Investigation of the OpenCL SYCL Programming Model

Angelos Trigkas

August 22, 2014

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2014
Abstract

OpenCL SYCL is a new heterogeneous and parallel programming framework created by the Khronos Group that tries to bring OpenCL programming into C++. In particular, it enables C++ developers to create OpenCL kernels, using all the popular C++ features, such as classes, inheritance and templates. What is more, it dramatically reduces programming effort and complexity, by letting the runtime system handle system resources and data transfers. In this project we investigate Syclone, a prototype SYCL implementation that we acquired from Codeplay Ltd. To do that we compare SYCL in terms of both programmability and performance with another two established parallel programming models, OpenCL and OpenMP. In particular, we have chosen six scientific applications and ported them in all three models. Then we run benchmarking experiments to assess their performance on an Intel Xeon Phi based platform. The results have shown that SYCL does great from a programmability perspective. However, its performance still does not match that of OpenCL or OpenMP.
## Contents

Chapter 1 Introduction ............................................................................................................. 1  
  1.1 Motivation .......................................................................................................................... 1  
  1.2 Project Goals ..................................................................................................................... 2  
  1.3 Report Structure ............................................................................................................... 3  

Chapter 2 Background Theory ................................................................................................. 5  
  2.1 OpenCL Programming model ............................................................................................. 5  
     2.1.1 Platform Model ......................................................................................................... 6  
     2.1.2 Execution Model ....................................................................................................... 6  
     2.1.3 Memory Model ......................................................................................................... 8  
     2.1.4 Programming Model ................................................................................................. 9  
  2.2 OpenCL SPIR ....................................................................................................................... 9  
  2.3 Intel Xeon Phi .................................................................................................................... 10  
     2.3.1 Hardware ................................................................................................................... 11  
     2.3.2 Software .................................................................................................................... 12  
     2.3.3 OpenCL on Intel Xeon Phi ......................................................................................... 13  

Chapter 3 The OpenCL SYCL Programming Model .............................................................. 15  
  3.1 Relevant C++ features ....................................................................................................... 16  
     3.1.1 Function Objects ....................................................................................................... 16  
     3.1.2 Lambda Expressions ................................................................................................. 17  
  3.2 SYCL Platform Model ....................................................................................................... 18
3.3 SYCL Execution Model .................................................................................................................. 18
3.4 SYCL Memory Model .................................................................................................................... 19
3.5 SYCL Programming Model .......................................................................................................... 19
3.6 SYCL Application Example ........................................................................................................... 20
Chapter 4 Application Development with SYCL ................................................................................. 24
4.1 Applications ..................................................................................................................................... 24
  4.1.1 27stencil ...................................................................................................................................... 25
  4.1.2 Le_core ...................................................................................................................................... 26
  4.1.3 Himeno ....................................................................................................................................... 27
  4.1.4 Mandel ....................................................................................................................................... 28
  4.1.5 GEMM ....................................................................................................................................... 28
  4.1.6 Bitonic ....................................................................................................................................... 29
4.2 Development with SYCL .................................................................................................................. 29
  4.2.1 Device Selection .......................................................................................................................... 29
  4.2.2 Kernel Definition ......................................................................................................................... 30
4.3 Comparison of OpenCL, SYCL and OpenMP .................................................................................. 32
Chapter 5 Performance Results and Analysis ....................................................................................... 40
5.1 Experimental Platform ..................................................................................................................... 40
5.2 SYCLONE Installation ..................................................................................................................... 41
5.3 Experimental Setup .......................................................................................................................... 41
5.4 Experimental Results ....................................................................................................................... 45
5.5 Discussion ......................................................................................................................................... 50
Chapter 6 Conclusion ............................................................................................................................... 59
Appendix A  Extensive OpenMP Results ............................................................................................... 63
  A.1 OpenMP Xeon Phi Results ............................................................................................................. 63
List of Tables

Table 1: Device query results ................................................................. 43
Table 2: OpenCL and SYCL execution parameters ................................. 44
Table 3: 27stencil OpenMP execution times on Xeon Phi ...................... 63
Table 4: Himeno OpenMP execution times on Xeon Phi ....................... 63
Table 5: Le core OpenMP execution times on Xeon Phi ....................... 64
Table 6: Mandel OpenMP execution times on Xeon Phi ....................... 64
Table 7: GEMM OpenMP execution times on Xeon Phi ....................... 64
Table 8: Bitonic OpenMP execution times on Xeon Phi ....................... 65
Table 9: 27stencil OpenMP execution times on Xeon ......................... 65
Table 10: Himeno OpenMP execution times on Xeon ........................... 66
Table 11: Le core OpenMP execution times on Xeon ........................... 66
Table 12: Mandel OpenMP execution times on Xeon ........................... 67
Table 13: GEMM OpenMP execution times on Xeon ........................... 67
Table 14: Bitonic OpenMP execution times on Xeon ........................... 68
List of Figures

Figure 1: OpenCL platform model (image source [3])..................................................6
Figure 2: Example of a 2-dimensional kernel space decomposition in OpenCL (image source [3]) .......................................................................................................7
Figure 3: OpenCL conceptual memory model (image source [3]) .........................8
Figure 4: SPIR compilation flow (image source [5]) .....................................................9
Figure 5: Intel Xeon Phi Coprocessor card (image source [6])............................10
Figure 6: Intel Xeon Phi high level architectural diagram (image source [7]) ....11
Figure 7: Intel Xeon Phi single core architecture (image source [8]) .................12
Figure 8: Vector addition implemented in SYCL programming language............22
Figure 9: 27stencil OpenACC version .....................................................................34
Figure 10: 27stencil OpenCL version .......................................................................35
Figure 11: 27stencil OpenCL kernels ......................................................................36
Figure 12: 27stencil SYCL version ...........................................................................37
Figure 13: 27stencil OpenMP version .....................................................................38
Figure 14: 27stencil execution times on Xeon Phi ..................................................46
Figure 15: Himeno execution times on Xeon Phi .....................................................46
Figure 16: Le core execution times on Xeon Phi ......................................................47
Figure 17: Mandel execution times on Phi ...............................................................47
Figure 18: Bitonic execution times on Xeon Phi .......................................................48
Figure 19: GEMM execution times on Xeon Phi ......................................................48
Figure 20: vadd_chain execution times on Xeon Phi .................................................50
Figure 21: vadd_abc execution times on Xeon Phi ......................................................51
Figure 22: matmul execution times on Xeon Phi ..........................................................51
Figure 23: 27stencil execution times on Xeon ...............................................................53
Figure 24: Himeno execution times on Xeon ...............................................................53
Figure 25: Le core execution times on Xeon .................................................................54
Figure 26: Mandel execution times on Xeon ...............................................................54
Figure 27: Bitonic execution times on Xeon .................................................................55
Figure 28: GEMM execution times on Xeon ...............................................................55
Figure 29: large_kernel execution times on Xeon Phi .................................................56
Acknowledgements

Upon the completion of this project, I would like to thank my supervisor Mark Bull for his constant guidance and support. I am also grateful to Codeplay Ltd for providing us the software that this dissertation was based upon and especially Paul Keir and Ruymann Reyes for answering all my questions. Finally, I would like to thank my family and friends for their unconditional love and support throughout my Master studies.
Chapter 1

Introduction

1.1 Motivation

Moore’s Law has been the fundamental driver behind application performance increase for over forty years. Shrinking transistor sizes, frequency escalation, microarchitecture and compiler advances have provided unprecedented performance gains. However, when frequency scaling reached its peak due to power limitations, the computing industry to be able to extend Moore’s Law turned to multiple core systems. Meanwhile, as multicore systems quickly began reaching their limits [1], Graphics Processing Units (GPUs), that outperform traditional Central Processing Units (CPUs) in both arithmetic throughput and memory bandwidth, upgraded their role from exclusively graphics rendering devices to general purpose computing devices. Alongside with GPUs we experienced the advent of many other unconventional architectures into traditional computing, called accelerators. These include Field Programmable Gate Arrays (FPGAs), Digital Signal Processors (DSPs), the IMB’s Cell Broadband Engine and more recently Intel’s Xeon Phi coprocessor.

Systems that include more than one kind of processor are called heterogeneous systems and the act of programming those systems, heterogeneous computing. Today’s personal computers are essentially heterogeneous systems, as they incorporate a multicore CPU for sequential tasks and a GPU for the computationally demanding graphic activities. The past few years, accelerators are also heavily used in High Performance Computing (HPC). The current No. 1 system, in the top500 list [2], Tianhe-2 is based on Intel Xeon Phi coprocessors and the No. 2, Titan, on Nvidia GPUs. A total of 62 systems on the list are using accelerator/co-processor technology. Accelerators could also constitute the building blocks of the next generation exascale systems as they deliver much more computing power per watt than CPUs. Therefore, the significance of heterogeneous computing and accelerator programming is very obvious.

To be able to exploit all the power that accelerators and heterogeneous systems provide, it is essential to have powerful programming models as well. Accelerators include fundamental differences in their hardware design, so the vendors that developed them also introduced a special programming model to program them. This has resulted in an abundance of programming models, such as CUDA for Nvidia GPUs, APP for AMD GPUs, Offload pragmas for Intel’s Xeon Phi just to name a
few. Developers are forced to learn new programming models for every new computing platform that quickly get out of date. This is clearly inefficient and has led to the need for a widely accepted heterogeneous programming model that will cover all the existing and future devices. Based on this vision the Khronos Group [3] established the OpenCL [4] programming model. OpenCL enables portable and efficient access to a wide range of devices, but it is a quite low level and difficult to learn programming model.

The ultimate goal of a programming model is to increase developer productivity. OpenCL’s low level characteristics, have prevented it from being developer’s top choice when it comes to parallel programming. Up to this point, OpenCL C is the only option to program OpenCL devices. C++ on the other hand, is a very popular widely spread, high level, object oriented programming model and in the same time, very close to plain C. As a result, C++ makes a very good candidate to be the base language for a high level model to program OpenCL devices.

The above reasons led the Khronos group to the development of a higher level model that targets OpenCL devices. The name of this new model is OpenCL SYCL [5] and the provisional specification released on March 19, 2014. SYCL enables developers to combine C++ and OpenCL for the creation of efficient libraries and applications. SYCL builds on top of OpenCL and the SPIR [6] intermediate representation and permits the use of many popular C++ techniques like templates, inheritance, operator overloading in OpenCL kernel development.

So far, SYCL is still in the phase of provisional specification. This means that a first draft of the SYCL specification has been released so that the Khronos group can gather community feedback. The next step is the release of the full specification. SYCL is an open standard which means that anyone can implement it. After the final specification is released vendors will be free to release their implementations.

1.2 Project Goals

Codeplay Ltd [7] is, up to this point, the only vendor that is publicly working on an implementation of SYCL. They have already implemented a first, yet unreleased, version, which was provided to us for investigation, as part of this project.

More specifically the overall goal of this project is to extensively assess this new programming paradigm, in terms of both performance and programmability. To achieve that we have developed several scientific applications from very simple to more complex ones in SYCL and other established parallel programming models, including OpenCL and OpenMP [8].

Currently SYCL can target only devices that support the SPIR intermediate representation. So far, Intel is the only vendor that supports SPIR, so the device we used to run the experiments is an Intel Xeon Phi coprocessor.
1.3 Report Structure

The dissertation includes a total of six chapters. Chapter 2 outlines the background for this project, including the OpenCL programming model, a few words about the SPIR intermediate representation and details on the hardware and software aspects of Intel Xeon Phi Coprocessor which is the device that we are running the experiments on.

Chapter 3 describes in detail the OpenCL SYCL programming model.

Chapter 4 covers application porting to SYCL. This includes, a brief description of the applications, design choices made and comments on the programmability of this new programming paradigm compared to OpenCL and OpenMP.

In Chapter 5 we present the performance results of SYCL, OpenCL and OpenMP, describe the experimental setup and try to explain performance differences.

Ultimately, Chapter 6 summarises SYCL programming model.
Chapter 2

Background Theory

2.1 OpenCL Programming model

Open Computing Language (OpenCL) [4] is an open, royalty free programming framework that targets heterogeneous systems. OpenCL essentially tackles the problem of programmability and portability. It enables developers to efficiently access systems that consists of multiple compute devices including multicore CPUs, GPUs and other accelerators like the Intel Xeon Phi. Applications developed in OpenCL are portable, meaning that they can correctly run on any OpenCL supported platform unmodified. OpenCL was created by the Khronos Group [3] in 2008 and has a wide industry support. The Khronos Group is a non for profit organisation comprised of many technology companies. Its goal is to create open standards that help accelerate computation on a wide variety of devices.

However, all these desirable, game changing features do not come without a certain cost. OpenCL is a rather low level model that targets expert developers. The low level characteristics make code development a tedious task, resulting in loss of productivity. What is more, portability also encompasses a high toll in the resulting performance. Most OpenCL compute devices include radical hardware differences. As a consequence, running an application on a different device than the one it was originally developed for, although it may produce correct results, performance most likely will be quite low. Therefore, OpenCL applications essentially need rewriting, when it comes to porting to other devices. This partially cancels the whole goal of OpenCL which is portability.

OpenCL is more than just a programming language. It is a programming framework that includes the OpenCL platform layer, runtime system and compiler. The platform layer is responsible for detecting OpenCL devices and to creating contexts. The runtime system allows the host to manage the context, e.g. to transfer data, launch kernels. OpenCL kernels are defined using the OpenCL C programming language, which sustains a fragment of the ISO C99 programming language with some additions for parallelism. Finally the OpenCL compiler creates executables from OpenCL kernels.
2.1.1 Platform Model

The OpenCL platform model concepts the underlying heterogeneous system as a host device connected to multiple OpenCL compute devices. A device incorporates a number of compute units (CU), which are further divided into processing elements (PE). Processing elements are the computational components that execute instructions, either in SIMD (Single Instruction Multiple Data) style or in SPMD style (Single Program Multiple Data). This model can be clearly depicted in Figure 1: OpenCL platform model.

![OpenCL platform model](image source [3])

2.1.2 Execution Model

OpenCL applications are split into two parts. The host program that executes on the host and kernels that execute on one or more devices. Usually, the sequential parts of an algorithm are executed on the host (CPU) and the computationally intensive parts are offloaded as kernels for execution on the devices.

There are four main concepts in the OpenCL execution model: contexts, command queues, kernels and memory objects. A context is a container that includes the host and one or more devices. All buffer allocations, kernels execution and data transfers take place within a specific context. The user can define multiple contexts that include the same or different devices and operate independently.

Creating a command queue related to a beforehand created context is the next step in design an OpenCL application. Command queues coordinate the execution of commands in an OpenCL application. There are two types of commands, memory transfer and kernel execution commands. The host program places commands in the queues, which subsequently schedule them for execution. Command queues can be in-order, where commands are pulled for execution in the order they were enqueued
or out-of-order, where the OpenCL runtime can rearrange commands for optimal execution. In the latter case the user is responsible for defining the dependencies between the commands that ensure correct execution.

Kernels in OpenCL express parallelism in its finer grain. They contain the computationally intensive part of the application that is executed on the device. All threads execute the kernel code in parallel, using different data. Also, different threads may follow different paths inside kernel code. Kernels are enqueued alongside with a “geometry” which is a 2-level 3-dimensional index space that describes the execution. Each thread represents an instance of the kernel and is called work-item. Work-items are identified on the index space based on a global ID. They are further organised into work-groups. Work-groups express a more coarse-grain level of parallelism and they are also assigned a unique work-group ID. Work-items inside a work-group are assigned a local ID. Work-items are executed in parallel on the processing elements, whereas work-groups are assigned on compute units.

OpenCL is equipped with a barrier synchronisation mechanism among the work-items inside a work-group. However, work-groups execute completely independent and cannot synchronise between each other. Figure 2: Example of a 2-dimensional kernel space decomposition shows an example of a 2-dimensional NDRange and describes how the global IDs can be calculated based on the local IDs, work-group IDs and work-group sizes. In this example, \((G_x, G_y)\) represent the global sizes, \((S_x, S_y)\) the local sizes and \((w_x, w_y)\) the work-group IDs and \((s_x, s_y)\) the work-item local IDs inside the work-group.

**Figure 2: Example of a 2-dimensional kernel space decomposition in OpenCL**
(image source [3])
Memory objects contain data that can be moved between the host and the devices and are created on the host. When a memory object is written, the corresponding data are transferred to the device. When it is read, data are transferred from the device to the host. Memory objects can be one of two types, buffer objects or image objects. The former can store one-dimensional arrays of elements, whereas the latter are used to store two or three-dimensional buffers or images.

### 2.1.3 Memory Model

The OpenCL memory model is based on the idea of a different address space between the host and the device. Figure 3: OpenCL conceptual memory model shows a diagram of the OpenCL memory hierarchy. Devices can incorporate four different memory regions:

- Global memory: This is the device’s DRAM memory, which is usually large and high-latency. All items in the global memory can be read and written by every work-item in every work-group. Global memory can be cached, depending on the capabilities of the device.
- Constant memory: This is a small, low-latency part of the global memory that remains constant during the execution. Items in constant memory are allocated and initialized by the host and visible by every work-item.
- Local memory: Each work-group contains a portion of local memory. Variables allocated in that region are shared inside a work-group and visible only to that specific work-group. This memory is usually small size, low-latency and close to the compute unit. If the device hardware does not support local memory, it can be simulated using regions of the global memory.
- Private memory: Each work-item comprises a small portion of memory visible only to that itself.

![Figure 3: OpenCL conceptual memory model](image source [3])
2.1.4 Programming Model

There are two types of programming models in OpenCL. The data parallel programming model, which expresses parallelism in its finer grain. In this model index space points, work-items and memory items are mapped one-to-one. On the other hand, the task parallel programming model expresses a coarser level of parallelism. It is proportionate to executing a kernel using work-groups that contain a single thread. The data parallel programming model maps more closely to the GPU architecture that contain large numbers of simplistic cores. Task parallelism maps better to CPU architecture that includes few, but powerful cores.

2.2 OpenCL SPIR

OpenCL was designed low level to enable developers access the full capabilities of the devices and with the thought that other higher level model that target OpenCL devices will be built on top of it. One way to support such higher level models, is through source-to-source compilers, translating new languages to OpenCL C. Khronos group, proposed a better approach by launching the Standard Portable Intermediate Representation (SPIR) [4]. SPIR is a portable encoding of OpenCL device programs in an intermediate representation (IR) that is based on LLVM IR. As a result, compilers for alternative new programming models can generate SPIR instead of OpenCL and have it run on any OpenCL backend that supports SPIR. A program compilation flow with SPIR can be seen in Figure 4: SPIR compilation flow.

![Figure 4: SPIR compilation flow (image source [5])](image source [5])
Moreover, SPIR serves yet another purpose, this of IP protection. To this point there are two options of shipping OpenCL kernels, either as source text or as a binary. The first option leaves the source code completely exposed and the second implies that the application will be run on a specific platform, or multiple binaries should be shipped, one for each available device. Shipping the kernel code as SPIR IR bitcode provides both legal protection and portability among devices.

2.3 Intel Xeon Phi

Intel with the Xeon Phi coprocessor, codenamed “Knights Corner”, introduces a completely new accelerator to the high performance computing market, which aims to directly compete with GPUs. Tianhe-2 the No. 1 system in the current top 500 list, comprises of 48,000 Xeon Phi coprocessors. There are another 16 systems in the top 500 list that are based on Xeon Phi acceleration [2]. Xeon Phi introduces a novel architecture with main characteristic a medium number of fairly powerful x86 cores, rather than the very large numbers of simplistic cores that GPUs contain. Like GPUs Xeon Phi is connected to the host via the PCIe bus. The greatest leverage of Xeon Phi over GPUs and other types of accelerators, is that it can be programmed using traditional programming models that software developers are accustomed to, such as C/C++, OpenMP and MPI. However, recompiling an existing application and running it on Phi, there is a high probability that the performance is going to be mediocre. To achieve high performance, a great deal of programming effort is required, mainly involving vectorization. An Intel Xeon Phi card can be seen in Figure 5: Intel Xeon Phi Coprocessor card.

![Intel Xeon Phi Coprocessor card](image source [6])
2.3.1 Hardware

Intel Xeon Phi is based on Intel’s many integrated core (MIC) architecture. It comes in many configurations regarding core count (57, 60, 61), main memory size (6, 8, 16GB) and clock speed (1053, 1100, 1238 MHz). The cores are connected via a 512-bit bidirectional ring interconnect. Each core comprises of 32KB L1 data cache, 32KB L1 instruction cache and 512KB L2 cache. L2 caches are connected via the ring interconnect resulting to a total of 32MB shared L2 cache. Hardware coherency is used to keep L2 caches fully coherent. A high level diagram of the Xeon Phi architecture can be seen in Figure 6: Intel Xeon Phi high level architectural diagram.

![Figure 6: Intel Xeon Phi high level architectural diagram (image source [7])](image)

Figure 7: Intel Xeon Phi single core architecture depicts the architectural diagram of a single Xeon Phi Knights Corner core. Each core is based on the P54C Pentium processor. It supports four concurrent threads via hyperthreading and can execute two instructions per clock cycle. Each core includes 512-bit wide vector processing unit (VPU). This means that it can pack 8 double-precision or 16 single precision floating point operations in the same instruction. The vector processing unit also supports Fused Multiply-Add (FMA) instructions increasing the number of floating point operation executed per cycle to 16 double precision and 32 single-precision. As a result, it is critical that the developers use both parallelism and vectorization to achieve high performance on Xeon Phi.
2.3.2 Software

Intel Xeon Phi runs a light-weight version of the Linux operating system and supports all the Intel software development tools, like compilers, debuggers, profilers and the Intel Math kernel Library. Currently, Xeon Phi acts as an accelerator to the host processor, to whom it is connected via PCIe. The next generation of Xeon Phi, named “Knights Landing”, as announced by Intel will be able to stand as a standalone processor. The maximum memory it can support is 16GB, thus it is not suitable for memory hungry applications. It supports most of the industry standard programming models, like C, C++ and Fortran. There are three ways of running an application on Xeon Phi:

- **Native execution**: In this mode the entire application runs completely on Phi without host interference. The application is first compiled on the host using the ‘mmic’ flag and then the user logs into and executes it on the Phi. All standard programming models are applicable here, including C, C++, Fortran, OpenMP, MPI.

- **Symmetric MPI execution**: In this mode both the host and the Phi act as MPI nodes. It requires caution to balance the work between host’s ‘big’ cores and Phi’s ‘small’ cores. This mode better suits hybrid MPI and OpenMP execution, with MPI threads running on host cores and OpenMP threads on the Phi. In this model Xeon Phi is viewed just as another node in a cluster.

- **Offload execution**: This the GPU-like mode where the host runs the application and offloads the computationally intensive parts to the
coprocessor. This can be done either using OpenCL, or the Intel Offload pragmas.

2.3.3 OpenCL on Intel Xeon Phi

As we will be running OpenCL applications on Xeon Phi it is worth mentioning a few words on programming and optimizing OpenCL code for Xeon Phi. As already mentioned, OpenCL is portable among devices, but that does not include portability in performance. OpenCL, which shares a lot with CUDA and is intuitively a programming model that fits better on the GPU architecture. Intel Xeon Phi architecture on the other hand, differs a lot from GPUs. Xeon Phi relies on fully coherent cache hierarchy, whereas GPUs include very fast local memory that needs to be programmed (although latest GPUs include cache memory). What is more, Xeon Phi relies on software scheduling of relatively small number of medium sized threads, while GPUs rely on hardware scheduling of very large numbers of tiny threads. As a result, porting applications to Xeon Phi and to GPUs require different kind of optimizations.

The main aspects that the developer needs to take care of include multi-threading and vectorization. For a 60-core Phi configuration, at initialization time the OpenCL driver creates 240 hardware threads, four for each physical core. It is important to notice here that unlike GPUs where every work-item is mapped on a thread/core, on Xeon Phi the smallest task that can be mapped on a thread is a work-group. Then the work-items inside the work-group are executed serially. As a result, running the kernel with less than 240 work-groups leaves the coprocessor underutilised. It is recommended to use more than 1000 work-groups for best performance.

The OpenCL compiler automatically vectorizes the implicit work-group loop over the work items in dimension zero. Currently, the vectorization width is 16. There is a limitation that the vectorized kernel is only used if the work-group size is greater than or equal to 16 and the also divisible by 16. More details on optimizing OpenCL for Xeon Phi can be found in [5].
Chapter 3

The OpenCL SYCL Programming Model

“You see, in creating OpenCL, we decided the best way to impact the industry would be to create a programming model for the performance-oriented programmer wanting full access to the details of the system. Our reasoning was that, over time, high-level models would be created to map onto OpenCL. By creating a common low-level target for these higher level models, we’d enable a rich marketplace of ideas and programmers would win.” – Tim Mattson [6]

OpenCL SYCL [7] comes to fill the missing part in the OpenCL software stack for heterogeneous programming. After the creation of the low-level OpenCL programming model and SPIR the cross-vendor intermediate representation, SYCL comes on top of them to provide a higher level programming model for heterogeneous systems. SYCL is a C++ programming model for OpenCL. It combines the ease of use and flexibility of C++ with the powerfulness and portability of OpenCL. The choice of C++ to build this high level programming model was not random. C++ is a very popular and widely used in the industry and academia high level, object oriented programming model. Moreover, OpenCL it is based on C and C++ is also based on C. Thus, it makes an excellent candidate to be used to build a higher level OpenCL programming model.

In SYCL developers can write standard C++ code and add very simple constructs to offload computationally intensive parts as kernels to OpenCL devices. It allows shared source design, which means that both C++ and SYCL code can be written inside the same source file. Furthermore, developers are able to use popular techniques like classes, inheritance and templating when defining kernels, making programming much more flexible. However, not all C++ features are enabled in SYCL, due to the underlying OpenCL framework limitations. These include virtual functions, function pointers, exceptions, and runtime type information. These limitations apply only inside kernels. The features can be used in the rest of the application as normal.

SYCL, like OpenCL is a programming framework. It includes a programming API, a runtime system and a device compiler. A SYCL application is compiled in a two phase compilation flow, where the source file that contains SYCL code is passed
from both the device compiler and a standard C++ compiler that the programmer chooses. The rest of the application is compiled only by the standard host compiler as normal. SYCL model does not exclude a single phase compilation, but is not designed to require it.

SYCL targets C++ libraries, middleware and applications developers who wish to accelerate their code in a seamless way. It builds on top of OpenCL, but it manages to hide much of its complexity. The SYCL model, handles the underlying platform and devices, as well as data movement. This results in reduced host side code and reduced complexity. Furthermore, all the OpenCL low-level features are available if the user wishes to use them.

To conclude, SYCL’s contribution can be summarized in the following points:

- Enable developing OpenCL kernels using C++ and using all C++ popular techniques like classes, inheritance, templating.
- Enable shared source design, where host and kernel code are in the same file.
- Added hierarchical parallelism constructs, which map better to CPU-like architectures.
- Hide much of OpenCL complexity by providing a runtime that automatically manages platforms, devices and data movement.

3.1 Relevant C++ features

It is worth devoting some space to describe the C++ features that constitute central concepts in the SYCL programming model.

3.1.1 Function Objects

In C++, function objects or functors [8] are objects that can act as functions. They are created as normal C++ classes that overload the function call operator. The following code snippet shows an example of a functor definition and its usage:

class adderFunctor {
    int k;
public:
    adderFunctor(int kk) : k(kk) {};
    int operator()(int l) {
        return k+l;
    }
};
int main(void) {
    adderFunctor add5(5);
    cout << add5(5) << endl;
    return 0;
}
In this example, we have created a functor that adds two numbers. The first operand of the addition is provided when the functor is created and the second when it is called. The example will print ‘10’ in the screen.

One might argue over what do functors offer more than regular functions. The key difference between a functor and a regular function is that the functor beyond local variables, global variables and parameters can also access the class data members (instance variable ‘k’ in the example above). In other words, functors contain state. In the example above, the functor add5 adds five to every argument it receives. However, the value five was not hardcoded and there can be created other functors that add different values. This characteristic makes functors nicely customizable. Functors are mostly used as arguments for Standard Template Library (STL) algorithms. For example:

```c++
int main(void) {
    vector<int> vec(10, 0);
    transform(vec.begin(), vec.end(), vec.begin(), adderFunctor(5));
    return 0;
}
```

The code above, uses the `adderFunctor` to add 5 to each element in a vector. STL algorithms require the last argument to be a unary or binary function. This means a function that receives one or two parameters respectively, which is quite restrictive. Functors help to solve this issue, as the additional information can be included as class-data.

### 3.1.2 Lambda Expressions

Lambda expressions were introduced in the C++11 programming standard. They are anonymous functions that can be written inline inside the source code. Anonymous functions, are functions that have a body, but do not have a name. Lambdas are usually passed to other functions, such as STL algorithms, the way functors or function pointers are passed. In cases where the function body is very simple, lambdas can be more compact, efficient and easier to maintain than using function objects.

A lambda expression is defined in the following way:

```
[ ] ( ) -> T { }
```

- **[ ]** Capture clause
  
  If the lambda has to access other local variables, except the ones in the parameter list, it has to capture these variables. Lambda can capture variables either by value (=) or by reference (&). For example:

  ```
  [=] capture all variables by value
  &[ ] capture all variables by reference
  [=, &foo] capture by value by default, but capture foo by reference
  ```
• ( ) Parameter list
  A parameter list is optional and resembles the parameter list of a regular function.
• → Return type
  The return type is usually deduced by the compiler. However, there is the option to define the return for more complex cases, where it cannot be deduced by the compiler.
• { } Lambda body
  The body of a lambda expression is the same as the body of a regular function. It includes the code that will be executed when the lambda is called.

The following example shows a lambda expression that adds two integers a and b, and returns the result into a variable named sum. The return type (type of variable sum) is automatically deduced by the compiler, hence we declare sum with the auto specifier:

```cpp
auto sum = [] (int a, int b) { return a+b; }
```

### 3.2 SYCL Platform Model

SYCL shares a very similar platform model with OpenCL. As in OpenCL, there is a host connected to multiple OpenCL compute devices. The devices include multiple compute units, which are further divided into multiple processing elements. The difference between the two is that in OpenCL the user creates a queue which is tied to a context. Then he adds the commands that he wishes to execute, like kernel invocations or data transfers in the queue. SYCL provides a different abstraction where the user submits commands groups that include OpenCL commands from the host to execute on the compute units.

### 3.3 SYCL Execution Model

To be able to run a kernel in OpenCL, the developer has to follow a long list of actions to set up the environment. In particular he has to create a platform, select a device, create a context, create a queue and enqueue the kernel in that queue. In SYCL the only object required to run a kernel is a queue. This enables quick and easy development of applications, avoiding the cumbersome OpenCL procedure. Platforms, devices, and contexts are set automatically by the runtime. However, this does not restrict the programmer from setting them manually if he wishes. There are seven main concepts in the SYCL execution model: Platforms, Contexts, Devices, Command groups, Kernels, Program Objects, Command queues. The only difference from the respective OpenCL model is the introduction of the concept of command
groups. Command groups include all the necessary OpenCL commands needed to correctly process host data on a device using a kernel.

In SYCL as in OpenCL the program is divided in two parts, the kernel that executes on the device and the host part that executes on the host. Unlike OpenCL, these two parts can be in the same source file. The kernel execution model is also the same. When a kernel is launched an index space is defined. Kernel instances called work-items are executed for each point of the index space. Work-items can be further organised into groups providing a coarser grain form of parallelism. Work-items inside a work-group can be organised in a one, two or three-dimensional grid and have a unique global ID that identifies them from every other work-item. Work-groups can be also organised in a one, two or three-dimensional grid. Each work-group has a unique ID and work-items inside that have a local ID.

3.4 SYCL Memory Model

One of the greatest facilitations that SYCL offers to developers is that it relieves them from having to manually move data from the host to the device and vice versa. The typical scenario in OpenCL is that the developer allocates and initializes data on the host. Then he allocates memory space on the device and transfers the data from the host memory to the device memory. After that the computation takes place on the device and then the user transfers the new data back from the device buffer to the host buffer.

In SYCL data storage is separated from data access. The developer creates a buffer on the device and associates it with host data. To access the data he needs to create an accessor object for the corresponding data, specifying the type of access (read, write, or both), the memory region that the access will take place (host or device) and the corresponding buffer. Then he can proceed to computation. The SYCL runtime system guarantees to have the data ready on the device when the kernel needs them, but no assumption can be made as to when exactly the data were transferred to the device. The data are transferred back to the host when the device buffer gets out of scope.

Memory hierarchy in SYCL is the same as in OpenCL, including the four distinct memory regions: global memory, constant memory, local memory and private memory.

3.5 SYCL Programming Model

A SYCL application is developed using standard C++. New C++11 features, like lambda functions are available and their usage is encouraged to improve program readability. As mentioned before a SYCL program is logically divided into kernels that execute on devices and host code. Kernels are invoked inside command groups and they can be defined inside or outside of a command group. There are four ways to define a kernel:

- **As a C++ functor:** In this case, class instance variables represent the kernel arguments. A class constructor is used to initialize those arguments.
kernel method is the method defined as the overloading of the function call operator. This option enables developers to create templated kernels.

- **As lambda function**: In C++11 functors can be defined as lambda functions. This is particularly helpful for short kernels to save space. Lambda functions are usually anonymous in C++, but in SYCL there is the requirement to provide a name to enable the linking between the SYCL device kernels and the host code that invokes them. The name is a C++ class.
- **As program object**: This option provided for OpenCL interoperability. It can be used when the developer needs to specify compiler flags or special linkage options for the kernel.
- **As OpenCL C string**: SYCL also provides this option for interoperability, as there are already many OpenCL kernels implemented are OpenCL Cstrings.

Moreover, SYCL provides three options for invoking a kernel:

- **Single task invoke**: This interface provides sequential execution of a kernel. It is equivalent to invoking a kernel with a single work-group that contains a single work-item.
- **Parallel for invoke**: This is the standard parallel invocation interface. The user can provide the index space organisation or he can just supply the total number of threads and the runtime will organise them as it sees fit.
- **Parallel for hierarchical invoke**: This interface offers two levels of parallelism. The work-group level for coarse grain parallelism where execution of all the commands follows single threaded approach. The work-item level allows execution among all the work-items inside the work-group. This is a new addition to the OpenCL model and maps better to CPU-like architectures.

### 3.6 SYCL Application Example

Figure 8: Vector addition implemented in SYCL programming language shows the implementation of a vector addition algorithm using SYCL.

In line 1, we include the header file “sycl.hpp” which provides all the SYCL features that we will need. These features are included in the `cl::sycl` namespace.

Inside the main function, in lines 4 through 11 we allocate and initialize memory on the host. This includes the two vectors to be added and a vector to store the result. After that follows the SYCL region. To better control the application flow and the construction and destruction of the various SYCL objects, SYCL specific code is wrapped in brackets, without this being compulsory.

In line 13 we declare a `queue`. A `queue` is the minimum object required to run a SYCL application. In this case, platforms, contexts and devices are all handled by the runtime. SYCL provides functionality for the developer to set them up if he wishes, but we are keeping this example simple.
Next, follows the definition of the device buffer objects, in lines 15 to 17. The template arguments in a buffer definition indicate the type of the values that the buffer holds and its dimensionality. The constructor parameters, point the host memory that the buffer is associated with and its size. For example, in line 15, we declare a one-dimensional buffer of $N$ integers, named $d_a$, that is associated with the array $h_a$ from the host. A device buffer being associated with host memory, means that for example, data from $h_a$ will be copied to $d_a$, if $d_a$ is requested for use in the kernel. Buffers can also be created without association to host memory.

After the buffers are defined, we define a `command_group` object in line 19, which enqueues the commands required to transfer and process the data on the given queue (`myQueue` in our case). Inside the `command_group` construct we firstly define the accessor objects for our buffers, providing the type of access we wish, read, write or both, in lines 20 to 22. Buffers only allocate memory in SYCL, whereas accessors are used to access data in buffers. Accessors are always declared inside the `command_group` construct.

After defining the accessor objects, we are ready to launch the kernel. In this example we invoke the kernel using a `parallel_for` construct, in line 24. The first argument of the `parallel_for` is the kernel execution range. The N-dimensional range as known from OpenCL is abstracted in SYCL by the `nd_range` object. The `nd_range` template argument indicates the dimensionality of the overall work space. A `nd_range` object receives two parameters, the global range and the local range. In this example, we define only the global range, to be a one-dimensional range of $N$ items. Leaving the local range undefined means that we let the runtime system decide the optimal work group size.

The second argument of the `parallel_for` is the kernel to be executed. This argument is passed as an instance of a `kernel_functor` object. The `kernel_functor` object is used when the kernel is defined either as a C++ functor or as a lambda expression. The template argument of the `kernel_functor` object defines the name of the kernel (`vaddKernel` in this example) in the form of a C++ class. This class does not have to be implemented and the name is used to link the kernel with the host code in a split compilation phase. The argument of the `kernel_functor` object is the name of the C++ functor that defines the kernel or a lambda expression. In our case the kernel is defined as a lambda expression. The kernel body spans from line 25 to 29. As it can be noticed, we pass an `item` object in the kernel parameters, which provides all the information around a work-item, such as its global id.

Line 32 points the end of SYCL application scope. Before, buffers are destroyed alongside with the rest of the SYCL objects, their contents are copied back to the host memory. This means that vector $h_c$ after line 32 will contain the result of the addition of vectors $h_a$ and $h_b$. 

21
```cpp
#include "SYCL/sycl.hpp"
using namespace cl::sycl;

int main(void) {
    int N = 16;
    int h_a[N];
    int h_b[N];
    int h_c[N];
    for (int i = 0; i < N; i++) {
        h_a[i] = h_b[i] = 1;
        h_c[i] = 0;
    }

    // SYCL region starts here
    queue myQueue;
    // Device buffers
    buffer<int, 1> d_a(h_a, N);
    buffer<int, 1> d_b(h_b, N);
    buffer<int, 1> d_c(h_c, N);

    command_group(myQueue, [&](){
        auto a = d_a.get_access<access::read>();
        auto b = d_b.get_access<access::read>();
        auto c = d_c.get_access<access::write>();

        parallel_for(nd_range<1>(range<1>(N)), kernel_functor<class vaddKernel>[m](item item)
        {
            int i = item.get_global_id(0);
            if (i < N) {
                c[i] = a[i] + b[i];
            }
        });
    });

    // End of SYCL region

    return 0;
}
```

Figure 8: Vector addition implemented in SYCL programming language
Chapter 4

Application Development with SYCL

4.1 Applications

To test SYCL we selected a total of six applications to port in three programming models, OpenCL, SYCL and OpenMP. There were two primary sources of applications, the EPCC OpenACC benchmark suite [9] and the Intel OpenCL code samples [10]. We have chosen to test SYCL against OpenCL because to our point of view this is the most relevant comparison, as SYCL is based on OpenCL and both models target heterogeneous systems. OpenMP was selected as an established, widely used parallel programming API, which is characterised for its ease of use and now in version 4.0 can target accelerators. Other programming models where not used either because implementing application would require substantial amount of time (MPI), or lack of compilers for Intel Xeon Phi (OpenACC).

EPCC OpenACC benchmark suite is designed to test the performance of compilers and systems using the OpenACC standard. OpenACC [11] is a relatively new parallel programming standard that enables directive programming for accelerators. The suite consists of three levels of applications. The first level comprises of low-level operations that measure raw performance of compilers and systems, including execution times for various data movement configurations and other OpenACC constructs. The second level measures performance for some BLAS-type routines. Finally, the third level includes OpenACC parallelization of three real scientific applications.

We focused our efforts on the third level of the EPCC OpenACC benchmark suite. The three applications that it consists of are “27stencil”, “le_core”, “himeno” and will be described later in this chapter. They there selected for porting as they are reasonably simple applications in terms of length and code complexity. Moreover, the already existing OpenACC directives were already exposing the parallelism, letting us focus all our attention on the language constructs. The three applications were first ported in OpenCL, then in SYCL and finally in OpenMP. Our goal, apart from comparing performance, was also to establish a programmability metric, i.e. test how easy it is to port an application in each model.
The second source of applications was the Intel OpenCL code samples provided alongside with the Intel OpenCL SDK. We used two applications from there, “GEMM” and the “BitonicSort”. Finally, the last application we used is an independent benchmark suite that provides an OpenCL implementation of the Mandelbrot set [19]. These three applications were already implemented in OpenCL and we ported them to SYCL and OpenMP. Several other benchmarks were considered for porting, but ultimately they were not used either because they were presenting compilation complexities or they required features that were not implemented in the SYCL version that we were using (e.g. calling methods from other libraries inside SYCL kernels).

4.1.1 27stencil

Stencil operations are typically used in Partial Differential Equations (PDEs) solvers, which are encountered in many scientific applications, such as fluid dynamics. In a stencil operation every point inside a multidimensional grid is updated in both time and space, based on the weighted contribution of a subgroup of its neighbours. In our case, the grid is 3-dimensional and each point is updated based on the values of 27 of its neighbours. The application uses Jacobi iteration to model time, which means it includes two arrays, one that is only written to and another that is only read from. Computation of each grid point is independent of every other point, which means that the application is embarrassingly parallel. The main computational part of the application is carried out inside the time integration loop, which includes two triple-for loops. It can be seen in the following piece of code:

```cpp
for (iter = 0; iter < ITERATIONS; iter++) {
    #pragma acc parallel loop
    for (i = 1; i < n+1; i++) {
        #pragma acc loop
        for (j = 1; j < n+1; j++) {
            #pragma acc loop
            for (k = 1; k < n+1; k++) {
                a1[i*sz*sz+j*sz+k] = /* Sum of 27 neighbouring points. */
            }
        }
    }
}
```

/* end iteration loop */
The first triple for-loop calculates every point in the grid and seconds copies the results back, for the next iteration. Both loops are offloaded to the device. As a result in our OpenCL implementation we created two kernels, each one for the calculation operation and one for the copy operation. In the OpenMP implementation we inserted directives to both loops, including the collapse(3) construct, which collapses the three nested loops into one larger loop, providing a larger iteration space.

4.1.2 Le_core

This application performs a linear elasticity simulation of materials. Linear elasticity [13] is a mathematical model of how solid materials perform under stress conditions. The simulation is modelled as a linear system of hyperbolic partial differential equations of the form:

$$\frac{du}{dt} + A_x \frac{du}{dx} + A_y \frac{du}{dy} = 0$$

During the time integration the system is solved in two step using the dimension split method:

1. Integrate system: $$\frac{dv}{dt} + A_x \frac{dv}{dx} = 0$$
2. Integrate system: $$\frac{du}{dt} + A_y \frac{du}{dy} = 0$$

All the information needed for the simulation is stored in a single data structure called ‘le_task’. This data structure contains information about the material, as well as the grid of nodes of the elastic structure, which is two dimensional. The time integration loop can be seen in the following code snippet, with the two functions implementing the two steps of the solution:

```c
for(i=0; i<noSteps; i++){ 
   le_step_x_mil(task->dt, task->mat, task->h, task->n.x, task->n.y, task->grid, task->tmpGrid); 
   le_step_y_mil(task->dt, task->mat, task->h, task->n.x, task->n.y, task->tmpGrid, task->grid); 
}
```

Each function uses a double for loop to run over each point of the grid and calculate its new value, which are parallelized in the OpenACC version:

```c
#pragma acc parallel loop gang, present(inGrid[0:nx*ny],outGrid[0:nx*ny])
for (j = 0; j < ny; j++) { 
   #pragma acc loop vector 
   for (i = 0; i < nx; i++) { 
      /* Computation for each point. */ 
   } 
}
```
Implementing this in OpenCL, we created two kernels, one for each step of the solution. Kernels are launched using as many threads as the nodes of the grid. Data need to transferred twice, before the beginning and at after end of the computation. For the OpenMP version we replaced the OpenACC directives with OpenMP directives. We used the construct collapse(2) to unleash more parallelism.

### 4.1.3 Himeno

Himeno [12] application was originally created by Dr. Ryutaro Himeno to measure the CPU performance in floating point operations per second, as well as memory access speed. Its computational kernel solves the Poisson equation using the Jacobi iteration. This code appears mostly in incompressible fluid analysis applications. It consists of a time integration loop, which models the steps in the Jacobi iteration. There are five main arrays involved in this application:

- a, b, c: coefficient matrices
- wrk1: source term of Poisson equation
- wrk2: working area
- p: pressure matrix

Each time point includes two triple for loops. The first calculates wrk2 matrix based on the coefficient matrices, the source term of the Poisson equation, as well as the pressure values from the previous iteration. The calculation also includes a reduction variable. The second triple for loop updates the pressure matrix with the new values from wrk2 matrix. The main computational part can be seen in the following piece of code:

```c
double gosa, s0, ss;
for(n=0;n<nn;++n){
    #pragma acc parallel loop private(i,j,k,s0,ss), reduction(+:gosa)
    for(i=1 ; i<imax-1 ; ++i){
        for(j=1 ; j<jmax-1 ; ++j){
            for(k=1 ; k<kmax-1 ; ++k){
                s0 = /* expression based on arrays a,b,c,wrk1,p. */
                ss = ( s0 * a[i][j][k][3] - p[i][j][k] ) * bnd[i][j][k];
                gosa = gosa + ss*ss;
                wrk2[i][j][k] = p[i][j][k] + omega * ss;
            }
        }
    }
    #pragma acc wait
    #pragma acc parallel loop
    for(i=1 ; i<imax-1 ; ++i)
        for(j=1 ; j<jmax-1 ; ++j)
            for(k=1 ; k<kmax-1 ; ++k)
                p[i][j][k] = wrk2[i][j][k];
} /* end n loop */
```


The computation of the reduction variable is quite easy to be implemented in both OpenMP and OpenACC as these models provide already implemented constructs for such computations. On the other hand, it involves a lot more complexity and development time in OpenCL. The OpenCL implementation was split in three kernels. The first kernel computes the values of ‘s0’ and stores them in another array so that the second kernel, which implements a parallel reduction algorithm in OpenCL, can reduce them. The third kernel copies the results to the pressure matrix.

4.1.4 Mandel

Mandel [14] is an OpenCL implementation of Mandelbrot set. The mandelbrot set is included in the category of fractals which are patterns that repeat themselves at every scale. A fractal can be a geometrical figure that as we zoom in, it exhibits the same shape and goes on until infinity. The particular application draws the mandelbrot set in a user defined dimensions window. To achieve that it loops through every pixel of the window and decides if it belongs to the mandelbrot set or not. Then it colours the pixel with a specific colour for each case. The mandelbrot set consists of complex numbers. To determine if a complex number $c$ is part of the mandelbrot set we use the iterative formula $z = z \ast z + c$. We begin from $z = 0$ and compute new values of $z$ in every iteration. If $z$ is bounded by some number it belongs to the mandelbrot set. If it goes to infinity, it does not. The number of iterations that we use to decide if a point is part of the mandelbrot set is not constant and is also user defined.

Calculation of the mandelbrot set is an embarrassingly parallel problem as all pixels in the image are independent and can be calculated in parallel. The OpenCL implementation consisted of one kernel that included the above code. The kernel was launched with as many threads as the pixels in the image.

4.1.5 GEMM

General Matrix Multiply (GEMM) [10] is an application included in the Intel OpenCL sample codes. It is an efficient implementation of the matrix multiplication operation of two dense matrices. More specifically this implementation is designed to utilize cache memory more efficiently by introducing a tiling or blocking algorithm. In such algorithms matrices are divided into blocks and the calculations take place between those blocks, resulting in better data locality.

When two matrices are multiplied, each point in the output matrix is based on the dot product of a column and a row of the two input matrices. The naïve matrix multiplication algorithm turns out to be quite cache inefficient as it brings much data in the cache that are not used and evict data that will be needed again later. The main idea of the tiled version is that we load blocks of elements from both input matrices inside the cache and reuse them until they are not needed for any further calculations. The size of the block is chosen based on the cache size. This application was already implemented in OpenCL and we ported it to SYCL and OpenMP.
4.1.6 Bitonic

This application comes also from the Intel OpenCL code samples. Bitonic sort [16] is one of the fastest sorting networks algorithms. Sorting networks [17] are special sorting algorithms where the sequence of comparisons is not data-dependent, thus making it ideal candidate for implementation on parallel systems. A nice explanation of the algorithm including examples can be found in [18].

In OpenCL the full sorting algorithm is implemented by repetitive calls to the sorting kernel. There is a limitation that the input array of integers to sort is of size $2^N * 4$, where N is a positive integer number. The local work-group size is left to the runtime to decide by providing ‘NULL’ when running the kernel.

4.2 Development with SYCL

4.2.1 Device Selection

Syclone includes an abstract base class called \textit{device_selector}. This class implemented can instruct the runtime on how to perform device selection. The selection process takes place inside the overloading of the function call operator which is a pure virtual function in the base class and needs to be implemented by the derived classes. Syclone includes some build-in options for device selection, such as, \textit{default_selector}, \textit{cpu_selector}, \textit{gpu_selector}. In our case, to select Xeon Phi as the device we created the \textit{phi_selector} class, which can be seen below:

```cpp
class phi_selector : public device_selector {
public:
    phi_selector() : device_selector() {}

    int operator()(const device &device) const {
        if (device.has_extension(strdup("cl_khr_spir"))) {
            if (device.get_info<CL_DEVICE_TYPE>() == CL_DEVICE_TYPE_ACCELERATOR) {
                return 1;
            }
        }
        return -1;
    }
};
```

This is the simplest case for selecting a device. However, this implementation enables the developer to create more complex schemes where the appropriate device can be selected automatically, based on the characteristics of the application.
4.2.2 Kernel Definition

We have used three different methods for defining our kernels, lambda functions, C++ functors and OpenCL C strings, depending on the size and the characteristics of the kernels. Below we present, in general, how kernels can be defined in each of the three methods:

- **Lambda function**

  ```cpp
  parallel_for(nd_range<>(range<>(), range<>(N)),
              kernel_functor<class kernel_name>([](item item)
     {
       int i = item.get_global_id(0);

       /* kernel code */
     }));
  ```

  In the example above, the kernel is defined inline, inside the `parallel_for` kernel invocation construct. Although lambda functions are anonymous, SYCL requires naming them, in order to enable linking between the device kernel and the host that invokes the kernel. The name is a C++ class, `class kernel_name` in our example. The class is only used to name the kernel and does not need to be implemented.

  Moreover, SYCL requires variables to be captured by value. The lambda parameter list is optional, but it usually includes an object of the class `item`, which enables us to identify a work-item. It provides all the work-item built-in functions, such as `get_global_id`, `get_local_id`, `get_group_id` etc.

- **C++ functor**

  The following code example shows the definition of a SYCL kernel that performs vector addition, as a templated C++ functor:
The kernel arguments, which are usually accessor objects, are declared as class instance variables and initialized by the constructor of the functor. The overloaded function call operator includes the kernel code and expects an item object argument. The following code snippet show how this kernel can be invoked:

```
command_group(myQueue, [&]() { 
  auto a = d_a.get_access<access::read>();
  auto b = d_b.get_access<access::read>();
  auto c = d_c.get_access<access::write>();

  auto functor = vadd(a, b, c, N);

  parallel_for(nd_range<>(range<>(N)), kernel_functor(functor));
});
```

- **OpenCL C String**

  This option is provided for interoperability with OpenCL.

  Kernel definition:
const char *kernel_src = R"EOK(
   __kernel void foo(__global int *k) {
    int i = get_global_id(0);
    k[i] = i;
   }
)EOK";

Kernel invocation:

program foo_program(kernel_src, myQueue.get_context());

kernel *foo_kernel = foo_program.compile_kernel_by_name("foo");

command_group(myQueue, [&]() {
   auto buf = input_buffer.get_access<access::write>();
   foo_kernel->set_kernel_arg(buf);
   parallel_for(nd_range<>()((range<>(N)), foo_kernel);
});

As it can be noticed the kernel is defined using the OpenCL C notation. This enables rapid and easy porting of already implemented OpenCL kernels to SYCL. Launching the kernel requires a few extra steps than the standard procedure, including defining program and kernel objects, as well as manually setting kernel arguments.

4.3 Comparison of OpenCL, SYCL and OpenMP

In this section we will provide a programmability comparison between the three parallel programming models, OpenCL, SYCL and OpenMP as experienced by developing applications in all of them. To do that, we will an example of an application as developed in all models. Figures 15 through 19 show the source codes for the OpenACC, OpenCL, SYCL and OpenMP versions of the 27Stencil application, respectively. In particular, figures expose only the computational part of the application rather than the whole application, which would have been too long.

Out of all the programming models studied OpenCL entails the higher programming costs. First of all, OpenCL encompasses considerable amount of time and effort to learn the language. Secondly, re-writing an application in OpenCL, requires significant restructuring and major changes compared to the original code. Moreover, OpenCL being a quite low-level model, application development turns out to be quite time consuming. This is mainly due to the high effort required by the developer to set up and manage the system i.e. choosing platforms, devices, contexts, creating queues, buffers,
moving data around, launching kernels. Finally, OpenCL, results in huge amounts of code, even for relatively simple applications, that can be burdensome to maintain. Re-writing 27\textit{stencil} in OpenCL resulted in 78 lines of code (including both host code and the kernels) compared to the 42 lines of code of the original OpenACC code.

SYCL on the other hand, presents a much more elegant and compact programming interface. It is easier to learn, although having previous OpenCL knowledge helps a lot. It reduces programming effort and results in better structured code that would be easier to maintain. Using SYCL the developer can very quickly run simple cases, using the minimum objects required (a \textit{queue}), letting the runtime system handle the rest of the resources. However, SYCL also enables the creation of more customized execution environments, by providing functionality that let the developer handle all the resources. SYCL requires significant code restructuring, although no major code additions are needed. Developing 27\textit{stencil} in SYCL resulted in only 44 lines of code, which is quite close to the 42 lines of code of the OpenACC code.

OpenMP belongs in the class of directive programming models. It is doubtlessly the simplest parallel programming API, as to run a computational kernel in parallel requires only a single line of code. This line of code points out that the following part of code, marked by the directive, will be executed in parallel. Then, the runtime system is responsible for creating and destroying threads, partitioning the work and ensuring work balance. Furthermore, for more customized usage, OpenMP provides simple constructs that enable the developer to instruct the compiler on how to partition the work. However, the OpenMP model, suites better for multicore systems, such as CPUs and its usage is restricted only for shared memory systems, although the new OpenMP 4.0 specification includes support for accelerators. Developing 27\textit{stencil} in OpenMP reduced the lines of code of OpenACC version to 34 lines of code. It only needed to add one line of code above each triple-for loop to execute it in parallel.
#pragma acc data copy(a0[0:sz*sz*sz]), create(a1[0:sz*sz*sz], i,j,k,iter), copyin(sz,fac,n)
{
    for (iter = 0; iter < ITERATIONS; iter++) {
        #pragma acc parallel loop
        for (i = 1; i < n+1; i++) {
            #pragma acc loop
            for (j = 1; j < n+1; j++) {
                #pragma acc loop
                for (k = 1; k < n+1; k++) {
                    a1[i*sz*sz+j*sz+k] = (a0[i*sz*sz+(j-1)*sz+k] + a0[i*sz*sz+(j+1)*sz+k] +
                                            a0[(i-1)*sz+sz+(j-1)*sz+k] + a0[(i+1)*sz+sz+(j+1)*sz+k] +
                                            a0[(i+1)*sz+sz+(j-1)*sz+k] + a0[(i+1)*sz+sz+(j+1)*sz+k] +
                                            a0[i*sz+sz+(j-1)*sz+(k-1)] + a0[i*sz+sz+(j+1)*sz+(k-1)] +
                                            a0[i*sz+sz+(j-1)*sz+(k+1)] + a0[i*sz+sz+(j+1)*sz+(k+1)] +
                                            a0[(i-1)*sz+sz+(j-1)*sz+(k-1)] + a0[(i+1)*sz+sz+(j-1)*sz+(k+1)] +
                                            a0[(i-1)*sz+sz+(j+1)*sz+(k+1)] + a0[(i+1)*sz+sz+(j+1)*sz+(k+1)] +
                                            a0[(i+1)*sz+sz+(j-1)*sz+(k+1)] + a0[(i+1)*sz+sz+(j+1)*sz+(k+1)] +
                                            a0[i*sz+sz+j*sz+(k-1)] + a0[i*sz+sz+j*sz+(k+1)]
                        ) * fac;
                    }
                }
            }
        }
    }
}
/* end iteration loop */
/* end data region */
#pragma acc wait

Figure 9: 27stencil OpenACC version
Figure 10: 27stencil OpenCL version
```
__kernel void stencil(__global double *in, __global double *out, double fac) {
  int i = get_global_id(0)+1;
  int j = get_global_id(1)+1;
  int k = get_global_id(2)+1;
  int sz = get_global_size(0)+2;

  out[i * sz * sz+j*sz+k] = {
    in[i*sz*sz+(j-1)*sz+k] + fac*in[i*sz*sz+(j+1)*sz+k] +
    in[(i-1) * sz * sz+j*sz+k] + in[(i+1) * sz * sz+j*sz+k] +
    in[(i-1) * sz * sz+(j-1) * sz+k] + in[(i-1) * sz * sz+(j+1) * sz+k] +
    in[(i+1) * sz * sz+(j-1) * sz+k] + in[(i+1) * sz * sz+(j+1) * sz+k] +
    in[i * sz * sz+j*sz+(k-1)] + in[i * sz * sz+j*sz+(k+1)] +
    in[(i-1) * sz * sz+(j-1) * sz+(k-1)] + in[(i-1) * sz * sz+(j+1) * sz+(k+1)] +
    in[(i+1) * sz * sz+(j-1) * sz+(k-1)] + in[(i+1) * sz * sz+(j+1) * sz+(k+1)] +
    in[i*sz*sz+(j-1)*sz+(k+1)] + in[i*sz*sz+(j+1)*sz+(k+1)] +
    in[(i-1)*sz*sz+j*sz+(k+1)] + in[(i+1)*sz*sz+j*sz+(k+1)] +
    in[(i-1)*sz*sz+(j-1)*sz+(k+1)] + in[(i+1)*sz*sz+(j-1)*sz+(k+1)] +
    in[i*sz*sz+j*sz+(k-1)] + in[i*sz*sz+j*sz+(k+1)] +
  } * fac;
}

__kernel void copy(__global double *a0, __global double *a1) {
  int i = get_global_id(0)+1;
  int j = get_global_id(1)+1;
  int k = get_global_id(2)+1;
  int sz = get_global_size(0)+2;

  a0[i*sz*sz+j*sz+k] = a1[i*sz*sz+j*sz+k];
}
```

Figure 11: 27-stencil OpenCL kernels


```cpp
    
    // SYCL region starts here
    phi_selector selector;
    queue myQueue(selector);
    buffer<double, 1> d_a0(a0, sz*sz*sz);
    buffer<double, 1> d_a1(a1, sz*sz*sz);

    command_group (myQueue, &[]() {
      auto a0 = d_a0.get_access(access::read_write);
      auto a1 = d_a1.get_access(access::read_write);

      for (iter = 0; iter < ITERATIONS; iter++) {
        parallel_for(nd_range<3>(range<3>(n,n,n), range<3>(182,1,1)), kernel_functor<class
                       kernel_compute>([=](item item) {
          int i = item.get_global_id(0) + 1;
          int j = item.get_global_id(1) + 1;
          int k = item.get_global_id(2) + 1;
          int sz = item.get_global_size(0) + 2;

          a1[i*sz*sz+j*sz+k] = (a0[i*sz*sz+(j-1)*sz+k] +
          a0[(i-1)*sz*sz+j*sz+k] + a0[(i+1)*sz*sz+j*sz+k] +
          a0[(i-1)*sz*sz+(j-1)*sz+k] + a0[(i+1)*sz*sz+(j-1)*sz+k] +
          a0[(i-1)*sz*sz+(j-1)*sz+k] + a0[(i+1)*sz*sz+(j-1)*sz+k] +
          a0[i*sz*sz+(j-1)*sz+(k-1)] + a0[i*sz*sz+(j+1)*sz+(k-1)] +
          a0[(i-1)*sz*sz+(j+1)*sz+(k-1)] + a0[(i+1)*sz*sz+(j+1)*sz+(k-1)] +
          a0[(i-1)*sz*sz+(j+1)*sz+(k-1)] + a0[(i+1)*sz*sz+(j+1)*sz+(k-1)] +
          a0[i*sz*sz+(j-1)*sz+(k+1)] + a0[i*sz*sz+(j+1)*sz+(k+1)] +
          a0[(i-1)*sz*sz+(j+1)*sz+(k+1)] + a0[(i+1)*sz*sz+(j+1)*sz+(k+1)] +
          a0[(i-1)*sz*sz+(j+1)*sz+(k+1)] + a0[(i+1)*sz*sz+(j+1)*sz+(k+1)] +
          a0[i*sz*sz+j*sz+(k-1)] + a0[i*sz*sz+j*sz+(k+1)])
          ) * fac;
        });

        parallel_for(nd_range<3>(range<3>(n,n,n), range<3>(182,1,1)), kernel_functor<class
                       kernel_copy>([=](item item) {
          int i = item.get_global_id(0) + 1;
          int j = item.get_global_id(1) + 1;
          int k = item.get_global_id(2) + 1;
          int sz = item.get_global_size(0) + 2;
          a0[i*sz*sz+j*sz+k] = a1[i*sz*sz+j*sz+k];
        });
      }
    } // SYCL region ends here
```
```c
for (iter = 0; iter < ITERATIONS; iter++) {
    #pragma omp parallel for default(none) private(i,j,k) shared(a0, a1, fac, n, sz) collapse(3)
    for (i = 1; i < n+1; i++) {
        for (j = 1; j < n+1; j++) {
            for (k = 1; k < n+1; k++) {
                a1[(i*sz*j+sz*k)] = (a0[(i*sz*(j-1)*sz+k)] + a0[(i*sz*(j-1)*sz+k)] + a0[(i-1)*sz*(j-1)*sz+k] + a0[(i-1)*sz*(j-1)*sz+k] + a0[(i-1)*sz*(j-1)*sz+k]) + a0[(i-1)*sz*(j-1)*sz+k] + a0[(i-1)*sz*(j-1)*sz+k] + a0[(i-1)*sz*(j-1)*sz+k] + a0[(i-1)*sz*(j-1)*sz+k] + a0[(i-1)*sz*(j-1)*sz+k] + a0[(i-1)*sz*(j-1)*sz+k];
            }
        }
    }

    #pragma omp parallel for default(none) private(i,j,k) shared(a0, a1, n, sz) collapse(3)
    for (i = 1; i < n+1; i++) {
        for (j = 1; j < n+1; j++) {
            for (k = 1; k < n+1; k++) {
                a0[(i*sz*j+sz+k)] = a1[(i*sz*j+sz+k)];
            }
        }
    }
} /* end iteration loop */
```

Figure 13: 27stencil OpenMP version
Chapter 5

Performance Results and Analysis

Apart from programmability and ease of use, performance is a vital metric for a programming language, especially when it targets accelerators and high performance computing systems. We evaluated SYCL performance-wise by executing the six benchmark applications described in Chapter 4 on the Intel Xeon Phi coprocessor that is hosted at the EPCC’s Hydra cluster [18]. In this chapter, we will provide details on the experimental evaluation setup and present the performance results gathered.

5.1 Experimental Platform

As SYCL is built on top of SPIR and Intel is the only vendor that supports SPIR to this point, the device used to test SYCL performance was an Intel Xeon Phi coprocessor. The platform we used was EPCC’s Hydra Cluster. Hydra Cluster took its name from “Lernaean Hydra”, a monster from the Greek mythology that possesses many heads. In this notion, the Hydra Cluster comprises of a single frontend and multiple different backend nodes. Each backend node comprises of a different technology, like multicore CPUs, GPUs or the Intel Xeon Phi. We will focus our attention on the Phi backend node, which is the one we used for the dissertation.

The Phi backend of the Hydra Cluster consists itself of a frontend and a backend. The frontend comprises of two Intel Xeon E5-2650 processors and the backend accommodates two Intel Xeon Phi coprocessor cards, named mic0 and mic1. Intel Xeon E5-2650 is an 8-core processor, supporting one thread per core at 2 GHz, resulting in a total of 16 threads for the whole frontend. Each core includes 32 KB L1 instruction cache, 32 KB L1 data cache and 256 KB L2 cache. There is a total of 20 MB shared L3 cache per processor. The frontend comprises of 64 GB of main memory. The version of both Intel Xeon Phi cards is 5110P. A Phi card of this version comprises of 60 cores that run at 1.053 GHz, 30 MB L2 cache and only 8 GB of main memory, which make the coprocessor unsuitable for memory hungry applications.
5.2 SYCLONE Installation

Codeplay Ltd provided us with its implementation of SYCL, named “Syclone”. The software package included the SYCL device compiler, runtime system and library, as well as a variety of example applications. We should notice that Syclone is still in development and the version we were provided does not include all the features that are described in the Khronos Group provisional specification. Installation on the Phi server of the Hydra machine turned out to be not a trivial procedure. Syclone’s requirements for the latest versions of many software packages and libraries, combined with the complexity of the Hydra Cluster, resulted in this procedure occupying substantial time at the beginning of the project.

Some of Syclone’s requirements include C++11 support, cmake 2.8 or greater, glibc version 2.15 or greater and glibcxx version 3.4.15 or greater. The main obstacles were the C and C++ libraries as the phi backend of the Hydra Cluster has older versions installed. glibc is the core library of a linux system, required by every single program, including the kernel. Therefore, it is not upgraded very often and updating it in a system like Hydra was not a viable option. The problem was diverted by installing newer versions of both glibc and glibcxx libraries on my directory and the setting LD_PRELOAD environment variable to point to them. LD_PRELOAD is used to load an object, before any other objects in the system.

The phi backend includes two sets of developer tools. To achieve C++11 support it is essential to load “devtoolset-2”, which includes newer compiler versions. Finally, as we successfully compiled Syclone using all the newest tools, when we attempted to compile SYCL applications, the Syclone was still looking at the old version compiler directories resulting in errors. This complication was resolved by setting the CPLUS_INCLUDE_LIBRARY environment variable to the newest version compiler directory.

5.3 Experimental Setup

To attain a deep perspective into SYCL application performance, we decided to compare it against OpenCL and OpenMP. In our point of view, comparing against OpenCL is the most straightforward and reasonable comparison, as SYCL is based on OpenCL and both are heterogeneous programming models that target the same SPIR intermediate representation. We also thought that it would very interesting to see how SYCL performs against another established, widespread and mature parallel programming model. Apart from OpenCL, there are plenty of other models to choose from, such as MPI, OpenMP, CUDA, OpenACC. As the platform we are running the experiments is an Intel Xeon Phi, CUDA and OpenACC were excluded. Moreover, MPI was ruled out due to the high complexity of development. As a result, we choose OpenMP to be the second model that we compare SYCL with.

Our main performance metric for the comparison between SYCL, OpenCL and OpenMP, was execution time measured in seconds. We run the six benchmark
applications described in section 4.1, for various configurations in terms of dataset size and kernel iteration count (how many times the kernel was run over). When measuring times, we focused on the computational part, instead of the whole application. To get more accurate, where system setup phase and data transfers do not dominate the execution time we executed the computational kernels repeatedly inside for loops. For all the experiments we used Intel Compilers to compile the applications on the host side, with the default optimization level. Device side code was compiled by the OpenCL compiler provided with the Intel OpenCL SDK for the OpenCL versions and with Syclone device compiler for the SYCL versions.

To run a kernel in OpenCL or SYCL the developer needs to determine the geometry of the execution, i.e. how work-items and work-groups are arranged. This is accomplished by passing two parameters in the kernel launch command, the global size and the local size. Both can be one, two or three dimensional spaces. Unlike CUDA, in OpenCL the global size is expressed in terms of the total number of work items. Then the runtime system divides it by the local size to calculate the number of work-groups. OpenCL devices present limitations regarding these sizes, as well as other parameters. To obtain information about both Intel Xeon Phi and Intel Xeon processors, we created and run a device query program in OpenCL. The results are presented in Table 1: Device query results.

Noteworthy is that from the 60 physical cores of Xeon Phi, OpenCL recognizes the 59 of them, multiplied by 4 hardware threads per core, results in the 236 OpenCL parallel compute units. The 60th core is not touched by the application and is only used to run the offload daemon. On the other hand, OpenMP native execution can use all 60 cores. Moreover, OpenCL recognizes only the 6 out of the 8 GB of Xeon Phi main memory. A single OpenCL memory allocation on Xeon Phi is limited to a maximum of 2 GB, which prevented us from running big datasets.

Table 2: OpenCL and SYCL execution parameters displays the all the parameters used to run the OpenCL and SYCL versions. These include, the global size, which accounts for the total numbers of work-items, the local size, which accounts for the size of the work-group, the total number of work-groups and the number of iterations for each kernel. The number of iterations was decided in order to provide meaningful execution times, neither to low in which case system setup would represent high percentage of that time, nor too high. Dataset refers to the total number of data elements in the basic data structure of each application. Finally, global and local size notation shows the number of elements in each dimension: (1st Dimension, 2nd Dimension, 3rd Dimension) or (1st Dimension, 2nd Dimension) for a 2D decomposition or 1st Dimension without parenthesis for an 1D decomposition.
Table 1: Device query results

<table>
<thead>
<tr>
<th>cl_device_info</th>
<th>Description</th>
<th>Phi</th>
<th>Host</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL_DEVICE_NAME</td>
<td>Name of the device</td>
<td>Intel(R) Many Integrated Core Acceleration Card</td>
<td>Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz</td>
</tr>
<tr>
<td>CL_DEVICE_VERSION</td>
<td>OpenCL version</td>
<td>OpenCL 1.2</td>
<td>OpenCL 1.2</td>
</tr>
<tr>
<td>CL_DEVICE_MAX_COMPUTE_UNITS</td>
<td>The number of parallel compute cores on the OpenCL device</td>
<td>236</td>
<td>16</td>
</tr>
<tr>
<td>CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS</td>
<td>Maximum dimensions that specify the global and local work-item IDs</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>CL_DEVICE_MAX_WORK_ITEM_SIZES</td>
<td>Maximum number of work-items that can be specified in each dimension of the work-group</td>
<td>(8192, 8192, 8192)</td>
<td>(8192, 8192, 8192)</td>
</tr>
<tr>
<td>CL_DEVICE_MAX_WORK_GROUP_SIZE</td>
<td>Maximum number of work-items in a work-group</td>
<td>8192</td>
<td>8192</td>
</tr>
<tr>
<td>CL_DEVICE_GLOBAL_MEM_SIZE</td>
<td>Size of global device memory</td>
<td>67520 MB</td>
<td>6053 MB</td>
</tr>
<tr>
<td>CL_DEVICE_MAX_MEM_ALLOC_SIZE</td>
<td>Max size of memory object allocation</td>
<td>16880 MB</td>
<td>2017 MB</td>
</tr>
</tbody>
</table>
## Table 2: OpenCL and SYCL execution parameters

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Dataset</th>
<th>Kernel</th>
<th>Global Size</th>
<th>Local Size</th>
<th># Groups</th>
<th>Kernel Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>27stencil</td>
<td>10^6</td>
<td>“stencil” and “copy”</td>
<td>(37,37,37)</td>
<td>(37,1,1)</td>
<td>1369</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>10 * 10^6</td>
<td></td>
<td>(83,83,83)</td>
<td>(83,1,1)</td>
<td>6889</td>
<td></td>
</tr>
<tr>
<td></td>
<td>50 * 10^6</td>
<td></td>
<td>(144,144,144)</td>
<td>(144,1,1)</td>
<td>20736</td>
<td></td>
</tr>
<tr>
<td></td>
<td>100 * 10^6</td>
<td></td>
<td>(182,182,182)</td>
<td>(182,1,1)</td>
<td>33124</td>
<td></td>
</tr>
<tr>
<td>Himeno</td>
<td>16x16x16</td>
<td>“jacobi” and “copy”</td>
<td>(16,16,16)</td>
<td>(16,1,1)</td>
<td>256</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32x32x32</td>
<td></td>
<td>(32,32,32)</td>
<td>(32,1,1)</td>
<td>1024</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64x64x64</td>
<td></td>
<td>(64,64,64)</td>
<td>(64,1,1)</td>
<td>4096</td>
<td></td>
</tr>
<tr>
<td></td>
<td>128x128x128</td>
<td></td>
<td>(128,128,128)</td>
<td>(128,1,1)</td>
<td>16384</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16x16x16</td>
<td></td>
<td>16 * 16 * 16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32x32x32</td>
<td></td>
<td>32 * 32 * 32</td>
<td>1024</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64x64x64</td>
<td></td>
<td>64 * 64 * 64</td>
<td>4096</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td></td>
<td>128x128x128</td>
<td></td>
<td>128 * 128 * 128</td>
<td>4096</td>
<td>512</td>
<td></td>
</tr>
<tr>
<td>Le core</td>
<td>128x128</td>
<td>“le_step_x_mil” and “le_step_y_mil”</td>
<td>(128,128)</td>
<td>(32,1)</td>
<td>512</td>
<td></td>
</tr>
<tr>
<td></td>
<td>256x256</td>
<td></td>
<td>(256,256)</td>
<td>(64,1)</td>
<td>1024</td>
<td></td>
</tr>
<tr>
<td></td>
<td>512x512</td>
<td></td>
<td>(512,512)</td>
<td>(128,1)</td>
<td>2048</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1024x1024</td>
<td></td>
<td>(1024,1024)</td>
<td>(256,1)</td>
<td>4096</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2048x2048</td>
<td></td>
<td>(2048,2048)</td>
<td>(512,1)</td>
<td>8192</td>
<td></td>
</tr>
<tr>
<td>Mandel</td>
<td>1024x1024</td>
<td>“mandelGPU” (width*height/4)+1</td>
<td>270336</td>
<td>8192</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2048x2048</td>
<td></td>
<td>1056768</td>
<td>8192</td>
<td>129</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4096x4096</td>
<td></td>
<td>4202496</td>
<td>8192</td>
<td>513</td>
<td></td>
</tr>
<tr>
<td>Bitonic</td>
<td>2^22</td>
<td>“BitonicSort”</td>
<td>Depends on stage</td>
<td>Depends on stage</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2^23</td>
<td></td>
<td>Depends on stage</td>
<td>Depends on stage</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2^24</td>
<td></td>
<td>Depends on stage</td>
<td>Depends on stage</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2^25</td>
<td></td>
<td>Depends on stage</td>
<td>Depends on stage</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2^26</td>
<td></td>
<td>Depends on stage</td>
<td>Depends on stage</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GEMM</td>
<td>256</td>
<td>“gemm_nn”</td>
<td>(256,2)</td>
<td>(16,1)</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td></td>
<td>512</td>
<td></td>
<td>(512,4)</td>
<td>(16,1)</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td></td>
<td>(1024,8)</td>
<td>(16,1)</td>
<td>512</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2048</td>
<td></td>
<td>(2048,16)</td>
<td>(16,1)</td>
<td>2048</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td></td>
<td>(4096,32)</td>
<td>(16,1)</td>
<td>8192</td>
<td></td>
</tr>
</tbody>
</table>

For 27stencil, himeno and le core benchmarks the global size is equal to the total number of elements in the dataset. As far as the local size is concerned we let it up to the OpenCL runtime system to split the global work size into groups, by providing `NULL` in the `clEnqueueNDRangeKernel()` function that launches the kernel. The rest of the applications were already implemented in OpenCL and the global and local sizes were already provided. It should be noted that for application bitonic, global and local sizes are not constant during the execution and depend on the current stage of the sorting algorithm.

Syclone implementation of SYCL, to this point, does not provide the functionality of automatically deciding the local size. Instead, if the local size is not provided it is considered to be one, i.e. it creates global size work-groups of a single work-item. Therefore, for a more fair comparison, we explicitly defined local sizes in SYCL equal to the ones in OpenCL.
OpenMP unlike OpenCL and SYCL, expresses parallelism at a coarser level. It enables the developer to easily vary the number of threads, which is usually proportional to the number of cores in the system. OpenCL/SYCL on the other hand, express parallelism at its finest level and the number of threads is tied to the number of data elements. For the OpenMP versions we run the benchmarks using 60, 120, 180, 240, 300, 360, 420, 480 threads, which account to 1, 2, 3, 4, 5, 6, 7 and 8 threads per Xeon Phi physical core, for each one of the datasets. We gathered the execution times and selected the best one to compare with OpenCL and SYCL. The complete OpenMP execution results can be seen in Appendix A, section A.1.

Comparing OpenCL and SYCL applications that run with millions of threads with OpenMP applications that run with only hundreds may be considered unfair or out of topic comparison. However, as mentioned in section 2.3.3, when we launch an OpenCL kernel on Intel Xeon Phi, the driver creates as many threads as the number of work-groups and not as the total number of threads defined by the developer. This greatly decreases the total number of threads and in that sense, OpenCL, SYCL and OpenMP performance is more comparable.

Ultimately, we should notice that we used the exact same computational kernels for execution in each programming model. The kernels were the same as adapted from the original code either OpenACC or OpenCL. In all models we used regular parallel code and no attempt on aggressive, Xeon Phi specific optimizations, was made.

5.4 Experimental Results

Figures 14 through 19 depict execution times for the six benchmark applications for serial execution on the host, OpenCL, SYCL and OpenMP execution on Xeon Phi, as well as OpenCL execution on Xeon Phi, but with optimizations disabled when compiling the kernel.

The performance results show a substantial performance gap between SYCL and the other models, with SYCL being significantly slower in the majority of the applications. For 27stencil, himeno and le core, SYCL achieves quite lower performance than both OpenCL and OpenMP, whereas for Mandel SYCL achieves good performance, better than OpenMP, but worse than OpenCL. From the Intel code samples, in bitonic SYCL achieves the best performance, very close to OpenCL, whilst for GEMM SYCL presents the worst performance, alongside with OpenMP, with OpenCL having much lower execution time.
Figure 14: 27stencil execution times on Xeon Phi

Figure 15: Himeno execution times on Xeon Phi
Figure 16: Le core execution times on Xeon Phi

Figure 17: Mandel execution times on Phi
Figure 18: Bitonic execution times on Xeon Phi

Figure 19: GEMM execution times on Xeon Phi
More specifically, in Figure 14: 27stencil execution times on Xeon Phi it can be seen that for application 27stencil, both OpenCL and OpenMP provide speedup versus the serial execution on the host. OpenMP provides the best performance, around 8 seconds for the largest dataset, whereas OpenCL is at 37 and the host at 61 seconds. SYCL does unexpectedly bad, requiring 211 seconds to run the largest dataset.

The graph in Figure 15: Himeno execution times on Xeon Phi presents the performance results for application himeno. This application is characterised by high PCIe traffic as it involves memory transfers in each iteration of the outer loop, which greatly limits performance. Therefore, as expected, OpenMP that runs natively on the Xeon Phi achieves the best performance with 13 seconds for the largest dataset, whereas OpenCL requires 82 seconds. What is more, serial execution on the host provides better performance than parallel OpenCL, with 61 seconds. SYCL performance is very poor for this application. Indicative is that it requires over 1000 seconds to run the largest dataset.

Le core performance results are displayed in the graph in Figure 16: Le core execution times on Xeon Phi. OpenMP and OpenCL provide almost similar performance, achieving an 8x speedup over the serial version for the largest dataset. SYCL also provides a 1.6x speedup over the host execution for the largest dataset, but again its performance can be characterised low compared to the other models. It also provides relatively poor performance over the smaller data sets.

For Mandel (Figure 17: Mandel execution times on Phi) it is interesting to notice that SYCL achieves better performance than OpenMP. In this application OpenCL achieves by far the best performance, requiring 11 seconds to compute the Mandelbrot set for an image of size 4096x4096, achieving a 47x speedup over the serial host execution. OpenMP and SYCL require 147 and 93 seconds respectively.

Bitonic is an application were the computational kernel lies inside a double loop. If we count the outer for loop that we have added to execute the kernel many times over, this accounts for large number of kernel invocations. The kernel contains mainly conditionals and comparative operations with relatively small amounts of computation. However, as it can be seen in Figure 18: Bitonic execution times on Xeon Phi, both SYCL and OpenCL achieve very good, almost identical performance, providing 20x speedup over the serial version for a dataset of $2^{24}$ elements. OpenMP achieves also speedup over the host, but it is still far behind OpenCL and SYCL. We should notice that the host execution time for the the $2^{25}$ dataset is omitted for the sake of visibility as it exceeds 1000 seconds.

For GEMM (Figure 19: GEMM execution times on Xeon Phi), both SYCL and OpenMP yield poor performance, requiring around 500 and 400 seconds respectively to multiply matrices of order 2048. OpenCL is extremely faster, requiring only 15 seconds for the same task.
5.5 Discussion

As an overall note, SYCL performs considerably worse than OpenCL in five out of the six benchmarks. As SYCL is built on top of OpenCL, with both models targeting the same intermediate representation, we were expecting them to present similar performance. As a next step, we tried to identify and provide an explanation for the observed performance gap.

To get a better perspective on SYCL performance, we also run three of the tutorial applications provided with Syclone software. The three applications are:

- **vadd_chain**: Includes a two-way vector addition kernel and adds five vectors by invoking the kernel three times.
- **vadd_abc**: Adds three vectors using a single three-way vector addition kernel.
- **matmul**: Multiplies two 2D matrices, using a naïve matrix multiplication algorithm.

In all cases, we have wrapped the computational part inside a for loop to increase device execution time. All applications were already implemented in both OpenCL and SYCL and the addition of the outer for loops was our only modification. The results are depicted in the graphs in figures 20, 21 and 22. As it can be noticed in all three applications SYCL performs worse than OpenCL.

![Figure 20: vadd_chain execution times on Xeon Phi](image)
Figure 21: vadd_abc execution times on Xeon Phi

Figure 22: matmul execution times on Xeon Phi
Intel Xeon Phi coprocessor relies heavily on vectorization to bring high performance, where the device compiler is responsible to produce vectorized machine code. Our first assumption was that, up to this point, Syclone’s device compiler might not apply optimizations, including vectorization. To test this assumption, we re-run the OpenCL versions of the six benchmarks and the three tutorial applications, but this time including the flag “-cl-opt-disable”, in the kernel build command, which instructs the compiler to not apply any optimization, including vectorization.

The results can be seen in the form of the bar under the name “OpenCL opt-dis Phi” for each application in the initial graphs, in figures 14 through 19. Unfortunately, the results did not provide any kind of specific pattern that would be useful to explain the cause of SYCL poor performance. For 27stencil, himeno and le core OpenCL with and without optimization times are considered very similar, indicating that SYCL does not perform poor in this case due to SYCL device compiler not applying vectorization or other optimizations. Regarding mandel, OpenCL optimization-free version and SYCL present almost the same performance, whilst for bitonic and GEMM OpenCL without optimizations performs considerably worse than SYCL.

All the applications we have tested so far include relatively small kernels in terms of single kernel execution time that are invoked repetitively. Our thought was that the performance gap might be derived from the overhead of the repetitive kernel calls. Our next assumption regarding of the root of the performance gap includes two components, either it is derived from high kernel launch overhead, or the Syclone runtime system might be transferring data back and forth between the host and the device, during the repetitive kernel calls. The first part is highly improbable as OpenCL and SYCL execution times differ up to 600 seconds in some cases, which makes it difficult to be originating solely from kernel launch overhead.

To investigate the second part of the scenario we re-run the OpenCL, SYCL and OpenMP versions of the six benchmarks using the host as the device. The host comprises of two 8-core Intel Xeon processors, with a total of 16 threads and 64 GB of main memory. The results can be seen in the graphs in figures 23 through 28. The graphs also include the SYCL version on Xeon Phi for comparison purposes. Once again, for the OpenMP versions we measured execution time for variable thread count, analytical results of which can be seen in Appendix A, section A.2. In the graphs we present the best OpenMP execution times.

For one more time the results did not reveal a clear pattern that would allow us to safely state that the assumption is true. It is noticeable that for four out of the six applications the SYCL Xeon version presents a 2x speedup over the SYCL Phi version. However, it is still behind the OpenCL version.
Figure 23: 27stencil execution times on Xeon

Figure 24: Himeno execution times on Xeon
Figure 25: Le core execution times on Xeon

Figure 26: Mandel execution times on Xeon
Figure 27: Bitonic execution times on Xeon

Figure 28: GEMM execution times on Xeon
As a final step we tried to identify what happens in the case where there is a single large kernel that is run only once and not a small kernel running multiple times. To do that, we modified a vector addition kernel by adding a for loop to increase kernel execution time. The kernel can be seen in the code snippet below:

```c
__kernel void large_kernel(__global int *a, __global int *b, __global int *c) {
  int i = get_global_id(0);
  int j;
  for (j = 0; j < 10000; j++) {
    c[i] = a[i] + b[i];
  }
}
```

The results of running the above kernel in both OpenCL and SYCL are displayed in the graph in Figure 29: large_kernel execution times on Xeon Phi. The graph also show execution times for the large_kernel OpenCL version compiled without optimizations, which present almost identical performance with the SYCL version. However, the performance gap is present for one more time, enhancing the point that the repetitive kernel calls or some kind of extra data transfers is not the only cause of poor performance.

![Figure 29: large_kernel execution times on Xeon Phi](image-url)
Ultimately, to our understanding the cause of SYCL poor performance as compared with plain OpenCL may be originating from two things. Either the device compiler produces non efficient code or the delay comes from the runtime system, or maybe both. However, none of our experiments was enough to provide a reasonable explanation. Doing so requires running more experiments, using applications with specific characteristics, such as memory bound or compute bound applications. An extensive code profiling might also have been very useful. However, to the best of our knowledge, up to this point there are no tools available that would allow us to profile SYCL device code. We should also acknowledge that we are working on a very early version of Syclone which is yet unreleased and might not have all features implemented yet.
Chapter 6

Conclusion

Developers’ long requesting demand for a higher-level parallel programming model based on OpenCL, is finally being answered. The Khronos group on the 19th of March, 2014 announced the release of the provisional specification of the OpenCL SYCL programming model. SYCL, like OpenCL, is an open source, royalty free, cross platform parallel programming framework with wide industry support. It provides an abstraction layer on top of OpenCL that enables single source development and combines the ease of use and flexibility of C++ with the efficiency and robustness of OpenCL. SYCL status is still in the phase of gathering community feedback. The next step is the release of the full specification and subsequently vendors will be able to release their implementations.

For the purposes of this project we acquired an initial implementation of SYCL, named “Syclone”, from Codeplay Ltd, an Edinburgh based company specialising in compiler development. Our main goal, was to assess this new programming paradigm and provide valuable feedback that would be useful to both Codeplay and the Khronos group. To accomplish that goal, we compared SYCL to another two established parallel programming models, OpenCL and OpenMP, in both terms of performance and programmability. More specifically, we developed six scientific applications in all three models and run them on an Intel Xeon Phi platform to compare performance.

The results have shown that SYCL greatly simplifies the development process compared to OpenCL. Using SYCL, a developer can achieve the same outcome with OpenCL in a considerably smaller program in terms of lines of code that is also more coherent, robust and easier to maintain. SYCL achieves these great programmability results by firstly, providing single source development. This means that both host and device code are in the same file, eliminating meaningless complexities of splitting the code. What is more, SYCL enables the use of popular C++ features, including classes, inheritance, templating and operator overloading inside kernel development. Therefore, this facilitates the creation of generic and robust code and makes SYCL ideal for libraries and middleware development. Finally, SYCL API encapsulates most of the complexity encountered in OpenCL. In particular, the SYCL runtime system handles the underlying resources, automatically sets the execution environment (platforms, devices, queues) and handles data transfers between the host and the device. Indicative is that the minimum object required to run a SYCL application in parallel is a queue,
whilst in OpenCL the developer needs to follow a long list of actions including selection of platform and device, creation of a queue, a program object, a kernel object. OpenMP, on the other hand, is undisputedly the simplest parallel programming model. However, until very recently it was only supported on shared memory systems, with accelerator support being added in the latest 4.0 version.

The benchmarking results revealed a performance gap between SYCL and OpenCL/OpenMP. More specifically SYCL performed considerably worse from both OpenCL and OpenMP in four out the six applications, better than OpenMP, but worse than OpenCL in one of them and similarly to OpenCL, better than OpenMP in the last. As a next step in the project we set out to explore the cause of the performance gap. We thought of two possible reasons for SYCL poor performance:

1. Syclone device compiler does not apply optimizations: Apart from massive parallelism, vectorization is the most critical step towards achieving high performance on an Intel Xeon Phi platform. The device compiler that Intel provides within the OpenCL SDK is known to apply vectorization on the kernel code. Thus, if Syclone device compiler do not apply vectorization, this perfectly explains the poor performance.

2. Repetitive kernel calls: In all six applications the kernels are launched repetitively inside for loops. However, data transfers occur only at the beginning (data are transferred from the host to the device) and the end (data are transferred from the device to the host) of the simulation. The above pattern takes place in all applications except Himeno, where data transfers occur in each iteration. In SYCL, data transfers are handled by the runtime system and the developer does not have any control on the procedure. Moreover, he cannot assume at which exact point of point the execution they take place. Our thought was that in the cases of SYCL poor performance the runtime may be moving data back and forth, consuming a lot of time.

To examine the first possible reason of poor performance, compiler optimizations, we re-run all the OpenCL versions of the applications using the flag “-cl-opt-disable” when building the kernels. This device compiler option turns off all optimizations, including vectorization. We expected that if this was the cause of the performance gap, OpenCL execution times would turn towards SYCL times. However, the results varied significantly and did not provide a clear pattern that would enable us to undoubtedly state that the first assumption is true.

Regarding our second assumption, we re-run the OpenCL, SYCL and OpenMP versions of the six benchmarks, using the host’s Intel Xeon processor as the device. Thereby, there would be no need to transfer data over the PCIe bus, as the data already reside on the host. The results in most cases showed a 2x performance increase for the SYCL Xeon version over the SYCL Phi version. However, SYCL performance was still worse than OpenCL.

As a final note, computer hardware is evolving rapidly leaving computer software behind. Indicative is that today there are only very few applications that are able to fully
exploit the world’s largest systems. Furthermore, accelerators represent higher percentage of the high performance computing ecosystem year after year, establishing their position. Therefore, the need for new universal parallel programming models that is characterised by ease of use, efficiency and can target a wide range of devices is bigger than ever and SYCL can be a step towards that direction.
Appendix A

Extensive OpenMP Results

A.1 OpenMP Xeon Phi Results

The following tables present the analytical performance results for the OpenMP versions of the six benchmarks on the Xeon Phi, using variable thread count. For each dataset the best time can be seen in bold annotation. All times are in seconds.

Table 3: 27stencil OpenMP execution times on Xeon Phi

<table>
<thead>
<tr>
<th># threads</th>
<th>1 Million</th>
<th>10 Million</th>
<th>50 Million</th>
<th>100 Million</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>0.246712</td>
<td>1.716667</td>
<td>7.1491</td>
<td>14.334294</td>
</tr>
<tr>
<td>120</td>
<td>0.190757</td>
<td>1.157204</td>
<td>4.96833</td>
<td>10.3029</td>
</tr>
<tr>
<td>180</td>
<td>0.245392</td>
<td>1.007359</td>
<td>4.655216</td>
<td>9.157185</td>
</tr>
<tr>
<td>240</td>
<td>0.211821</td>
<td><strong>0.963087</strong></td>
<td><strong>4.070962</strong></td>
<td><strong>8.412051</strong></td>
</tr>
<tr>
<td>300</td>
<td>2.514795</td>
<td>3.584067</td>
<td>8.198658</td>
<td>13.102889</td>
</tr>
<tr>
<td>360</td>
<td>3.189082</td>
<td>4.0784</td>
<td>8.442022</td>
<td>13.106068</td>
</tr>
<tr>
<td>420</td>
<td>3.584495</td>
<td>4.490375</td>
<td>8.705486</td>
<td>13.208967</td>
</tr>
<tr>
<td>480</td>
<td>3.945478</td>
<td>4.770374</td>
<td>9.208111</td>
<td>13.229945</td>
</tr>
</tbody>
</table>

Table 4: Himeno OpenMP execution times on Xeon Phi

<table>
<thead>
<tr>
<th># threads</th>
<th>16x16x16</th>
<th>32x32x32</th>
<th>64x64x64</th>
<th>128x128x128</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td><strong>0.121038</strong></td>
<td>0.354723</td>
<td>2.8067</td>
<td>16.681333</td>
</tr>
<tr>
<td>120</td>
<td>0.138536</td>
<td><strong>0.299815</strong></td>
<td>2.053507</td>
<td>16.384666</td>
</tr>
<tr>
<td>180</td>
<td>0.15207</td>
<td>0.306197</td>
<td>1.878494</td>
<td>14.036642</td>
</tr>
<tr>
<td>240</td>
<td>0.173487</td>
<td>0.300035</td>
<td><strong>1.690149</strong></td>
<td><strong>14.005947</strong></td>
</tr>
<tr>
<td>300</td>
<td>4.168159</td>
<td>4.180259</td>
<td>6.288005</td>
<td>19.284706</td>
</tr>
<tr>
<td>360</td>
<td>5.040088</td>
<td>5.158422</td>
<td>6.657049</td>
<td>19.001321</td>
</tr>
<tr>
<td>420</td>
<td>5.620751</td>
<td>5.793346</td>
<td>7.147279</td>
<td>19.630662</td>
</tr>
<tr>
<td>480</td>
<td>6.060212</td>
<td>6.380678</td>
<td>7.740195</td>
<td>19.345854</td>
</tr>
</tbody>
</table>
Table 5: Le core OpenMP execution times on Xeon Phi

<table>
<thead>
<tr>
<th># threads</th>
<th>Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>0.180204</td>
</tr>
<tr>
<td>120</td>
<td>0.129703</td>
</tr>
<tr>
<td>180</td>
<td>0.106574</td>
</tr>
<tr>
<td>240</td>
<td>0.097563</td>
</tr>
<tr>
<td>300</td>
<td>0.566911</td>
</tr>
<tr>
<td>360</td>
<td>0.687541</td>
</tr>
<tr>
<td>420</td>
<td>0.790597</td>
</tr>
<tr>
<td>480</td>
<td>0.798472</td>
</tr>
</tbody>
</table>

Table 6: Mandel OpenMP execution times on Xeon Phi

<table>
<thead>
<tr>
<th># threads</th>
<th>Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>1024x1024</td>
</tr>
<tr>
<td>120</td>
<td>25.438148</td>
</tr>
<tr>
<td>180</td>
<td>16.75055</td>
</tr>
<tr>
<td>240</td>
<td>12.524034</td>
</tr>
<tr>
<td>300</td>
<td>9.240479</td>
</tr>
<tr>
<td>360</td>
<td>12.54071</td>
</tr>
<tr>
<td>420</td>
<td>10.6452</td>
</tr>
<tr>
<td>480</td>
<td>9.232847</td>
</tr>
</tbody>
</table>

Table 7: GEMM OpenMP execution times on Xeon Phi

<table>
<thead>
<tr>
<th># threads</th>
<th>Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>256x256</td>
</tr>
<tr>
<td>120</td>
<td>1.30367</td>
</tr>
<tr>
<td>180</td>
<td>0.938781</td>
</tr>
<tr>
<td>240</td>
<td>1.82067</td>
</tr>
<tr>
<td>300</td>
<td>1.72222</td>
</tr>
<tr>
<td>360</td>
<td>1.36451</td>
</tr>
<tr>
<td>420</td>
<td>1.87099</td>
</tr>
<tr>
<td>480</td>
<td>1.87254</td>
</tr>
</tbody>
</table>
Table 8: Bitonic OpenMP execution times on Xeon Phi

<table>
<thead>
<tr>
<th># threads</th>
<th>2^22</th>
<th>2^23</th>
<th>2^24</th>
<th>2^25</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>27.158912</td>
<td>60.677154</td>
<td>130.799569</td>
<td>280.969521</td>
</tr>
<tr>
<td>120</td>
<td>18.059581</td>
<td>41.486419</td>
<td>89.329712</td>
<td>191.475428</td>
</tr>
<tr>
<td>180</td>
<td>17.333902</td>
<td>39.207458</td>
<td>82.183318</td>
<td>179.927159</td>
</tr>
<tr>
<td>240</td>
<td><strong>17.087016</strong></td>
<td>52.973148</td>
<td>97.508712</td>
<td>224.411629</td>
</tr>
<tr>
<td>300</td>
<td>33.271527</td>
<td>60.098626</td>
<td>112.238299</td>
<td>222.515287</td>
</tr>
<tr>
<td>360</td>
<td>33.839237</td>
<td>58.473465</td>
<td>106.688253</td>
<td>210.050429</td>
</tr>
<tr>
<td>420</td>
<td>36.23256</td>
<td>59.162506</td>
<td>108.589599</td>
<td>212.334539</td>
</tr>
<tr>
<td>480</td>
<td>38.132128</td>
<td>62.167116</td>
<td>119.823758</td>
<td>249.356161</td>
</tr>
</tbody>
</table>

A.2 OpenMP Xeon Results

The following tables present the analytical OpenMP performance results of the six benchmarks on the host (Intel Xeon processor). We measured times, using different thread count, including 8, 16, 24, 32, 40, 48, 56, 64, 72, 80 threads. All times presented below are in seconds. The best time for each dataset is annotated in bold.

Table 9: 27stencil OpenMP execution times on Xeon

<table>
<thead>
<tr>
<th># threads</th>
<th>1 Million</th>
<th>10 Million</th>
<th>50 Million</th>
<th>100 Million</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td><strong>0.3336</strong></td>
<td>3.381723</td>
<td>17.286541</td>
<td>34.858993</td>
</tr>
<tr>
<td>16</td>
<td>0.516298</td>
<td>3.7512</td>
<td>18.547328</td>
<td>37.561191</td>
</tr>
<tr>
<td>24</td>
<td>0.792357</td>
<td>4.207041</td>
<td>16.340658</td>
<td>31.977807</td>
</tr>
<tr>
<td>32</td>
<td>0.845515</td>
<td>4.405095</td>
<td>17.061885</td>
<td>32.224498</td>
</tr>
<tr>
<td>40</td>
<td>0.82312</td>
<td>4.04158</td>
<td>14.142584</td>
<td>28.53173</td>
</tr>
<tr>
<td>48</td>
<td>0.804503</td>
<td>3.639151</td>
<td>14.927282</td>
<td>29.359232</td>
</tr>
<tr>
<td>56</td>
<td>0.76758</td>
<td>3.755854</td>
<td>13.568197</td>
<td>27.174486</td>
</tr>
<tr>
<td>64</td>
<td>0.761364</td>
<td>3.778826</td>
<td>13.982186</td>
<td>27.848069</td>
</tr>
<tr>
<td>72</td>
<td>0.744311</td>
<td><strong>3.626949</strong></td>
<td><strong>13.206505</strong></td>
<td><strong>26.455104</strong></td>
</tr>
<tr>
<td>80</td>
<td>0.724376</td>
<td>3.848608</td>
<td>13.398724</td>
<td>26.684306</td>
</tr>
</tbody>
</table>
Table 10: Himeno OpenMP execution times on Xeon

<table>
<thead>
<tr>
<th># threads</th>
<th>16x16x16</th>
<th>32x32x32</th>
<th>64x64x64</th>
<th>128x128x128</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0.054982</td>
<td>0.263672</td>
<td>1.827893</td>
<td>12.525246</td>
</tr>
<tr>
<td>16</td>
<td>0.268869</td>
<td>0.455144</td>
<td>2.536708</td>
<td>16.980309</td>
</tr>
<tr>
<td>24</td>
<td>0.265033</td>
<td>0.607707</td>
<td>3.001627</td>
<td>13.572655</td>
</tr>
<tr>
<td>32</td>
<td>0.316268</td>
<td>0.672865</td>
<td>3.04591</td>
<td>12.883406</td>
</tr>
<tr>
<td>40</td>
<td>0.363159</td>
<td>0.702934</td>
<td>2.947082</td>
<td>12.280958</td>
</tr>
<tr>
<td>48</td>
<td>0.411064</td>
<td>0.74535</td>
<td>2.981425</td>
<td>11.865853</td>
</tr>
<tr>
<td>56</td>
<td>0.414642</td>
<td>0.734993</td>
<td>2.892946</td>
<td>12.138547</td>
</tr>
<tr>
<td>64</td>
<td>0.445492</td>
<td>0.719703</td>
<td>2.900917</td>
<td>11.61535</td>
</tr>
<tr>
<td>72</td>
<td>0.494695</td>
<td>0.8072</td>
<td>2.858832</td>
<td>11.658453</td>
</tr>
<tr>
<td>80</td>
<td>0.514727</td>
<td>0.786529</td>
<td>2.872545</td>
<td>11.54453</td>
</tr>
</tbody>
</table>

Table 11: Le core OpenMP execution times on Xeon

<table>
<thead>
<tr>
<th># threads</th>
<th>128x128</th>
<th>256x256</th>
<th>512x512</th>
<th>1024x1024</th>
<th>2048x2048</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0.120302</td>
<td>0.475957</td>
<td>1.606691</td>
<td>6.414355</td>
<td>24.788495</td>
</tr>
<tr>
<td>16</td>
<td>0.274598</td>
<td>0.542584</td>
<td>1.712021</td>
<td>6.471989</td>
<td>25.870437</td>
</tr>
<tr>
<td>24</td>
<td>0.226479</td>
<td>0.492938</td>
<td>1.654961</td>
<td>5.696364</td>
<td>20.430861</td>
</tr>
<tr>
<td>32</td>
<td>0.222056</td>
<td>0.573655</td>
<td>1.591989</td>
<td>5.906038</td>
<td>16.469321</td>
</tr>
<tr>
<td>40</td>
<td>0.230934</td>
<td>0.623431</td>
<td>1.376638</td>
<td>5.081933</td>
<td>18.830661</td>
</tr>
<tr>
<td>48</td>
<td>0.231649</td>
<td>0.610513</td>
<td>1.37365</td>
<td>5.289211</td>
<td>18.501469</td>
</tr>
<tr>
<td>56</td>
<td>0.222519</td>
<td>0.616587</td>
<td>1.298519</td>
<td>4.960709</td>
<td>18.14554</td>
</tr>
<tr>
<td>64</td>
<td>0.230255</td>
<td>0.601246</td>
<td>1.291549</td>
<td>5.06745</td>
<td>14.934245</td>
</tr>
<tr>
<td>72</td>
<td>0.21729</td>
<td>0.600733</td>
<td>1.241931</td>
<td>4.863708</td>
<td>17.738712</td>
</tr>
<tr>
<td>80</td>
<td>0.224972</td>
<td>0.611048</td>
<td>1.27284</td>
<td>4.765579</td>
<td>17.508612</td>
</tr>
</tbody>
</table>
Table 12: Mandel OpenMP execution times on Xeon

<table>
<thead>
<tr>
<th># threads</th>
<th>1024x1024</th>
<th>2048x2048</th>
<th>4096x4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>36.738403</td>
<td>147.367503</td>
<td>586.662228</td>
</tr>
<tr>
<td>16</td>
<td>27.609202</td>
<td>94.98015</td>
<td>384.25451</td>
</tr>
<tr>
<td>24</td>
<td>21.978479</td>
<td>83.38148</td>
<td>334.96067</td>
</tr>
<tr>
<td>32</td>
<td>20.62549</td>
<td>79.16168</td>
<td>321.452999</td>
</tr>
<tr>
<td>40</td>
<td>19.838422</td>
<td>80.242686</td>
<td>316.469431</td>
</tr>
<tr>
<td>48</td>
<td>19.948773</td>
<td>79.342763</td>
<td>312.648702</td>
</tr>
<tr>
<td>56</td>
<td>19.803658</td>
<td>78.149376</td>
<td>314.314858</td>
</tr>
<tr>
<td>64</td>
<td>19.636988</td>
<td>78.442313</td>
<td>311.60209</td>
</tr>
<tr>
<td>72</td>
<td><strong>19.438734</strong></td>
<td><strong>77.723865</strong></td>
<td><strong>309.41659</strong></td>
</tr>
<tr>
<td>80</td>
<td>19.48572</td>
<td>77.743049</td>
<td>312.278873</td>
</tr>
</tbody>
</table>

Table 13: GEMM OpenMP execution times on Xeon

<table>
<thead>
<tr>
<th># threads</th>
<th>256x256</th>
<th>512x512</th>
<th>1024x1024</th>
<th>2048x2048</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0.554275</td>
<td>5.12876</td>
<td>35.5089</td>
<td>296.135</td>
</tr>
<tr>
<td>16</td>
<td>0.67974</td>
<td>5.27427</td>
<td>36.7296</td>
<td><strong>163.119</strong></td>
</tr>
<tr>
<td>24</td>
<td>0.635606</td>
<td><strong>3.73698</strong></td>
<td>21.977</td>
<td>188.902</td>
</tr>
<tr>
<td>32</td>
<td>0.572714</td>
<td>4.05075</td>
<td>21.4478</td>
<td>183.952</td>
</tr>
<tr>
<td>40</td>
<td>0.512486</td>
<td>4.1251</td>
<td>20.9166</td>
<td>180.415</td>
</tr>
<tr>
<td>48</td>
<td>0.496692</td>
<td>4.15671</td>
<td>20.7462</td>
<td>178.914</td>
</tr>
<tr>
<td>56</td>
<td><strong>0.47677</strong></td>
<td>3.98925</td>
<td><strong>20.3724</strong></td>
<td>178.521</td>
</tr>
<tr>
<td>64</td>
<td>0.499431</td>
<td>3.96861</td>
<td>20.4521</td>
<td>170.553</td>
</tr>
<tr>
<td>72</td>
<td>0.781874</td>
<td>6.61258</td>
<td>34.6069</td>
<td>245.358</td>
</tr>
<tr>
<td>80</td>
<td>0.733785</td>
<td>5.92549</td>
<td>34.7614</td>
<td>239.795</td>
</tr>
</tbody>
</table>
Table 14: Bitonic OpenMP execution times on Xeon

<table>
<thead>
<tr>
<th># threads</th>
<th>Dataset</th>
<th>$2^{22}$</th>
<th>$2^{23}$</th>
<th>$2^{24}$</th>
<th>$2^{25}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td></td>
<td>15.226551</td>
<td>33.768459</td>
<td>80.249231</td>
<td>177.055403</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>17.43146</td>
<td>38.901127</td>
<td>100.448109</td>
<td>228.213792</td>
</tr>
<tr>
<td>24</td>
<td></td>
<td>18.612999</td>
<td>37.410045</td>
<td>76.689853</td>
<td>174.967345</td>
</tr>
<tr>
<td>32</td>
<td></td>
<td>23.951118</td>
<td>40.340223</td>
<td>79.729312</td>
<td>171.644763</td>
</tr>
<tr>
<td>40</td>
<td></td>
<td>21.800153</td>
<td>34.061289</td>
<td>69.772942</td>
<td>161.300684</td>
</tr>
<tr>
<td>48</td>
<td></td>
<td>22.597314</td>
<td>32.480081</td>
<td>69.589777</td>
<td>152.748749</td>
</tr>
<tr>
<td>56</td>
<td></td>
<td>23.181088</td>
<td>32.137482</td>
<td>67.38165</td>
<td>144.146781</td>
</tr>
<tr>
<td>64</td>
<td></td>
<td>22.304028</td>
<td>31.31997</td>
<td>67.159011</td>
<td>151.353761</td>
</tr>
<tr>
<td>72</td>
<td></td>
<td>23.288869</td>
<td>31.360909</td>
<td>66.360724</td>
<td>148.837407</td>
</tr>
<tr>
<td>80</td>
<td></td>
<td>22.994601</td>
<td>31.747114</td>
<td>65.671813</td>
<td>145.892195</td>
</tr>
</tbody>
</table>
References


