Dissertation Type: Research

DEPARTMENT OF COMPUTER SCIENCE

Exploring the performance portability of the SYCL language

William Hawkins

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

Wednesday 29th July, 2020

Contents

List of Figures

List of Listings

Executive Summary

High Performance Computing (HPC) is an extremely important subject as it underpins the running of all scientific codes and is widely used in industry. By improving HPC, we are benefiting a large range of sectors. SYCL is a new language in High Performance Computing which aims to solve issues faced by existing languages and to make programs performance portable. This means that they can perform well on a wide range of machines, making them more future proof.

My research hypothesis is that implementations using the SYCL programming model will have a performance portability score similar to that of implementations in OpenCL. To evaluate this, the project aimed to quantitatively assess the performance portability of the new programming language SYCL and to compare and contrast its different implementations. To do this the following objectives were met: to port a computational fluid dynamics (CFD) code which uses the Lattice Boltzmann methods (LBM) to the language SYCL; to port Tealeaf [22], a mini-app representing iterative sparse linear solvers, to the language SYCL and to rigorously test both Tealeaf and the LBM codes against their OpenCL counterparts to measure the overheads of using the SYCL language. I discovered that with large input sizes (more reflective of real world usage), the overheads are minimal. At small input sizes there are large overheads, largely due to the in-order versus out-of-order queuing mechanism. This was something that the Bristol HPC group was unaware of and would not have expected to be so detrimental to performance. I performed 477 tests, having a combined total runtime of over 25 hours. Using these results, I applied the performance portability metrics defined by Pennycook et al [25] to measure the performance portability of SYCL across a diverse range of architectures. I have shown that SYCL can achieve high performance portability scores, similar to OpenCL, but as the program size and complexity grows, the performance and thus the performance portability of SYCL falls. From these results, I have concluded that to get the full benefits of the SYCL language, hipSYCL plus either LLVM or ComputeCPP needs to be used. I believe that this project has shown that SYCL is a promising new language, which is already achieving good results. I predict that within five years SYCL will be as performance portable, if not more performance portable than OpenCL for all sizes and complexities of program.

In addition to the results presented in this document, the project has several outputs through which it will have impact:

- A Lattice Boltzman Code in SYCL
- A SYCL version of the TeaLeaf mini-app
- A guide, with examples, to aid other developers to port their codes to SYCL

These are freely available from: https://github.com/WSJHawkins/ExploringSycl. These outputs can be used in further research and to drive forward the SYCL language. The porting guide will facilitate developers to start using SYCL.

Supporting Technologies

- I used the Bristol HPC Zoo extensively. This contains a diverse range of hardware and software and was where all my code was tested and run.
- I used Codeplay's ComputeCPP compiler to compile my SYCL code.
- I used the hipSYCL compiler to compile my SYCL code.
- I used Intels's LLVM SYCL compiler to compile my SYCL code.
- I used Intels's OpenCL Intercept Layer to debug my code and produce the traces that are shown in my results.
- I used Intels's DevCloud. This provided more hardware and software for me to test on.

Notation and Acronyms

AMD	:	Advanced Micro Devices - A semiconductor company
API	:	Application Programming Interface
ARM	:	A British global semiconductor and software design company
CFD	:	Computational Fluid Dynamics
ComputeCPP	:	An implementation of the SYCL programming language
CPU	:	Central Processing Unit
CUDA	:	A parallel computing platform and programming model developed by Nvidia
FLOPS	:	Floating Point Operations Per Second
GPU	:	Graphics Processing Unit
hipSYCL	:	An implementation of the SYCL programming language
HPC	:	High Performance Computing
HPC Zoo	:	A small cluster containing many different accelerator technologies
IWOCL	:	International Workshop on OpenCL
Kokkos	:	A HPC programming language from Sandia National Laboratories
LBM	:	Lattice Boltzmann
LLVM SYCL	:	An implementation of the SYCL programming language
NEC	:	A Japanese multinational information technology and electronics company
Nvidia	:	An American technology company specialising in GPU design
OpenMP	:	A HPC programming model
OpenCL	:	Open Computing Language - A programming language used in HPC
PC	:	Personal Computer
ROCm	:	A universal platform for gpu-accelerated computing from AMD
SYCL	:	A higher-level programming model for OpenCL
SYCLcon	:	A SYCL Conference which is linked with IWOCL
TeaLeaf	:	A mini-app that solves the linear heat conduction equation on a spatially decomposed regularly grid using a 5 point stencil with implicit solver
UK MAC	:	UK Mini-App Consortium
Φ	:	Performance Portability
$\mid H \mid$	:	Size of the set of platforms used in Performance Portability
$e(a,p)$	:	Efficiency of application a on platform p

Acknowledgements

I would like to thank my supervisor, Simon McIntosh Smith, who throughout the project provided direction when needed and ensured that I had access to the required resources for progression. I would also like to thank Tom Deakin for sharing his extensive knowledge on SYCL and TeaLeaf. Thanks must also go to Ben Ashbaugh, from Intel, who helped look into the overheads I experienced in SYCL and to Tristan Warren for helping proof read this work. Finally, I would like to thank my family for their continued support in all my endeavours.

Chapter 1

Contextual Background

1.1 High Performance Computing

High Performance Computing (HPC) is the development of technologies that allow people to run codes quickly and efficiently, often on large specialist machines called supercomputers. The use of HPC varies widely, as shown by the 'Application Area Performance Share' on the Top500 Website [6], which tracks the 500 most powerful supercomputers.

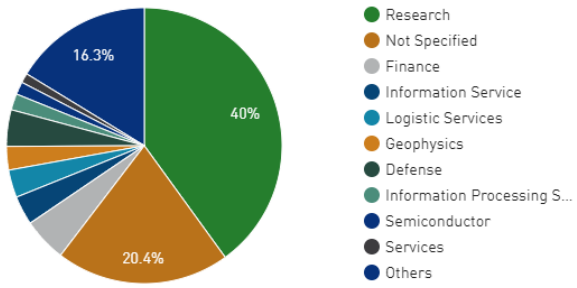

Figure 1.1: Application Area Performance Share, June 2010 [6]

It identified that there were 31 sectors using the top 500 machines plus those not specified. This illustrates the huge reach and therefore importance of High Performance Computing. The reason the above figure is from 2010 and not a more recent graphic, is due to the fact that most of the Top500 machines have become more generalised and thus no longer specify their main workloads, 93% of the application area performance share was unspecified in the November 2019 data [6]. By researching and improving HPC methods, an even wider range of sectors are being affected and improved.

The need for High Performance Computing is growing due to the increasing parallelism present in hardware and software. Originally, improvements in computation came from Moore's Law [24], where the number of transistors in a dense integrated circuit would double every two years. Before 2005, these transistors were used to increase the clock speed, however, this also meant that the power consumption grew at an exponential rate, leading to overheating issues. Since 2005 the performance has been increasing by utilising concurrency [26]. Waldrop points out that although the growth from the number of transistors doubling has come to an end, the industry is looking for new functionality to continue improving the performance of their chips [27]. This trend in increasing the number of cores in CPU's is still occurring with AMD releasing their '64 Core Rome CPU' in 2019. These multiple cores are allowing a greater performance level but only if the programs using them can exploit their parallelism. On top of the multiple cores many CPUs have wide SIMD (Single Instruction, Multiple Data) lanes, requiring further parallelism to exploit.

This growth in the consumer section has led to changes in the HPC sector. The original HPC machines were expensive bespoke machines with a single vendor and proprietary software. If you wanted to move

between machines you may have had to completely rewrite your code just to get it to work with the different hardware and software stack. As consumer PCs improved the HPC market was able to build computers out of commodity hardware. This was done by linking them together with Ethernet networking which had been developed and had become cheaply available. The free operating system Linux, allowed for a standardised software across the machines and the industry had multiple vendors to choose from. The fact that these computers were being produced in their millions for retail, meant the price was much lower than the previous bespoke machines. As the PCs gained 64 bit architecture, hardware floating point support and hardware error correction, this hardware became increasingly interesting to HPC.

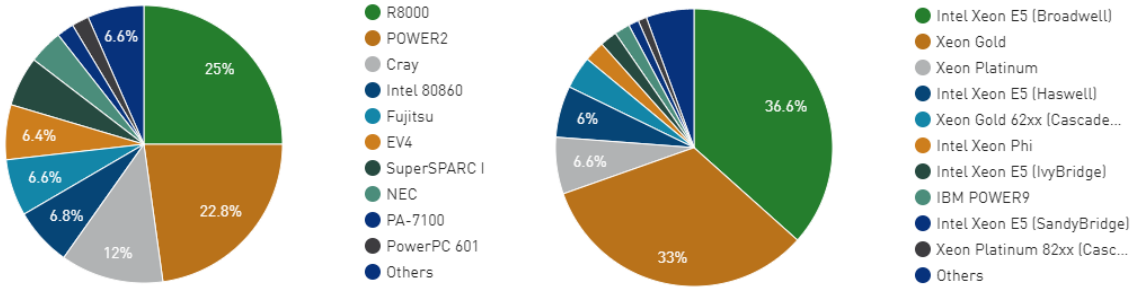

Figure 1.2: Processor Generation System Share: Left - 1995, Right - 2019 [6]

Comparing 1995 with 2019, it can be seen that the Top 500 machines have left behind the bespoke proprietary world and progressed on to harnessing today's commodity hardware.

Another interesting development has been the increased use of accelerator technologies by the HPC industry. GPU (General Processing Units), made widely available through their large use in computer vision, gaming and cryptocurrency-mining industries are now being widely adopted in HPC [18]. In 2018, 5 0f the top 7 supercomputers in the world were powered by GPUs [29]. These GPUs are massively parallel, often having over a thousand simple cores, so fit nicely into the HPC landscape. A GPU can perform around 10 times as many operations per second (FLOPS) than the equivalently price CPU and can move data around 5 times faster (memory bandwidth). They are optimised for throughput and can hide the latency of waiting for data by overlapping the work. This means they do exceptionally well on massively parallel applications.

Taking into consideration all the above factors, it can be seen that HPC machines are becoming increasingly heterogeneous. Modern day programs need to be able to exploit massive parallelism across a multitude of architectures.

1.2 Performance Portability

The range of diverse architectures used today presents a significant challenge to developers. They need to ensure that their code runs and achieves a high level of performance on a given architecture. Many machines contain multiple devices of different architectures, making for even more work. Finally, if a program needs to move to a new machine, significant work may be necessary so that it still can run and maintain a high level of performance. Being able to run and produce a correct result on different machines is what makes a code portable. A code is performance portable if it is able to produce a correct result and still achieve a high level of performance on many machines. Many different studies have investigated performance portability, often using different definitions. To make this study rigorous I will use the definition defined by Pennycook et al [25], which has subsequently been used by McIntosh-Smith in "the broadest and most rigorous performance portability study to date" [21]. The definition is explained in Section 2.5 of this report. As there is no clear consensus of what hardware future machines will have, it is very important that programs are performance portable so that they are future proof.

1.3 SYCL

SYCL is a new language designed to be used in High Performance computing. It is an abstraction layer on top of a previous HPC language OpenCL. OpenCL is a 'close to metal' language, meaning the language has constructs to directly interact with the underlying hardware. It allows programs to execute on heterogeneous platforms and thus can be deemed portable. Since SYCL is built upon this, it inherently can run on a heterogeneous set of hardware.

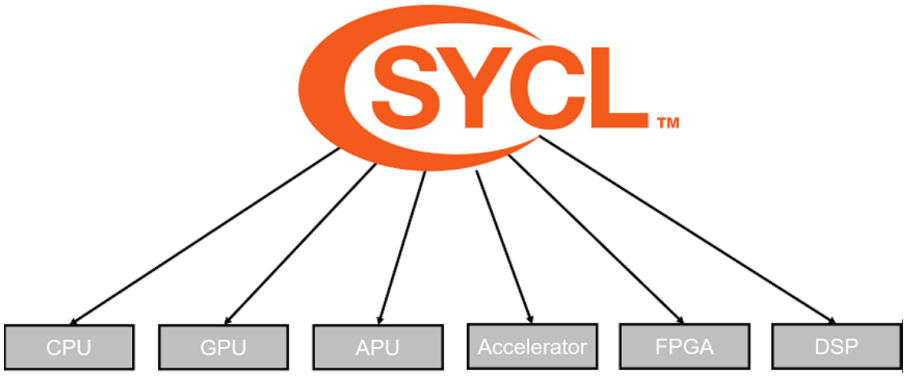

Figure 1.3: Different Architectures SYCL can target [5]

As it is an abstraction layer it allows developers to produce parallel programs without having to worry about the low level details. This makes writing an OpenCL style program far easier. As this is a young language, this project is assessing the state of SYCL and quantitatively looking at what SYCL brings as a new language. Section 2.2 takes a deeper look into the benefits SYCL brings.

1.4 Challenges

This project will have a number of significant challenges to overcome:

1. Learning the SYCL language and overcoming any bugs or nuances in it due to it being a very young standard.
2. Learning Kokkos to understand the TeaLeaf program structure to then enable me to port it to SYCL
3. Standing at 4883 lines of code, TeaLeaf will be the biggest program I have ever worked on.
4. Due to the nature of porting code, having to do it all in one go makes it a significant challenge.
5. I will have to learn how to use many different compilers and understand how to run them on different platforms.
6. All my results will have to be methodically obtained and analysed so that they are consistent with existing research.

1.5 Benefits of Work

This project will add to the existing performance portability study by McIntosh-Smith et al [21]. This will give other researchers and users a quantitative look on the performance of SYCL compared with existing languages. Ultimately giving a greater understanding of the language choice in High Performance Computing. Where applicable, any findings during my project will be fed back to the relevant bodies, so that the SYCL implementations can be improved. Finally, this project will produce two guides in porting to SYCL code, one starting from OpenCL and one from Kokkos. All the code for my project will be made available to aid the understanding of SYCL.

1.6 Project Objectives

My research hypothesis is that implementations using the SYCL programming model will have a performance portability score similar to that of implementations in OpenCL. To do this the project aims to quantitatively assess the performance portability of the new programming language SYCL and to compare and contrast its different implementations. More specifically the following objectives will be met:

1. To port a computational fluid dynamic (CFD) code which uses the Lattice Boltzmann method (LBM) to the language SYCL.
2. To port Tealeaf [22], a mini-app representing iterative sparse linear solvers, to the language SYCL.
3. To rigorously test both Tealeaf and the LBM codes against their OpenCL counterparts to measure the overheads of using the SYCL language.
4. To apply the performance portability metrics defined by Pennycook et al [25] and demonstrated by McIntosh-Smith et al. [21] to measure the performance portability of SYCL across a diverse range of architectures.
5. To test the current implementations of SYCL and assess which is the most performance portable.

1.7 Covid-19 Mitigation Plan

Issue	Impact on Project	Actions to Mitigate Impact	Remaining Impact
The Intel Iris Pro Graphics 580 in Bristol's HPC Zoo went offline. Without physical access it could not be brought back online. Due to Covid-19 physical access was not possible.	The Iris Pro was my main testing device and was needed throughout the testing stages of the project.	I contacted my supervisor to raise the issue. He reached out to Tom Deakin, a Senior Research Associate at Bristol. Tom suggested that I got access to the Intel DevCloud. This contained a similar device.	I was not able to get Codeplay's ComputeCPP compiler to work on the Intel DevCloud. As such these results are missing from my analysis.

Table 1.1: Covid-19 Mitagation Table.

Chapter 2

Technical Background

2.1 OpenCL

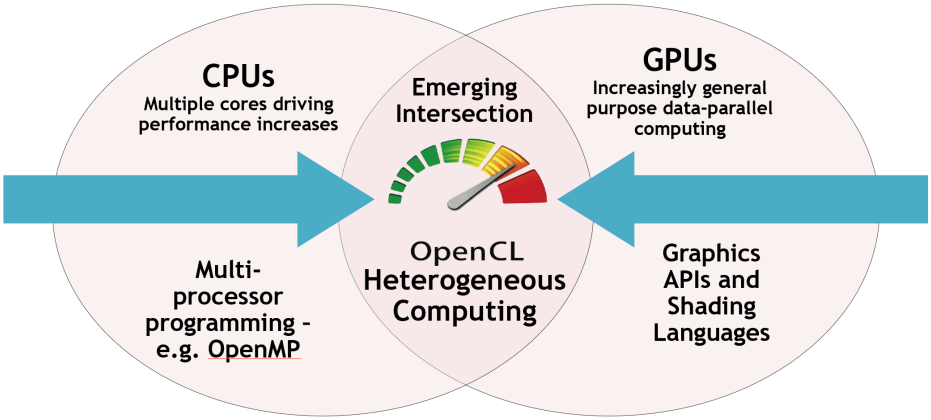

Figure 2.1: Origins of OpenCL[20]

As CPUs began using more cores to drive their performance increase, GPUs were becoming more general purpose and offering their massively parallel structures to scientific computing. This led to the emerging intersection of heterogeneous computing. A vendor-agnostic language was needed that was able to exploit and target the parallelism of all the different architectures. This led to the creation of OpenCL. "OpenCL (Open Computing Language) is an open, royalty-free standard for cross-platform, parallel programming of a diverse range of architectures" [3]. Nowadays OpenCL has widespread industrial support and is used in everything from embedded chips to supercomputers. OpenCL is driving the camera pipelines of the majority of smartphones sold today [23]. Due to the fact the language is not just used in HPC, it is widely supported and documented, making it an attractive option.

The standard, developed by the Khronos Group [3], is a low level framework based on the `c` language. It allows simultaneous targeting of multiple supported devices, not necessarily of the same architecture, allowing for truly heterogeneous computing. This is very different to most of the previous models which just offloaded the code to a single GPU or CPU. It is therefore clear that OpenCL can be used to create performance portable code. The 'Hands On OpenCL' course sums up the importance of OpenCL - "The future is clear: OpenCL is the only parallel programming standard that enables mixing task parallel and data parallel code in a single program while load balancing across ALL of the platform's available resources" [20].

The fundamental idea behind OpenCL is that you exploit all the parallelism in your program. You find loops where each iteration is independent. E.g. the result of the second iteration does not rely on the first and so on. You replace these loops with a function. This function is referred to as a kernel. This kernel function is then executed at every point in the problem domain.

```
//Loop with independent iterations
void squared( float* x ){
    for( i = 0; i < n; i++ ) {
      x[i] = x[i] * x[i];
    }
}

//OpenCL equivalent
_kernel void squared( _global float* x){
    int id = get_global_id(0);
    x[id] = x[id] * x[id];
}
```

Listing 2.1: Converting a loop to an OpenCL kernel.

By changing the program over to a kernel structure it allows all the iterations of the loop to be run at the same time in parallel and therefore you can get your result much quicker. These kernel functions are written in a separate file and then called from the host code when needed. The `_global` tags added to the variables in Listing 2.1 are to define in which memory the variables are held.

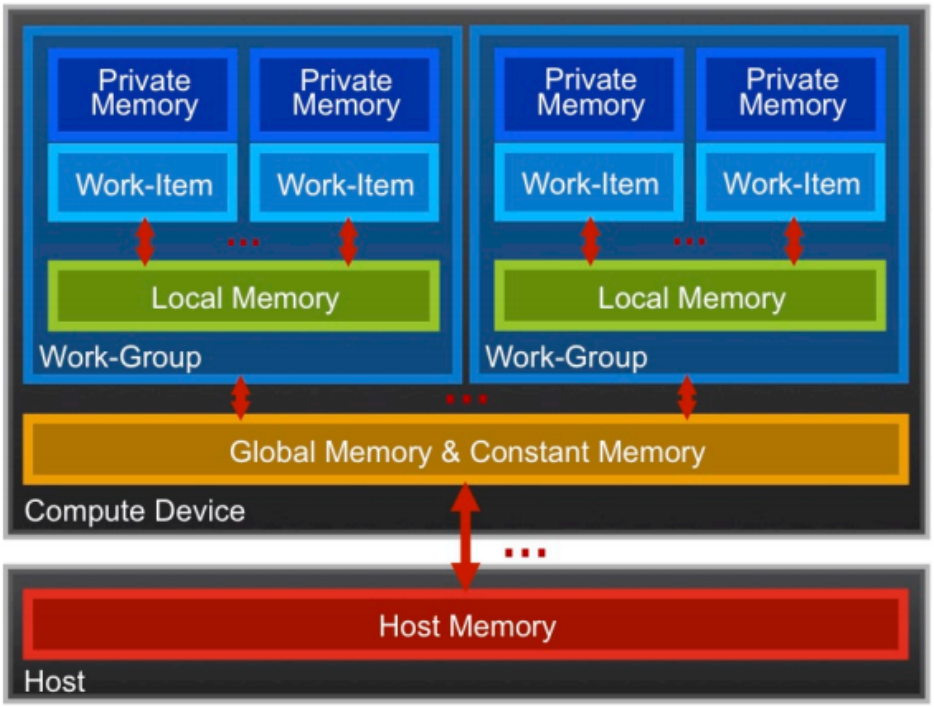

Figure 2.2: Memory Model of OpenCL[20]

As the program is executing across multiple devices, you have to ensure that the memory the program requires to operate, is in the same location as the execution of that bit of code. As Figure 2.2 shows, each device has its global memory which is accessible to all processing elements on that device. It then has memory local to work groups and the memory private to each processing element. A work group is a team of processing elements that you can synchronise between. The host device then has its own separate memory space. OpenCL has an explicit memory management system. This means that any data which needs to be moved between memory spaces, has to be done by the programmer specifying when and where.

2.2 SYCL

As good as OpenCL is, there are a few weak points in the model. SYCL has been designed to combat these. Firstly, SYCL is based on standard `C++`, which is a more modern style of programming and a highly requested feature for OpenCL. `C++` has a better defined memory model and the standard is currently being updated to support more concurrency and parallelism. The SYCL language is completely standard `C++`. This is different to many other parallel languages which rely on pragmas, language extensions or attributes. This allows standard `C++` compilers to be used to produce a standard executable object. This is without OpenCL which will run on any CPU. The SYCL compiler compiles with two passes; one for the host and one for the device. This is then combined into a single executable.

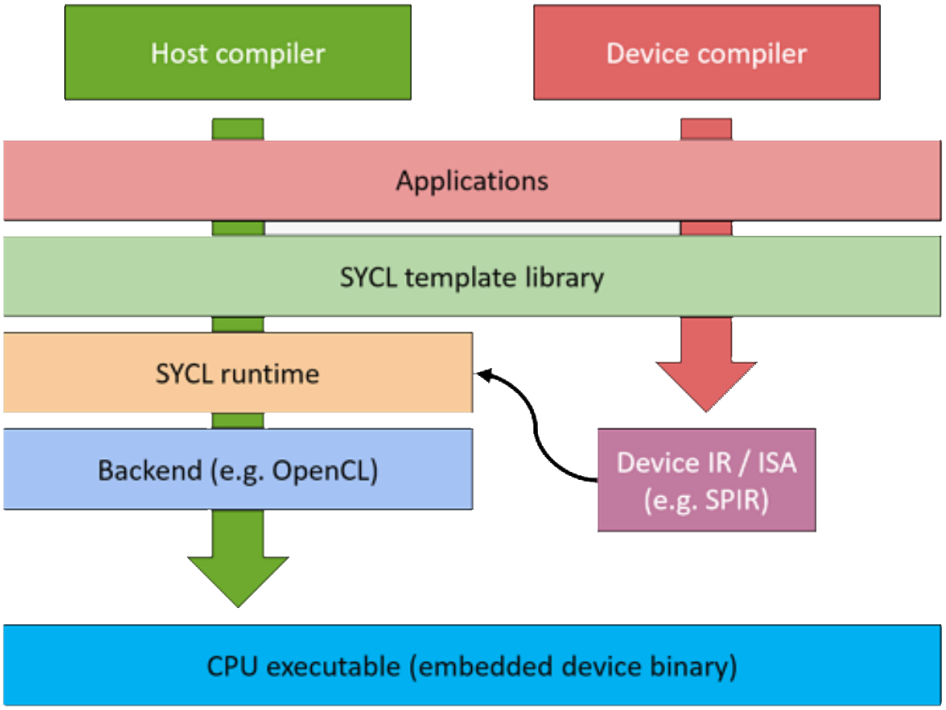

Figure 2.3: Sycl Compiler[5]

SYCL is single source, this means the host code and the device code are written together in the same file. This reduces bugs which occur from the separation of the files. These errors and bugs often occur due to types and numbers not matching between the two files. These result in crashes at runtime, leading to much frustration. Having all the code in a single source system means the compiler will be able to detect and throw the errors instead. It provides a more rigorous interface between host and device code, making it easier for programmers.

Due to the abstraction layer, SYCL code has far fewer lines of code than the equivalent OpenCL code. For example, the basic vector addition code for OpenCL, from HandsOnOpenCL [20], has 243 lines of code, whereas, the SYCL vector addition code, from SYCL Academy [5], has 59 lines of code. SYCL is 4 times shorter to write than the equivalent OpenCL in this case. This makes writing SYCL programs quicker and easier. SYCL handles most of the device set up and has an implicit memory management model. This means that when you want to access data, you request access through an `accessor` construct. SYCL then looks at the `accessors` you are using and makes sure that the data you need is located where you need it, when you need it. This is different from OpenCL as the programmer does not have to move any of the data around manually. SYCL uses these `accessors` to create a dependency graph for the program. This graph shows which data is used and altered where, from this it automatically works out the order of execution and what kernels can run concurrently. An important side note is that OpenCL does have a `C++ host interface` which can handle the device setup. When used this makes the host code more compact.

```
// declare host arrays
double *Ahost = new double[10];
double *Bhost = new double[10];
double *Chost = new double[10];

// Init arrays
for (int i = 0; i < 10; ++i) Ahost[i] = 1.0;
for (int i = 0; i < 10; ++i) Bhost[i] = 2.0;
for (int i = 0; i < 10; ++i) Chost[i] = 0.0;

using namespace cl::sycl;

// Initializing the devices queue with a gpu_selector
queue device_queue(gpu_selector{});

// Creating 1D buffers which are bound to host arrays
buffer<double, 1> aBuff{Ahost, range<1>{10}};
buffer<double, 1> bBuff{Bhost, range<1>{10}};
buffer<double, 1> cBuff{Chost, range<1>{10}};

device_queue.submit([&] (handler& cgh) {
        // Create accessors to read from a and b, write to c
        auto a_acc = aBuff.get_access<access::mode::read>(cgh);
    auto b_acc = bBuff.get_access<access::mode::read>(cgh);
    auto c_acc = cBuff.get_access<access::mode::discard_write>(cgh);

    cgh.parallel_for<class vector_addition>( range<1>(10), [=] (id<1> idx){
                c_acc[idx] = a_acc[idx] + b_acc[idx];
    });
});
```

Listing 2.2: An example SYCL Program.

The key constructs to note in Listing 2.2 are: `buffers`, the `device queue`, `accessors` and the `parallel for`. The `buffers` are objects which hold the memory that can be used on the device. They can be bound to an array on the host as in the example, or they can be initialised on the device. The `device queue` is set up by choosing the device you want to execute your kernels on. Every time you want to run a kernel on said device, you use a `queue submit` to do this. Within each `queue submit` you need to get access for the data that you want to use. The SYCL `accessor` construct allows you to specify what type of access you need to each buffer. The `parallel for` construct specifies that its contents should be executed in parallel across the given range.

2.3 Kokkos

Kokkos is a parallel language from Sandia National Laboratories [12]. It is a single source language based on `C++` similarly to SYCL. It compiles down to OpenMP and CUDA, to allow a range of devices to be targeted. One of the main differences to SYCL is its memory model is very transparent. A `deep copy`, moving data from device to host or vice versa, is never made unless the programmer specifies one. A big advantage of Kokkos over SYCL is that it has a `parallel reduce` function. Reductions are a common pattern in HPC computing and having to write your own reductions in SYCL is time consuming and can result in a sub-optimal configuration. A reduction is where you reduce all the elements of an array down to one value, e.g. find out the sum of all the values in the array. This is not a straight forward task in parallel computing as the traditional criteria of each loop body being independent does not exist here. Therefore special algorithms have been developed to allow reductions to occur in parallel. Reductions have been requested for SYCL and it has been recently announced at IWOCL and SYCLcon 2020 that they will be present in SYCL 2020. The reason Kokkos is included in this project is that TeaLeaf has a

version of its kernels written in Kokkos. This means the code is already in `C++` and this will be used as the starting point for the SYCL Tealeaf port.

```
kokkosInit();
// declare host arrays
double *Ahost = new double[10];
double *Bhost = new double[10];
double *Chost = new double[10];

// Init arrays
for (int i = 0; i < 10; ++i) Ahost[i] = 1.0;
for (int i = 0; i < 10; ++i) Bhost[i] = 2.0;
for (int i = 0; i < 10; ++i) Chost[i] = 0.0;

using namespace Kokkos;

// Creating Kokkos views
View<double* [1]> aBuff("A", 10);
View<double* [1]> bBuff("B", 10);
View<double* [1]> cBuff("c", 10);

//Transfer Arrays to Device
deep_copy(aBuff, Ahost);
deep_copy(bBuff, Bhost);

parallel_for(10, KOKKOS_LAMBDA (const int idx){
        c(idx) = a(idx) + b(idx)
}
kokkosFinalize();
```

Listing 2.3: An example Kokkos Program.

Listing 2.3 is an example Kokkos program. It will look very similar to the SYCL example in Section 2.2. The key differences are that: Kokkos has a construct called `views`, these are similar to `buffers` and look after the data which can be used on the device; Kokkos does not require `accessors` as the programmer has to manually copy the data across before use and finally, there is no `queue` construct as Kokkos handles that all behind the scenes.

2.4 The Seven Dwarfs of HPC

The Seven Dwarfs of HPC was a phrase first coined by Phil Colella in 2004 [11]. It is used to represent that within scientific codes there are 7 very commonly used algorithmic patterns. The Seven Dwarfs are:

1. Dense Linear Algebra (data is in dense matrices)
2. Sparse Linear Algebra (data sets contain many zero values)
3. Spectral Methods (data is in the frequency domain)
4. N-Body Methods (interactions between many discrete points)
5. Structured Grids (spatial grid, discretised in a regular way)
6. Unstructured Grids (spatial grid, discretised in an irregular way)
7. Monte-Carlo (numerical approximations for mathematical functions)

This was subsequently expanded to 13 dwarfs in 2006 by a team at Berkeley [9]. As they are all very common, they have all been parallelised and solved in the past. This means that on encountering one, it

is possible to utilise past solutions to help solve a current problem. Each dwarf reveals its complexities in terms of computation, communication and memory requirements and thus it can be seen what the bottle neck will be.

The Lattice-Boltzmann code used in this project is a Structured Grid Dwarf. This means that the code will be memory bandwidth bound. This is the sort of code used in fluid simulations. It is particularly useful as it parallelises really well compared with other fluid methods such as the Navier-Stokes Equation. The version used was the 'D2Q9' version meaning it is a 2 dimensional grid with 9 speeds of fluid flow per cell.

TeaLeaf is a Sparse Linear Solver Dwarf. "It is an open-source mini-app released as part of the R&D 100 award-winning Mantevo suite. TeaLeaf is a vehicle to explore the design of new solvers, and for comparing the efficiency of new parallel programming languages" [22]. TeaLeaf is memory-bandwidth bound when run on one node as done in this project. At large scale (when strong scaling) it becomes communication bound.

2.5 Performance Portability

In this project I explore the performance portability of the SYCL programming language and thus performance portability needs to be defined. Many papers have been published on this topic but there is still not an agreed method to quantitatively assess it. Pennycook et al. [25] provided a rigorous definition to create a shared metric by which everything can be compared fairly. This definition was then used by McIntosh-Smith et al. [21] in "the broadest and most rigorous performance portability study to date" [21]. By using the same metric I am allowing my results to be easily understood and built upon by future researchers. Pennycook et al.'s definition [25] is quoted from their paper below:

$$\Phi(a,p,H) = \begin{cases} \frac{|H|}{\sum_{i \in H} \frac{1}{e_i(a,p)}} & \text{if } i \text{ is supported } \forall i \in H \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

$e(a,p)$ is the performance efficiency of application a on platform p. The above metric works is the harmonic mean of performance efficiency for all platforms, p, in the set of platforms H. Performance Efficiency is then defined as either of the following:

- Architectural Efficiency – the performance efficiency is the performance of the program as a percentage of the theoretical peak hardware performance.
- Application Efficiency – the performance efficiency is the performance of the program as a percentage of the performance a variant of the code hand tuned for that device can achieve. This allows the portability of a program to be assessed without taking into account how efficient the algorithm is.

Application Efficiency will be used in the analysis of this project's results in Section 4. The McIntosh-Smith et al. paper [21] contains a section on the TeaLeaf mini-app used in this report. They have the application written in 4 languages and have assessed the performance portability across 11 devices. This project will produce a SYCL version of the application and add to these results.

2.6 Methodology

To analyse and compare different programs on different architectures, recording the programs runtime was critical. I tested multiple programs at varying stages of development, which were written in multiple languages. I used different compilers and different input sizes, and executed them on different architectures. Each test was run five times and the results were averaged and checked to ensure that they were correct. If there was large variability in the timings, this was investigated and further testing was carried out to gain an accurate runtime. This resulted in an extremely large amount of data. This was effectively

id	program	language	inputSize	version	hardware	compiler	flags	time
1	LBM	OpenCL	128	N/A	Iris Pro 580	GCC	Ofast march=native	2.554
2	LBM	OpenCL	128	N/A	Iris Pro 580	GCC	Ofast march=native	2.517
3	LBM	OpenCL	128	N/A	Iris Pro 580	GCC	Ofast march=native	2.588
4	LBM	OpenCL	128	N/A	Iris Pro 580	GCC	Ofast march=native	2.51
5	LBM	OpenCL	128	N/A	Iris Pro 580	GCC	Ofast march=native	2.635
6	LBM	OpenCL	256	N/A	Iris Pro 580	GCC	Ofast march=native	12.12
7	LBM	OpenCL	256	N/A	Iris Pro 580	GCC	Ofast march=native	12.185
8	LBM	OpenCL	256	N/A	Iris Pro 580	GCC	Ofast march=native	12.105

Figure 2.4: A snippet from the timing database

managed by creating a database to store it, enabling ease of analysis at a later point.

To store my code I used GitHub and Dropbox. Dropbox was set up to automatically synchronise all the files related to the project between my desktop and laptop, ensuring that I always had the most recent code whichever machine I was using without any intervention. This also provided an automatic backup service in case of lost access to my laptop and desktop. GitHub was used as a version control system. It kept track of changes I made and allowed me to revert to previous versions if necessary. GitHub also provided a platform on which to share my code with others. This was used when troubleshooting or sharing interesting results and bugs with the relevant parties.

To ensure the results from this project could be easily replicated. All the code used was be uploaded to: https://github.com/WSJHawkins/ExploringSycl. This repository explains how to get each file to run and provides Makefiles to compile each program. Unless explicitly mentioned all results for LBM were from using the 256x256 input, sometimes referred to as 'the small input size' or from the 4096x4096 input referred to as the 'large input size'. TeaLeaf ships with 6 benchmark inputs to check the program is working correctly. For my tests I used 'Benchmark 2' as the small input and 'Benchmark 5' as the large input.

2.7 Software

The main type of software used by this project was compilers. As SYCL is an open standard it is reliant on others to implement the standard and write the compiler which will turn users' SYCL code into an executable object which can run on different machines.

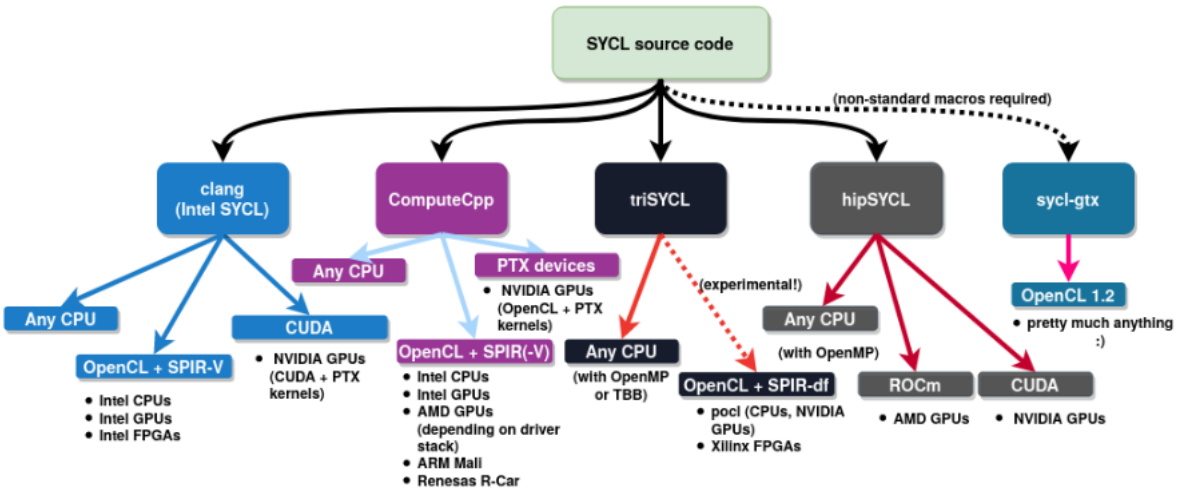

Figure 2.5: The current SYCL ecosystem [7]

Figure 2.5 shows the current SYCL ecosystem and the different hardware each implementation al-

lows you to target. Each offers different benefits, whilst also being in different stages of development. Objective 5 of this project was to discover which of the SYCL implementations is the most performance portable. To do this all the current implementations were explored.

Clang (Intel SYCL) and ComputeCPP are the most mature implementations. They work well on Intel CPUs and GPUs using OpenCL as the underlying framework. They can both can target any CPU either serially or using parallelisation using OpenCL. ComputeCPP can target a few other devices such as certain AMD GPUs and ARM Mali chips. The latest versions of both support Nvidia GPUs. However, this is a brand new addition and is not working on the Bristol HPC Zoo yet. Due to this, this project does not include results of Intel SYCL or ComputeCPP on NVIDIA GPUs. triSYcl is a single developer led project in a very early phase. It does not add many platforms and is becoming increasingly focused on targeting Xlinix FPGAs. This could be useful to look at in a year's tine. It is also not on the HPC Zoo either and as such this project did not use it. hipSYCL is in a pre-release state but has a large range of architectures it can target. It is an interesting implementation as it is classed as 'non-conformant'. This is due to it not using OpenCL as its backend. Using OpenMP it can target any CPU; it can target AMD GPUs and Nvidia GPUs using ROCm and CUDA respectively. Being in prelease it has not yet implemented all the features of the SYCL specification. During this project, I used hipSYCL to see if I could circumnavigate the limitations. For the OpenCL code, the GNU compiler `gcc` was used. For Kokkos, the CUDA compiler was used, `nvcc`.

As described above, Section 2.6, I selected Dropbox and GitHub. This report was written in LaTeX, using the online platform Overleaf. The database used was a MySQL database hosted locally using the XAMPP software. Microsoft Excel was used to produce the graphs you see in this report.

2.8 Hardware

The main source of hardware for this project was the Bristol HPC Zoo [14]. This is a small cluster containing a variety of different architectures and accelerator technologies used to test performance portability. The main device used for the majority of testing during the project execution was the Intel Iris Pro 580. This was chosen as the main device as it is a stable platform for both the Intel LLVM compiler and the ComputeCPP compiler. For the analysis and testing carried out in the latter stages of this project the Zoo provided both AMD and Nvidia GPUs plus server CPUs and a vector engine.

Chapter 3

Project Execution

3.1 Lattice Boltzmann

3.1.1 SYCL

The first stage of this project was gaining an understanding of the SYCL programming language. This started by watching an 'Overview of SYCL ComputeCPP' from the SuperComputing 17 conference [15] and 'A Modern C++ Programming Model for GPUs using Khronos SYCL' from 'CPPCon2018' [10]. These videos were both great resources for understanding the need for and the ideas behind the language. It also provided sample code and the slides were used as a reference point during the start of my SYCL journey. From past experience, I have learnt a good way to get into a language is by being hands on and starting to write code. A good resource for this is the 'SYCL Academy' [5], especially the online interactive tutorial [2]. This allows users to write, compile and execute SYCL code from their browser, enabling them to focus on learning the language without having to worry about the details of setting it up. Having completed this tutorial, a base level understanding has been gained allowing me to start to tackle the Lattice Boltzmann code.

3.1.2 Coding Stages

I started from my OpenCL version of the Lattice Boltzmann code. I had this code from 'COMS30006 - Advanced High Performance Computing', a unit I had previously undertaken in my degree. This meant I was familiar with how this code worked and as such would be a good starting point. This code had been hand optimised [16] and thus provided a good benchmark for the SYCL performance. The desired outcome would be that the SYCL code would compile down into OpenCL code which would perform as well as the original OpenCL code.

The OpenCL version is written across two files, the kernels that run on the 'OpenCL Device' are in the second file. SYCL is written in a single source, so the first stage of the port was to move them both together into one file. Secondly, SYCL is based on `C++`, efforts were made to convert the `C` style OpenCL across to `C++`. Then the actual porting began. Instead of having `cl_mem` objects SYCL uses `buffers`. For all of the data that would be accessed on the device, a buffer was set up mapping on to the host memory. In OpenCL, the memory management is explicit so every time you need data on the device or host you have to ensure it is there and if not you need to read or write from the buffer. SYCL manages this for you with a dependency graph, this meant all the reads and writes could be removed from the code. To allow SYCL to build this dependency graph `accessors` needed to be set up. Each `accessor` allows the code to access the contents of the `buffers` and specifies what permissions it needs, e.g. read or write. Work was undertaken to look at the kernels and understand the memory they needed to access and with what permissions. Setting up the minimal amount of accessors was key, as unnecessary dependencies can slow execution.

Within the actual kernel code, very little needed to change to get the program into a working state. The OpenCL based functions for getting the indices and other maths based functions needed to be swapped over to SYCL equivalents. To allow efficient memory access the optimised LBM code required the swapping of the inputs into the kernel for every iteration. For example, on the first iteration of the kernel, you read from object1 and write into object2, on the next operation you read the results

from object2 and write back into object1. This removes an unnecessary copy operation and thus improves efficiency. In the OpenCL code this is done by switching the input arguments given to the kernel based on the iteration count. With SYCL there were a few different ways to implement this. Only one kernel execution, e.g. one `parallel_for`, is allowed per SYCL `queue.submit`. This means each of the `accessors` are set up per kernel execution. The direct comparison to the method used in OpenCL, would be to swap the memory the `accessors` point to each execution. Another option would be to keep the `accessors` the same but have a different version of the kernel which reads and writes the other way round. Merging two kernels executions into one would simplify the situation but makes the output incorrect so is not an option. Both these possible solutions were tested. Using different kernels was quicker, but only up to 1.5% which is not statistically significant. Going forward the different kernel pattern was used.

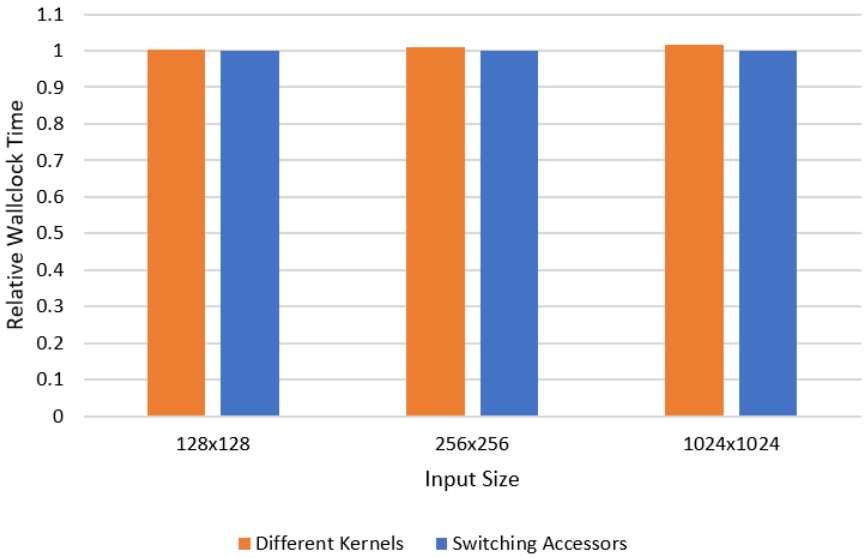

Figure 3.1: Comparison Of Kernel Options

During the coding stage I found out several things about SYCL which had not been initially obvious from tutorials. Firstly, each SYCL `queue.submit`, should only contain one kernel. This makes a `queue.submit` approximately equal an OpenCL `clEnqueueNDRangeKernel` rather than matching a SYCL `parallel_for` to an OpenCL `clEnqueueNDRangeKernel` as I initially had thought. If more than one kernel is placed in the `queue.submit` the program still compiles and runs, but does not produce the correct result. I believe that the compiler should throw an error to raise awareness of the issue. This was put into my recommendations in Section 5.4. I also believe that there should be built in support for switching the input and output parameters without having to redefine the whole kernel. This is a common pattern of code and thus should have some support, this was also mentioned in my recommendations. Another surprising discovery was that all compiler error messages are repeated twice. This can be disconcerting and confusing at first as it was for me. I investigated the issue and it became apparent this occurs due to the double pass compile mechanism SYCL uses. This mechanism is described in 2.2.

After finishing both codes the verbosity of both languages can be compared. As mentioned in the technical background, Section 2.2, the SYCL code would be expected to be significantly shorter.

From Figure 3.2, it can be seen that the LBM code is shorter when written in SYCL compared to OpenCL, but it is not as big of a difference as was expected. The figure breaks down the program into code that would execute on the host and the code that executes on the device. Here, as expected, it can be seen that the SYCL host code is significantly shorter than the OpenCL equivalent. This is not as large as the 4 times difference mentioned in the vector add example in Section 2.2, however this is expected as the LBM program contains lots of host code that is compromised of generic functions, nothing to do with the setup of the two languages. The unexpected result is that the device code for SYCL is twice the length of the device code for OpenCL. This is due to the kernel code having to be written twice in SYCL to swap over the variables, compared to swapping the kernel arguments in OpenCL. On further

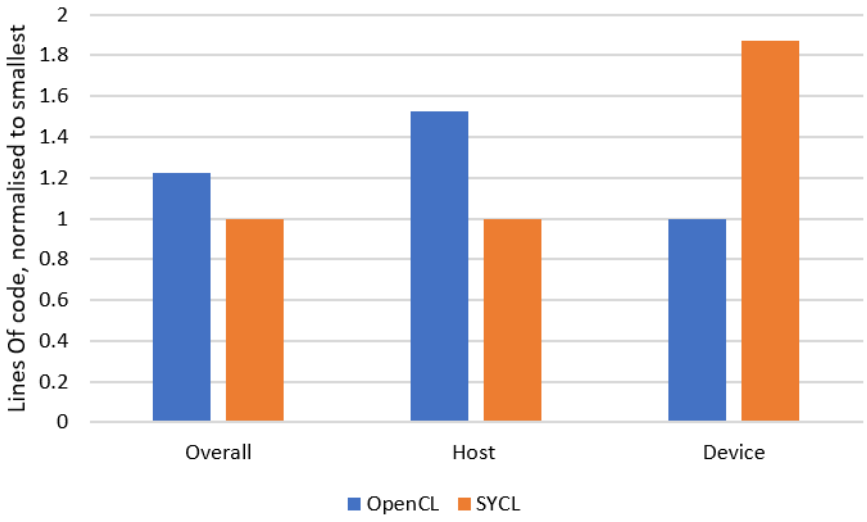

Figure 3.2: Comparing the lines of code, breaking it down by parts

inspection the SYCL device code can be seen to be approximately twice the size of the OpenCL code and thus the equivalent kernel code is in the same ballpark in terms of its length.

An issue I encountered with my code is that on the Skylake CPU, using the LLVM compiler, it compiles successfully but at runtime the program crashes with a segmentation fault while it is parsing the program. I discovered which lines are causing the issue and experimented with different solutions to fix this. I shared this problem with Tom Deakin, a Senior Research Associate at Bristol University working with SYCL. He was also unable to see the issue. I believe it is happening due to some mismatch between the OpenCL runtime for this specific platform and the LLVM SYCL compiler. This would explain why it only occurs on this platform using this compiler. I have fully detailed this issue and documented the work which I have carried out in an attempt to fix it. This has been logged as an issue in the repository containing this code. This makes its existence clear to any future users and allows them to offer a fix if they know of one.

3.2 TeaLeaf

3.2.1 Kokkos

TeaLeaf is a heat diffusion mini-app, which uses a Conjugate Gradient linear solver on a 2D structured grid [22]. It comes with kernels in a variety of parallel languages. Unfortunately, an OpenCL kernel is not present in the implementation used. Kokkos was chosen as the kernel language to start from. This is due to the reasons discussed in Section 2.3, especially the fact the code is already written in `C++`. Starting from Kokkos provided the opportunity to learn a new language and enabled me to document porting from Kokkos to SYCL, another useful output of this project.

To start learning Kokkos I used the 'Kokkos-Tutorials' GitHub page [4]. This contained slides which provided a crash course in Kokkos. Focus was particularly paid to what constructs Kokkos used in relation to their SYCL counterparts. Another good resource I used was Lin's article on porting CloverLeaf [19], another mini-app, from Kokkos to SYCL. From these resources I had a base level of understanding and as such I started working on the TeaLeaf code.

3.2.2 Coding Stages

TeaLeaf is the largest piece of code I have worked on to date, it has 4883 lines of code counting the host code and only one set of kernels (for this count the SYCL kernels were used). Due to this I wanted to be very careful and methodically work through it. I only had to work within the `c-kernels` folder and if the interface stayed the same all the host code could remain. I then tried to break down the task into different stages. The stages were: port all the `parallel_for` loops over to SYCL; port the `parallel_reduces`

over to SYCL; set up the `device queue` so all functions can access it; remove the Kokkos `mirror` and `deep copy` constructs; write the Makefile and successfully get the program to compile. An interesting point to note here is that I did not compile until the end as I needed to have successfully completed the port to do this. This is different to usual coding practice of compiling regularly to try to prevent bugs. This was further reason to ensure a methodical approach.

In the SYCL implementation I ensured that I had a consistent naming convention for my variables in the kernels. This reduced the chance of bugs being produced from using the wrong variables used in the wrong place. All `buffers` had the word 'Buff' on the end of the variable. The `accessors` attached to each `buffer` had the same name minus the 'Buff'. This made the distinction clear and as such made it obvious which to use where.

The first task of tackling the `parallel_for` loops was a relatively simple one but there were lots of loops to replace. I took an existing Kokkos version from the code and wrote a SYCL equivalent. The major difference, as shown in Listing 3.1, is the need to submit the function to the `device queue` and to set up the `accessors` to manage the data flow. I then used this SYCL loop as a template from which to do all the rest. This meant that when I got to the compilation stage if there was an error it was easier to fix as they would all be in the same format. The Kokkos code used a `typedef` to shorten the `view` construct to a `KView`. I followed this pattern making `SyclBuffer` stand for a `buffer` of type `double` with 1 dimension. This reduced the amount of typing that needed to be done. If the type needed to be changed, then it only needed to be changed in one place.

```
//Kokkos Parallel For
parallel_for(x*y, KOKKOS_LAMBDA (const int index)
{
    u(index) = energy(index)*density(index);
});

//SYCL Parallel For
device_queue.submit([&](handler &cgh){
    //Set up accessors
    auto u              = uBuff.get_access<access::mode::write>(cgh);
    auto density        = densityBuff.get_access<access::mode::read>(cgh);
    auto energy         = energyBuff.get_access<access::mode::read>(cgh);

    auto myRange = range<1>(x*y);
    cgh.parallel_for<class example>( myRange, [=] (id<1> idx){

        u[idx[0]] = energy[idx[0]]*density[idx[0]];
    });
});//end of queue
```

Listing 3.1: Kokkos `parallel for` construct and SYCL equivalent.

Next was the `parallel_reduce` loops. SYCL does not have a built in reduce function so these had to be undertaken manually. The main loop changes were as above adding the `device queue` and `accessors`. The actual reduction was a bit more complicated. The first solution was to make a generic reduction function. This took a `buffer` as an argument, reduced it in parallel and then returned the result. This meant all the reductions in the code used the same function and as such any bugs in the reduction were easier to find. In Kokkos the reduction function cannot handle multiple reductions out of the box. In this case a class has to be set up with specific functions to allow Kokkos to handle this. The `Field Summary` function in TeaLeaf used this set up in the Kokkos Kernels. When converting this to SYCL, the class setup was removed and the code standardised to the same `parallel for` lambdas system that the rest of the mini-app uses. For the reductions the generic helper function was used on each buffer individually. This method was later found to be inefficient. Instead the reductions could be done in place, meaning that one less kernel invocation was needed per reduction. The results of this change are discussed in Section 4.1.4

As mentioned in the above tasks, SYCL requires all kernels to be submitted to a `device queue`. To make sure all functions had access to the queue, it had to be passed as a function parameter to all the kernel functions. To set the queue up, it was defined as a global variable in the `kernel interface` file, allowing it to be passed to the `kernel initialise` function where it was set up.

The next task was to tackle the Kokkos `mirror` construct and `deep copy` functions. A `mirror` is a Kokkos `view` construct on the host that has a compatible layout to that of the `view` on the device. A Kokkos `deep copy` is when you explicitly instruct the program to move data between the `device view` and the `host mirror` or vice versa. The `mirror` constructs were deleted as they were not needed in SYCL. Instead a SYCL `host accessor` was created at the point of the `deep copy`. This meant that SYCL would automatically transfer the data across for use at this point.

Compiling the program was the next stage of development. Compilation threw up many simple bugs, these were due to the nature of porting. The methodical and consistent process used meant that these were easy to solve. Following these 'typo' style bugs being fixed, a segmentation fault bug arose. To debug this a smaller version of the code was built. This program was only 20 lines but was able to replicate the segmentation fault seen in the full code. Using this, experimentation and trialling fixes was far easier. This smaller version was quicker to compile and run, allowing for faster changes. Although the issue which caused the segmentation fault was identified, a solution was not immediately apparent. I reached out to Tom Deakin. Having it in this smaller form meant that he could spot the solution to the problem. This was passing the `buffers` to the functions by reference instead of using pointers and de-referencing. How the `buffers` were referenced on creation also needed to change. This was very easy to do due to the `typedef` already set up. Once it was working in the small program it was rolled out on scale to the TeaLeaf code. This resulted in the program running albeit not producing the correct output, a big improvement. Building a simple version was extremely helpful in solving this problem.

The program was now running but not producing the correct output. To debug this a function which would print out the variables from a buffer was created. To do this the data needed to be brought back onto the host before printing it to screen. The Kokkos version had a similar function which was modified to match. This allowed both programs to be run on the same input and the outputs to be compared. Using these functions I methodically worked through the program comparing the outputs. The first issue found was that an `integer` was being used in a calculation for a variable of type `double`. This was producing the wrong output and was fixed by casting the `integer` variable before use. The other bug found was that the reductions were not working in some cases. Having looked at the reduction function I could see the local memory for the reduction was using a 32 bit integer type not a double. Having the generic function meant changing this one line fixed all the reductions. The program now ran and produced the correct output for Benchmark 2. However on the larger input, Benchmark 5, the program died during execution, sometimes throwing a `bad alloc` warning. A `bad alloc` is thrown when the program fails to allocate memory that the programmer has asked for. After lots of debugging it was found that the temporary arrays used in the reductions were being created and not deleted. On the smaller program this is fine as the size of the array is smaller and the number of iterations is smaller. On Benchmark 5, this memory leak was causing the memory supply of the heap to be exhausted and thus causing the program to fail. Deleting the temporary arrays as soon as the reduction was complete fixed this issue and the program then successfully ran on all the provided benchmarks.

TeaLeaf comes with a built in profiler to see how long each kernel takes to execute. Knowing that this would be useful for the analysis stage of my project I made sure that it worked with my SYCL kernels. Most of this was basic work copying the structure from the existing Kokkos kernels. During this process I made a couple of discoveries, the first being that the profiler is not activated by a 'profiler_on' flag like the repository instructs. Instead the profiler is enabled through a flag passed to the Makefile. Secondly, the memory used by the profiler was never initialised and thus sometimes the results would be wrong. I fixed this issue within my own code. I shared the fix to the uninitialised variable bug, back to the original repository, through the medium of a pull request. I also notified the repository owners of the way in which the profiler is turned on. This was by email, in addition to adding a note in the readme included in the pull request.

On the Intel Skylake I encountered an issue which I was unable to solve. Whilst the code produces

the correct output on the smaller benchmarks, on larger ones (such as Benchmark 5), the code produces the wrong output. Looking in depth at the code, the error value (which is what determines how many iterations are run at each timestep) decreases at a slower rate than it does on other platforms. This means that the program takes a very long time to run, hitting the iteration limit every time step and producing the wrong output. I documented this issue and raised it on the repository containing the code. This will allow researchers to look into this issue and hopefully fix it in future.

3.3 Testing

Now I had both my codes working I could move onto the next phase. To analyse the performance of my codes, the runtime of the programs was measured. This section does not detail the results as they are discussed in Section 4. Instead the testing methodology is discussed.

To satisfy Objective 4 of this project I needed to run the programs on a diverse range of architectures. The hardware used was the Iris Pro 580, Nvidia GTX 2080TI, AMD Radeon VII and the Intel Skylake. This group contains a CPU, Nvidia GPU, AMD GPU and an embedded GPU. This provided a diverse group and as such should reflect performance on a much wider range of devices. I used the 3 different SYCL compilers. I used the hipSYCL compiler to allow me to target the GTX 2080TI and AMD Radeon VII. I used LLVM SYCL and ComputeCPP compilers to target the Iris Pro 580 and Intel Skylake.

Throughout the coding stages I used LLVM SYCL as my main compiler and ran the code on the Iris Pro 580 due to this being the easiest configuration to get working. For the testing stages I tried compiling with hipSYCL. As mentioned in Section 2.7, hipSYCL has not fully implemented the entire SYCL specification. I had to remove some of the built-in functions I had used in my code. I had used the `recip` and `select` functions from the SYCL specification. As hipSYCL did not support these I had to use manual versions. `recip(x)` became `1/x` and `select(a,b,c)` became `c ? b : a`. The reason for using the built in functions in the first place is often the SYCL implementation can perform the operation in more efficient ways.

To gain insight into how well SYCL performs, I needed something to compare it with. For this I used versions of the codes in different languages. For the LBM code, I had OpenMP and OpenCL versions. The TeaLeaf code came with Kokkos, OpenMP and CUDA kernels. To compare the SYCL TeaLeaf implementation to OpenCL, I used a Fortran version of TeaLeaf with OpenCL kernels, this came from MAC UK - the UK Mini App consortium. For large input sizes the kernel time would dominate and therefore the fact the host code was in Fortran and not `C` would make little difference. When I tried to use this Fortran version I found some bugs in the OpenCL code. It would not run on the Radeon VII due to a variable scope error. I managed to fix these bugs so that I could get my results. I also created a pull request to contribute my fixes to the original repository.

Unfortunately the Iris Pro 580 I was using as my main testing device went offline at this stage of the project. The hardware did not respond to attempts to bring it back online remotely. Due to the Covid-19 Pandemic, physical access to the hardware was not possible. The device was still needed as I had not yet run all tests. Fortunately, Tom Deakin was able to suggest an equivalent replacement. I was able to sign up to the 'Intel DevCloud' [17]. Here I remotely accessed another Intel Integrated Graphics device. This one was an Intel UHD Graphics P630 compared to the Intel Iris Pro Graphics 580 on the HPC Zoo. This new device had 1.1 times more memory bandwidth but 0.33 times amount of Compute Units. This meant on TeaLeaf and other memory bandwidth codes it would perform better but on compute bound codes worse. Due to the differences all the results for TeaLeaf were rerun on this device to ensure consistency.

Codeplay's ComputeCPP compiler could not be set up on the DevCloud due to the compiler not being able to find dependent packages in the environment. The DevCloud lacks a module system and as a user with few permissions I was unable to find or setup these packages. This means that for TeaLeaf there are no ComputeCPP results on the Intel UHD Graphics P630. This issue is documented according to Bristol Universities guidelines in Section 1.7.

Finally, to produce my graphs Microsoft Excel was used. I ran queries on my database to pull out the relevant records I needed for each stage of analysis. I entered these into Excel and manipulated them to produce graphs. Where possible I followed the Gestalt Principles for Data Visualisation [28].

I used consistent colours throughout all of the results and clustered data together in a way which aids understanding. Coblis [1] was used to check colourblindness compatibility. This ensures that the graphs are intuitive to all.

3.4 Sharing of my code

The final task I undertook on this project was to clean up all the code and make it freely accessible. This will hopefully allow others to benefit from my work. I created a public repository on my GitHub: https://github.com/WSJHawkins/ExploringSycl. Here I placed all my code in a neat and organised fashion. Each program has with it the necessary instructions and Makefiles such that an outsider could come along and run my code to reproduce my results. This repository also contains the guides I have written to help others port their codes to SYCL.

Chapter 4

Critical Evaluation

4.1 SYCL Overheads

4.1.1 SYCL versus OpenCL

After successfully porting the Lattice Boltzmann code from OpenCL to SYCL, the code was run on the Intel NUC (Iris Pro 580). Using the same platform for both codes allowed for a fair comparison. Ideally the SYCL code would have been compiled down to OpenCL code which would have been comparable in performance to the hand written OpenCL code.

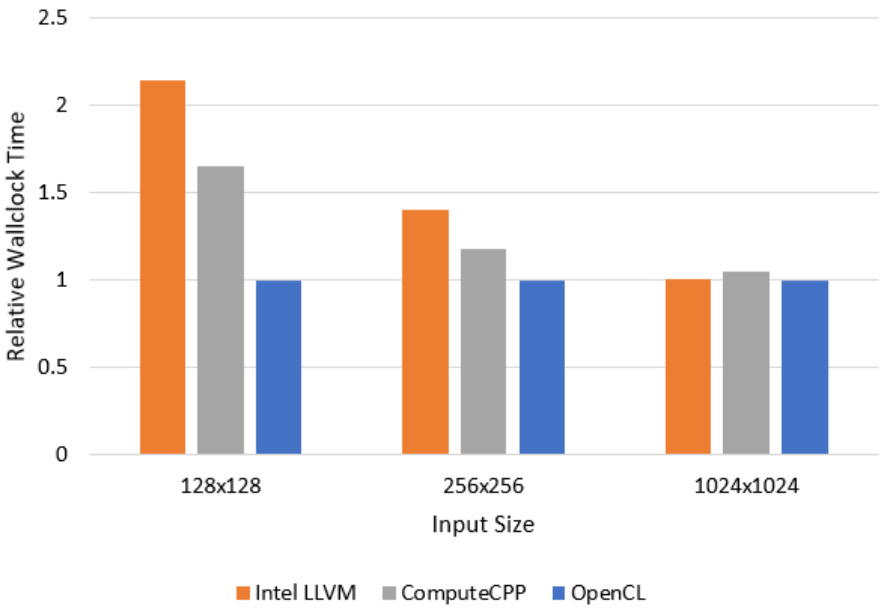

Figure 4.1: Comparison Of OpenCL and SYCL performance on the Intel Iris Pro 580

On the 128x128 problem size the SYCL code is achieving 0.5 times the performance of the OpenCL equivalent. However, as the problem gets larger, the two programs become more comparable. The 256x256 problem size on SYCL achieves 0.7 times the OpenCL performance. Finally, with the 1024x1024 input size SYCL is achieving 0.96 times the performance of the OpenCL version. This is an interesting result as the difference is not consistent. In the larger problem sizes the runtime is dominated by the kernel execution times, whereas in the smaller problems the setup and host code takes a bigger percentage share of runtime. Therefore these results suggest that there are some overheads in the SYCL setup which are becoming very noticeable for the small problem size.

To investigate what is going on here the Intel OpenCL Intercept Layer was used. This program produces a timeline of all the OpenCL API calls allowing users to see the kernel execution times and

memory transfers. As SYCL is sitting on top of OpenCL this will allow direct comparison between the two.

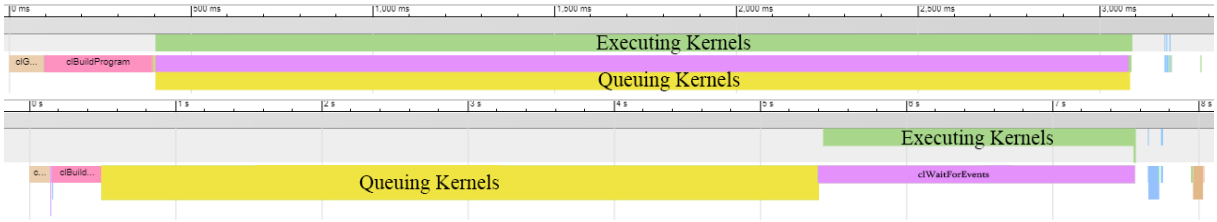

Figure 4.2: Comparison Of OpenCL Intercept Layer Traces. Top: OpenCL, Bottom: SYCL

Traces for both runtimes were generated for both languages on the 128x128 input size. Before inspecting the traces, one difference became clear, the file size for the SYCL trace was 5 times larger. This suggested far more calls to the underlying OpenCL APIs were being made by the SYCL runtime. In Figure 4.2 the green section is the execution of the kernels and the yellow section is the queuing of the kernels. Admittedly you cannot see much detail from Figure 4.2, however, it can clearly be seen in SYCL the kernels were not running until after they had all been queued. In OpenCL you can see the queuing and execution of the kernels is overlapped. Waiting for all the kernels looks to be slowing the execution.

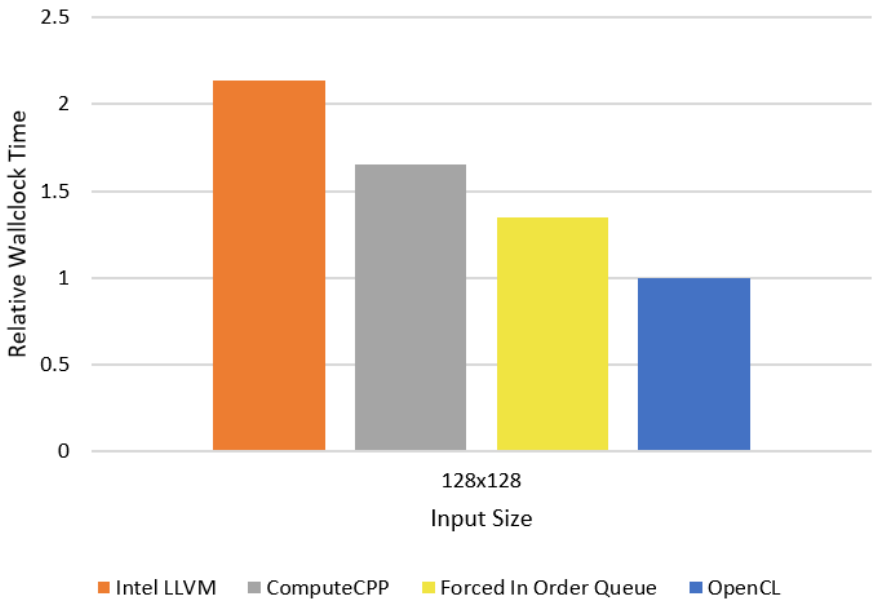

Figure 4.3: Comparison of the performance of different queue styles in SYCL

To try solve the issue, I contacted the author of the OpenCL Intercept Layer, Ben Ashbaugh. Ben works for Intel and has a role in the development of the Intel LLVM SYCL compiler. Ben discovered a large difference between the SYCL and OpenCL versions. This was due to the SYCL implementation using an out-of-order queue, whereas the OpenCL version used an in-order queue. An in-order queue is where the kernels are run in the order they are launched. If two kernels are launched one after another, the first completes before the second starts. An out-of-order queue does not come with that guarantee. The only constraints on the order of execution come from the programmer. In the case of SYCL, this is from the dependency graphs created from the accessors. Ben stated "for our GPU implementation we're much more aggressive about batching commands for out-of-order queues, hoping to execute kernels concurrently as much as possible, whereas for in-order queues we try to get commands onto the GPU as quickly as possible". This explained the delaying of the execution of kernels. Using the OpenCL Intercept Layer, Ben forced the SYCL implementation to use an in-order queue which gave a 1.4 times improvement in wall-clock time. This result is shown in Figure 4.3. This queuing behaviour causing a detriment to runtimes had not been experienced by the Bristol HPC group and as such this project has

raised their attention to the issue.

This forced in-order queue still had not closed the gap to the OpenCL implementation. In the traces the SYCL has 90+ kernel arguments compared to the 25 of OpenCL. Also the SYCL kernels have around 30 events in their wait list which is an artefact from the out-of-order queue. All these events need to have completed before execution and the tracking of this could result in the overheads observed. This can be tested when in-order queues are implemented in SYCL, which, according to Ben, is in beta stages for Intel's compiler.

4.1.2 hipSYCL on GPUs

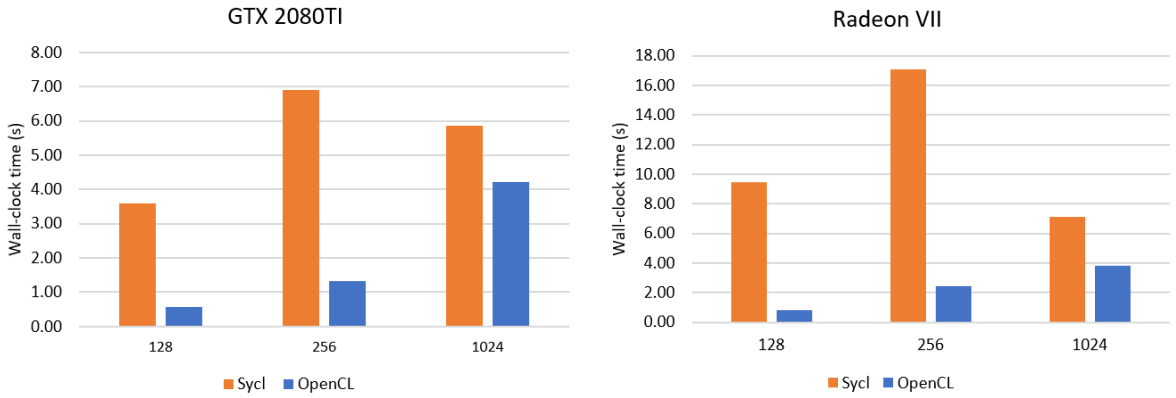

Figure 4.4: LBM Timings using hipSYCL on a Nvidia GPU (left) and an AMD GPU (right)

Initially, the results in Figure 4.4 are extremely confusing. I could not make sense of the fact that some of the smaller input sizes were taking longer than the larger ones. I contacted Aksel Alpay to help explain these results. Askel is the inventor of hipSYCL and as such the best person to understand them. From these discussions, some interesting information came to light. hipSYCL is based on top of CUDA when compiling for Nvidia GPUs and on top of ROCm when compiling for AMD GPUs. As such, in hipSYCL the out-of-order queue behaviour is manually implemented, since ROCm/CUDA only have in-order queues. This takes a noticeable amount of time. In ideal circumstances, if the kernel is empty with only a single accessor, it takes 0.075 ms to submit a kernel to the queue on a Radeon VII. For the smaller kernels in my code, the number of work items is very small, meaning that on powerful GPUs the runtime is limited by the kernel invocation latency. As OpenCL can use an out-of-order queue it does not suffer in the same way, hence the considerably faster runtimes on small inputs. This still did not explain the reason why smaller input sizes were taking longer than the larger ones. Armed with this new knowledge about kernel invocation times, I dug a little deeper into the code. The reason for this observation turned out to be an artefact from the original implementation of the code. Each problem size has an input parameters file. Delving into this you can see that the 128 problem is run 40,000 times, the 256 problem has 80,000 iterations and finally the 1024 problem has only 20,000. This explains why 256 has the worst relative performance and why the 128 code is slower than the 1024 code on the Radeon VII.

To double check this reasoning I constructed two new input sizes. 2048 and 4096. I made sure to construct the input and obstacle files in the same way as the originals, for both I fixed the iteration count at 5000. In Figure 4.5, you can see there is scaling as you expect, this is due to the larger sizes making the kernel running time dominate the invocation time. The input size of the 4096 problem is 4 times bigger than the 2048 problem as the problem is square. All of the programs take roughly 4 times longer except the SYCL code on the Radeon which only takes 3.3 times longer on the larger input. This suggests the overheads are still having an effect here. The running times are closer to the OpenCL running times, at 4096 both versions achieve 0.9 times the performance of the OpenCL version. This shows the two languages produce kernels which run at equivalent speeds and the only difference are the hipSYCL overheads, which become obvious on the smaller input sizes.

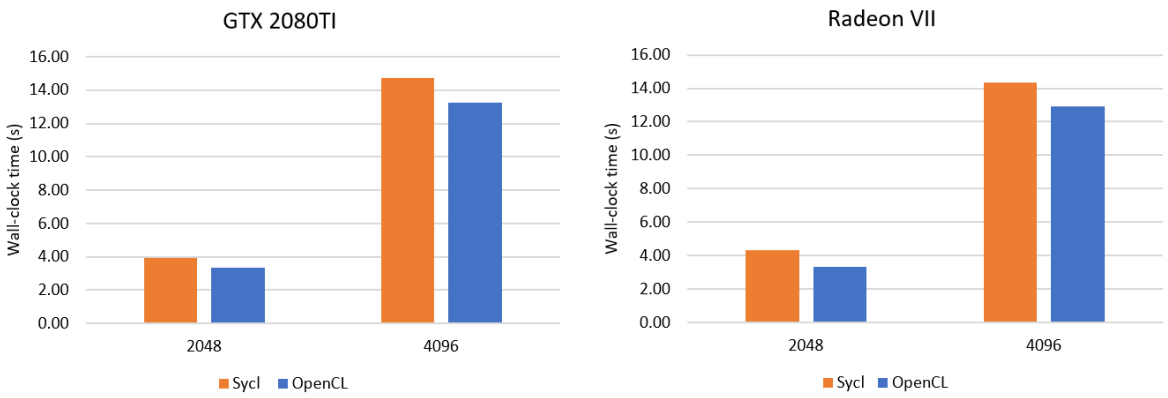

Figure 4.5: hipSYCL timings - LBM, Large Input Sizes

TeaLeaf comes with a set of CUDA kernels allowing for direct comparison between hand written CUDA and the CUDA produced by the hipSYCL compiler. For small input sizes the performance of hipSYCL is 0.16 times that of the CUDA implementation. This is expected due to the kernel invocation overheads just discussed. On Benchmark 5, a 4000x4000 grid, hipSYCL is achieving 0.63 times that of CUDA implementation. This is a lot better but there is still a considerable difference. I used the Nvidia `nvprof` profiler to profile both the codes on Benchmark 5. This allowed me to directly compare how long each kernel took to run. The main difference was in the time taken moving data around. CUDA took 0.08 seconds transferring data from device to host, whereas the hipSYCL version took 13.37 seconds. The data movement occurs after the reduction finishes so that the answer can be communicated back to the host. The hipSYCL code brings back the whole buffer to the host, in which the answer is contained. The buffer is the size of one workgroup, set at 64 for this example. CUDA moves a smaller amount of data around and thus the time is less. I tried creating another smaller buffer to be used to transfer the data back to the host but the creating and managing of this new buffer created more overheads than the reduced data movement saved. Another solution would be to use the `reinterprets` function on the original SYCL buffer to define a smaller buffer which could be transferred. This would reduce the overheads of having an entirely new buffer. However, it adds a significant layer of complexity to the code. As reductions are soon going to be present in SYCL, I decided to leave this problem open and allow it to be reassessed after reductions are standard. I predict that they will reduce this issue by providing an optimal solution as part of the SYCL standard.

Adding up the total time spent on the GPU for CUDA it comes to 135 seconds, this is compared with the total runtime of the program being 137 seconds. For SYCL, the time spent on the GPU is 154 seconds. This is 0.87 times the performance of the CUDA kernels. However, the total runtime of the program is 217 seconds. A quarter of the runtime is not being spent on the GPU. I then used the built-in TeaLeaf profiler to see if this could account for the remaining time. The TeaLeaf profiler accounted for all the runtime. The 'missing' time was spread out across all the results. The difference between the two profilers is that `nvprof` measures the amount of time taken on the GPU whereas the TeaLeaf profiler measures the whole execution time, this includes the SYCL setup overheads. As the time was spread out across all the kernels I concluded that although the GPU Kernel execution of CUDA and SYCL is only 10% different, the SYCL kernel invocation and queue management overheads dominate the difference in performance.

4.1.3 hipSYCL on CPUs

hipSYCL can target CPUs by using OpenMP as its back-end. This allows hipSYCL to target more architectures. I compiled my LBM code using hipSYCL specifying the target to be a CPU. I then tried to run the 128x128 problem on the Intel Skylake CPU on the HPC Zoo. After 10 minutes of runtime it still had not finished. This was a confusing result as the hand written OpenMP version finished the task in just over 1 second for this problem size. This was around the same time I virtually attended IWOCL and SYCLcon 2020. At this conference I watched a talk on hipSYCL from its founder Askel Alpay.

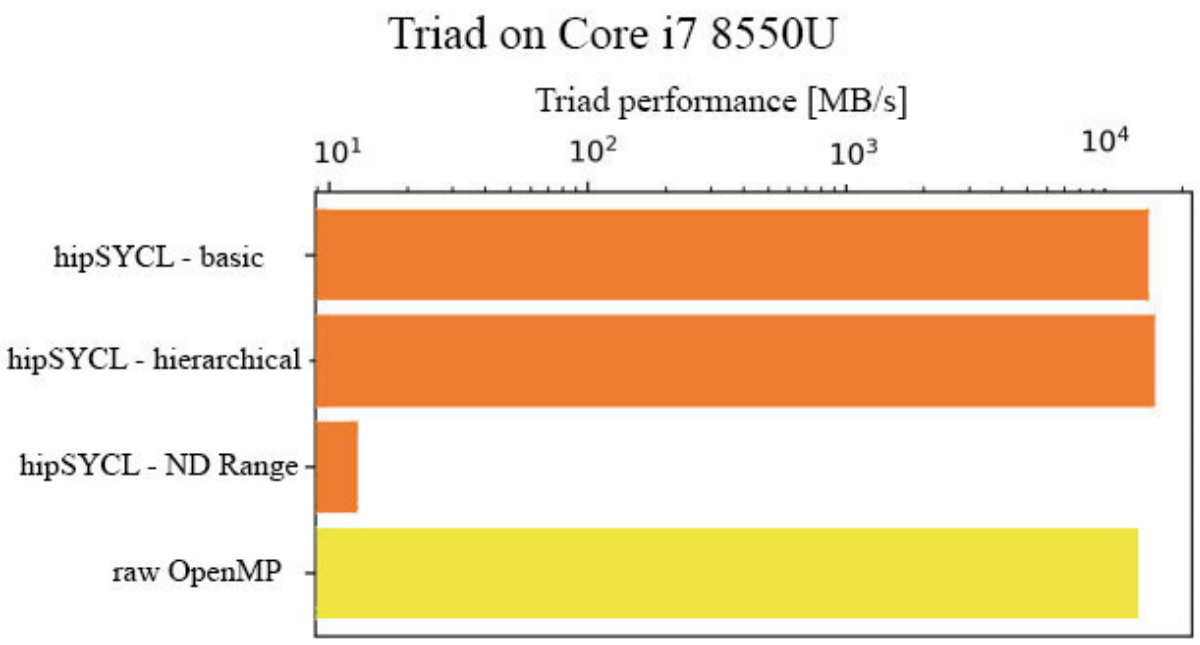

Figure 4.6: hipSYCL comaprison to OpenMP, SYCLcon 2020 [8]

Figure 4.6 was shown during his talk. It shows hipSYCL can match the performance of OpenMP in most cases except if a `ND-Range` construct is used. The `ND-Range` pattern allows for explicit barriers. For a barrier every work thread has to be guaranteed to hit the barrier. This means as many OpenMP threads as there are work items in the loop are needed. An extremely inefficient pattern for CPUs. The LBM kernel uses `ND-Range` kernels and thus this explains the poor performance observed.

TeaLeaf consists of many kernels, mostly just using standard `parallel for` loops. The only exception to this is all the reduction kernels are using an `ND-Range` construct. This means the hipSYCL for CPU implementation does not work efficiently for TeaLeaf either. On the Skylake CPU, Benchmark 1, a 10x10 grid, takes around 4500 seconds to run with hipSYCL OpenMP. This is in comparison to normal OpenMP which takes 0.5 seconds. To investigate this further I spoke to Askel Alpay, the creator of hipSYCL. The first suggestion was the ND Range kernel reduction could be changed to a hierarchical `parallel for` structure. This performs better with hipSYCL as shown in Figure 4.6. The reduction could be written in an optimal way for CPUs but this would lose the performance portability.

```
parallel_for_work_group([]{
  T mysum=0
  parallel_for_work_item([]{
     mysum += input[i] // This is only race-free on hipSYCL CPU backend
  });
  finalsum += mysum
})
```

Listing 4.1: An optimal reduction for hipSYCL's CPU Backend [7]

To make this solution performance portable you can check if you are using the hipSYCL CPU backend. If you are, you can use the code in Listing 4.1, if not you can use a standard reduction method as usual. I would have implemented a solution like this, however after discussion with Askel, a reduction mechanism is going to be added to hipSYCL soon. When this is added, this will be the optimal solution.

4.1.4 Reduction Methods

In the completed TeaLeaf port all the items to be reduced were stored in a temporary buffer. This was then passed to a reduction function, which would carry out the reduction and return the result. This uses an extra kernel invocation per reduction but is less verbose and means finding bugs is easier as all the reduction code is in one place. However, I was aware that this could carry a performance penalty so I investigated further. I rewrote all the reductions so that they were carried out in the kernel that calculated the results which needed reducing.

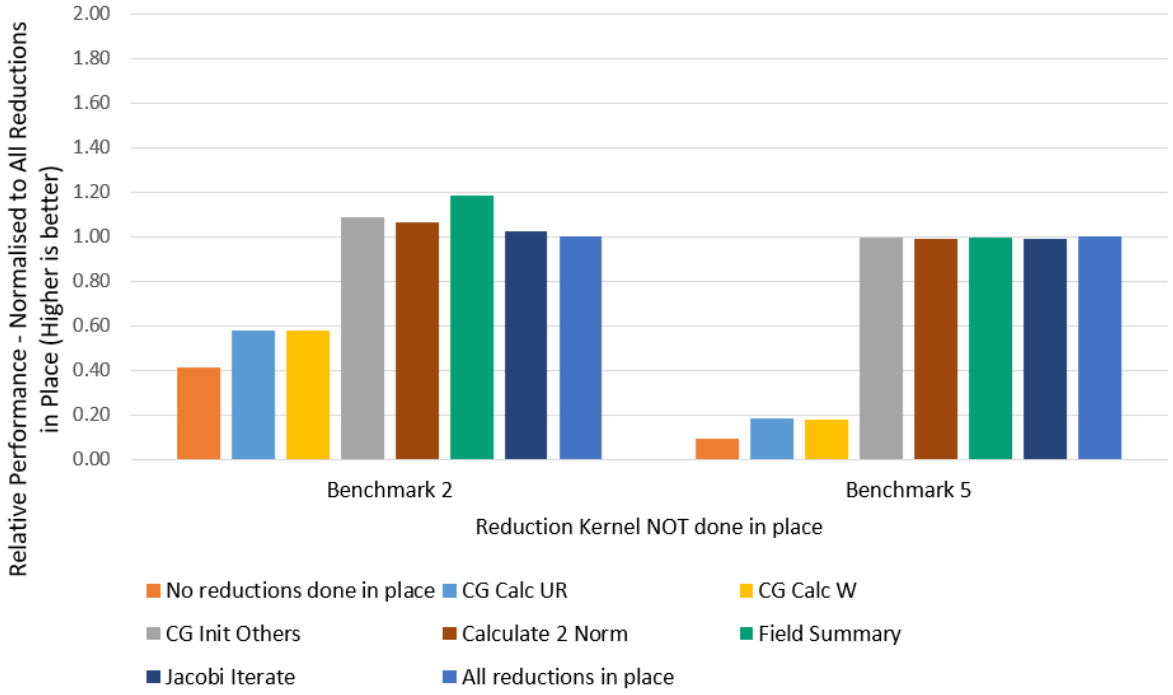

Figure 4.7: Comparing methods of carrying out the reductions in TeaLeaf

Figure 4.7 shows that when the reductions were performed in place, the runtime of the program was significantly shorter. For Benchmark 2, a 250x250 grid, doing the reductions in place gave 2.5 times speedup. For the larger 4000x4000 grid used in Benchmark 5, doing the reductions in place gave a 10.5 times speedup. The speedup for Benchmark 5 is much larger than that of the one for Benchmark 2. This is because for the small input the kernel invocation overheads are the limiting factor.

During the process of changing reduction methods I was regularly compiling and running my code to ensure that the output remained correct. It was from this process that I noticed that not all kernels benefited from this reduction change, a counter intuitive discovery. I decided to investigate this further. Figure 4.7 breaks down the runtimes by kernels not using the in place reduction method. Different kernels are run different amounts of times during one execution of the TeaLeaf program. As the benchmarks use the `CG` kernels, the `Jacobi Iterate` kernel is not run and as such the reduction method has no effect. The `CG Init Others` Kernel and the `Calculate2Norm` Kernels are run once per time step, for both the benchmarks there are 10 time steps. On the large benchmark no difference is seen from changing the reduction method but when the reduction are done in place on the smaller input the runtime increases by roughly 10%. This is counter intuitive as it seems to be the opposite to the overarching trend seen from doing the reductions in place. The `Field Summary` kernel is run twice per execution of TeaLeaf and exhibits the same trends. Doing `Field Summary` in place takes 20% longer on the small input size but has no difference on the large input. The larger input does not show the differences which the smaller input does in these cases, as the number of times the kernels are run are minimal and thus the small changes are dwarfed by the execution time of the rest of the program. For the smaller input the fact that doing the reductions in place is causing a slow down suggests that having one larger kernel is worse than two smaller kernels. Digging deeper into the pattern of execution for TeaLeaf it can be seen that in the case of all three kernels the reduction output is not used immediately. By splitting the kernel into two kernels, one doing any calculations and the other doing the reduction, a branch is created in the SYCL

dependency graph. Any kernels that do not need the reduction result can start immediately after the first kernel finishes and need not wait for the reduction to complete. This hides the time taken for the reduction by overlapping it with other work items. When the reductions are in place any subsequent kernels are forced to wait for both the calculations and reduction to occur and thus the runtime increases.

The `CG Calc UR` and `CG Calc W` kernels are run frequently during execution: 2345 times for Benchmark 2 and 42297 times for Benchmark 5. Doing the reductions in place in these kernels makes a huge difference, taking approximately 1/5th of the time of an execution where the reductions are handled by a singular function. This is what was expected, as changing these over reduced the number of kernel invocations and in turn reduced the dependency graph size leading to the faster runtimes.

4.2 Portability

Key
Architecture Supported
Experimental Support
No Support

	Iris Pro 580 (Intel Embedded GPU)	GTX 2080TI (NVIDIA GPU)	Radeon VII (AMD GPU)	NEC SX-Aurora (Vector Engine)	Intel Skylake (Intel CPU)	Ampere (ARM CPU)
Kokkos						
CUDA						
OpenMP						
OpenCL						
hipSYCL						
LLVM						
ComputeCPP						
SYCL Overall						

Figure 4.8: Portability of Different Languages

Figure 4.8 shows which languages can target which of the architectures on the HPC Zoo. It shows OpenMP and OpenCL have very good coverage. hipSYCL has the greatest portability of any SYCL implementation, being able to target both AMD and Nvidia GPUs as well as CPUs. Once the Nvidia support comes to LLVM SYCL and ComputeCPP they will be equal in terms of the number of devices from this sample which they can target. However, the key is to look at the SYCL ecosystem as a whole. In this case it has great portability, being able to target all the platforms except the vector engine and the ARM CPU. Targeting both architectures may become possible in SYCL down the line. This is because OpenMP can run on the NEC SX-Aururo and ARM CPU. hipSYCL can use OpenMP as a backend. At the moment linking these systems together is too complicated as the NEC only supports its NEC compilers, but in the future this could become possible by chaining together hipSYCL and the NEC compiler. The ARM CPU is in a similar situation and it is likely it will soon be possible to use hipSYCL to target this.

4.3 Performance Portability

Although portability is essential for modern programs, the most important statistic is the performance portability. If a program achieves a very low performance on a given architecture it could be seen to be of very limited use. To work out the performance portability I used the application efficiency and the Pennycook definition discussed in Section 2.5. The graphs in this document are plotted in the style of McIntosh et al [21] to aid comparison. The performance portability is plotted for all architectures in the set. As you move across the x-axis you remove the architecture that has the most results missing. I have plotted SYCL by taking the best times for a given platform from any of the available implementations. The architecture group used is the Intel Iris Pro 580, Nvidia GTX 2080TI, AMD Radeon VII and the Intel Skylake. This collection provides a good sample of architectures as it contains a CPU, GPUs from both major vendors and an embedded GPU.

4.3.1 Lattice Boltzmann

In this paper I have discussed how SYCL has overheads which become visible at small input sizes. To make the following results transparent I have calculated the performance portability results twice. Once

at a small input size and once at a large input size. The LLVM SYCL result is not present for the Skylake CPU as detailed in 3.1.2. As the ComputeCPP compiled version works on this platform, a result is still present for the SYCL language.

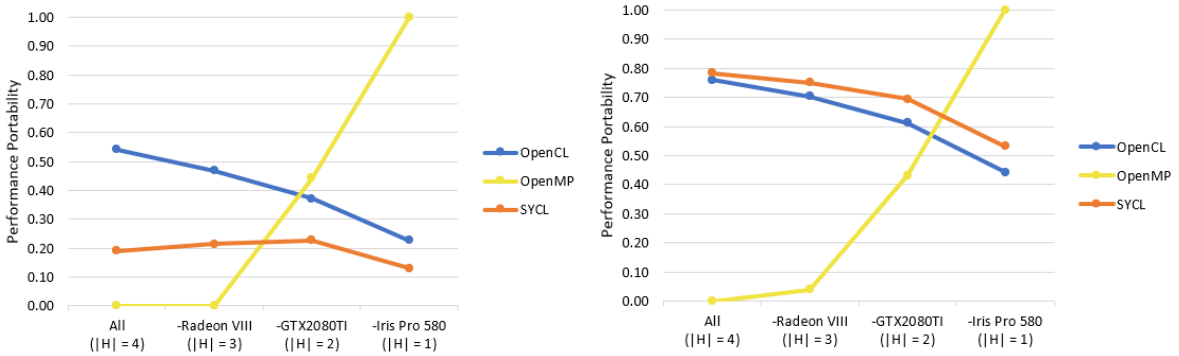

Figure 4.9: Performance Portability of the LBM Code on input size 256x256 (left) and input size 4096x4096 (right)

In Figure 4.9 it can be seen that at smaller input sizes the performance portability of SYCL is relatively poor, far below that of OpenCL. However, with the larger input size, which is more reflective of real world problems, SYCL is actually outperforming OpenCL in its performance portability. SYCL's $\Phi = 0.78$ and OpenCL's $\Phi = 0.76$ when considering all 4 architectures. In this case my research hypothesis is correct, as the SYCL performance portability is within 10% of the OpenCL performance portability, which is reasonable to class as similar. Initially, the fact that SYCL has a higher performance portability may seem counter intuitive as SYCL is designed on top of OpenCL. Delving deeper it can be seen that OpenCL is better than SYCL on all the hardware except the Intel Skylake CPU. This is due to the OpenCL implementation being hand tuned to target GPUs whereas the SYCL runtime can change how it calls the underlying OpenCL APIs based on the hardware it is targeting. At large input sizes, using ComputeCPP, SYCL is achieving 1.2 times the performance of the OpenCL version. Looking at the graph again, this can be inferred by the fact that the OpenCL and SYCL lines diverge as the number of platforms decreases, ultimately ending at just the Skylake CPU. This highlights an important point that a more performance portable program does not necessarily mean that it is faster. On observing this graph a user may pick SYCL as their language of choice. This would be a poor choice given the fact that OpenCL is faster on all the platforms except the Skylake. However, on the Skylake both OpenCL and SYCL have poor performance, 0.53 times OpenMP and 0.44 times OpenMP respectively. This one data point shifts the performance portability and can lead to a completely different impression.

4.3.2 TeaLeaf

For the TeaLeaf results, the Iris Pro 580 is swapped out for the Intel UHD P630 as discussed in Section 3.3. Also discussed in Section 3.3 is the lack of results for the ComputeCPP compiler on the Intel UHD P630. On the Iris Pro 580, ComputeCPP outperformed LLVM on a previous version of the TeaLeaf code with both Benchmark 2 and 3. From this, it is expected that ComputeCPP would be comparable if not better than LLVM on the Intel UHD P630. Therefore the SYCL performance portability would likely increase if this result was obtained. As detailed in Section 3.2.2, the result for the Skylake CPU on Input 5 is missing.

Figure 4.10 shows OpenCL dominating the other implementations. For Benchmark 2 it is the fastest version on every platform except the CPU, where it achieves 0.78 times the performance of Kokkos. On Benchmark 5, using the Radeon VII, the performance of OpenCL is worse than that of hipSYCL. SYCL has a very low but consistent performance portability on the smaller input size, $\Phi \approx 0.15$. This is where the execution time of SYCL is heavily dominated by setup overheads. As shown in the previous results, the performance of SYCL compared to other implementations increases as the input size becomes larger. On Benchmark 5, SYCL achieves the highest performance on the Radeon VII out of any tested implementation. This is what gives the SYCL line its peak. However once this result is excluded, the performance portability falls with SYCL achieving 0.8 times the performance of OpenCL and 0.6 times

the performance of OpenCL on the GTX 2080TI. The result on the GTX 2080TI is surprisingly low and has been discussed in Section 4.1.2. Once the SYCL implementation issue is fixed with large inputs on the Skylake, SYCL will have good performance portability across all the hardware in the set.

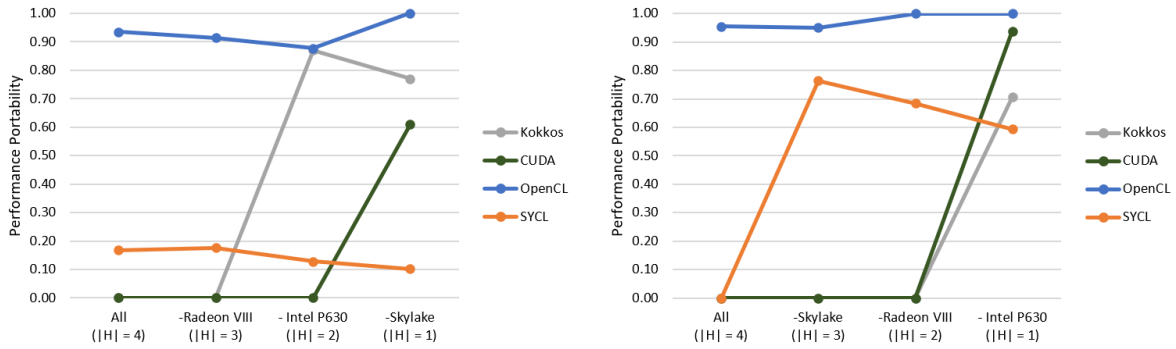

Figure 4.10: Performance Portability of the TeaLeaf Code on Benchmark 2 (left) and Benchmark 5 (right)

Comparing these results to the ones from the Lattice Boltzmann implementation allows for an interesting discussion. Overall, the performance portability of SYCL is worse on TeaLeaf than on LBM. TeaLeaf is over three times the size of the LBM code and as such is much more complex. It has many different types of kernels compared with just one large one in LBM. This complexity appears to degrade the performance of SYCL. It is likely to be due to SYCL having to handle the memory movement and ordering of kernels. Event tracking will take time and thus be an overhead of using SYCL. The one result that stands out is that SYCL is the fastest implementation on the Radeon VII. I believe that this is because hipSYCL is using the AMD ROCm platform not OpenCL as the backend. AMD ROCm is designed specifically for AMD GPUs (such as the RadeonVII) and in this case this is allowing for a greater acceleration. For the three architectures left after removing the Skylake, SYCL's $\Phi = 0.76$ and OpenCL's $\Phi = 0.91$. These are not within 10% of each other as I originally hypothesised.

4.3.3 Compilers

In this project three SYCL implementations have been used: LLVM SYCL, ComputeCPP and hipSYCL.

Key
Architecture Supported
No Support

Hardware	Iris Pro 580	GTX 2080TI	AMD Radeon VII	Intel Skylake
LBM				
OpenCL	1.00	1.00	1.00	0.83
hipSYCL		0.90	0.90	
LLVM	1.00			
ComputeCPP	0.98			1.00

Figure 4.11: Application Efficiency of LBM implementations on input size 4096

Figure 4.11 shows how the SYCL overhead compared to OpenCL is minimal, even outperforming OpenCL on the Intel CPU. This suggests that the SYCL implementations are able to tune their output to the target architecture more easily than OpenCL, which was hand tuned for GPUs. On the Nvidia and AMD GPUs the performance of SYCL is around 10% worse than OpenCL but in these cases SYCL is using CUDA and ROCm respectively so this is not a fair comparison. An important note is that hipSYCL does run on Intel Skylake CPU but due to the ND Range issue, discussed in Section 4.1.3, it is extremely slow and I was not able to get a time for it. No time for LLVM on the Skylake was obtained due to issues discussed in 3.1.2. Each compiler can target two out of the four platforms. As such, I have calculated the performance portability for each compiler on its two platforms. Φ(hipSYCL) = 0.90 and Φ(ComputeCPP) = 0.99. LLVM does not get a result due to the lack of time on the Skylake.

Figure 4.12 shows the performance of the SYCL implementations of the TeaLeaf mini-app. As detailed in Section 3.2.2, the result for the Skylake CPU on Input 5 is missing. For the two platforms hipSYCL can target, its $\Phi = 0.75$. The scores for LLVM and ComputeCPP cannot be calculated due to the lack

Key
Architecture Supported
No Support

Hardware	Intel UHD P630	GTX 2080TI	AMD Radeon VII	Intel Skylake
TeaLeaf				
Kokkos	0.00	0.71	0.00	1.00
OpenCL	1.00	1.00	0.86	0.97
hipSYCL		0.59	1.00	0.00
LLVM	0.81			
ComputeCPP				

Figure 4.12: Application Efficiency of TeaLeaf implementations for Benchmark 5

of results. In this project I have managed to obtain more times using hipSYCL on both the LBM and TeaLeaf codes than with the other compilers. This means hipSYCL is the most portable implementation used in this project. I cannot do a performance portability comparison of the three compilers due to the fact they operate on different platforms. However, I can draw the conclusion that the performance portability of LLVM and ComputeCPP are very similar. They target the same platforms and in all tests they had a maximum difference in runtime of 5%. At this stage in the development of SYCL, no singular compiler provides all the benefits of SYCL. To reap the full benefits of SYCL, hipSYCL plus either LLVM or ComputeCPP needs to be used.

4.4 Verbosity of SYCL

The purpose of this section is to compare SYCL to other programming models in terms of the number of lines of code needed to express the same program. I used TeaLeaf to do this where the host code could be ignored and only the kernels analysed. To count the lines of code the Linux command line command `wc` was used for each set of kernels in a given language. The comments were included in this count as they are vital in explaining what is happening in a mini-app. An important note here is that even though a language is less verbose it does not mean that it is quicker to write. For example, in a lot of the low level parallel languages, a large section of the code is boilerplate. This means that the amount of code is large but when writing a program in that language you can often copy and paste chunks from past programs.

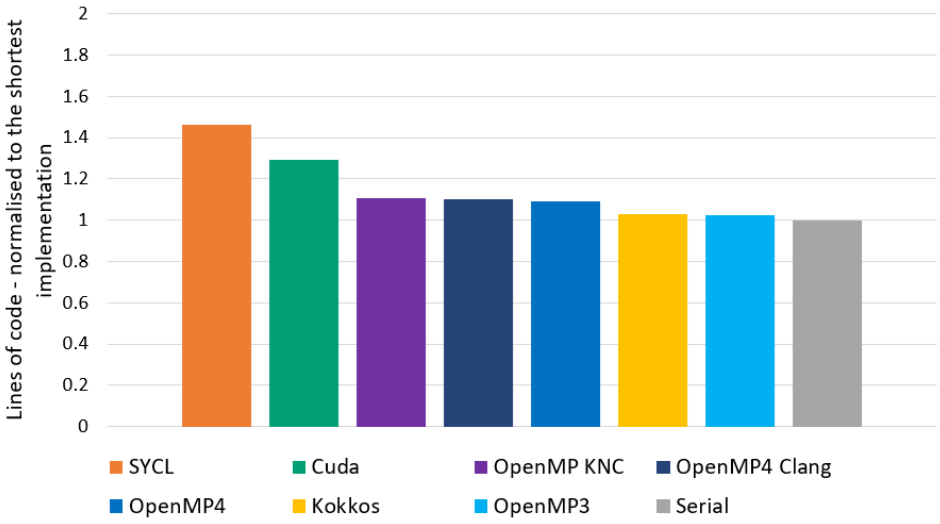

Figure 4.13: Lines of code used in the TeaLeaf kernels, given for each language.

From Figure 4.13 it can be seen that the SYCL kernel has the most lines of code. There is not much difference in length between most of the kernels as they are high level abstraction languages. One of the main reasons SYCL is longer than the other high level languages is due to SYCL's reliance on `accessors`. Every kernel needs a list of `accessors` for its required `buffers`. These are responsible for a large number of lines in the program. It is surprising that the SYCL code is longer than the low level language CUDA. A reason for this could be that the SYCL reductions used are very verbose and a lot of the code is repeated. CUDA has a centralised reduction function which reduces the number of lines of code required.

Chapter 5

Conclusion

5.1 Achievements and Contributions

The main achievements and contributions of my project are listed below:

- Successfully ported the Lattice Boltzmann code to SYCL (710 lines of code)
- Successfully ported the TeaLeaf code to SYCL (2267 lines of code)
- Investigated the overheads in SYCL with the help of Intel and highlighted the in-order versus out-of-order queue issues. This was something that the Bristol HPC group was unaware of and would not have expected to be so detrimental to performance.
- Carried out 477 tests, having a combined total runtime of over 25 hours. These have all been logged and stored in my database.
- Identified bugs in both the University of Bristol TeaLeaf repository and the UK Mini-App Consortium's repository for TeaLeaf implemented in OpenCL. I fixed them and shared the code with the respective owners by the means of a pull request.
- Quantitatively assessed the performance of SYCL showing that SYCL can achieve high performance portability scores similar to OpenCL, but as the program size and complexity grows the performance and thus the performance portability of SYCL falls.
- Produced a guide, with examples, to aid other developers to port their codes to SYCL

The ported codes, timings database and guide to porting SYCL are freely available from: https://github.com/WSJHawkins/ExploringSycl.

5.2 Project Status

My original project aim was to quantitatively assess the performance portability of the new programming language SYCL and to compare and contrast its different implementations. I specified that I would do this by completing the following objectives:

1. To port a computational fluid dynamics(CFD) code which uses the Lattice Boltzmann methods (LBM) to the language SYCL.
2. To port Tealeaf [22], a mini-app representing iterative sparse linear solvers, to the language SYCL.
3. To rigorously test both Tealeaf and the LBM codes against their OpenCL counterparts to measure the overheads of using the SYCL language.
4. To apply the performance portability metrics defined by Pennycook et al [25] and demonstrated by McIntosh-Smith et al. [21] to measure the performance portability of SYCL across a diverse range of architectures.

5. To test the current implementations of SYCL and assess which is the most performance portable.

Objectives 1 and 2 have been completed. Both the LBM and TeaLeaf codes have working SYCL versions which are available from https://github.com/WSJHawkins/ExploringSycl.

For Objective 3, I have used the LBM code to assess the overheads of going through SYCL compared with writing the code directly in OpenCL. I have found at large problem sizes (more reflective of real world usage), the overheads are very minimal. At small input sizes there are some overheads, mainly due to the out-of-order queuing mechanism. These results can be found in Section 4.1.1. SYCL is looking at adding in-order queue support and hopefully having raised the issue, this overhead will be looked at and resolved in future implementations.

For Objective 4, I have successfully tested and analysed the performance portability of both the SYCL and TeaLeaf codes for both small and large input sizes. To do this I used the portability metrics defined by Pennycook et al [25] and plotted my graphs in the style of McIntosh-Smith et al. [21]. I have shown that on programs with a simple structure SYCL can achieve comparable performance portability to OpenCL. However, as the program size and complexity grows the performance and thus the performance portability of SYCL falls. This can be found in Section 4.3.

For Objective 5, I have been using the three most mature SYCL implementations throughout my project. I have found that to reap the full benefits of the SYCL language, hipSYCL plus either LLVM or ComputeCPP needs to be used. The performance portability cannot directly be used to compare the implementations as they target different platforms. This can be seen in Section 4.3.3. I expect these results to change when the Intel LLVM Compiler and Codeplay's ComputeCPP compiler have stable support for Nvidia GPUs. This will boost the portability of those implementations and allow for direct comparison with hipSYCL. Once the reductions are present in SYCL and in particular hipSYCL, I expect the performance of hipSYCL on CPUs to dramatically increase. This is due to the issues discussed in Section 4.1.3 being resolved. I believe that hipSYCL will then outperform ComputeCPP and LLVM on CPUs. This is due to OpenMP executing faster than OpenCL in my results and hipSYCL using OpenMP on CPUs.

My original project aim was to quantitatively assess the performance portability of the new programming language SYCL and to compare and contrast its different implementations. By completing all of the objectives, this aim has been achieved.

My research hypothesis is that implementations using the SYCL programming model will have a performance portability score similar to that of implementations in OpenCL has turned out to be false. Although this does occur in some cases as shown with the SYCL LBM implementation (where SYCL actually had the higher performance portability), it does not happen in all cases. With larger more complex implementations the performance portability score of SYCL falls away from the OpenCL score.

5.3 Future Work

SYCL is a quickly evolving standard and from taking part in IWOCL and SYCLcon 2020, I have seen lots of improvements which are to be implemented soon. The one I am most excited about is a reduction construct within SYCL. I have felt the need for this commonly used pattern to be included in the standard throughout my project. This will reduce the verbosity of the code and ensure that the reduction is undertaken in the most efficient way. This will lead to greater performance portability in implementations such as hipSYCL, where I have seen that the reductions are causing a sub-optimal performance on CPUs. Another development within the SYCL implementation that I believe will be of great benefit, is the fact that the standard is opening up to allow other back-ends. Currently it is designed to only use OpenCL as the back-end but allowing for other back-ends will help SYCL combat adoption friction. Therefore it will be able to run on more devices without the underlying devices having to explicitly support SYCL or OpenCL. This is what hipSYCL has been doing and during this project I have seen that it has been successfully adding to the performance portability of the SYCL ecosystem.

As a future project for myself or someone else, I would suggest that the following tasks are undertaken:

1. The issue regarding the SYCL version of the TeaLeaf mini-app on the Intel Skylake should be investigated. Work should be undertaken to understand what is causing the code to produce the wrong output. Once this is understood a fix should be applied.

2. Once reductions are added to the standard and supported in the implementations they should be added to the LBM and TeaLeaf codes.

3. LBM and TeaLeaf should then be tested on a range of hardware, this should include running executables from the LLVM/SYCL and ComputeCPP compilers on Nvidia GPUs. hipSYCL's performance on CPUs should also be tested.

4. The performance portability should be recalculated and compared to the results in this project.

5. Further investigation should take place into the effect of in-order versus out-out-order queues. Once SYCL implementations support in-order queues the results from this project should be re run to see any differences.

I predict that the results of these will show that the gaps between SYCL and the underlying back-ends will reduce. This means SYCL will become more performance portable and therefore a more attractive language choice for future applications.

5.4 Recommendations

As I have gone through this project I have made notes on things that have not worked as expected and things that I thought should be simple but were not. Fortunately, the announcements at SYCLcon showed other developers had had the same thoughts and as such many of the issue are being fixed in the next version of SYCL, SYCL 2020. However, there are two things that I have not seen mentioned and as such I wish to highlight them in this report:

- When I was starting out with SYCL I found it confusing that only one `parallel for` was allowed per `queue submit`. This would have been fine but when I made the mistake of having multiple `parallel for` constructs in on `queue submit` the code still ran, just with the incorrect answer. I believe that an error should be thrown to raise attention to this issue.

- In my LBM code the same kernel is used every iteration, although the input buffers are swapped with the output buffers. This is a common pattern as it reduces the amount of memory movement and thus increases performance. As this is a common pattern I feel there should be a built in way of executing this.

5.5 Reflection

As Einstein said "Pure logical thinking cannot yield us any knowledge of the empirical world; all knowledge of reality starts from experience and ends in it" [13]. From undertaking this project I have greatly increased my knowledge of High Performance Computing and in particular SYCL.

If I were able to do this project again, I would use my experience to approach some tasks in a different manner. During bug fixing, I would spend more time hands on, trying to build simple examples that replicate the bug. This proved to be more effective than sitting thinking of what could be the cause. In this project having a enquiring mind helped drive success. Results were questioned to understand why they had occurred. Testing these theories experimentally helped to consolidate knowledge.

Reflecting on the project journey, I have thoroughly enjoyed taking my interest in High Performance Computing further. This is not to say that I did not experience difficulties but to say that the successes far outweighed them. Gaining and applying this knowledge allowed for a true sense of accomplishment. Looking forward I hope that my research will contribute to the progression of HPC and in particular the SYCL ecosystem.

5.6 Concluding Remarks

I have developed a keen interest in SYCL and will definitely be keeping up to date with it as it matures. From my experiences and from listening to the many talks at SYCLcon 2020, I believe that this project has shown that SYCL is a promising new language already achieving good results. I predict that within five years SYCL will be as performance portable, if not more performance portable than OpenCL for all sizes and complexities of program. This will make it a key player in the future of High Performance Computing.

Bibliography

[1] Coblis — Color Blindness Simulator.

[2] Introduction to SYCL - Tech.io.

[3] Khronos Group's Website.

[4] Kokkos Tutorials - GitHub.

[5] SYCL Academy.

[6] Top500 Website.

[7] Aksel Alpay. hipSYCL, 2020.

[8] Aksel Alpay and Vincent Heuveline. SYCL beyond OpenCL: The architecture, current state and future direction of hipSYCL, 2020.

[9] Krste Asanovic, Ras Bodik, Bryan Catanzaro, Joseph Gebis, Parry Husbands, Kurt Keutzer, David Patterson, William Plishker, John Shalf, Samuel Williams, and Katherine Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. *EECS Department, University of California, Berkeley*, EECS-2006-183, 12 2006.

[10] Gordon Brown and Michael Wong. A Modern C++ Programming Model for GPUs using Khronos SYCL.

[11] Phil Colella. Defining Software Requirements for Scientific Computing, 2004.

[12] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[13] A. Einstein. *Ideas and opinions*. Bonanza Books, 1954.

[14] Bristol HPC Group. Bristol HPC Zoo.

[15] The Khronos Group. Overview of SYCL ComputeCPP.

[16] William Hawkins. Porting an implementation of Lattice Boltzmann to a GPU using the OpenCL language. 2019.

[17] Intel. Intel OneAPI DevCloud.

[18] Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. GPUs and the Future of Parallel Computing. 2011.

[19] WeiChen Lin. CloverLeaf SYCL port. 2019.

[20] Simon McIntosh-Smith and Tom Deakin. Hands On OpenCL.

[21] Simon McIntosh-Smith, Tom Deakin, James Price, Andrei Poenaru, Patrick Atkinson, Codrin Popa, and Justin Salmon. Performance Portability across Diverse Computer Architectures. 2019.

[22] Simon McIntosh-Smith, Matthew Martineau, Tom Deakin, Grzegorz Pawelczak, Wayne Gaudin, Paul Garrett, Wei Liu, Richard Smedley-Stevenson, and David Beckingsale. TeaLeaf: A Mini-Application to Enable Design-Space Explorations for Iterative Sparse Linear Solvers. 2017.

[23] Roberto Mijat. Smile to the camera, it's OpenCL!, 2015.

[24] Gordon E. Moore. Cramming more components onto integrated circuits. 1965.

[25] S.J. Pennycook, J.D. Sewall, and V.W. Lee. A Metric for Performance Portability. 2016.

[26] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. 2009.

[27] M. Mitchell Waldrop. The chips are down for Moore's law. 2016.

[28] Max Wertheimer. Gestalt theory. 1938.

[29] Tiffany Trader HPC Wire. GPUs Power Five of World's Top Seven Supercomputers. 2018.