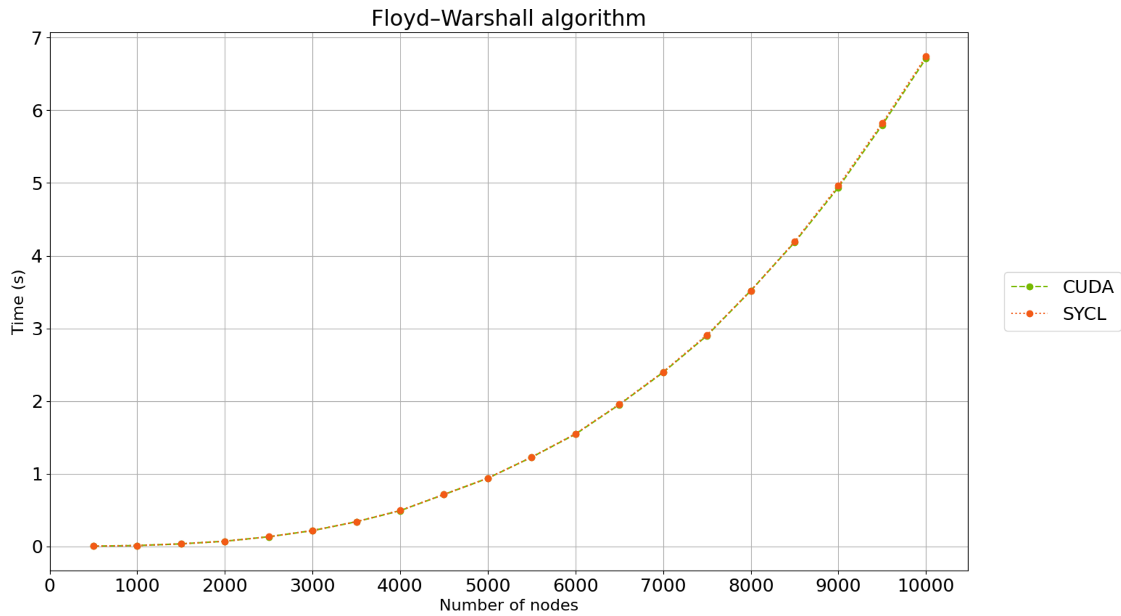


Figure 4.5: Floyd–Warshall algorithm benchmark graph for SYCL and CUDA executions.

In terms of speed-up, Figure 4.6 reflects an almost identical graph for SYCL and CUDA, tracing a logarithmic-like representation in both cases.

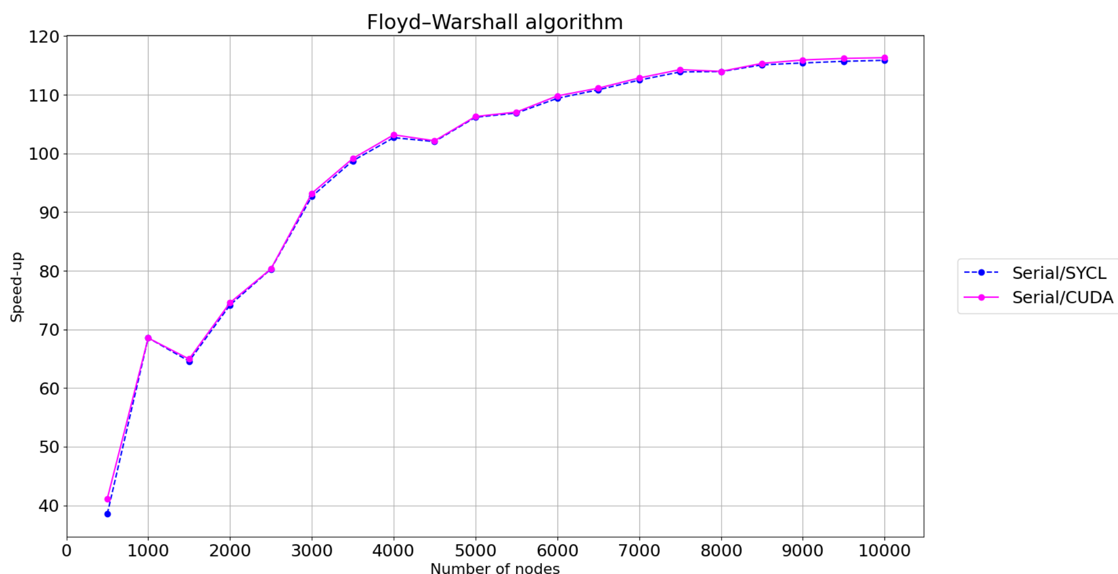


Figure 4.6: Floyd–Warshall algorithm speed-up graph for SYCL and CUDA executions.

4.4 Molecular dynamics

Molecular dynamics (MD) is a powerful computational simulation technique used to study the physical movements of atoms and molecules over time. By applying the principles of classical mechanics, MD allows scientists to predict the behavior of matter at the molecular level, offering insights into the structural, dynamic, and thermodynamic properties of complex systems such as proteins, nucleic acids, and materials.

Once more, the original benchmark had to be slightly modified to be able to input different problem sizes. There are two parameters we can work with:

- `problem size`: Number of atoms in the simulation. Using sizes 2500 to 1000000.
- `iterations`: How many times the MD kernel is executed, then averaged over all times. Using a fixed amount of 1000.

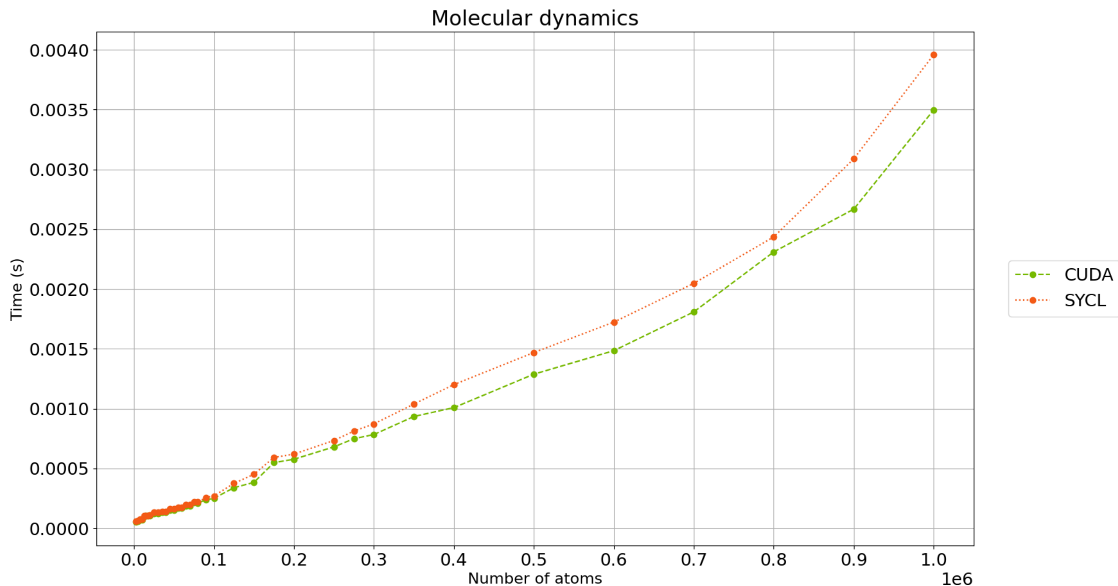


Figure 4.7: MD benchmark. Results for SYCL and CUDA executions.

In Figure 4.7 we measured the average kernel execution time for each selected size. From this graph we can see that both implementations run at similar speeds, although CUDA becomes up to 11.65% faster (on problem size 1000000) than SYCL as the number of atoms increases.

4.5 Backpropagation

Backpropagation, is a fundamental algorithm in machine learning, particularly in the training of artificial neural networks. It enables the development of complex models that can perform tasks such as image recognition and natural language processing. The algorithm works by iteratively adjusting the weights of the network to minimize the difference between the predicted output and the actual target output, which is measured by a loss function.

In this case, the algorithm has a single parameter: the `number of input nodes` (number of nodes in the neural network). The code requires a value divisible by 16 and for these experiments the value ranges from 512 to 1024000. What is being measured is the device offloading time for all execution instances.

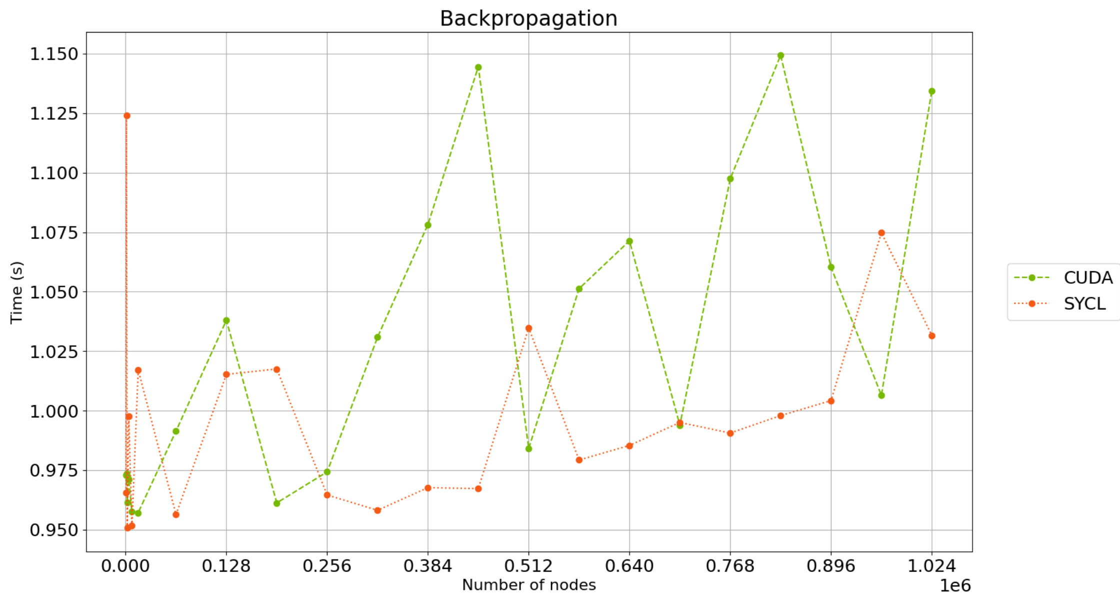


Figure 4.8: Backpropagation benchmark. Results for SYCL and CUDA executions.

The results in Figure 4.8 show a large variation in CUDA/SYCL runtimes for each problem size. In this experiment, SYCL seems to be a bit more stable in terms of time fluctuation.

The SYCL version of this algorithm has a limitation since the maximum problem size supported is near 1.048.000 nodes, while the CUDA implementation surpasses this constraint.

```
1  Random number generator seed: 7
2  Input layer size : 1048800
3  Starting training kernel
4  Performing GPU computation
5
6  UR CUDA ERROR:
7    Value:           1
8    Name:            CUDA_ERROR_INVALID_VALUE
9    Description:     invalid argument
10   Function:        urEnqueueKernelLaunch
11   Source Location: /home/bejeque/acabrera/sources/llvm/build/_deps/unified-runtime-src/source/
         adapters/cuda/enqueue.cpp:474
12
13 terminate called after throwing an instance of 'sycl::_V1::nd_range_error'
14   what(): Number of work-groups exceed limit for dimension 1: 65550> 65535-30 (
         PI_ERROR_INVALID_VALUE)
15 /home/verode/alu0101436784/slurm-submit.sh: line 57: 15450Aborted          ./main 1048800
16 srun: error: verode21: task 0: Exited with exit code 134
```

Listing 4.1: NDRange error on backpropagation benchmark.

This limitation can be evidenced by using `1,048,800` as the problem size. Listing 4.1 is the output of this experiment and line 14 shows the root of this

limitation. The problem is that the number of resulting work-groups that the program pretends to create exceeds the maximum established by the compiler.

This constraint is not tied to the SYCL specification but the compiler implementation, which is `clang++` in this case. Since the hardware is always a physical and tangible limit, this particular compiler has decided to set an arbitrary limit for this very reason.

One solution to this situation would be to assign more work to each work-group. This would reduce the number of work-groups so that it does not hit the limit set by the compiler.

Chapter 5

An Industry Case Study: Image Processing with SYCL

In this chapter we will take a deep look into the development of a SYCL application. For this purpose, we have contacted Wooptix[1], a Canary Islands based company with expertise in image processing. To illustrate their needs, Wooptix suggested the implementation of the erosion operation on FITS format images and they provided us with one of the images they use for real purposes.

The erosion is one of the fundamental operations in morphological image processing [26]. This branch of knowledge studies the structure and components of an image with the objective of creating useful descriptions of its shape, while also providing tools for signal processing.

On the other hand, the FITS (Flexible Image Transport System) file format is the standard astronomical data format supported by NASA and the IAU (International Astronomical Union). This standard supports multi-dimensional images, although we will only focus on two-dimensional FITS files.

The application presented in this chapter was developed using C++. To open and process images in FITS format, we will use CFITSIO[2], a library of C and Fortran subroutines for reading and writing FITS files.

5.1 The erosion operation

The erosion operation is used to shrink or erode the boundaries of objects within an image applying a structuring element.

Structuring Element

[1] https://wooptix.com

[2] FITSIO Home Page https://heasarc.gsfc.nasa.gov/fitsio/

A structuring element (SE) is a mask which determines how the operation is applied to the image. This object is usually represented by a matrix, where non-zero values describe the mask and a center origin to apply the mask from. We can classify them in two types:

- **Flat SE**: Composed of 0s and 1s only, simply establishes which pixels are relevant and which are not.
- **Non-flat SE**: Contains grayscale values that describe how meaningful the pixels are.

For this application, we will work with two-dimensional grayscale images using flat structuring elements.

The erosion operation consists of sliding a SE over an entire image, applying the SE to each pixel. This procedure takes the lowest value in the vicinity established by the SE (with a value of 1) and assigns it to the evaluated pixel, creating an output image whose bright regions are shrunk and dark regions are enlarged.

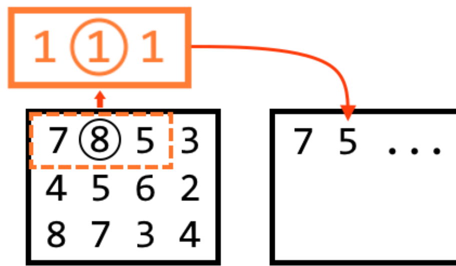


Figure 5.1: Simplified erosion operation.

In Figure 5.1 we can see a graphical example of this transformation. In this illustration, the second element of the input image is evaluated. According to the SE, the neighborhood of the element is 7, 8 and 5, from which we can conclude that the lowest value is 5, resulting in the value of erosion at the second pixel.

Note that certain border regions of the image make the SE go out of bounds. In order to process image borders correctly, we add padding around the image with a very high value, ensuring that the algorithm only considers pixels within the original image.

5.2 Supporting Code

In order to implement the erosion operation, we need to create an environment to work on images in FITS format, as well as to represent the structuring element and make everything work together.

The supporting code is based on three main classes:

- `FitsImage`: Encapsulates the functionality of the CFITSIO library.
- `StructuringElement`: Represents a SE and provides access to every detail of it.
- `Morphology`: Base class for the morphology operations, this makes the project easily expandable.

One important aspect to note is that the FITS format supports many pixel sizes, so the number of bytes a pixel occupies may vary from file to file. This is important to have into account because we need to maintain format consistency when reading, operating and writing the image we work with.

Depending on the pixel size, the application will decide on a data type in runtime, which can be one of the following: `unsigned char`, `short`, `long`, `long long`, `float` or `double`.

Because we are using C++, we need to use polymorphism in combination with the factory method pattern[3] to achieve dynamic type behaviour in our program.

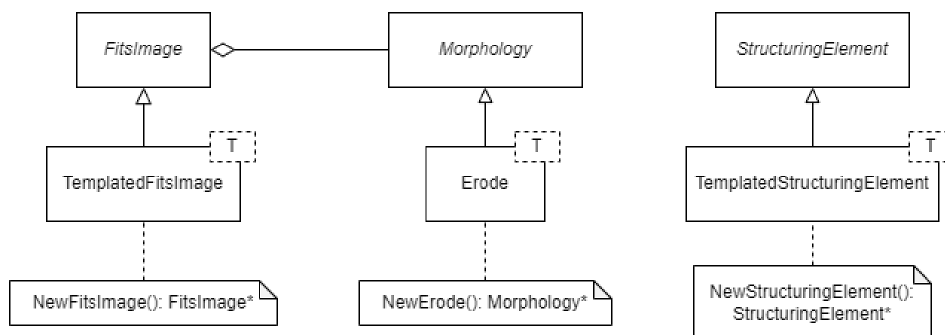


Figure 5.2: Conceptual UML class diagram for the erosion application.

This dynamic typing was made possible thanks to the implementation of the abstract base classes described in Figure 5.2. Each of these classes derive in a class template that can be used to dynamically create instances by using a non-member function, also indicated in the diagram with a note for every concrete class.

[3] Factory method `https://refactoring.guru/design-patterns/factory-method`

5.3 Erosion Solution Development

We present two implementations for the erosion operations, this section is dedicated to explain in detail how the operation is performed in both serial and SYCL approaches.

It would be beneficial to keep a few notes in mind when reading the implementations:

- Both `FitsImage` and `StructuringElement` data are stored in generic type one-dimensional arrays.
- Data arrays are stored in row-major order, starting from the bottom left.
- The code shown operates directly on the image representation, modifying the `FitsImage` content.

5.3.1 Serial Implementation

The serial implementation of the erosion operation presented here is quite straightforward. This approach provides a baseline against which the performance of the parallel version can be measured, while also serving as a reference to see whether its parallel version is worth the effort.

In addition, writing a naive implementation can provide insight into the complexity and potential bottlenecks of the algorithm, and help understand how the algorithm scales with increasing data or computational load.

The algorithm can be divided in two main zones. The first is the preparation for the erosion and the second is the erosion itself.

```
1  void Operate(FitsImage* fits_image, StructuringElement* operation_sel) override
      {
2    TemplatedFitsImage<T>& image =
3      *dynamic_cast<TemplatedFitsImage<T>*>(fits_image);
4    TemplatedStructuringElement<T>& sel =
5      *dynamic_cast<TemplatedStructuringElement<T>*>(operation_sel);
6    T* image_data = image.GetData();
7    T* image_data_copy = new T[image.PaddedTotalElements()];
8    std::copy(image_data, image_data + image.PaddedTotalElements(),
         image_data_copy);
9    T* sel_data = sel.GetData();
10   long origin = image.Padding() * image.PaddedColumns() + image.Padding();
11   for (long row{0}; row < image.Rows(); ++row) {
12     long image_row = origin + row * image.PaddedColumns();
13     for (long column{0}; column < image.Columns(); ++column) {
14       long pixel_index = image_row + column;
15       long local_origin = pixel_index -
16                           sel.CenterRow() * image.PaddedColumns() -
17                           sel.CenterColumn();
18       T minimum = std::numeric_limits<T>::max();
19       for (long local_row{0}; local_row < sel.Rows(); ++local_row) {
20         long local_image_row = local_origin + local_row * image.PaddedColumns();
21         long sel_row = local_row * sel.Columns();
22         for (long local_column{0}; local_column < sel.Columns(); ++local_column)
               {
23           long local_pixel = local_image_row + local_column;
24           if (sel_data[sel_row + local_column] == 1&&
25               image_data_copy[local_pixel] <= minimum) {
26             minimum = image_data_copy[local_pixel];
27           }
28         }
29       }
30       image_data[pixel_index] = minimum;
31     }
32   }
33   delete[] image_data_copy;
34 }
```

Listing 5.1: Serial erosion function. *See on Github*.

The first action of the preparation code is to cast both the FITS image and the structuring element pointers into their corresponding concrete class, which have the accessors to the template dependant arrays of data. This happens in lines 2-5. Then, lines 6 and 9 retrieve both image and SE data pointers.

We cannot read and write over the same image data, since it would modify the neighbourhood of pixels we will operate on later. To avoid this situation, we create a copy of the image data in line 8.

To conclude the set-up, we calculate the location of the first pixel of the image in line 10.

The erosion operation itself starts from line 11, where the main loops (lines 11 and 13), traverse the whole image. After every row, we calculate the first index of said row for the image data array in line 12, and for every column, we obtain the pixel index in line 14 for the pixel we will evaluate. Another relevant index is the location of the first pixel that corresponds to the SE evaluation, calculated in lines 15-17.

The region of code that covers lines 18-30 determines the output pixel value of the current pixel, which calculates the minimum pixel value in the neighbourhood set by the SE.

Starting in line 18, we set a very high minimum value for the output pixel. Then, the loops in lines 19 and 22 traverse the whole structuring element. For every row, we calculate the first index of said row for the image data array as well as for the SE, accomplished in lines 20-21.

In the innermost loop, we compute the neighbour pixel index to be evaluated in line 23 and check if the structuring element considers that pixel and whether its value is the new minimum. If it passes the checks, the new minimum is assigned in line 26. After the calculation of the minimum, it is set as the new pixel value in line 30. As a last action, we delete the copy of the image in line 33.

5.3.2 SYCL Implementation

The parallel implementation using SYCL is based on the naive algorithm presented in the serial implementation. Nevertheless, a parallel approach to this algorithm is clear to give significant benefits if we take into account the embarrassingly parallel nature of it. To be more specific, the fact that each pixel of the image is processed independently of each other, makes it an ideal candidate for parallel execution. Moreover, if we implement memory tiling to this process, the the efficiency can be further enhanced.

Memory tiling involves dividing the image into smaller blocks that fit into the fast, local memory dedicated to every work-group. By doing so, we can significantly reduce the latency associated with accessing global memory. Each tile can be loaded into the local memory, processed independently, and then the results can be written back to global memory. This approach minimizes memory access delays and improves data locality, ensuring that each work-item spends more time computing and less time waiting for data.

For this implementation we will use the buffer/accessor model, covered in Section 3.3 of Chapter 3.

```
sycl::queue queue{sycl::gpu_selector_v};
TemplatedFitsImage<T>& image =
  *dynamic_cast<TemplatedFitsImage<T>*>(fits_image);
TemplatedStructuringElement<T>& sel =
  *dynamic_cast<TemplatedStructuringElement<T>*>(operation_sel);
T* image_data = image.GetData();
T* sel_data = sel.GetData();

const long kTwicePadding = 2* image.Padding();
auto twice_padding_range = sycl::range(kTwicePadding, kTwicePadding);
auto padding_range = sycl::range(image.Padding(), image.Padding());
auto sel_offset =
  padding_range - sycl::range(sel.CenterRow(), sel.CenterColumn());

{ // Buffer scope
// CG Ranges
auto local_range = sycl::range(sel.Rows(), sel.Columns());
int column_work_groups_amount =
  FitsUtils::DivisionCeiling(image.Columns(), local_range[1]);
int row_work_groups_amount =
  FitsUtils::DivisionCeiling(image.Rows(), local_range[0]);
auto global_range = sycl::range(local_range[0] * row_work_groups_amount,
                                local_range[1] * column_work_groups_amount);
auto nd_range = sycl::nd_range(global_range, local_range);
// Buffer Ranges
auto image_buffer_range =
  sycl::range(image.PaddedRows(), image.PaddedColumns());
auto output_buffer_range =
  sycl::range(image.PaddedRows(), image.PaddedColumns());
auto sel_buffer_range = sycl::range(sel.Rows(), sel.Columns());
auto tile_range = local_range + twice_padding_range;
// Buffers
auto image_buffer = sycl::buffer{image_data, image_buffer_range};
image_buffer.set_final_data(nullptr);
auto output_buffer = sycl::buffer<T, 2>{output_buffer_range};
output_buffer.set_final_data(image_data);
auto sel_buffer = sycl::buffer{sel_data, sel_buffer_range};
```

Listing 5.2: Set-up for SYCL parallel execution. *See on Github*.

In Listing 5.2 we have the preparation code for the kernel execution. Firstly, we create a queue and select a GPU device in line 1. Then we retrieve both image and SE pointers and their data (lines 2-7), the same way we did for the serial version.

Lines 9-13 set padding related ranges and offsets. Line 15 starts the buffer scope, outside of which the buffers are destroyed.

Following, we have two sets of ranges.

In the first one, line 17 specifies the size of the work-groups of the NDRange,

which is defined to be the same size as the SE. Then we define how many work-groups we need by dividing the image dimensions by the local range and taking the ceiling of the result. This is calculated in lines 18-21. After this, we can calculate the global range and set the NDRange in lines 22-24.

The second set of ranges calculates buffer sizes. Both input and output image buffer ranges are the same size (lines 26-29) and is set to the size of the original padded image. The SE range, in line 30 has the same dimensions as the SE itself. On the other hand, in line 31, the tile range is set to be the local range plus twice the padding, ensuring that the structuring element does not to surpass the tile range.

The next 33-37 lines simply create the buffers using the previously defined ranges and data pointers. Note that line 34 indicates that the image buffer has nowhere to write back when the buffer is destroyed, this is specified to ensure that there is no time lost writing unnecessary data back. On the other hand, line 36 gives the output buffer a pointer to write back to, which is the pointer to the original data array, successfully overwriting the data.

```
38 // Command Group Submission
39 queue.submit([&](sycl::handler& handler) {
40   sycl::accessor image_accessor{image_buffer, handler, sycl::read_only};
41   sycl::accessor output_accessor{output_buffer, handler, sycl::write_only};
42   sycl::accessor sel_accessor{sel_buffer, handler, sycl::read_only};
43   auto tile = sycl::local_accessor<T, 2>(tile_range, handler);
44 
45   handler.parallel_for(nd_range, [=](sycl::nd_item<2> item) {
46     auto global_id = item.get_global_id();
47     auto group_id = item.get_group().get_group_id();
48     auto local_id = item.get_local_id();
49     auto global_group_offset = group_id * local_range;
50 
51     // Load tile
52     for (auto row = local_id[0]; row < tile_range[0]; row += local_range[0]) {
53       for (auto column = local_id[1]; column < tile_range[1]; column +=
             local_range[1]) {
54         tile[row][column] = image_accessor[global_group_offset + sycl::range(row
               , column)];
55       }
56     }
57     sycl::group_barrier(item.get_group());
58     auto tile_index_origin = local_id + sel_offset;
59 
60     // Erode
61     T minimum = std::numeric_limits<T>::max();
62     for (int row = 0; row < sel_buffer_range[0]; ++row) {
63       for (int column = 0; column < sel_buffer_range[1]; ++column) {
64         auto tile_index = tile_index_origin + sycl::range(row, column);
65         if (sel_accessor[row][column] == static_cast<T>(1) &&
66             tile[tile_index] < minimum) {
67           minimum = tile[tile_index];
68         }
69       }
70     }
71     // Write output
72     output_accessor[global_id] = minimum;
73   });
74 });
75 queue.wait_and_throw();
76 }
```

Listing 5.3: Erosion operation kernel using SYCL. *See on Github*.

The erosion operation occurs in Listing 5.3, where we submit the command group to the queue (line 39). Just before the kernel starts, we define a series of accessors in lines 40-42, which correspond to the image (read-only), output (write-only) and SE (read-only). A fourth accessor is present in line 43, which grants

access to the local memory to perform the memory tiling with.

Line 45 starts the definition of the kernel. First, we set various indexes related to the current work-item (lines 46-49):

- **Global ID**: Unique global identifier. It matches with the output pixel index since the `parallel_for` range coincides with the image range.
- **Group ID**: Identifies the work-group. Useful to calculate where the work-group tile begins within the image.
- **Local ID**: Index of the work-item within the work-group. Determines which pixels the work-item has to copy to the tile and the offset to read within the tile.
- **Global group offset**: First index of the tile within the image.

In order to write the whole tile, each work-item of the work-group contributes to copy from the image to the tile without repeating. This is done in lines 52-56, where each work-item starts at a unique position determined by the local ID and iterates in increments of the size of the local range. This approach ensures that no work item repeats the copy operation on the same pixel.

To guarantee that every work-item has access to the same exact full information of the tile, line 57 sets a group barrier which makes all work-items in the work-group wait until each one has reached this point in the code. Right after, we calculate the starting point where the SE will be applied within the image. Line 58 determines this starting point, obtaining the index of the first element to be read, which will serve as an offset in the next calculations.

The computation for the new pixel value starts by setting a very high number as the new minimum in line 61. Following, we iterate over the SE using loops 62 and 63. In the innermost loop, we start by identifying which pixel of the image to analyze in line 64. Then, if the current position of the SE indicates that such pixel may be considered, we evaluate whether the pixel value is the new minimum. This process happens in lines 65-68. The new minimum value is written in the output buffer in line 72.

Finally, we wait in line 75 for this task to finish.

5.4 Results

In this section, we will present the application of the erosion operation on various images. We will also display the global compute time of each program as well as the isolated time for the operation only. The code presented in the previous section was run on *Verode*, the same platform described in Chapter 4.

The first image we will process was created using the the Faint Object Camera (FOC) from the Hubble Space Telescope in July 1996, with the size of 1024 by 1024 pixels. This picture is part of a sample collection[4] listed by NASA. In Figure 5.3 we can see this image.

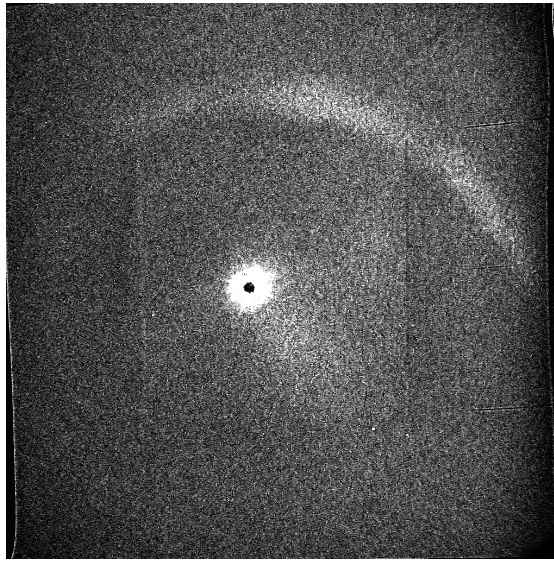


Figure 5.3: Original FOC sample image.

For this first experimentation, we will use the structuring element shown in Listing 5.4. It defines a simple cross shape whose chosen center is (1, 1), the second element of the second row.

```
0 1 0
1 1 1
0 1 0
```

Listing 5.4: Cross-shaped structuring element.

Figure 5.4 illustrates a magnified comparison of the original (left) and eroded (right) images. Given that both serial and SYCL implementations yield the same final image, there is no need to display or differentiate the resulting images.

[4] Sample FITS Files `https://fits.gsfc.nasa.gov/fits_samples.html`

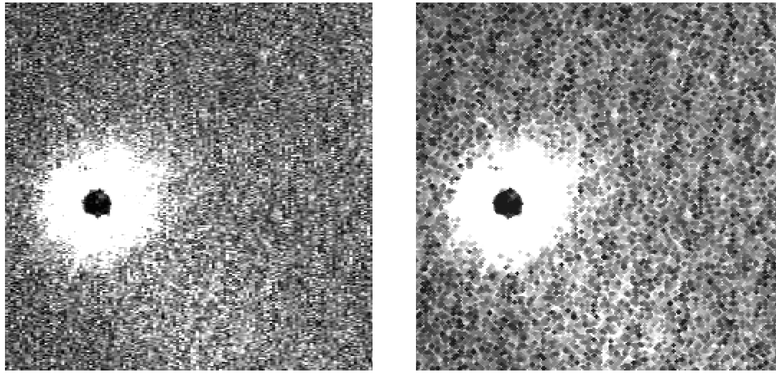


Figure 5.4: Zoomed in FOC image. Original (left) and eroded (right).

Executing both serial and parallel versions 10 times and averaging the times, we find the following results:

- The overall program execution time is 1.10 seconds for serial and 1.30 seconds for SYCL.
- The operation execution time is 0.04 seconds for serial and 0.17 seconds for SYCL.

For this specific test, both overall and operation times are lower for the serial version. This is expected since smaller images may suffer from the overhead introduced by the parallel SYCL version, while larger images would greatly benefit from the parallel implementation.

To prove that the SYCL implementation can actually outperform the serial version, we worked with a large sample image provided by Wooptix. The image size is is 6388 by 6388 pixels and you can see it in 5.5.

Figure 5.5: Original Wooptix sample image.

The following testing will be performed using a 8 by 8 circle-shaped structuring element, represented in Listing 5.5, whose chosen center is (3, 3) or the fourth element of the fourth row.

```
0 0 1 1 1 0 0
0 1 1 1 1 1 0
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
0 1 1 1 1 1 0
0 0 1 1 1 0 0
```

Listing 5.5: Circle-shaped structuring element.

The result of the operation can be observed in Figure 5.6, distinguishing the original image from the eroded one. Although it appears to be a minor change, the difference can be seen by observing the dark spot near the center of the image.

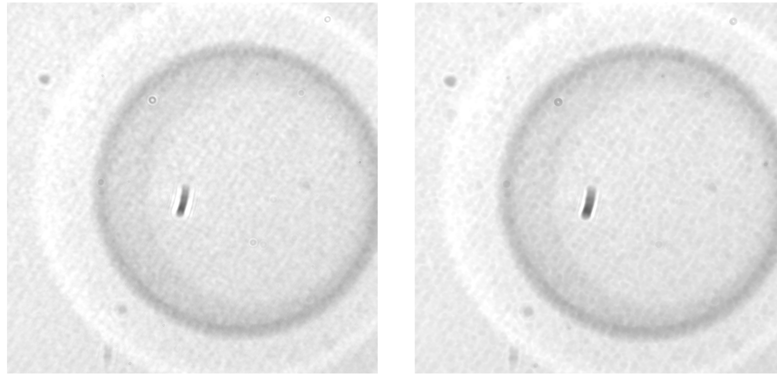


Figure 5.6: Zoomed in Wooptix sample image. Original (left) and eroded (right).

Once again, executing the algorithm 10 times and averaging the resulting times, we can see the following differences in time:

- The overall program execution time is 58.61 seconds for serial and 56.30 seconds for SYCL.
- The operation execution time is 3.32 seconds for serial and 0.29 seconds for SYCL.

Considering all 10 execution instances, the overall program execution time varies from 45 to 70 seconds for both serial and SYCL implementations. On the other hand, the SYCL version clearly outperforms the serial version when it comes to the execution of the erosion operation itself, gaining a speed-up of factor 11.5.

Chapter 6

Conclusiones y Líneas de Trabajo Futuras

En este capítulo se presentarán las conclusiones alcanzadas tras la realización de este trabajo y discutiremos posibles líneas futuras de trabajo. Las conclusiones y resultados obtenidos de este proyecto de investigación y desarrollo pueden resumirse en los siguientes puntos:

- Un primer valor añadido del trabajo realizado ha sido la contextualización y aprendizaje realizado por el estudiante en el ámbito de la computación de altas prestaciones. Dada la escasez de materias relacionadas con este tópico en el Grado en Ingeniería Informática en la Universidad de La Laguna, el estudiante ha tenido que realizar inicialmente un esfuerzo significativo en el aprendizaje de conceptos y técnicas que le eran ajenos.

- Se ha utilizado SYCL como vehículo para elevar el nivel de conocimiento en programación paralela y contextualizar el aprendizaje en HPC. El modelo de programación de SYCL permite el desarrollo de aplicaciones HPC con portabilidad y facilidad, ejecutándose en múltiples plataformas de hardware (CPUs, GPUs, FPGAs) con mínimas modificaciones, simplificando y acelerando el desarrollo.

- Los conocimientos de los aspectos técnicos de SYCL se han adquirido básicamente a través del texto *Data Parallel C++* [1] y la realización del tutorial práctico de *SYCL Academy*[1].

- Se han evaluado diferentes aplicaciones del la colección de benchmarks *HeCBench* utilizando *Verode* como plataforma de desarrollo. A partir de estos experimentos concluimos que SYCL no afecta significativamente

[1]SYCL Academy https://github.com/codeplaysoftware/syclacademy

al rendimiento de las aplicaciones, posicionándola como una plataforma competitiva en su nicho tecnológico.

- Se ha implementado en SYCL un algoritmo de procesado de imágenes. El algoritmo elegido ha sido una transformación morfológica, pero resulta inmediato utilizarlo como punto de partida para otros algoritmos similares. Los resultados computacionales obtenidos al procesar imágenes de tamaño realista para ciertas aplicaciones industriales muestran que SYCL es una aproximación relevante para este tipo de tareas cuando el número de imágenes a procesar es elevado.

Existen algunas líneas de trabajo futuras abiertas a exploración que podrían ayudar a ampliar y profundizar en los beneficios y aplicaciones de SYCL en el ámbito de la computación de altas prestaciones. Son las siguientes:

1. Optimizar algoritmo de erosión implementado en SYCL.

2. Implementación de otros algoritmos paralelos de procesamiento de imágenes utilizando SYCL.

3. Investigación y puesta en práctica de técnicas avanzadas de optimización para aplicaciones SYCL.

4. Implementación y benchmarking para diferentes back-ends y/o hardware de destino empleando SYCL. Debido a las limitaciones en cuanto al hardware disponible, las únicas plataformas con las que se ha experimentado son CPU y GPU. Sería interesante extender los experimentos realizados a otras plataformas.

Chapter 7

Conclusions and Future Lines of Work

This chapter will present the conclusions reached after the completion of this work and discuss possible future lines of work. The conclusions and results obtained from this research and development project can be summarised in the following points:

- A first added value of the work done has been the contextualisation and learning carried out by the student in the field of high performance computing. Given the scarcity of subjects related to this topic in the Degree in Computer Science at the Universidad of La Laguna, the student has initially had to make a significant effort in learning concepts and techniques that were alien to him.

- SYCL has been used as a vehicle to raise the level of knowledge in parallel programming and contextualise learning in HPC. The SYCL programming model allows the development of HPC applications with portability and ease, running on multiple hardware platforms (CPUs, GPUs, FPGAs) with minimal modifications, simplifying and accelerating development.

- Knowledge of the technical aspects of SYCL has been acquired mainly through the *Data Parallel C++* [1] text and the completion of the *SYCL Academy*[1] practical tutorial.

- Different applications from the *HeCBench* benchmark collection have been evaluated using *Verode* as the development platform. From these experiments we conclude that SYCL does not significantly affect application performance, positioning it as a competitive platform in its technological niche.

[1] SYCL Academy https://github.com/codeplaysoftware/syclacademy

- An image processing algorithm has been implemented in SYCL. The chosen algorithm has been a morphological transformation, but it is immediate to use it as a starting point for other similar algorithms. The computational results obtained when processing images of realistic size for certain industrial applications show that SYCL is a relevant approach for this type of task when the number of images to be processed is high.

There are some future lines of work open to exploration that could help to broaden and deepen the benefits and applications of SYCL in the field of high-performance computing. They are the following:

1. Optimize the erosion algorithm implemented in SYCL.
2. Implementation of other parallel image processing algorithms using SYCL.
3. Research and implementation of advanced optimization techniques for SYCL applications.
4. Implementation and benchmarking for different back-ends and/or target hardware using SYCL. Due to limitations in terms of available hardware, the only platforms that have been experimented with are CPU and GPU. It would be interesting to extend the experiments to other platforms.

Chapter 8

Budget

In this chapter we will present an estimated budget for the development and execution of a complex SYCL project on an HPC platform.

This budget has been prepared with a company like Wooptix[1] in mind, which has a strong technology profile and experience in developing CUDA applications for image processing. The following is a quotation for the adoption of SYCL by a company with such a profile.

The main considerations in calculating the cost of this project are the cost of working hours, which includes preparing the development environment, coding the actual SYCL program, writing documentation and testing. There is also the cost of the execution platform, which can vary greatly depending on the computing power required and the hardware purchased.

Working hours

The cost of working hours is estimated to be 12,46€ per hour. This is the average price of a junior full-stack developer in Spain based on the data given by web platforms that collect and display information related to job salaries. These platforms are: Glassdoor[2] $(10,94€/h)$, Talent[3] $(12,44€/h)$ and Jooble[4] $(13,99€/h)$, whose average value is $12{,}46€$.

For this project, the weekly working time is 40 hours, with a total time of 8 weeks. This time would be divided into the following tasks:

- **Preparation**: Getting everything ready in the working environment includes server/machine configuration and the installation of SYCL along with the

[1] https://wooptix.com

[2] https://www.glassdoor.es/Sueldos/espa%C3%B1a-desarrollador-full-stack-junior-sueldo-SRCH_IL.0,6_IN219_KO7,38.htm

[3] https://es.talent.com/salary?job=desarrollador+full+stack+junior

[4] https://es.jooble.org/salary/desarrollador-full-stack-junior#hourly

proper compilation back-ends and other additional tools. This process may take 16 hours of work, which translates to $16h \times 12,46€/h = 199,36€$.

- **Project design**: Planning the code and creating the basic structure of the program might require 24 hours of work time, which costs: $24h \times 12,46€/h = 299,04€$.

- **Project development**: Involves programming and also writing the associated documentation and tests. This is the longest process lasting about 7 weeks, costing $40h/week \times 7weeks \times 12,46€/h = 3488,80€$

In total, the cost of working hours is $199,36€ + 299,04€ + 3488,80€ = \mathbf{3987.20}€$.

Execution platform

We have two possible options for acquiring hardware. The first one to consider is to purchase all the necessary equipment, which would be composed by the following:

- **Basic server**: A pre-built server with all the essentials like the *Smart Selection PowerEdge T150 Tower Server*[5] from Dell. This is a customizable server that can be personalized before buying. The base cost is 868,88€, coming with an Intel® Pentium® CPU and 1TB of HDD storage. For better performance, switching the CPU for an Intel® Xeon® would cost around 300,00€ and adding 480GB of SSD storage costs around 600,00€.

- **Accelerator device**: A suitable option for a GPU is the *NVIDIA Tesla L4*[6] which offers good performance with over 7000 CUDA cores while also having a low-power consumption of 72W. This device costs about 3000,00€.

By these means, the cost of acquiring hardware sums $868,88€ + 300,00€ + 600,00€ + 3000,00€ = 4768,88€$.

Another option is to use cloud computing, where you can rent hardware and resources using virtual machines. There are many companies that offer this service, on Table 8.1 we can see three possible candidates: Google Cloud[7], Tencent Cloud[8]

[5] https://www.dell.com/es-es/shop/enterprise-products/servidor-torre-t150-intel/spd/poweredge-t150/pet1501a

[6] https://www.amazon.com/-/es/Nvidia-Tensor-Tarjeta-Accellerator-Gr%C3%A1ficos/dp/B0CCNM2WY2

[7] Google Cloud price calculator https://cloud.google.com/products/calculator

[8] Tencent Cloud price calculator https://www.tencentcloud.com/pricing/cvm/calculator

and IBM Cloud[9]. For these budgets, we estimated 150 hours for regular testing and 250 hours for experimentation with real or large problem instances.

	Google Cloud	**Tencent Cloud**	**IBM Cloud**
CPU	General Purpose "N1" x4 vCPU	8-core Intel Xeon Skylake 6133 (2.5 GHz)	16-core Intel Xeon 4110 (2.10 GHz)
RAM	16 GB	40 GB	32 GB
Storage	SSD 400 GB		SSD 960 GB
GPU	NVIDIA V100		
Price	1059,25 €/400h	1314,73 €/400h	3183,94 €/2 months

Table 8.1: Specifications for cloud computing and budget.

To ensure a fair comparison between service providers, a single NVIDIA V100 GPU was chosen as the accelerator unit for each budget.

As we can see, renting a virtual machine is cheaper than buying a new one in this case. To guarantee that the performance is acceptable without the price tag going through the roof, we opted to use the Tencent Cloud virtual machine service to supply the hardware.

Working hours			Execution platform (Tencent Cloud)	Total
Preparation	Design	Development		
199.36 €	299.04 €	3488.8 €	1314,73 €	5301.93 €

Table 8.2: Total budget breakdown.

Finally, the total budget is shown in Table 8.2, which is the sum of both working hours and execution platform costs, making a total of 5301.93 €.

[9]IMB Cloud catalog https://cloud.ibm.com/catalog

Bibliography

[1] J. Reinders, B. Ashbaugh, J. Brodman, M. Kinsner, J. Pennycook, and X. Tian, *Data Parallel C++: Programming Accelerated Systems Using C++ and SYCL*. Apress, 2023. https://link.springer.com/book/10.1007/978-1-4842-9691-2 [Electronically Available. Accessed November 2023]. iii, 5, 17, 20, 28, 53, 55

[2] P. Assiroj, A. L. Hananto, A. Fauzi, and H. L. Hendric Spits Warnars, "High Performance Computing (HPC) Implementation: A Survey," in *2018 Indonesian Association for Pattern Recognition International Conference (INAPR)*, pp. 213–217, 2018. 1, 5

[3] P. Bose and D. Padua, *Power Wall*, pp. 1593–1608. Boston, MA: Springer US, 2011. 2

[4] G. Fox, R. Williams, and G. Messina, *Parallel Computing Works!* Morgan Kaufmann, 1994. 2

[5] J. L. Hennessy and D. A. Patterson, "A New Golden Age for Computer Architecture," *Commun. ACM*, vol. 62, p. 48–60, jan 2019. 3

[6] The Khronos Group Inc., "SYCL Main Page," 2014. https://www.khronos.org/sycl/ [Electronically Available. Accessed November 2023]. 3, 5

[7] R. C. Martin and J. O. Coplien, *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2009. 4

[8] A. dos Santos Moreira, F. de Sande González, and A. Cabrera Pérez, "TFG-SYCL," 2023–2024. https://github.com/AdrianoMoreira08/TFG-SYCL. 4

[9] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. P. Thibault, "Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model," *IEEE Transactions on Parallel and Distributed Systems*, pp. 1–1, 2017. 7

[10] F. Sainz, S. Mateo, V. Beltran, J. L. Bosque, X. Martorell, and E. Ayguadé, “Leveraging OmpSs to Exploit Hardware Accelerators,” in *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*, pp. 112–119, 2014. 7

[11] O. S. Lawlor, “Message passing for GPGPU clusters: CudaMPI,” in *2009 IEEE International Conference on Cluster Computing and Workshops*, pp. 1–8, 2009. 7

[12] J. E. Stone, D. Gohara, and G. Shi, “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems,” *Computing in Science & Engineering*, vol. 12, no. 3, pp. 66–73, 2010. 7

[13] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, “SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters,” in *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS ’12, (New York, NY, USA), p. 341–352, Association for Computing Machinery, 2012. 7

[14] G. R. Luecke, N. T. Weeks, B. M. Groth, M. Kraeva, L. Ma, L. M. Kramer, J. E. Koltes, and J. M. Reecy, “Fast Epistasis Detection in Large-Scale GWAS for Intel Xeon Phi Clusters,” in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 3, pp. 228–235, 2015. 7

[15] L. Dagum and R. Menon, “OpenMP: An Industry Standard API for Shared-Memory Programming,” *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998. 7

[16] M. Viñas, B. B. Fraguela, D. Andrade, and R. Doallo, “Heterogeneous distributed computing based on high-level abstractions,” *Concurrency and Computation: Practice and Experience*, vol. 30, no. 17, p. e4664, 2018. e4664 cpe.4664. 8, 12

[17] T.-Y. Liang and Y.-W. Chang, “GridCuda: A Grid-Enabled CUDA Programming Toolkit,” in *2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications*, pp. 141–146, 2011. 10

[18] R. Reyes Castro, *Directive-based Approach to Heterogeneous Computing*. PhD thesis, Universidad de La Laguna, La Laguna, Spain, oct 2012. 12, 16

[19] L. Grillo, J. J. Fumero, R. Reyes, and F. de Sande, “Programming for GPUs: the Directive Based Approach,” in *Proceedings of the 2013 Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing* (Fatos Xhafa et. al., ed.), (Compiègne, France), pp. 612–617, Oct 2013. 14

[20] S. Lee and R. Eigenmann, "OpenMPC: Extended OpenMP Programming and Tuning for GPUs," in *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, 2010. 14

[21] T. D. Han and T. S. Abdelrahman, "hiCUDA: High-Level GPGPU Programming," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 78–90, 2011. 15

[22] The Portland Group, "PGI Fortran & C Accelerator Programming Model," tech. rep., The Portland Group, mar 2010. 15

[23] The Khronos(R) SYCL(TM) Working Group, *SYCL™ 2020 Specification (revision 8)*. The Khronos(R) SYCL(TM) Working Group, 2023. https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html [Electronically Available. Accessed April 2024]. 17

[24] J. Hoberock, M. Garland, O. Giroux, V. Grover, U. Kapasi, and J. Marathe, "Parallelizing the Standard Algorithms Library," tech. rep., NVIDIA Corporation, 2012. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3408.pdf [Electronically Available. Accessed December 2023]. 17

[25] J. Järvi and J. Freeman, "C++ lambda expressions and closures," *Science of Computer Programming*, vol. 75, no. 9, pp. 762–772, 2010. 26

[26] Rafael C. González and Richard E. Woods, *Digital image processing, 3rd Edition*. Pearson Education, 2008. 39