Dissertation Type: research

DEPARTMENT OF COMPUTER SCIENCE

A Performance Evaluation of the SYCL Programming Model

Andrei Nițu

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.

September 2, 2020

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of MEng in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Andrei Nițu, September 2, 2020

Contents

List of Figures

List of Tables

List of Listings

Executive Summary

This project aims to investigate the performance portability of SYCL, a relatively new parallel programming model developed and maintained by the Khronos consortium. SYCL is built on top of the low-level abstraction layer of OpenCL and based on pure C++, enabling developers to write single source code for heterogeneous processors while being able to use standard ISO C++11 features, such as inheritance, templating and namespaces.

Since it was first announced and released in 2014, SYCL has slowly gained popularity as more implementations (both open source and commercial) have been released, reaching a level of support that is now close to that of other well-established parallel programming models. SYCL applications can be run on a broad range of hardware platforms, such as mobile, desktop and HPC-grade CPUs, discrete and integrated GPUs or FPGAs.

The research hypothesis is that, as a high-level heterogeneous model, SYCL can achieve comparable performance to OpenCL, whilst also offering the convenience of using C++ abstractions and a less verbose single source style, therefore lowering the overall complexity of heterogeneous programming in comparison to other similar models. To test this hypothesis, two scientific applications with different computational characteristics were selected: a structured grid lattice-boltzmann fluid flow simulation code and a Monte Carlo neutron particle transport simulation code. The study involved porting those two applications to SYCL and benchmarking them on various platforms and compilers in order to examine their performance.

The main contributions of this project are as follows:

- I implemented a SYCL port of the lattice-boltzmann method for CFD simulation.
- I implemented a SYCL port of the `neutral` mini-app for particle transport simulation.
- I assessed the performance of three SYCL implementations on a wide range of hardware platforms and presented a detailed comparative analysis between SYCL and two other similar models: OpenCL and Kokkos.

Supporting Technologies

This section outlines the third-party resources that were used during this project.

- I used the following compilers for building SYCL codes:
  - hipSYCL [1]
  - Codeplay's ComputeCpp compiler [2]
  - Intel's SYCL compiler [3]
- I used the Khronos OpenCL API headers [4] and OpenCL ICD Loader [5] for building and running OpenCL codes.
- I used the open-source Kokkos Core implementation (release 2.8.00) [6] for building and running Kokkos codes.
- I used the Intercept Layer for OpenCL [7] for debugging my code and for performance evaluation.
- I used several existing implementations of the `neutral` mini-app [8,9].
- University's of Bristol HPC Zoo[10] provided the aforementioned compilation platforms along with a selection of hardware devices and was used for running different implementations of the `neutral` mini-app and the lattice-boltzmann code.

[1] https://github.com/illuhad/hipSYCL
[2] https://developer.codeplay.com/products/computecpp/ce/home/
[3] https://github.com/intel/llvm/tree/sycl
[4] https://github.com/KhronosGroup/OpenCL-Headers
[5] https://github.com/KhronosGroup/OpenCL-ICD-Loader
[6] https://github.com/kokkos/kokkos/releases/tag/2.8.00
[7] https://github.com/intel/opencl-intercept-layer
[8] https://github.com/UoB-HPC/neutral
[9] https://github.com/UoB-HPC/neutral_kokkos
[10] https://uob-hpc.github.io/zoo/

Notation and Acronyms

ALU	:	Arithmetic Logic Unit
API	:	Application Programming Interface
AVX	:	Advanced Vector Extensions
BW	:	Memory Bandwidth
CFD	:	Computational Fluid Dynamics
CGH	:	Command Group Handler
CPU	:	Central Processing Unit
CU	:	Compute Unit
DAG	:	Directed Acyclic Graph
DLP	:	Data-Level Parallelism
DPC++	:	Data Parallel C++
GPGPU	:	General-Purpose Graphics Processing Unit
GPU	:	Graphics Processing Unit
FLOPS	:	Floating Point Operations Per Second
FPGA	:	Field Programmable Gate Array
HPC	:	High Performance Computing
ICD	:	Installable Client Driver
ILP	:	Instruction-Level Parallelism
ISA	:	Instruction Set Architecture
LBM	:	Lattice-Boltzmann Method
LSU	:	Load/Store Unit
MOSFET	:	Metal–oxide–semiconductor field-effect transistor
OI	:	Operational Intensity
PCI-e	:	Peripheral Component Interconnect Express
PE	:	Processing Element
PTX	:	Parallel Thread Execution
ROCm	:	Radeon Open Compute
SIMD	:	Single Instruction Multiple Data
SM	:	Streaming Multiprocessor
SMCP	:	Single-source multiple compiler-passes
SPIR	:	Standard Portable Intermediate Representation
SVE	:	Scalable Vector Extension
VE	:	Vector Engine

Acknowledgements

I would like to thank my supervisor, Prof. Simon McIntosh-Smith, for his excellent inputs and consistent support throughout the project and to Tom Deakin for sharing his extensive knowledge on SYCL. I would also like to thank my parents for all their continuous support.

CHAPTER 1

Contextual Background

1.1 High Performance Systems

High Performance Computing (HPC) is a field that studies the optimal application of supercomputers (a class of extremely powerful computers) for solving computationally intensive problems that would otherwise take too long to run on ordinary computers. HPC systems are primarily used for complex engineering and scientific simulations requiring vast amounts of data processing in areas like physics, chemistry or biology.

For supercomputers to achieve those high processing power requirements, they are typically constructed as *clusters* of numerous *compute nodes* (also called blades) linked together by a high bandwidth and low latency *interconnect* [5] (such as InfiniBand or Ethernet) which allows the compute nodes to behave as a single large system. Each node contains several CPU sockets (usually between one and four) and each socket houses a multi-core server-grade processor with associated RAM. Some of the nodes are also equipped with additional memory, while others have access to *hardware accelerators* (specialised components designed to perform specific functions more efficiently), such as Graphics Processing Units (GPUs), Vector Engines (VEs) or Field Programmable Gate Arrays (FPGAs).

Consequently, a supercomputer is able to provide a level of performance that would not be achievable in desktop-grade computers. For example, the main Arm system of GW4's Tier 2 HPC service, Isambard [6], has 164 nodes, each with two 32-core Marvell ThunderX2 CPUs and 256 GB of DDR4-2666 memory. The nodes are connected via a Cray Aries interconnect with a Dragonfly topology and a Cray Sonexion 3000 storage cabinet provides 480 terabytes of Lustre storage.

1.1.1 Exascale Computing

The fastest computing systems in the world are currently able to solve problems at petascale [7]; that is 1 quadrillion floating point operations per second (FLOPS). While these petascale systems are already quite powerful, one of the most important milestones in the HPC community is building a system that reaches exascale (1 exaFLOPS) capabilities.

What kind of architecture is best suited for exascale computing remains an open ended question [8] and, to this end, the choices in both software (i.e. parallel programming languages and frameworks) and hardware (i.e. processors, accelerators and network) among different HPC services are increasingly multifaceted.

1.2 Performance Portability

Performance portability describes the ability of applications to perform well on a range of different target device platforms, with minimal development and maintenance costs; it is an increasingly important requirement, given the high degree of hardware diversity that is resulting from the pursuit of exascale computing.

Even though there is no universally accepted definition of a "performance portable" program, the following loose traits are generally agreed as being desirable [9, 10, 11]:

- it can achieve some notional level of performance on a broad range of hardware platforms (leveraging architecture-specific features where possible).
- it can run with acceptable performance using a single version of source code or a substantial amount of shared code.
- the code is more maintainable and allows greater productivity than platform specific variants.

It's important to note that *portability* and *performance portability* are not the same: an application may compile and run on multiple architectures, but not be scalable to several thousands of concurrent threads on GPUs.

A rigorous and objective definition of performance portability was proposed by Pennycook et al. in 2016: "A measurement of an application's performance efficiency for a given problem that can be executed correctly on all platforms in a given set" [12]. Along with this formal definition, a metric for the performance portability $\varphi(a, p, H)$ of an application a solving problem p, for a given set of platforms H was also presented (Figure 1.1). The equation is based on the harmonic mean of an application's performance efficiency observed across a set of platforms.

$$\varphi(a,p,H) = \begin{cases} \dfrac{|H|}{\sum_{i \in H} \dfrac{1}{e_i(a,p)}}, & \forall i \in H, i \text{ is supported} \\ 0, & \text{otherwise} \end{cases}$$

Figure 1.1: The Pennycook performance portability metric, where $e_i(a, p)$ represents the performance efficiency of application a solving problem p on platform i.

1.3 Performance Evaluation

Performance evaluation (or performance analysis) is a crucial part in the development of any software application, especially in the field of HPC. Running the same application on multiple device platforms often results in a certain degree of performance variability, given the differences in architecture. Furthermore, processor vendors generally design their products by focusing on a specific kind of software applications.

One of the main objectives of performance analysis is to identify *bottlenecks* — inefficient code sections that consume the majority of the execution time of the program. This concept can be looked at from two different viewpoints:

1. Optimising a particular software application, given a fixed hardware platform to run it on. This scenario is very common in software engineering.
2. Selecting the hardware platform that yields the maximum performance, given a fixed application that needs to be run. This is vital for a scenario where — for instance — a research centre or

laboratory is considering building a new supercomputer and has to make a choice in regards to the best architecture for their desired target workload, especially given the large budgets involved in such projects (potentially tens or even hundreds of millions of dollars).

1.3.1 Profilers

The performance issues of applications are commonly diagnosed using *profiler tools*, which work by collecting data and presenting it though an interface that simplifies its analysis and interpretation. For example, the Intel VTune Profiler can generate reports on the percentage of usage taken by a function, module or individual code statement, multithreaded performance, achieved bandwidth and even suggest potential improvements [13].

1.3.2 Roofline Analysis

A roofline model is a conceptual tool that can be used to determine the limiting factor of an application's performance [14]. It is based on the *operational intensity* (OI) of a code, which is defined as the ratio between the number of operations performed and the number of bytes transferred between the last level of cache and main memory. The operational intensity can be visualised as a vertical line that "hits" either the flat section of roof or the oblique section of the roof. If it first intersects the flat part then the code is *compute bound* and if it first intersects the oblique part then the code is *memory bandwidth bound*.

Figure 1.2 shows an example roofline model for a fluid flow simulation code that is memory bandwidth bound.

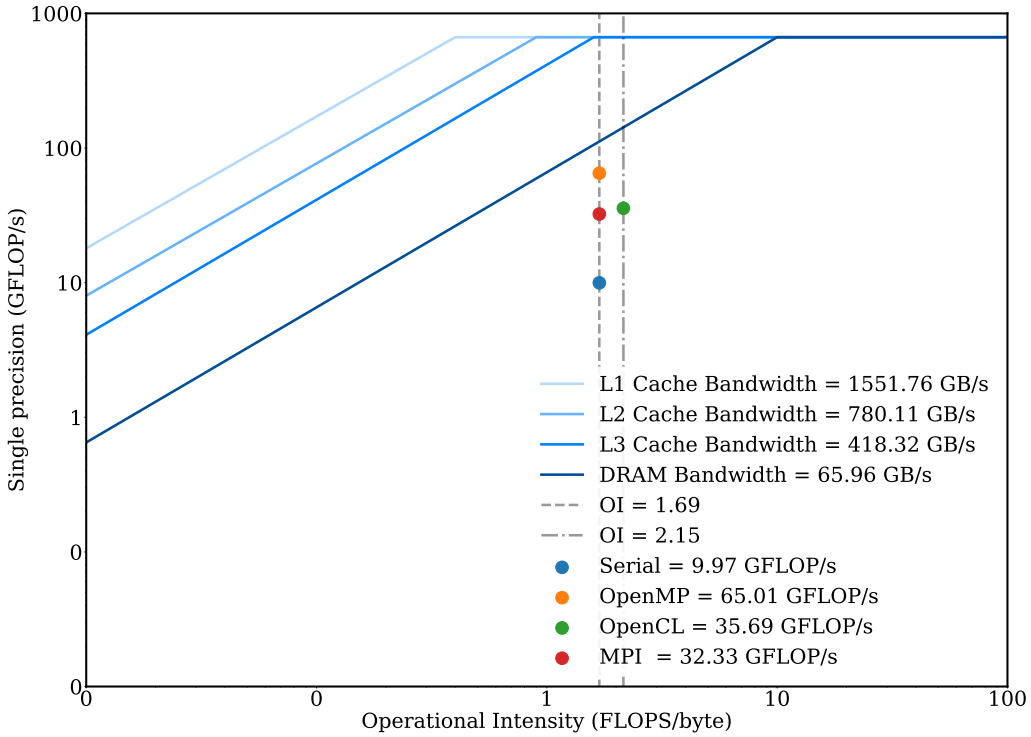


Figure 1.2: Roofline model for implementations of a code running on a Xeon E5-2670 CPU.

1.3.3 Synthetic Benchmarks

Besides profiling reports and roofline analysis, performance can also be measured using *benchmarks*.

Synthetic benchmarks are artificial applications that are constructed to try to match the characteristics of a broad range of programs and give an indicative measure of the performance capabilities of the machine being tested. For instance, `BabelStream` [15] can measure the attainable memory bandwidth of GPUs (i.e. the memory transfer rates to and from global device memory on GPUs) when using parallel programming models.

The main drawback of synthetic benchmarks is that they are, by definition, unrealistic. Such a program would not be run by an average user as an application because it does not actually compute anything useful. Moreover, they usually do not reflect the more unpredictable patterns exhibited by a real program and, in a lot of cases, compiler optimisations inflate their efficiency and exceed what the same optimisations would achieve on average applications.

1.3.4 Proxy Applications

Another class of benchmarks extensively used in high performance computing are *application benchmarks*, more commonly called proxy applications ("proxy apps"). They are relatively small and basic codes that represent the salient characteristics of large applications without the need for developers to assimilate large and complex code bases. Unlike synthetic benchmarks, which focus on a specific performance attribute, proxy apps measure the overall performance of a system by testing the cumulative contribution of each component.

Broadly speaking, there are 3 types of proxy apps:

Kernels

Benchmark kernels are portable, lightweight benchmarks that generate the same kind of calls as the original application, but execute much faster. Used on HPC systems in order to track their performance, they are decoupled from other application components and consist of only a few small pieces of code. Thanks to their small size, they are ideal for quickly detecting hardware and software issues at the node level. However, kernel benchmarks by themselves are rarely sufficient for evaluating cluster performance and in some cases can deliver deceptive results.

Skeleton Apps

Skeleton apps are communication accurate and computation fake benchmarks. They only replicate the memory and communication aspects of a larger program, while omitting to investigate the mathematical or algorithmic characteristics and often even "faking" the computation performed between the communication epochs, which makes them somewhat restricted in their purpose. They can however be useful for examining inter-node/inter-process communication performance, input and output overheads and memory and bus transfer bandwidth and latency.

Mini-Apps

Mini-apps (sometimes called *compact apps* or *representative apps*) are reduced versions of large production programs that combine some or all of the numerical kernels in a stand-alone application [16]. Their role as a co-design tool is to encapsulate the key computational aspects that would have a significant influence on the performance of the larger program from which they are derived. Unlike other kinds of proxy apps, mini-apps not only attempt to track the performance characteristics of the full scale program, but also seek to reflect the actual physical phenomena being simulated.

The much more manageable size of compact apps (typically a few thousands lines of code) facilitates the exploration of performance portability, as optimising and improving the algorithms and implementation of the compact app translates to a similar performance boost in the production code.

Due to the advantages they offer, mini-apps have drawn a considerable amount of interest over the last few years, particularly in the context of exascale computing planning. The Mantevo Project [17] is one of largest — and possibly one of the first ever developed — suites of mini-apps.

1.4 Overview of SYCL

This project aims to explore the performance benefits of SYCL [18, 4] (pronounced 'sickle'), which is a high-level C++ programming model developed by the Khronos consortium. SYCL builds on the underlying concepts, portability and efficiency of OpenCL and enables code for heterogeneous processors to be written in a single source style. It combines the flexibility of C++ features such as inheritance, templating and exception handling with the power of the OpenCL execution model, opening a wide scope for innovation in software design for heterogeneous systems.

There are 5 SYCL implementations currently available: ComputeCpp, Intel/LLVM SYCL, hipSYCL, triSYCL and sycl-gtx. The technical details of those implementations, the SYCL language and how it differs from other parallel programming languages are further discussed in Section 2.3.

1.5 Project Aims

The main goal of this project is to evaluate the performance portability of the available SYCL implementations across a range of different architectures and compare it to other similar heterogeneous programming languages.

Summary of Objectives

- Starting from an OpenCL version, implement a SYCL port of the lattice-boltzmann method to evaluate its performance portability and to get a better understanding of SYCL's main characteristics and common performance pitfalls. A brief overview of lattice-boltzmann is given in Section 3.2.
- Starting from a Kokkos version, implement a SYCL port of `neutral`, a Monte Carlo simulation mini-app. The implementation details of `neutral` and Monte Carlo particle transport codes are covered in Section 2.5.
- Test and compare across three of the five available implementations: ComputeCPP, Intel SYCL and hipSYCL.
- Analyse the performance of various platform classes (CPUs, GPUs) and compare different vendor offerings (e.g. Intel Xeon vs AMD Ryzen, Nvidia RTX vs AMD Radeon).

Chapter 2

Technical Background

2.1 Overview of Computer Architectures

For the last half century computing performance has been increasing at an exponential rate in concordance with Moore's law [19] and Dennard scaling [20]. Moore's law is an empirical observation stating that the number of transistors in a dense integrated circuit doubles approximately every two years. Dennard's scaling law (originally devised for describing MOSFETs) states that the voltage and current scale down with the length of a transistor and that the power density of transistors stays constant as their size decreases, hence the power usage is proportional to the surface area of a transistor, rather than its density.

Starting around 2005, transistors have became so small that Dennard scaling no longer applied, slowing down the potential for further increases in single-core performance, mainly due to current leakage and heat problems [21].

This section serves as a brief introduction to the parallel computer architectures that have been used over the years to increase performance while managing the consequences of thermal and power dissipation issues.

2.1.1 Pipelined Processors

Pipelining is one of the two main techniques used for exploiting *instruction-level parallelism* (ILP). It can boost performance by dividing instructions into a series of consecutive stages which are then processed in parallel. This method greatly increases instruction throughput in comparison to an un-pipelined processor, where the next instruction can only be fetched after the previous one has been retired. Figure 2.1 shows an example of the operation of a basic pipelined processor.

2.1.2 Superscalar Processors

Superscalar CPUs are another standard class of ILP-processors. Unlike pipelining, which employs several functional units to perform each stage of a computation, superscaling works by replicating the execution units and operating them concurrently to allow more than one instruction to be executed every clock cycle. Figure 2.2 depicts the model of a simple superscalar processor.

Current processors also use *out-of-order execution*, dynamically dispatching instructions that are waiting in the reservation stations. In this scheme, a non-executable instruction does not hinder the dispatch of succeeding instructions that are eligible for execution.

Fetch	**Decode**	**Execute**	**Write Back**

Cycle	Next →	In Progress				→ Retired
1	Instr 1	F1				
2	Instr 2	F2	D1			
3	Instr 3	F3	D2	E1		
4	Instr 4	F4	D3	E2	WB1	
5	Instr 5	F5	D4	E3	WB2	Instr 1

Figure 2.1: The progression of 5 instructions through a simple pipeline with 4 stages, where each stage takes one clock cycle to complete. After the first 3 cycles, assuming no dependency between instructions, the pipeline can theoretically process a full instruction per cycle.

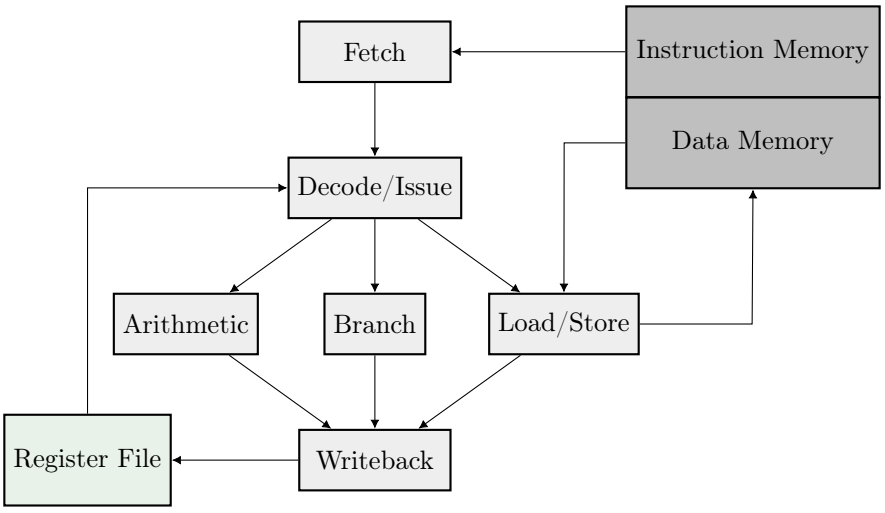
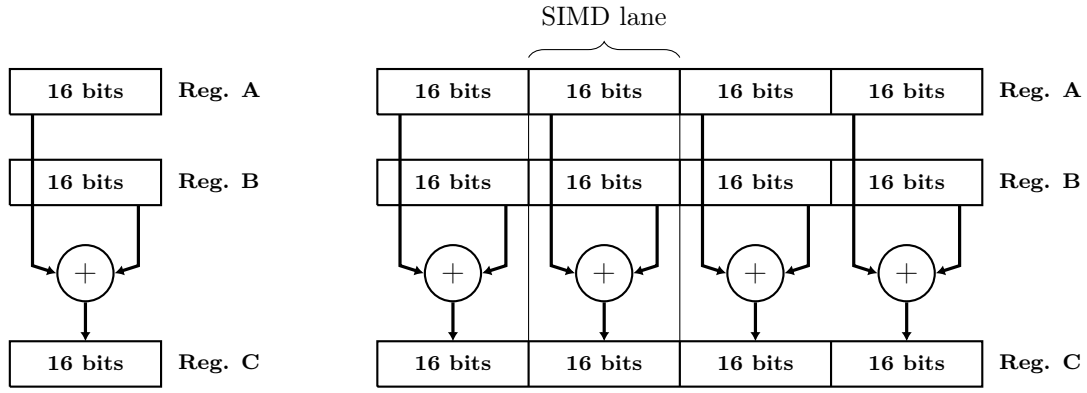


Figure 2.2: Diagram for a simple 3-way superscalar CPU with an arithmetic logic unit (ALU), a branch unit and a load/store unit (LSU) for memory instructions.

2.1.3 SIMD Architectures

Single instruction, multiple data (SIMD) systems contain a number of processing elements which are capable of all simultaneously executing the same instruction on multiple data items. SIMD architectures therefore take advantage of the *data-level parallelism* (DLP) of a program. They were the basis for the first supercomputers to use *vectorisation*, introduced by Cray in 1976 [22]. Vector processors use *wide registers* which can store multiple values, with each value being processed in a separate *lane*. Figure 2.3 illustrates the basic concept of how a vector processor works.

Modern CPUs include extensions for SIMD instructions (e.g. AVX [23] on the x86 ISA or SVE [24] on ARMv8) which can be either explicitly issued using intrinsics or automatically generated by the compiler using SIMD directives.



(i) 16-bit scalar registers.

(ii) 64-bit wide vector registers with 4 SIMD lanes.

Figure 2.3: Comparison between scalar computation (left) and vectorisation (right).

2.1.4 Multi-core Processors

Reaching the limits of Dennard scaling for single-core chips lead to the introduction of the first dual-core processor (IBM POWER4) in 2001 [25]. Since then, CPUs have started to incorporate multiple cores on the same die. The reasoning behind this approach is that, while a single-core architecture is limited by its clock frequency, adding an extra core to it can preserve the performance while reducing the frequency by half, resulting in improved power and heat dissipation. However, multi-core architectures also add a lot of complexity in their design and required programming techniques, with multi-threaded code being much harder to debug.

2.1.5 Graphics Processing Units

GPUs were initially developed as highly parallel hardware accelerators for image rendering. In the beginning they were strictly fixed-function, which made them unusable for anything else other than graphics specific tasks. However, in the early 2000s, vendors began to drop the fixed functionality and provide programmable shader cores, leading to general-purpose computing on GPUs [26]. The introduction of the CUDA and OpenCL programming frameworks further increased the popularity and availability of GPGPUs (general-purpose GPUs) and most modern graphics cards are now fully programmable, with some models even completely abandoning their fixed-function hardware.

In contrast to CPUs, which use branch prediction, re-order buffers and caches to optimise the *latency* (the delay between an instruction being fetched and its result becoming available to the next instruction) of any single thread, GPUs are designed to maximise their *throughput* (the amount of data processed per unit of time).

In terms of their architecture, GPUs can be broadly thought of as wide SIMD machines. They contain several *streaming multiprocessors* (SMs) which are grouped into processing clusters. In turn, each SM holds multiple CUDA cores, which process single threads and can be viewed as individual SIMD lanes. Parallel threads are combined together into groups of independent blocks (called *warps* by NVIDIA and *wavefronts* by AMD). Each warp contains 32 threads which perform the same instruction on different data, operating in lockstep. Each SM unit is then able to process one or more warps. Figure 2.4 shows the scheme of the TU102 (on which the RTX 2080 Ti is based), the highest performing GPU of the NVIDIA Turing line. Data can be transferred between the CPU and GPU through a PCI-e bus interface.

GPUs are therefore well suited for many scientific codes as they are able to "hide" memory latency by leveraging their massively-parallel nature. A common measure of thread parallelism on graphics cards is *occupancy*, which defines the ratio between the number of active warps on an SM and the number of warps supported by the SM.

Figure 2.4: A high-level block diagram of the Turing TU102 GPU [1], which contains 72 SM units with 64 CUDA cores each, for a total of 4,608 CUDA cores. The SMs share access to a 6MB L2 cache.

2.2 Heterogeneous Computing

Heterogeneous computing refers to the usage of systems that incorporate multiple kinds of machines or processors. Conversely, *homogeneous computing* describes systems which use a number of the same class of machines. As previously mentioned in Section 1.1, rather than attempting to boost performance by adding more of the same kind of processors, HPC systems are becoming more diverse — mainly through the use of hardware accelerators and coprocessors. One of the main advantages of those systems is their superior per watt and per dollar performance over clusters which only use CPUs.

The need to unlock the potential of heterogeneous machines has led to the development of programming languages that are able to run efficiently on a broad array of devices. The rest of this section presents an overview of some of the major parallel programming models.

2.2.1 OpenCL

Introduced in 2009 as one of the first of such models, OpenCL (Open Computing Language) is a low-level, open, vendor-agnostic standard for programming a wide range of computational devices [27, 28].

The main ideas that describe the OpenCL architecture can be explained using the following models:

Platform Model

The platform model is an abstraction that establishes a mapping between OpenCL and the physical hardware. The model is comprised of a host and one or more devices; each device contains multiple *compute units* (CUs) and each compute unit contains several *processing elements* (PEs), which are responsible for performing the computations. For example, when using a GPU, each CU corresponds to a SM and each PE corresponds to a CUDA core.

Execution Model

OpenCL's execution model is split in two separate components: one or more *kernels* that perform the main computational work and run on devices and a *host program* that orchestrates the execution context of the kernels.

Each instance of a kernel is called a *work item* and can be uniquely identified within the global index space by a set of coordinates. A work item is analogous to a single iteration of a loop. OpenCL also combines multiple work items into *work groups*. It is important to note that while work items can be synchronised if they are part of the same work group using *barriers* and *memory fences*, there is no mechanism for synchronising work items across different work groups (see Figure 2.5). Kernels can be submitted to the desired device through the use of a *command queue*.

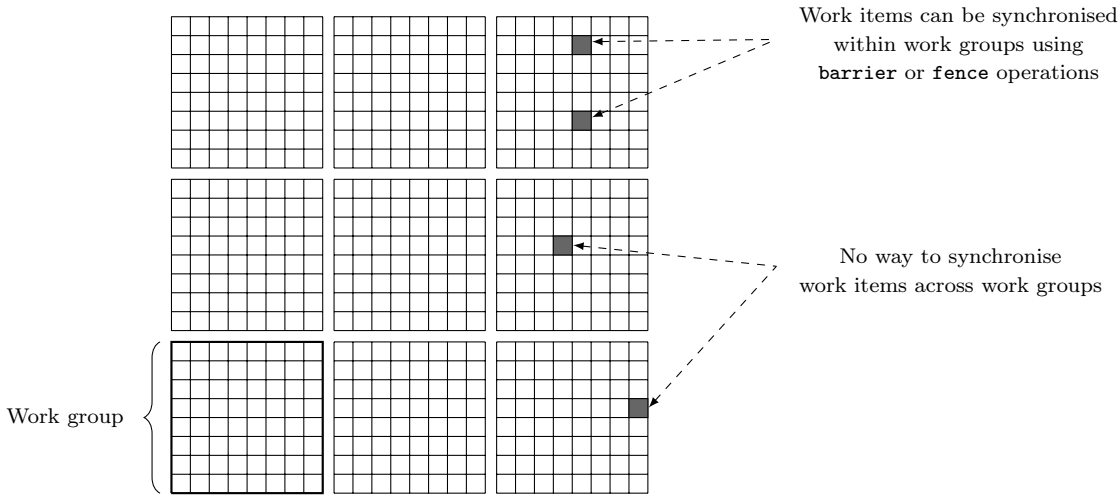


Figure 2.5: An example of an OpenCL ND-Range index space with 3x3 work groups, each containing 8x8 work items.

Memory Model

The memory of an OpenCL application is split into two main regions: *host memory* (accessible in the host code) and *device memory* (which is used by kernels when executing on a device). Memory objects have to be explicitly transferred by the programmer from the host to a device (and vice versa) through API calls.

Device memory is further subdivided into four memory regions:

1. *Global memory* is shared by all work items across the global index space.
2. *Local memory* is shared by all the work items within a work group.
3. *Private memory* is a region visible only to a specific work item. A work item's private variable cannot be accessed by any another work item. Variables declared inside kernels are marked as `__private` by default.

4. *Constant memory* is a global, read-only segment of memory, visible to all the work items.

Programming Model

The syntax of OpenCL is based on a subset of standard C99, with further extensions that add support for parallel execution. Kernels are declared in a separate file (e.g. `kernels.cl`) and have to be marked using the `__kernel` keyword.

Listing 2.1 shows a kernel code example for an OpenCL program.

```
__kernel void vadd(__global const float *A,
                   __global const float *B,
                   __global float *C)
{
  int i = get_global_id(0);
  C[i] = A[i] + B[i];
}
```

Listing 2.1: Example OpenCL kernel for vector addition.

2.2.2 CUDA

CUDA (Compute Unified Device Architecture) is a proprietary parallel framework, developed by NVIDA [29] with syntax, structure and features that are very similar to OpenCL. While its associated software toolkit makes it a very attractive choice for deep learning an AI applications, CUDA can only target NVIDIA GPUs, which makes it inadequate for writing performance portable codes.

```
__global__ void vadd(float *A,
                     float *B,
                     float *C,
                     int n)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (id < n) {
    C[i] = A[i] + B[i];
  }
}
```

Listing 2.2: Example CUDA kernel for vector addition, where the number of threads per block is `blockDim.x` and number of blocks per grid is `gridDim.x`.

2.2.3 HIP

Launched by AMD as an alternative to CUDA, HIP is a C++ parallel programming model and API [30]. HIP applications are able to target both AMD and NVIDIA GPUs from a single source code. Additionally, it includes a tool called "hipify" which can convert an existing source from CUDA to the HIP layer. HIP is a part of AMD's ROCm (Radeon Open Computing) open-source ecosystem for HPC-grade accelerated computing [31].

2.2.4 OpenMP

OpenMP (Open Multi-Processing) is an application programming interface that provides a model for multi-platform parallel codes written in C, C++ and Fortran [32]. It extends those languages to

```
__global__ void vadd(const float *A,
                     const float *B,
                     float *C,
                     unsigned n)
{
  unsigned gid = hipThreadIdx_x;
  if (gid < n) {
    C[gid] = A[gid] +B[gid];
  }
}
```

Listing 2.3: Example HIP kernel for vector addition. Similar to CUDA, grid and block coordinates are determined though the use of builtin variables.

expresses shared memory parallelism through the use of *compiler directives* (i.e. `#pragma` statements) along with environment variables and callable run-time library routines.

From OpenMP 4.0 onwards, support was added for offloading code regions to accelerator devices, through the use of `target` constructs [33].

```
#pragma omp parallel for
for (int i = 0; i < n; i++) {
  C[i] = A[i] + B[i];
}
```

Listing 2.4: Example OpenMP parallel loop for vector addition in C. The `#pragma` directive marks the parallel code region.

2.2.5 OpenACC

Similarly to OpenMP, OpenACC (which stands for *open accelerators*) comprises a set of compiler directives that allow developers to annotate blocks of code in standard C, C++ and Fortran that are meant to be executed on a target accelerator device [34].

```
#pragma acc parallel loop
for (int i = 0; i < n; i++) {
  C[i] = A[i] + B[i];
}
```

Listing 2.5: Example OpenACC parallel loop for vector addition in C.

2.2.6 Kokkos

Initially developed by Sandia National Laboratories, Kokkos is a single-source high-level programming model built on top of standard C++ that provides performance portability across a wide range of manycore computer architectures [35].

Kokkos provides an additional abstraction that is crucial for performance portability in the form of *polymorphic data layouts*. The main motivation behind this concept is that an efficient memory access pattern is critical in order for a parallel implementation to be performant on a variety of devices. However, most languages (e.g. C) have a predefined data layout, through built-in multidimensional arrays with a set indexing structure. Kokkos solves this problem by automatically choosing the optimal data layout (e.g row major or column major ordering) during compilation depending on the targeted

architecture. This also removes the need for users to aware of the implementation details particular to specific architectures. The *View* class, used for storing and communicating buffer to/from the device, incorporates data types into the data layout.

Kernels in Kokkos are typically implemented as *C++ functors*. A functor (which stands for "function object") is an object that defines the `operator()` function and "behaves like" a function. Kernels can also be written as *C++ lambdas*, which are essentially nameless functors.

Listing 2.6 shows an example program, with the host and device code being in the same file.

```
// Host buffers
Kokkos::View<float*>::HostMirror h_a(N);
Kokkos::View<float*>::HostMirror h_b(N);
Kokkos::View<float*>::HostMirror h_c(N);

// Initialise host data
for (int i = 0; i < 10; i++) h_a[i] = i;
for (int i = 0; i < 10; i++) h_b[i] = i;

// Device buffers
Kokkos::View<float*> d_a(N);
Kokkos::View<float*> d_b(N);
Kokkos::View<float*> d_c(N);

// Transfer data to device
deep_copy(d_a, h_a);
deep_copy(d_b, h_b);

// Add vectors on the device, using a lambda
Kokkos::parallel for(N, KOKKOS_LAMBDA (int idx) {
  d_c[idx] = d_a[idx] + d_b[idx];
});

// Wait for code to finish executing
Kokkos::fence();

// Transfer data back to device
deep_copy(h_c, d_c);
```

Listing 2.6: Vector addition in Kokkos, using a function lambda.

2.2.7 Raja

RAJA (developed at the Lawrence Livermore National Laboratories) is a relatively new programming model, currently best optimised for running on CPUs [36, 37]. Like Kokkos, it expresses parallelism through loop kernels defined as C++ lambda expressions.

```
RAJA::forall<RAJA::omp_parallel_for_exec>(RAJA::RangeSegment(0, N), [=] (int i) {
  c[i] = a[i] + b[i];
});
```

Listing 2.7: Raja vector addition loop kernel variant, using the `omp_parallel_for_exec` execution policy, which uses CPU multithreading.

2.2.8 LLVM

LLVM is an open-source compiler framework consisting of collection of compiler and toolchain technologies, which can be used to develop frontends for programming languages. [38]. A number of parallel programming model implementations (notably Intel's SYCL implementation) use LLVM technologies for their compilers.

2.3 The SYCL Programming Model

SYCL, the model of interest for this thesis, is an open standard for programming a heterogeneous system. The latest version is the SYCL 2020 provisional specification, which was launched on June 30th [39]. Though it introduced many welcome changes which have long been requested by the community, it was released toward the end of this thesis project and the only available implementation at this time is through an early beta version of Intel's DPC++. Therefore, this section only covers features available up to SYCL version 1.2.1 (revision 7), which is used by most implementations employed during the execution of this project.

The rest of this section outlines some of the main aspects of SYCL (primarily in comparison to OpenCL, since in many ways, they are similar and most of the terminology used when discussing SYCL is inspired from that of OpenCL), followed by a very brief overview of the available compilers and implementations.

2.3.1 Host Setup

One of the major disadvantages of OpenCL is that the portion of host code required to setup the environment (i.e. defining the platform, context, command queue and memory objects) is very verbose. The main appeal behind SYCL is that it abstracts away those cumbersome parts, combining the host and device code in a single file, while still offering the same low-level optimisations as OpenCL. To achieve this, SYCL compilers use a single-source multiple compiler-passes (SMCP) approach, making two separate passes: one for the host portion of the program and another one targeting just the kernels.

In SYCL, the only object that needs to be defined before launching a kernel is a *command queue* (`cl::sycl::queue`). An annotated example of the host setup for a simple program can be seen in Listing 2.8.

2.3.2 Kernels and Data Accessors

Compared to OpenCL, which can provide either in-order or out-of-order queues, SYCL always executes kernels out-of-order. To ensure that the correct results are produced at the end, the SYCL runtime constructs a *directed acyclic graph* (DAG) of dependencies between the submitted kernels. Thus users need to specify the inputs and outputs of each kernel through the use of *data accessors*. Depending on the type of access required, they can provide different access modes (read-only, read-write, atomic etc.).

In order to capture all the device-side operations and be able to build the DAG of kernels, SYCL uses a *command group handler* object (CGH). Unlike OpenCL, which requires explicit data operations, SYCL handles all the movement of data buffers implicitly.

Like Kokkos, SYCL kernels can be submitted either as a functor or lambda using a `parallel_for` construct. The grid space is then divided into work groups and work items, using the same model as OpenCL. Furthermore, all parameters passed to a SYCL kernel have to by captured by value inside the scope of the functor/lambda or passed through accessors.

2.3.3 Error Handling

One of the benefits of using C++11 is that SYCL has support for error handling. There are two types of errors in SYCL:

- *synchronous errors* are typical C++ exceptions that can be caught by the runtime with a `try{..} catch{..}` block. They can for example be caused when the wrong arguments are given to a function call.

- *asynchronous errors* (e.g. providing an invalid work group size to a kernel) are reported after they occur through an error handler provided by the user.

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

int main() {
  using namespace cl::sycl;
  int data[1024]; // Data to be worked on

  // Include all the SYCL work in a {} block to ensure all
  // SYCL tasks are completed before exiting the block.
  {
    // Create a queue to enqueue work to queue
    myQueue;

    // Wrap the data variable in a buffer.
    buffer<int, 1> resultBuf(data, range<1>(1024));
    // Create a command_group to
    // issue commands to the queue.
    myQueue.submit([&](handler& cgh) {

      // Request access to the buffer
      auto writeResult = resultBuf.get_access<access::write>(cgh);

      // Enqueue a parallel_for task.
      cgh.parallel_for<class simple_test>(range<1>(1024), [=](id<1> idx) {
        writeResult[idx[0]] = static_cast<int>(idx[0]);
      }); // End of the kernel function
    }) // End of the queue commands
  } // End of scope, so:wait for the queued work to complete

  // Print result
  for(int i = 0; i < 1024; i++) {
    std::cout<< "data[" << i << "] = " << data[i] << std::endl;
  }

return 0;
}
```

Listing 2.8: Example of a typical SYCL application which schedules a job to run in parallel on an OpenCL accelerator [4].

2.3.4 Current Implementations

Intel SYCL

Intel provides an open source implementation of SYCL that uses Clang as a C++ frontend and LLVM as a portability layer [40]. In the future, this implementation will be incorporated into the oneAPI unified programming model.

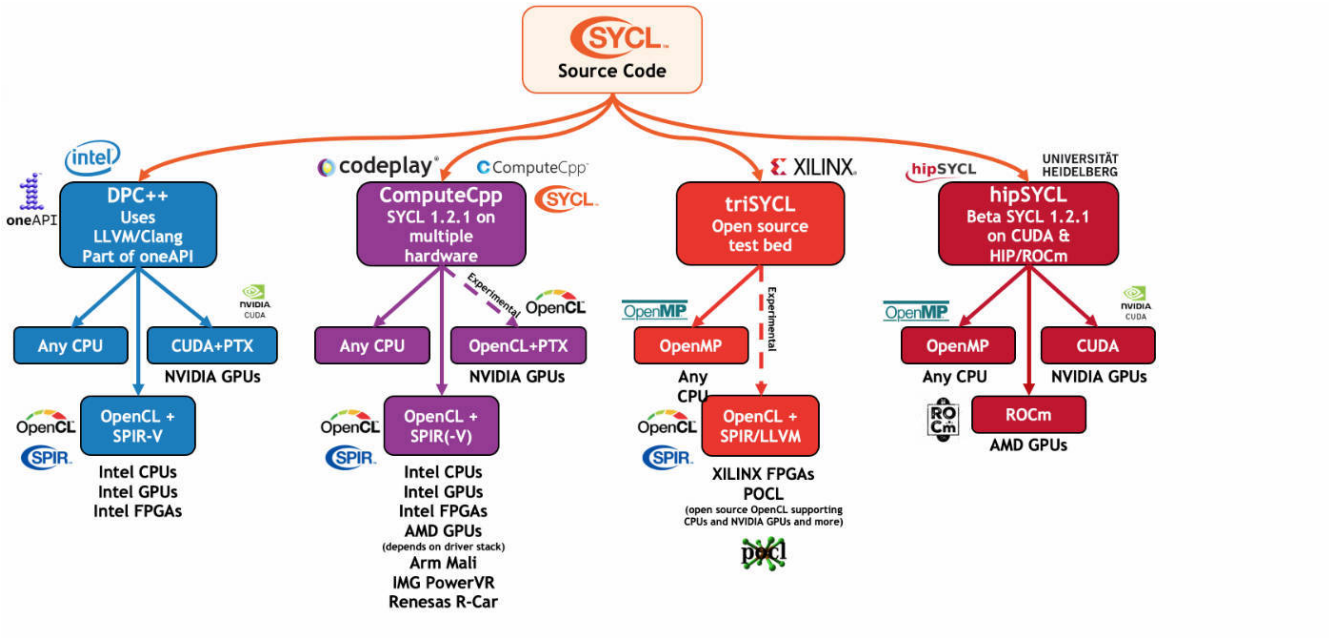


Figure 2.6: The current state of the SYCL ecosystem [2].

ComputeCpp

The first commercially available fully compliant implementation, developed by Codeplay [41]. Although it is not open-source, a Community Edition is available for free.

hipSYCL

HipSYCL (funded by Heidelberg University) is an implementation that uses HIP/CUDA instead of OpenCL as a backend [42]. This approach offers a level of performance close to what could be achieved using CUDA/HIP (by using platform specific optimisations, such as intrinsics) while allowing developers to use a vendor neutral programming model.

triSYCL

TriSYCL is an incomplete SYCL implementation that was created for the purpose of experimentation with the specification [43]. It it not meant to be used by regular end-users.

sycl-gtx

Developed as a masters project, this implementation consists of a set of header files defining macros (i.e. `#define` preprocessor statements) which map SYCL specific functions to their OpenCL counterparts. It requires non-standard macros to work and is very limited in its scope [44].

2.4 The Seven Dwarfs of HPC

In 2004, Phil Colella identified seven distinct classes of computational patterns frequently encountered in engineering and scientific codes, which he called "the seven dwarfs of HPC" [45]:

1. Dense linear algebra
2. Sparse linear algebra
3. Spectral methods
4. N-body methods
5. Structured grids
6. Unstructured grids
7. Monte Carlo

This taxonomy is useful to take into consideration when reasoning about the performance attributes of HPC-style applications. The lattice-boltzmann code (discussed in Section 3.2) is a structured grid dwarf and the `neutral` mini-app is a Monte Carlo dwarf.

2.5 Monte Carlo Neutron Transport

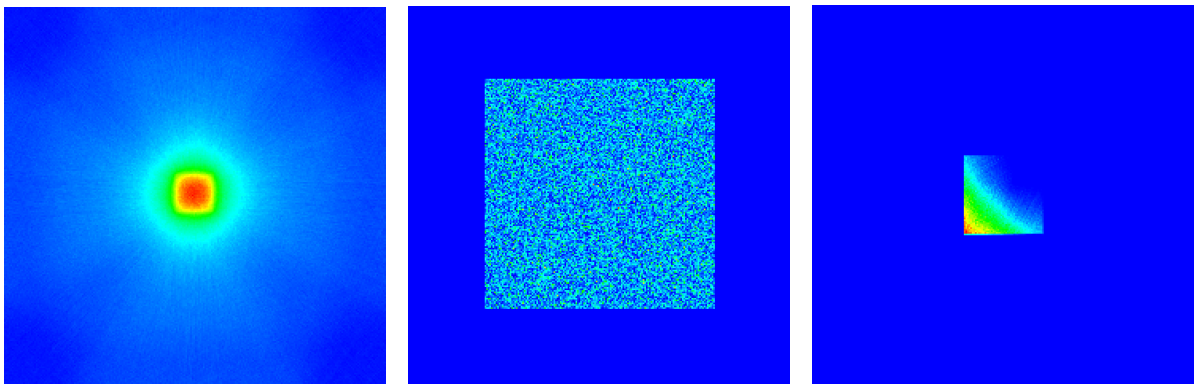
Monte Carlo is a massively parallel method used to numerically approximate intractable mathematical functions, relying on repeated random sampling from a probability distribution. Although most HPC applications are either memory bandwidth bound or compute bound, Monte Carlo codes do not fit into either of those categories. Because of their non-deterministic nature, the memory access pattern of most Monte Carlo applications is hard to predict, leading to sub-optimal performance due to poor cache usage and memory coalescence [46].

2.5.1 Neutral Mini-App

`neutral` is a Monte Carlo neutral particle transport mini-app, developed at the University of Bristol, that encapsulates the computational traits of physical processes simulated by much larger neutron transport codes [47]. It is included in the `arch` project, which contains the shared architectural code for a number of proxy apps [48].

The `neutral` mini-app includes three synthetic problems (illustrated in Figure 2.7) designed to reveal various performance aspects of the application.

- The *streaming* problem initialises particles in a dense section in the middle of the grid. Neutrons are then free to move across a very low-density material ($10^{-30} Kg/m^3$) at very high speeds, without any collisions between them. This test case underlines one of the major shortcomings of Monte Carlo codes, which is the lack of spatial locality of the particles as they move autonomously without a clearly defined pattern.
- The *scattering* problem spreads the particles in a high-density ($10^3 Kg/m^3$) material, producing only collision events. In `neutral`, a particle can 'die' when its energy falls below a certain threshold and becomes irrelevant to the simulation. Most particles in this case stay in the cell where they were initialised.
- The *central square problem* (*csp*) models a more realistic scenario than the previous two. Particles are sourced in the lower left corner of the grid and move through the low density mesh, until they start colliding with a high density block in the middle.



(i) `stream` (ii) `scatter` (iii) `csp`

Figure 2.7: Plots for the energy deposition of each of the three test problems used by `neutral`, after a single timestep [3].

CHAPTER 3

Project Execution

3.1 Hardware Platforms

The platforms used during both the development and testing stages of the project were provided through University of Bristol HPC Research Group's cluster, the Zoo [1]. Table 3.1 describes those hardware platforms, which were purposely selected to represent various devices from all major vendors: Intel and AMD for consumer and server-grade CPUs; NVIDIA, AMD and Intel for GPUs. The table also includes the theoretical peak memory bandwidth that can be attained by each device [49, 50, 51], which is taken from the official technical specifications where possible [2].

The Iris Pro 580 integrated graphics unit available on the Intel NUC was used as the main platform for development during this stage of the project. This is because it is the only GPU that supports both ComputeCPP and Intel SYCL, which are the most mature implementations at the moment. Although hipSYCL can run on both of the dedicated GPUs and supports a lot of the major features, it is not yet fully compliant with the Khronos specification and still lacks some useful functionalities (especially in terms of available debugging options).

Device	Type	Architecture	Peak BW (GB/s)
Intel Xeon Gold 6126	Server CPU (12-core)	Skylake	119
Intel i7-6770HQ	CPU (4-core)	Skylake	34
AMD Ryzen 7 2700	CPU (8-core)	Zen Plus	44
Intel Iris Pro 580	Integrated GPU	Generation 9.0	34
NVIDIA RTX 2080 Ti	Discrete GPU	Turing	616
AMD Radeon VII	Discrete GPU	RX Vega 20	1024

Table 3.1: List of devices used on the HPC Zoo.

[1] https://uob-hpc.github.io/zoo/

[2] AMD does not include the peak memory bandwidth in its official specifications page and neither does Intel for its Xeon line of processors, so the values for the Xeon Gold and Ryzen 7 are taken from https://en.wikichip.org/wiki/intel/xeon_gold/6126 and https://en.wikichip.org/wiki/amd/ryzen_7/2700

3.2 Lattice-Boltzmann

Lattice-Boltzmann methods (LBM) are a class of computational fluid dynamics (CFD) simulations which use the Boltzmann equation as an alternative to solving the Navier-Stokes equation [52]. Referring back to the classification of computational patterns described in Section 2.4, LBM is a *structured grid* dwarf and therefore memory bandwidth bound. The simulation domain is partitioned into square cells, where each cell contains nine speeds (represented as floating-point numbers) characterising the fluid flow.

The initial LBM code used in this section was previously developed by me as part of the Advanced High Performance Computing unit and was available in OpenMP and OpenCL versions [3]. The OpenCL version of the code was chosen as a starting point for the port due to its similarity to SYCL. Both implementations had already been optimised and achieved a good fraction of peak bandwidth on the tested CPUs, for the OpenMP version, and respectively on GPUs, for the OpenCL version.

3.2.1 Comparison between OpenCL and SYCL

Before migrating the code base from OpenCL to SYCL, it is important to note the correspondence between the main functions used when writing programs in those two models. A high-level overview of the mapping between homologous API calls and constructs in OpenCL and SYCL (further detailed in the rest this section) is presented in Table 3.2.

Type	OpenCL	SYCL
Host API Calls	`clGetPlatformIDs()`	`platform::get()` (optional)
	`clCreateContext()`	`cl::sycl::context` (optional)
	`clGetDeviceInfo()`	`cl::sycl::device`
	`clCreateCommmandQueue()`	`cl::sycl::queue`
	`clCreateBuffer()`	`cl::sycl::buffer`
Memory Management	`N/A`	`cl::sycl::accessor`
	`clEnqueueReadBuffer()`	`cgh::copy()` (optional)
	`clEnqueueWriteBuffer()`	`cgh::copy()` (optional)
	`clEnqueueCopyBuffer()`	`cgh::copy()` (optional)
	`clReleaseMemObject()`	`N/A`
Kernels & Indexing	`clCreateProgramWithSource()`	`cl::sycl::queue::submit()`
	`clBuildProgram()`	`cl::sycl::queue::submit()`
	`clCreateKernel()`	`cl::sycl::queue::submit()`
	`clSetKernelArg()`	`cl::sycl::queue::submit()`
	`clEnqueueNDRangeKernel()`	`cl::sycl::queue::submit()`
	`N/A`	`cl::sycl::nd_range`
Synchronisation	`barrier()`	`cl::sycl::nd_item::barrier()`
	`mem_fence()`	`cl::sycl::nd_item::mem_fence()`
	`read_mem_fence()`	`cl::sycl::nd_item::mem_fence()`
	`write_mem_fence()`	`cl::sycl::nd_item::mem_fence()`

Table 3.2: Equivalence between OpenCL and SYCL constructs.

In terms of the host platform configuration, SYCL is significantly easier to setup than OpenCL. All that is required is a `device_selector` which is then supplied to the `queue`. Although there are

[3] https://github.com/AndreiCNitu/HPC

equivalent API calls available in SYCL for `clGetPlatformIDs()` and `clCreateContext()`, they can be managed by the runtime through the `device_selector` and are entirely optional.

As the OpenCL model does not include a unified memory view, there is no matching construct for SYCL's `accessor`. In contrast to OpenCL, where buffer transfers and deletions must be explicitly specified, memory management in SYCL is handled implicitly by default. If needed, SYCL command group handlers can be used for explicit copy operations (`cgh::copy()`), which can for example be helpful when a user wishes to exclude the measurement of data transfers from the timing of the application.

The submission of kernels to devices is also considerably more streamlined in SYCL, with the kernels parameters and options being specified via the `queue::submit()` function. Global and local sizes are set in SYCL using the `nd_range` class and individual work items are encapsulated by the `nd_item` class.

3.2.2 Porting Lattice-Boltzmann to SYCL

Following the previously described equivalence between the OpenCL and SYCL API, the process of porting the LBM code is relatively straightforward. This section outlines the design decisions that were made in order to ensure that the comparison between the two versions is fair. It then addresses some of the issues and performance pitfalls encountered, which are not immediately apparent when migrating to a SYCL code.

Design/General Considerations

Firstly, since OpenCL and SYCL share the same view of the index space terms of work items and work groups, it is important to set the same local size for each work group (even though the runtime can choose a default local size, it might vary between different implementations). The work group sizes were fixed to 128×1, which was the optimal configuration that was found after benchmarking multiple variants.

Secondly, the LBM OpenCL implementation requires the use of a *parallel reduction* (illustrated in Figure 3.1) in order to sum the average cell velocities at the end of each time step, which had to be translated to SYCL. In OpenCL this is implemented such that each cell velocity value is stored in local device memory, which can be accessed in SYCL by passing the `cl::sycl::access::target::local` modifier to an accessor. The reduction is performed locally across all the work groups and the results for each of them are kept in global memory. They are then added up on the host at the end of the simulation.

When timing the LBM program, it is essential to make sure that the same parts of the computation are measured. The original OpenCL code excludes the buffer construction and data allocation on the device, which are not necessarily lightweight. However, it does include the total time taken by the initial data transfers to the device (i.e. the `clEnqueueWriteBuffer()` calls). This can be difficult to replicate in SYCL, because the runtime handles data transfers implicitly and therefore gives no guarantee for when they are actually performed. Explicit copy operations were first tried, but at the time of implementing the code they were not working correctly on all the compiler platforms. The solution chosen to resolve this issue was to include buffer construction, allocation and transfers in the timings on both versions.

Finally, the SYCL code was compiled using the `-O3` flag. Adding optimisation options is much more important when using a SYCL compiler, which behaves like a regular C/C++ compiler and needs them for generating optimised code, unlike OpenCL where adding those flags would only improve the host code without any benefit to the device code. After first reaching a working version of the program, synchronous and asynchronous exception handlers were added to catch any further errors.

Other Issues

While the ComputeCpp and Intel compilers generally worked smoothly, there were some difficulties when compiling with hipSYCL. Prototyped function declarations would cause linker errors, unless they

were marked as `__host__ __device__`. This was raised as an issue and later fixed through a pull request [4]. Also, hipSYCL does not support the `acc[i][j]` notation for buffer accessors [5], so all such accesses had to be substituted with `acc[cl::sycl::id<2>{i,j}]` (there was no impact on performance when using other platforms that support both notations).

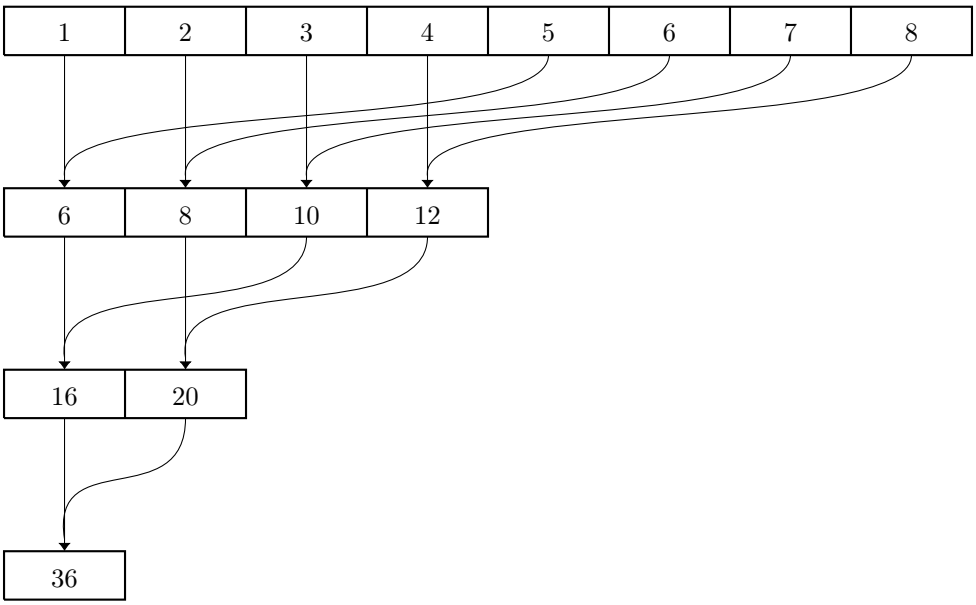


Figure 3.1: An example of a contiguous parallel reduction. Reducing a vector of size n requires $n-1$ operations. The final result is stored in the vector at index 0.

An unfortunate difference between OpenCL's and SYCL's execution models is the discrepancy in the ordering of work item indices. When passing an id value to each dimension of an accessor of size `{r0, r1, r2}` using subscript operators (`acc[id0][id1][id2]`), the SYCL specification [18] states that the index is expanded according to the following equation:

$$linear\_id = id_2 + (id_1 \times r_2) + (id_0 \times r_2 \times r_1) \tag{3.1}$$

Thus, in SYCL, the index returned by `get_global_id(0)` is the *slow moving index* and the index returned by `get_global_id(1)` is the *fast moving index*, which should match the contiguous index in memory. However, this is the opposite of the ordering implemented by OpenCL. Using the incorrect ordering can result in a significant hit on performance due to the long stride of consecutive memory accesses. To account for this, a 1D-range SYCL version was written, with accessors indexed as `acc[j*nx+i]`. This approach can also highlight potential issues stemming from different compiler implementation choices in regards to the linear id formula (Equation 3.1). For instance, because an accessor relies on the `operator[]` function (unlike a C array which is just a pointer type) a compiler implementation might resolve a `acc[i][j]` call as `(acc[i])[j]`, instead of `acc[j*nx+i]`. In this example, the nested calls to the `operator[]` function might carry a noticeable performance impact.

3.3 Neutral

The SYCL port of `neutral` was started from the master branch of the `neutral_kokkos` [6], which has a total of 5519 lines of code across 30 files.

Similarly to the previous section, a comparison between the Kokkos and SYCL concepts and APIs is first presented, followed by a more in-depth description of the porting process.

[4] https://github.com/illuhad/hipSYCL/pull/207
[5] https://github.com/illuhad/hipSYCL/issues/127
[6] https://github.com/UoB-HPC/neutral_kokkos

3.3.1 Comparison between Kokkos and SYCL

In a number of ways, Kokkos and SYCL are very similar. Both models enable single-source programming on a wide range of devices and are based on C++, expressing parallelism via functors and lambdas. Table 3.3 illustrates the equivalence between the core concepts and API calls in those two models.

Type	Kokkos	SYCL
General Terminology	Thread	Work item
	Thread team	Work group
	Team scratch pad memory	Local memory
Host API Calls	`Kokkos::initialize()`	`{` (begin scope)
	`Kokkos::finalize()`	`}` (end scope)
Memory Management	`Kokkos::View<T*>`	`cl::sycl::buffer`
	`Kokkos::View<T*>`	`cl::sycl::accessor`
	`Kokkos::View<T*>::HostMirror`	`T*`
	`Kokkos::deep_copy()`	`cgh::copy()` (optional)
Kernels & Indexing	`Kokkos::parallel_for()`	`cl::sycl::parallel_for()`
	`Kokkos::parallel_reduce()`	`N/A`
	`Kokkos::parallel_scan()`	`N/A`
	`Kokkos::RangePolicy`	`cl::sycl::range`
	`Kokkos::MDRangePolicy`	`cl::sycl::nd_range`
	`KOKKOS_LAMBDA, (size_t i) {...}`	`(cl::sycl::id<N> i) {...}`
Synchronisation	`Kokkos::fence()`	`cl::sycl::nd_item::mem_fence()`
	`team_member.team_barrier()`	`cl::sycl::nd_item::barrier()`

Table 3.3: Equivalence between Kokkos and SYCL nomenclature and constructs.

One of the most important differences between Kokkos and SYCL is that Kokkos offers builtin reduction operations through the `Kokkos::parallel_reduce()` construct. An analogous method is only available in SYCL through an open source implementation of the `ParallelSTL` library [53], currently only supporting triSYCL and a beta implementation of ComputeCPP.

Another relevant difference is that — similarly to OpenCL — Kokkos never performs a hidden copy and requires explicit data transfers between the host and device by using the `Kokkos::deep_copy()` function.

3.3.2 Porting Neutral to SYCL

Although the two models might appear to be very similar on the surface, there are some non-obvious distinctions between them, making the migration of this mini-app significantly harder then the previous one. A big challenge behind the development of this port was the fact that Kokkos offers more high-level abstractions which need to be decomposed when coding the SYCL version. This section outlines the main steps taken to implement a SYCL port for the `neutral` mini-app.

The host code was relatively simple to migrate, mainly by applying the changes summarised in Table 3.3. A global device queue was declared such that it could be passed to any parallel kernels.

One detail that turned out to be very inconvenient is that throughout the `neutral` mini-app `Views` are often declared at one point in the code and then allocated at another later point. However, in SYCL the buffer size has to be determined upon declaration. To solve this issue while still keeping the

same overall structure as the original version, SYCL buffers were stored as pointer references and then initialised later in the execution. This method is exemplified in Listing 3.1.

```
Kokkos::View<T*> buf;
...
new(buf) Kokkos::View<T*>("device", N);
```

(i) Kokkos

```
cl::sycl::buffer<T, 1>* buf;
...
*buf = new cl::sycl::buffer<T, 1>(cl::sycl::range<1>(N));
```

(ii) SYCL

Listing 3.1: Example of porting the declaration and initialisation of a buffer from Kokkos to SYCL.

The strategy used for porting the Kokkos kernels is demonstrated in Listing 3.2. Each kernel submission was first enclosed inside a `queue.submit()` call and all the necessary accessors were defined. The `KOKKOS_LAMBDA` was replaced with a capture by value (`[=]`). Where present, Kokkos builtin types were substituted with SYCL equivalents.

```
Kokkos::parallel_for(N, KOKKOS_LAMBDA (int i) {
  buf[i] = 0.0;
});
```

(i) Kokkos

```
queue.submit([&] (cl::sycl::handler& cgh) {
  // Capture accessors using the most restrictive mode
  auto buf_acc = buf->get_access<cl::sycl::access::mode::write>(cgh);

  cgh.parallel_for<class init_kernel>(cl::sycl::range<1>(N),
                                      [=](cl::sycl::id<1> idx) {
    buf_acc[idx] = 0.0;
  });
});
```

(ii) SYCL

Listing 3.2: Example of porting a simple kernel that zero initialises a buffer from Kokkos to SYCL.

Kokkos allows the use of *mirror views*, which are views of equivalent buffers that are located in different memory spaces. The idea behind this concept is to enable direct bitwise deep copies between memory spaces without the need for a temporary buffer, since the mirror view has the same layout as the original view.

The Kokkos version of `neutral` uses `HostMirror` views and `deep_copy()` calls in order to transfer data that was initialised on the host to the device. This construct has no direct equivalent in SYCL that can be used inside the buffer scope, since the ownership of host buffer pointers is handled by the SYCL runtime. However, the memory managed by a SYCL buffer can be modified directly in the host code through the use of a *host accessor*. Listing 3.2 shows a simple example of this approach.

3.3.3 Debugging SYCL

Encountering bugs is almost inevitable when migrating non-trivial code bases such as `neutral` from one language to another, especially when working with parallel programs. In this particular case, the

```
// Device buffer
Kokkos::View<double*> mesh_edgex;
init_data(&mesh_edgex);

// Host mirror of the buffer
Kokkos::View<double*>::HostMirror rank;
allocate_host_data(&rank);

deep_copy(*mesh_edgex, *rank);
```

(i) Kokkos

```
double rank;

// Access a buffer immediately in host code
auto edgex_acc = mesh_edgex.get_access<cl::sycl::access::mode::read>();
rank = edgex_acc[idx];
```

(ii) SYCL

Listing 3.2: Example of porting the mirror views from Kokkos to SYCL using host accessors.

likelihood of bugs is further exacerbated by the fact that the SYCL port could not be compiled and tested until it was fully completed, in contrast to other development scenarios where features can be incrementally added and tested on an individual basis. The purpose of this section is to briefly cover some of the available options for debugging SYCL that were employed during this project.

Unlike the OpenCL model, which only allows kernels to execute on a compatible device backend, the SYCL specification requires every implementation to provide a *host device* [18]. The host device emulates the execution of a SYCL program on a real device, reducing SYCL API calls to pure C++ and permitting the use of standard debuggers such as `gdb` and `valgrind`. Running `neutral` on the host device in conjunction with gdb was the main debugging technique used during the project.

Alternatively, the SYCL specification allows the use of output operations inside the scope of a kernel via the `cl::sycl::stream` object, providing buffered output streams for a number of standard types. Although there is no guarantee for when the output will be displayed and in some cases outputs are intertwined, this can still be a useful feature for logging kernel events. Unfortunately, at the time of writing, most implementations either do not support kernel streams and manipulators or only implement a small subset of them.

Chapter 4

Results and Evaluation

4.1 SYCL Compilers

An overview of all the available SYCL implementations has already been presented in Section 2.3.4. In order to encompass all the hardware platforms described in Section 3.1, the following three SYCL compilers were used during this project for collecting results:

- hipSYCL (release 0.8.1)
- ComputeCpp (version 2.0.0)
- LLVM SYCL (provided as part of Intel's oneAPI 2021.1.8 beta release)

Collectively, these compilers are able to target all the devices listed in Table 3.1 (on page 19). HipSYCL can support AMD GPUs (via HIP/ROCm), NVIDIA GPUs (via CUDA) and any CPU (via OpenMP). ComputeCpp and LLVM SYCL can both target Intel integrated GPUs (via OpenCL and SPIR) as well as any CPU. ComputeCpp also provides experimental support for NVIDIA devices with PTX.

The drivers used by each device are as follows:

- The NVIDIA RTX 2080 Ti has NVIDIA's driver 418.39 installed.
- The AMD Radeon VII has AMD's ROCm driver 2982.0 (HSA1.1,LC) installed.
- The Intel Iris Pro 580 has Intel's OpenCL Graphics Driver 20.28.17293 installed.
- The Intel i7-6770HQ, AMD Ryzen 2700 and Intel Xeon 6126 use the Intel OpenCL Runtime 18.1.0.0920.

4.2 Lattice-Boltzmann

All of the timings for LBM were collected as the average of five runs in order to take into consideration any potential variability. As lattice-boltzmann codes are memory bandwidth bound, the results in this section are displayed as the attained percentage of the theoretical peak memory bandwidth for each of the three inputs available.

The effective bandwidth is calculated by timing the execution of the kernels and by knowing the rate at which the program reads data from or stores data into memory. Each kernel performs a read and a write operation for each of the nine speeds, which are stored as floats (4 bytes). The following formula for memory bandwidth is used:

$$\frac{(B_r + B_w) \times N_x \times N_y \times numIterations \ (bytes)}{time \ (s)}$$

where the result is in units of bytes/second, $B_r = 9 \times 4$ is the total number of bytes read per kernel, $B_w = 9 \times 4$ is the total number of bytes written per kernel and $N_x \times N_y$ is the size of the input matrix.

4.2.1 GPU Results

The initial results for GPUs are shown in Figure 4.1. Since there was a noticeable difference between the SYCL 2D-range and 1D-range times on some of the platforms used, both variants are included here.

The results on the discrete graphics cards are up to an order of magnitude slower when using hipSYCL compared to the OpenCL implementation, especially on the 256×256 matrix. The smallest performance delta relative to the OpenCL version is obtained on the Iris Pro Graphics when using the 1024×1024 matrix, reaching 0.88× of the baseline using the ComputeCpp compiler and 0.78× using the Intel compiler. However, this still falls short of the expected results. The difference in execution times between the two models also appeared to be roughly constant across all input sizes, which could suggest that SYCL introduces a fixed overhead compared to OpenCL.

To examine the source of those problems, the Intel OpenCL Intercept Layer [54] was used. The Intercept Layer is a profiler tool that can intercept and modify OpenCL calls for debugging and performance analysis. It not only works for OpenCL applications, but also for any other compilers using an OpenCL backend, including the ComputeCpp and LLVM runtimes which are using OpenCL and SPIR-V to target Intel GPUs.

The Intercept Layer generates a `trace.json` file which can be displayed as a visual timeline in either Intel VTune or Google Chrome. Unfortunately, in this particular situation, the information supplied by the graphic mode was not detailed enough, with both traces looking almost identical (except for the SYCL one being longer), so the raw trace files had to be manually inspected. Listing 4.1 shows a representative snippet of the json traces for each of the two programs, specifically chosen to highlight the differences between them. The traces were generated by running the smallest test problem (128×128) on the Intel Iris Pro 580, compiled with Intel's SYCL implementation.

Observing the entries in Listing 4.1, it becomes apparent that SYCL uses a callback system, relying on `clSetEventCallback()`, `clGetEventInfo()` and `clReleaseEvent()` API calls to verify whether a kernel has finished executing before starting another kernel that is dependent on the completion of the previous one. In some cases, there is a substantial delay between the point when all the dependencies of a kernel have been resolved (i.e. all prerequisite kernels have completed) and the kernel being enqueued. In addition, the SYCL trace contains more than 3 times as many API function calls as the OpenCL trace. Most of those extra calls are `clSetKernelArg()`, suggesting that more arguments than necessary are sent. The host overheads are further compounded by the small size of the inputs included with the LBM code and the high total number of kernels that are submitted (40000 for the 128×128 input, 80000 for the 256×256 input and 20000 for the 1024×1024 input).

Although the Intercept Layer cannot be used to profile the AMD and NVIDIA runs, because hipSYCL — which is currently the only official option for targeting discrete graphics cards— does not compile down to OpenCL, it is still possible to estimate the GPU host overheads by removing all the content inside the kernels, as well as all the buffer accessors. Submitting 80000 empty kernels on the Radeon VII takes 6s using SYCL, compared to just 0.7s using OpenCL. Therefore, even if no computations are carried out on the GPU device, the SYCL version of LBM will still not be able to match the OpenCL version. One of the reasons for hipSYCL's high scheduling latency could be the

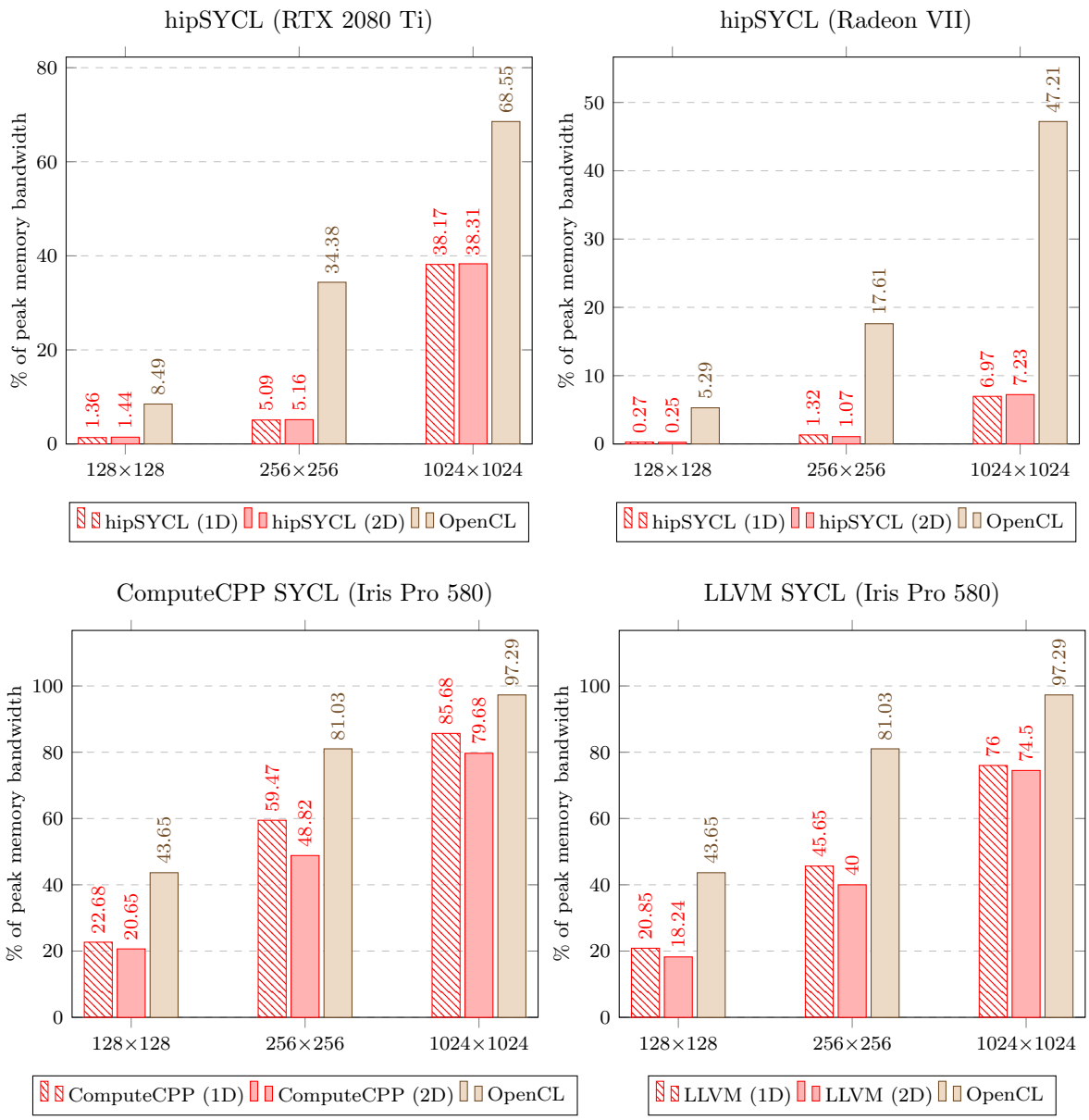


Figure 4.1: Initial GPU results for lattice-boltzmann.

usage of out-of-order queues. Because the CUDA and HIP backends only support in-order queues, the out-of-order behaviour and kernel task graph must be manually implemented in hipSYCL, which is difficult and can result in inefficiencies. In contrast, OpenCL already provides in-order queues and can also sometimes have access to lower level, more efficient driver APIs.

Re-examining some of the entries in the trace files, the offloaded SYCL and OpenCL kernels themselves achieve runtimes that are close to each other. The RTX 2080 Ti and Radeon VII results in Figure 4.1 also show that only a relatively small fraction of the peak bandwith is achieved, indicating that the smaller input sizes used by LBM are not sufficient to take advantage of the massively parallel nature of modern GPUs. The 128×128 and 256×256 problems were therefore discarded and another two larger input grids (2048×2048 and 4096×4096) were constructed in order to saturate task throughput on the GPUs. The number of iterations was also reduced from a few tens of thousands to just 1000 for all input sets to ensure that the runtimes are dominated by the kernels execution instead of host and scheduling overheads.

```
{"name":"clEnqueueNDRangeKernel( lbm )", "ts":500992, "dur":26}
{"name":"clSetKernelArg", "ts":501021, "dur":0}
{"name":"clSetKernelArg", "ts":501022, "dur":0}
{"name":"clSetKernelArg", "ts":501023, "dur":0}
{"name":"clSetKernelArg", "ts":501024, "dur":0}
{"name":"clSetKernelArg", "ts":501025, "dur":0}
{"name":"clSetKernelArg", "ts":501026, "dur":0}
{"name":"clSetKernelArg", "ts":501027, "dur":0}
{"name":"clSetKernelArg", "ts":501027, "dur":0}
{"name":"clSetKernelArg", "ts":501028, "dur":0}
{"name":"clSetKernelArg", "ts":501029, "dur":0}
{"name":"clEnqueueNDRangeKernel( lbm )", "ts":501030, "dur":20}
{"name":"clSetKernelArg", "ts":501052, "dur":0}
```

(i) OpenCL trace

```
{"name":"SYCL_lbm", "ts":448540, "dur":31}
{"name":"clGetEventInfo", "ts":448608, "dur":0}
{"name":"clReleaseEvent", "ts":448668, "dur":0}
{"name":"clSetKernelArg", "ts":448678, "dur":0}
{"name":"clSetKernelArg", "ts":448679, "dur":0}
{"name":"clSetKernelArg", "ts":448680, "dur":0}
{"name":"clSetKernelArg", "ts":448681, "dur":0}
{"name":"clSetKernelArg", "ts":448682, "dur":0}
{"name":"clSetKernelArg", "ts":448683, "dur":0}
{"name":"clSetKernelArg", "ts":448684, "dur":0}
{"name":"clSetKernelArg", "ts":448685, "dur":0}
{"name":"clSetKernelArg", "ts":448686, "dur":0}
{"name":"clSetKernelArg", "ts":448686, "dur":0}
{"name":"clSetKernelArg", "ts":448687, "dur":0}
{"name":"clSetKernelArg", "ts":448688, "dur":0}
{"name":"clSetKernelArg", "ts":448689, "dur":0}
{"name":"clSetKernelArg", "ts":448690, "dur":0}
{"name":"clSetKernelArg", "ts":448690, "dur":0}
{"name":"clSetKernelArg", "ts":448691, "dur":0}
{"name":"clSetKernelArg", "ts":448692, "dur":0}
{"name":"clEnqueueNDRangeKernel( SYCL_lbm )", "ts":448693, "dur":28}
{"name":"clSetEventCallback", "ts":448724, "dur":2}
{"name":"clFlush", "ts":448726, "dur":0}
{"name":"clGetEventInfo", "ts":448730, "dur":0}
{"name":"clReleaseEvent", "ts":448746, "dur":0}
{"name":"clSetKernelArg", "ts":448763, "dur":0}
```

(ii) SYCL trace

Listing 4.1: Comparison between OpenCL and SYCL traces. The `ts` field displays the tracing clock timestamp of each event and the `dur` field displays the tracing clock duration of completed events (both at microsecond granularity). For simplicity, the thread and process IDs of each trace event have been omitted. The extra API calls used by the SYCL runtime are shaded in light red.

The results obtained after using the modified inputs are shown in Figure 4.2 and the performance of SYCL is more in line with what was initially expected. On the 4096×4096 problem, ComputeCpp and LLVM SYCL both achieve 0.98× of the OpenCL baseline. HipSYCL also reveals a large improvement compared to the previous results, achieving up 0.9× of the baseline on the RTX 2080 Ti and up 0.8× on the Radeon VII, where the most significant difference is seen. In almost all instances, the 2D-range version performs slightly better that the linear one, contrary to the results observed in Figure 4.1.

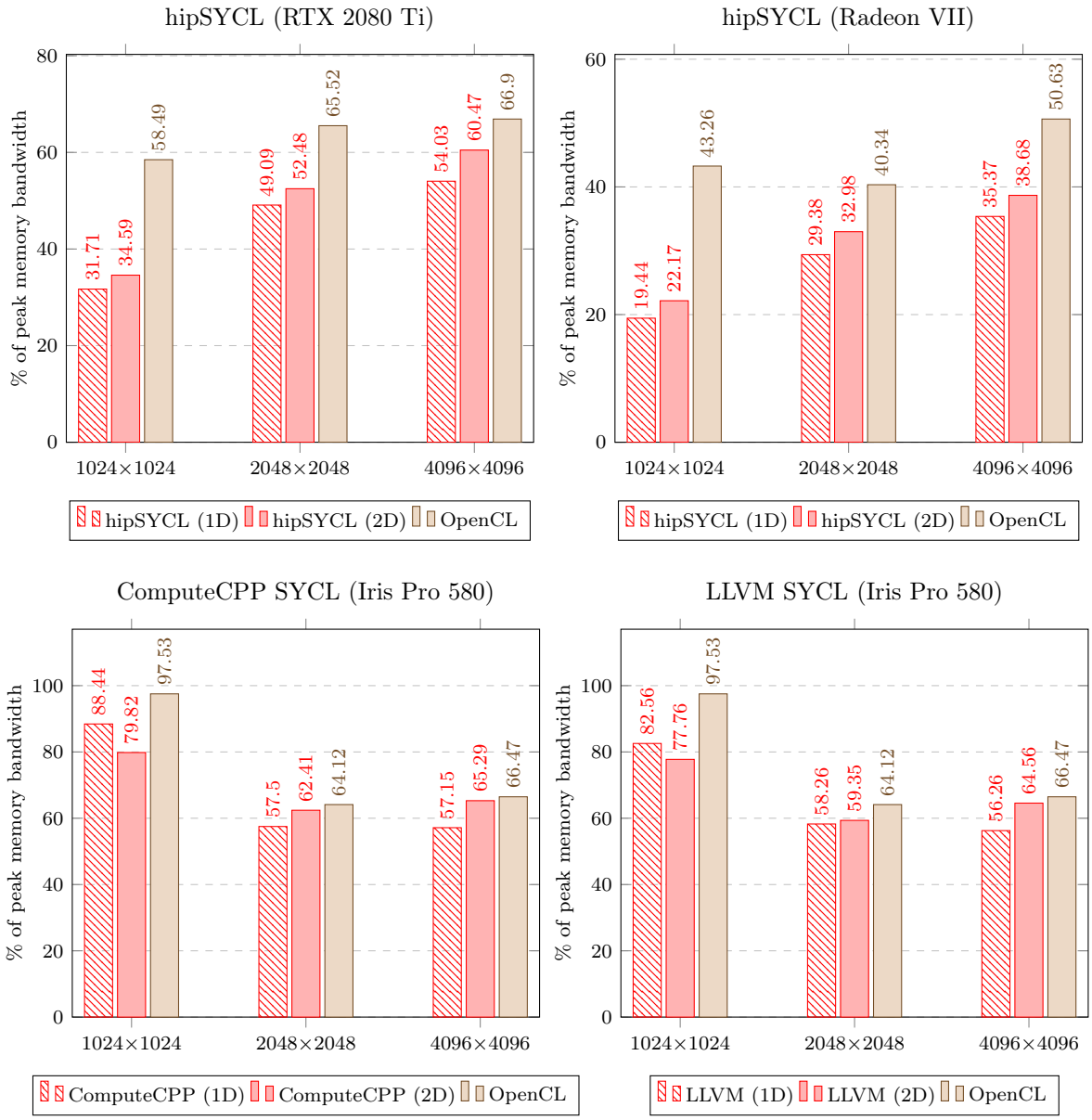


Figure 4.2: GPU results for lattice-boltzmann using custom inputs.

4.2.2 CPU Results

All three of the SYCL compilers can target any CPU. Figure 4.3 shows the CPU results, using ComputeCpp for compiling SYCL. The custom inputs developed in the previous section are used in order to account for the issues which affected the GPU performance. OpenMP results are also included for a complete comparison.

SYCL only reaches up to 4% of the peak memory bandwidth, which is several orders of magnitude below the bandwidth achieved by both the OpenCL and OpenMP versions. Very similar performance drops can also be observed when using the LLVM SYCL and hipSYCL compilers. The poor performance of the LBM code is contradictory to other recent studies which have proven that SYCL is able match raw OpenMP [55, 56].

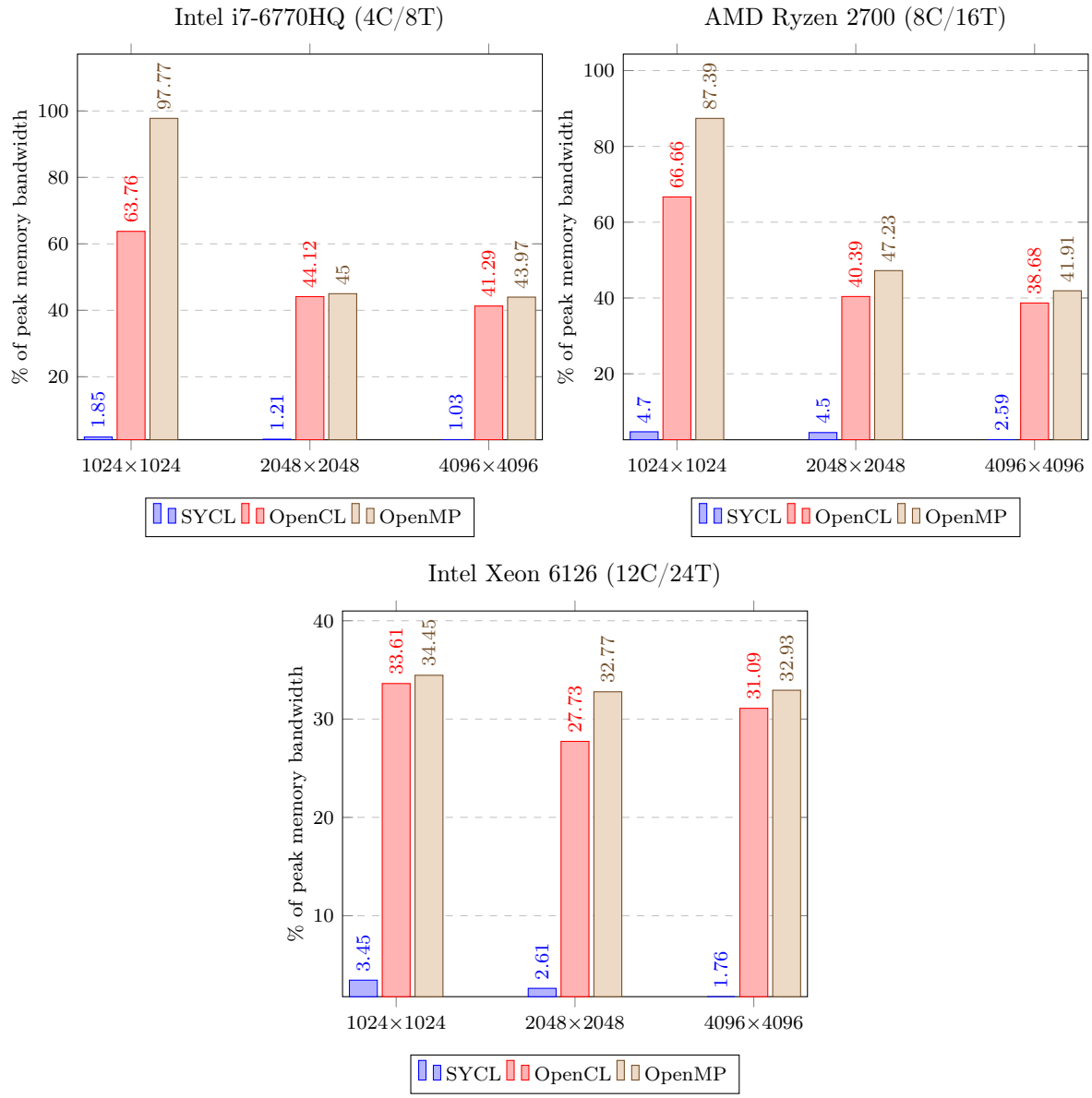


Figure 4.3: CPU results for lattice-boltzmann.

This turns out to occur due to a known issue that manifests in pure-library SYCL implementations when running `nd_range` kernels [42]. The reduction method in LBM requires the use of multiple explicit work group barriers, which are only available through the `barrier()` function of the `nd_item` class. In order to guarantee that each work item will be able to reach the barrier at some point in time, the SYCL runtime must spawn as many threads as there are work items. Launching such a high number of threads results in a severe impact on CPU performance and explains the results in Figure 4.3.

One potential solution is offered by hipSYCL via the hierarchical parallel execution interface. The hierarchical `parallel_for` invoke exposes the same functionality as the `nd_range` class, albeit structured in a different way. It requires an outer `parallel_for_work_group` call to instantiate the work groups, followed by one or more inner `parallel_for_work_item` calls to loop through the work items. The work groups are implicitly synchronised at the end of each inner parallel kernel. Listing 4.2 shows an example of a hierarchical kernel compared to an `nd_range` equivalent. This approach of multi-threading across work groups and looping over work items is much more efficient on CPUs, although it reduces performance on GPUs and other devices (unless conditional compilation directives are used to integrate both solutions).

```
cgh.parallel_for<class kernel>(sycl::nd_range<2>{global_size, local_size},
                               [=](sycl::nd_item<2> item) {
  // [kernel code]
  // Explicit work group barrier
  item.barrier(sycl::access::fence_space::local_space);
  // [kernel code]
});
```

(i) `nd_range` kernel

```
cgh.parallel_for_work_group<class kernel>(num_work_groups,
                                          local_size,
                                          [=](sycl::group<2> group_handler) {
  // [work group code]
  group_handler.parallel_for_work_item([=] (sycl::h_item<2> item) {
    // [work item code]
  });
  // Implicit work group synchronisation
  group_handler.parallel_for_work_item([=] (sycl::h_item<2> item) {
    //[work-item code]
  });
  // [workgroup code]
});
```

(ii) hierarchical kernel

Listing 4.2: Synchronisation inside an `nd_range` kernel (top) and a hierarchical kernel (bottom).

4.3 Neutral

As with the LBM results, the runtimes showed in this section are averaged across five runs to remove any variability. The `neutral` mini-app implements a reduction for tracking the number of collision, facet and census events that have occurred. Unfortunately, as explained in Section 4.2.2, SYCL reductions cannot be written in a manner that achieves good performance on both CPUs and GPUs. As the reduction in this case only serves logging purposes and does not impact the outcome of the particle simulation, it was excluded from the SYCL port and removed from all of the other implementations when collecting timings for this comparison.

The hipSYCL version fails to build due to the `threefry2x64` macro definition in the `Random123` library not being marked as `__host__ __device__` by the CUDA backend, which means that no discrete GPU results can be presented. LLVM SYCL could not be built on the Intel Xeon 6126 using Intel's OpenCL 18.1 runtime, so only the ComputeCpp results are shown.

Figure 4.4 shows a comparison between SYCL, Kokkos and OpenMP on the Intel Xeon CPU [1]. On the central square and streaming problem, SYCL matches the performance of OpenMP and achieves a speed-up of 1.15× compared to the Kokkos baseline. The scattering problem exhibits the largest disparity; SYCL reaches 0.85× of the Kokkos performance and 0.75× of the OpenMP performance. One possible explanation for the decreased performance of SYCL compared to the other two inputs could be a missed optimisation in the `collision_event()` function, which is the main path taken by the `solve_transport_2d` kernel when running the scattering problem.

[1] Although the SYCL port also runs successfully on the Intel i7-6770HQ, AMD Ryzen 2700 and Intel Iris Pro 580, the HPC Zoo only has Kokkos modules available for Intel Skylake E-class CPUs and NVIDIA Volta GPUs. This limits the comparison to only the Intel Xeon processor, due to the previously mentioned hipSYCL issue when building `neutral_sycl` for GPUs.

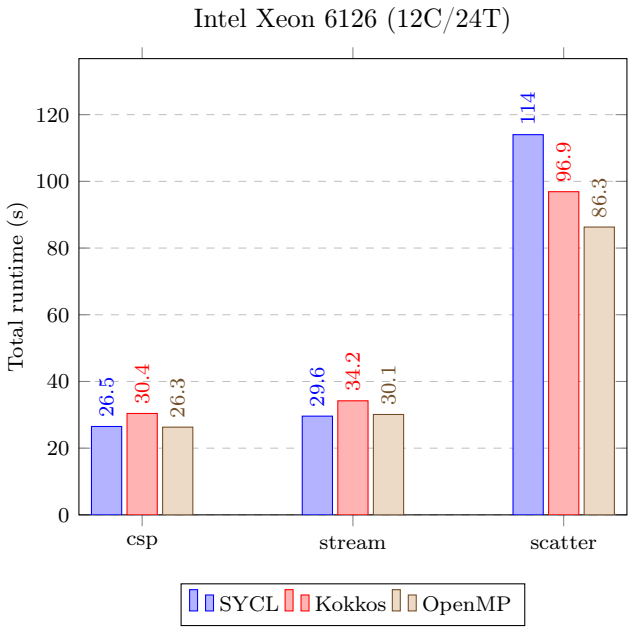


Figure 4.4: CPU results for `neutral`.

4.4 Productivity

The productivity of developing and maintaining various code bases is a key aspect to consider before writing a new port of a mini-app. Figure 4.5 shows the total lines of source code for each version of the LBM and `neutral` applications, normalised to the shortest implementation of the respective code. Although other more elaborate qualitative metrics exist for evaluating productivity, such as the ones proposed by Funk [57] and Pennycook [58], the source lines of code capture the verbosity of a language, which can still give a good indication for the level of developer effort required to write an application in a given model. To this end, the `wc -l` Unix command was used. The count covers both the host code and the kernels and includes comments, a crucial part of the code that can often aid productivity.

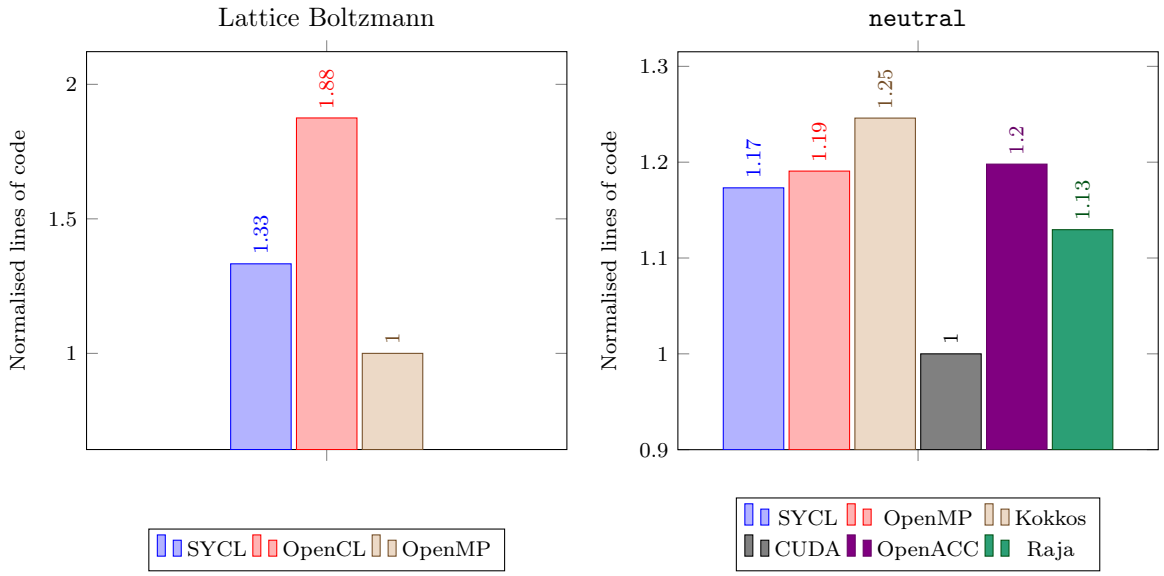


Figure 4.5: Lines of code for each implementation of the lattice-boltzmann and `neutral` applications (normalised to the smallest implementation).

As expected, the OpenMP implementation of LBM is the shortest, followed by SYCL and OpenCL. The difference observed between the lines of code of each version is accentuated by the smaller size of the code, highlighting the section of boilerplate host code needed for the initial setup of OpenCL and SYCL applications.

In the case of the `neutral` mini-app, the verbosity of SYCL is comparable to that of the other models. The reason for the significantly smaller size of the CUDA implementation is the fusion of the `collision_event()`, `facet_event()` and `census_event()` functions inside the body of the `solve_transport_2d()` function. Although it is a high-level programming model, the Kokkos version has the most lines of code due to the choice of defining a separate functor for the main computational kernel.

Chapter 5

Conclusion

5.1 Achievements and Reflection

This section addressed the state of the project with respect to the original objectives that were presented in Section 1.5. The majority of the project was focused on the process of migrating the lattice-boltzmann CFD application and the `neutral` Monte Carlo particle transport mini-app to SYCL implementations. Both ports were successfully completed and the performance of SYCL was tested across a wide range of hardware platforms.

The original research hypothesis stated that SYCL applications would be able to achieve a similar level of performance to OpenCL, while also offering the advantages associated with the use C++ features and abstractions. The results of this study demonstrate that, in most cases, SYCL is capable of attaining similar performance to OpenCL. In the case of the `neutral` mini-app, the performance of the SYCL implementation was comparable with the OpenMP version and often surpassed the performance of Kokkos.

The most mature and reliable implementation is currently Codeplay's ComputeCpp ecosystem. It unfortunately is fairly restrictive in terms of the devices that can be targeted, due to its requirement for SPIR-V, which is only met by some CPUs and a limited number of Intel integrated GPUs. In particular, this limits the support available for NVIDIA and AMD GPUs, which do not provide support for standard intermediate representations such as SPIR or SPIR-V. In 2017, Codeplay added experimental support for NVIDIA hardware via PTX (NVIDIA's intermediate ISA) [59] and earlier this year they have also announced that updated support will be added for CUDA devices through DPC++, Intel's SYCL compiler, which would remove the reliance on an OpenCL layer [60].

Intel aims to incorporate SYCL within DPC++ (Data Parallel C++) as part of the oneAPI unified programming model. The oneAPI project is still in beta and under heavy development [40], but the approach of leveraging the LLVM/Clang compiler infrastructure shows great promise in the future and will allow targeting a much broader range of devices.

5.2 Future Work

There is significant scope for extending the SYCL ports implemented during this project and for further exploring the performance of SYCL implementations as they evolve. This section provides some examples for potential routes to expand the work undertaken during this study.

5.2.1 Extending the SYCL Ports

As the SYCL programming standard matures and improves, it will be possible to add new features to the LBM and `neutral` implementations. The latest on-coming version of SYCL is the SYCL 2020 Provisional Specification (release 1) [61], which aims to introduce a number of new concepts:

- SYCL 2020 will be based on C++17 instead of C++11, allowing the use of more modern abstractions and simplifying the code.
- A new reduction operation that can be launched through a `parallel_reduce` kernel will be introduced. At the moment, two alternatives for implementing this construct are being considered, although the final version of the SYCL specification will probably only include one approach. It would be interesting to investigate how well the chosen reduction mechanism will perform compared to the builtin reductions used by Kokkos and other similar parallel programming models.

Although not mentioned in the provisional specification, there have been proposals from Intel and Codeplay that suggested introducing support for multiple kernel API calls within a single command group [62]. This is a common use case and could potentially improve the structure of the lattice-boltzmann code, which requires accessors to be redefined for every queue submission [1].

5.2.2 Targeting other Architectures

Unlike the previous specifications, which required an implementation to be built on top of an OpenCL backend in order to be considered fully compliant, SYCL 2020 introduces the concept of generic backends. Currently, the only SYCL implementation not defined on top of OpenCL is hipSYCL, although this change could prompt other implementations to adopt a similar approach, expanding the range of hardware platforms that can be targeted.

Another avenue that has not been explored during this project is the performance of SYCL on ARM hardware. At the time of writing this, the only supported ARM accelerators are the Mali line of GPUs, which can be targeted using ComputeCpp since version 1.1.0, although new options might emerge in the future.

[1] It's interesting to note that submitting multiple kernels to a single command group happens to work correctly in hipSYCL, even though it is illegal according to the specification — and not officially supported even in hipSYCL. Having said that, because of the way hipSYCL is implemented, the total size of all local memory allocations needs to be determined before executing the kernels. Therefore, using the same local memory accessor in multiple kernels might cause it to allocate a local memory segment of the requested size for each submitted kernel, increasing the total allocation, which can have a substantial negative effect on GPU occupancy.

References

[1] NVIDIA. NVIDIA Turing GPU Architecture Whitepaper. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[2] Khronos SYCL Working Group. SYCL Overview. https://www.khronos.org/sycl/.

[3] M. Martineau and S. McIntosh-Smith. Exploring on-node parallelism with neutral, a monte carlo neutral particle transport mini-app. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 498–508, 2017.

[4] Khronos Group. SYCL 1.2.1 API Reference Guide. https://www.khronos.org/files/sycl/sycl-121-reference-card.pdf, 2020.

[5] Onur Celebioglu, Ramesh Rajagopalan, and Rizwan Ali. HPC Cluster Interconnect. *POWER*, 7, 2004.

[6] GW4 Alliance. GW4 Isambard. https://gw4.ac.uk/isambard/.

[7] TOP500 List. https://www.top500.org/lists/top500/2020/06/.

[8] Mathieu Lobet, Matthieu Haefele, Vineet Soni, Patrick Tamain, Julien Derouillat, and Arnaud Beck. High-Performance Computing at Exascale: Challenges and Benefits, 2018.

[9] Simon J Pennycook, Jason D Sewall, and Victor W Lee. Implications of a metric for performance portability. *Future Generation Computer Systems*, 92:947–958, 2019.

[10] JR Neely. DOE centers of excellence performance portability meeting. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2016.

[11] Stephen Lien Harrell, Joy Kitson, Robert Bird, Simon John Pennycook, Jason Sewall, Douglas Jacobsen, David Neill Asanza, Abaigail Hsu, Hector Carrillo Carrillo, Hessoo Kim, et al. Effective performance portability. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 24–36. IEEE, 2018.

[12] Simon J Pennycook, Jason D Sewall, and Victor W Lee. A metric for performance portability. *arXiv preprint arXiv:1611.07409*, 2016.

[13] James Reinders. VTune performance analyzer essentials. *Intel Press*, 2005.

[14] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[15] Tom Deakin, James Price, Matt Martineau, and Simon McIntosh-Smith. Evaluating attainable memory bandwidth of parallel programming models via BabelStream. *International Journal of Computational Science and Engineering*, 17(3):247–262, 2018.

[16] Richard F Barrett, Paul S Crozier, Douglas W Doerfler, Simon D Hammond, Michael A Heroux, Paul T Lin, Heidi K Thornquist, Timothy G Trucano, and Courtenay T Vaughan. Summary of work for ASC L2 milestone 4465: Characterize the role of the mini-application in predicting key performance characteristics of real applications. *Sandia National Laboratories, Tech. Rep. SAND*, 4667:2012, 2012.

[17] Michael A Heroux et al. Improving performance via mini-applications. *Sandia National Laboratories, Technical Report SAND2009-5574*, 3, 2009.

[18] Khronos Group. SYCL 1.2.1 Khronos Specification. https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf, Apr. 2020.

[19] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965.

[20] Robert H Dennard, Fritz H Gaensslen, Hwa-Nien Yu, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[21] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual international symposium on computer architecture (ISCA)*, pages 365–376. IEEE, 2011.

[22] Richard M Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, 1978.

[23] Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri, and Shihjong Kuo. Intel AVX: New frontiers in performance improvements and energy efficiency. *Intel Whitepaper*, 19(20), 2008.

[24] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, et al. The ARM scalable vector extension. *IEEE Micro*, 37(2):26–39, 2017.

[25] Joel M Tendler, J Steve Dodson, JS Fields, Hung Le, and Balaram Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, 2002.

[26] John D Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E Lefohn, and Timothy J Purcell. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library, 2007.

[27] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering*, 12(3):66–73, 2010.

[28] Khronos Group. The OpenCL Specification. https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf, Apr. 2020.

[29] David Kirk et al. NVIDIA CUDA software and GPU parallel computing architecture. In *ISMM*, volume 7, pages 103–104, 2007.

[30] AMD. HIP Documentation. https://rocmdocs.amd.com/en/latest/Programming_Guides/Programming-Guides.html#hip-documentation.

[31] AMD. AMD ROCm Platform Documentation. https://rocmdocs.amd.com/en/latest/.

[32] Leonardo Dagum and Ramesh Menon. OpenMP: An industry standard API for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[33] Matt Martineau, Simon McIntosh-Smith, and Wayne Gaudin. Evaluating OpenMP 4.0's effectiveness as a heterogeneous parallel programming model. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 338–347. IEEE, 2016.

[34] CAPS Enterprise. Cray Inc. and NVIDIA and the Portland Group: The OpenACC application programming interface, version 3.0. https://www.openacc.org/specification.

[35] 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.

[36] David A Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J Kunen, Olga Pearce, Peter Robinson, Brian S Ryujin, and Thomas RW Scogland. RAJA: Portable performance for large-scale scientific applications. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 71–81. IEEE, 2019.

[37] RAJA Performance Portability Layer. https://github.com/LLNL/RAJA.

[38] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[39] Ruyman Reyes, Gordon Brown, Rod Burns, and Michael Wong. SYCL 2020: More than meets the eye. In *Proceedings of the International Workshop on OpenCL*, pages 1–1, 2020.

[40] Intel. Intel Project for LLVM technology. https://github.com/intel/llvm/tree/sycl.

[41] Codeplay. ComputeCpp Community Edition. https://developer.codeplay.com/products/computecpp/ce/home/.

[42] Aksel Alpay and Vincent Heuveline. SYCL beyond OpenCL: The architecture, current state and future direction of hipSYCL. In *Proceedings of the International Workshop on OpenCL*, pages 1–1, 2020.

[43] Ronan Keryell. triSYCL—An open source implementation of OpenCL SYCL from Khronos Group, 2014.

[44] Peter Žužek. Implementation of the SYCL heterogeneous computing library (Masters Thesis). https://github.com/ProGTX/sycl-gtx.

[45] Phillip Colella. Defining software requirements for scientific computing, 2004.

[46] Paul K Romano, Nicholas E Horelik, Bryan R Herman, Adam G Nelson, Benoit Forget, and Kord Smith. OpenMC: A state-of-the-art Monte Carlo code for research and development. In *SNA+ MC 2013-Joint International Conference on Supercomputing in Nuclear Applications+ Monte Carlo*, page 06016. EDP Sciences, 2014.

[47] Matt J Martineau. *On the porting and optimisation of physics simulations for heterogeneous parallel processors*. PhD thesis, University of Bristol, 2019.

[48] Matthew Martineau and Simon McIntosh-Smith. The arch project: physics mini-apps for algorithmic exploration and evaluating programming environments on HPC architectures. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 850–857. IEEE, 2017.

[49] Intel. Intel Core i7-6770HQ Processor. https://ark.intel.com/content/www/us/en/ark/products/93341/intel-core-i7-6770hq-processor-6m-cache-up-to-3-50-ghz.html.

[50] NVIDIA. NVIDIA 2080 Ti. https://www.nvidia.com/en-gb/geforce/graphics-cards/rtx-2080-ti/.

[51] AMD. AMD Radeon VII. https://www.amd.com/en/products/graphics/amd-radeon-vii.

[52] Shiyi Chen and Gary D Doolen. Lattice Boltzmann method for fluid flows. *Annual review of fluid mechanics*, 30(1):329–364, 1998.

[53] Khronos Group. SYCL Parallel STL. https://github.com/KhronosGroup/SyclParallelSTL.

[54] Ben Ashbaugh. Debugging and Analyzing Programs using the Intercept Layer for OpenCL Applications. In *Proceedings of the International Workshop on OpenCL*, pages 1–2, 2018.

[55] Sohan Lal, Aksel Alpay, Philip Salzmann, Biagio Cosenza, Nicolai Stawinoga, Peter Thoman, Thomas Fahringer, and Vincent Heuveline. SYCL-Bench: A versatile single-source benchmark suite for heterogeneous computing. In *Proceedings of the International Workshop on OpenCL*, pages 1–1, 2020.

[56] Tom Deakin and Simon McIntosh-Smith. Evaluating the performance of HPC-style SYCL applications. In *Proceedings of the International Workshop on OpenCL*, pages 1–11, 2020.

[57] Andrew Funk, Victor Basili, Lorin Hochstein, and Jeremy Kepner. Application of a development time productivity metric to parallel software development. In *Proceedings of the second international workshop on Software engineering for high performance computing system applications*, pages 8–12, 2005.

[58] Stephen Lien Harrell, Joy Kitson, Robert Bird, Simon John Pennycook, Jason Sewall, Douglas Jacobsen, David Neill Asanza, Abaigail Hsu, Hector Carrillo Carrillo, Hessoo Kim, et al. Effective performance portability. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 24–36. IEEE, 2018.

[59] Codeplay. Adding experimental PTX support to ComputeCpp for NVIDIA hardware. https://www.codeplay.com/portal/news/2017/12/06/adding-experimental-ptx-support-to-computecpp-for-nvidia-reg-hardware.html.

[60] Codeplay. Codeplay contribution to DPC++ brings SYCL support for NVIDIA GPUs. https://www.codeplay.com/portal/news/2020/02/03/codeplay-contribution-to-dpcpp-brings-sycl-support-for-nvidia-gpus.html.

[61] Khronos Group. SYCL 2020 Khronos Provisional Specification. https://www.khronos.org/registry/SYCL/specs/sycl-2020-provisional.pdf, June 2020.

[62] Ruymán Reyes. Towards an Asynchronous Data Flow Model for SYCL 2.2. https://sycl.tech/assets/files/Ruyman_SYCL_Asynchronous_Dataflow.pdf, 2017.

APPENDIX A

Reproducibility

This appendix details the shell environment modules and build commands needed to run the LBM and `neutral` codes on each platform that was used the HPC Zoo. The LBM code is available for download from https://github.com/AndreiCNitu/HPC/tree/develop/lattice-boltzmann. The `neutral_sycl` code is available for download from https://github.com/AndreiCNitu/neutral_sycl.

A.1 LBM

A.1.1 Intel Xeon 6126

```
# COMPUTECPP
module load computecpp/2.0.0
module load khronos/opencl/headers
module load intel/opencl/18.1
make computecpp
./lbm <paramfile> <obstaclefile>

# HIPSYCL
module load hipsycl/0.8.1-prerelease
./lbm <paramfile> <obstaclefile>

# LLVM SYCL
source /nfs/software/x86_64/inteloneapi-beta/2021.1.8/setvars.sh
make llvm
./lbm <paramfile> <obstaclefile>

# OPENCL
module load intel/parallel_studio/2020.2
module load khronos/opencl/headers
module load intel/opencl/18.1
make
./lbm <paramfile> <obstaclefile>
```

```
# OPENMP
module load intel/parallel_studio/2020.2
make
./lbm <paramfile> <obstaclefile>
```

A.1.2 Intel i7-6770HQ

```
# COMPUTECPP
module load computecpp/2.0.0
module load khronos/opencl/headers
module load intel/opencl/18.1
make computecpp
./lbm <paramfile> <obstaclefile>

# HIPSYCL
module load hipsycl/0.8.1-prerelease
./lbm <paramfile> <obstaclefile>

# LLVM SYCL
source /nfs/software/x86_64/inteloneapi-beta/2021.1.8/setvars.sh
make llvm
./lbm <paramfile> <obstaclefile>

# OPENCL
module load intel/parallel_studio/2020.2
module load khronos/opencl/headers
module load intel/opencl/18.1
make
./lbm <paramfile> <obstaclefile>

# OPENMP
module load intel/parallel_studio/2020.2
make
./lbm <paramfile> <obstaclefile>
```

A.1.3 AMD Ryzen 2700

```
# COMPUTECPP
module load computecpp/2.0.0
module load khronos/opencl/headers
module load intel/opencl/18.1
make computecpp
./lbm <paramfile> <obstaclefile>

# HIPSYCL
module load hipsycl/0.8.1-prerelease
./lbm <paramfile> <obstaclefile>

# LLVM SYCL
source /nfs/software/x86_64/inteloneapi-beta/2021.1.8/setvars.sh
make llvm
./lbm <paramfile> <obstaclefile>
```

```
# OPENCL
module load intel/parallel_studio/2020.2
module load khronos/opencl/headers
module load intel/opencl/18.1
make
./lbm <paramfile> <obstaclefile>

# OPENMP
module load intel/parallel_studio/2020.2
make
./lbm <paramfile> <obstaclefile>
```

A.1.4 Intel Iris Pro 580

```
# COMPUTECPP
module load computecpp/2.0.0
module load khronos/opencl/icd-loader
module load gcc/7.4.0
make computecpp
./lbm <paramfile> <obstaclefile>

# LLVM SYCL
source /nfs/software/x86_64/inteloneapi-beta/2021.1.8/setvars.sh
make llvm
./lbm <paramfile> <obstaclefile>

# OPENCL
module load intel/parallel_studio/2020.2
module load khronos/opencl/icd-loader
make build
./lbm <paramfile> <obstaclefile>
```

A.1.5 NVIDIA RTX 2080 Ti

```
# HIPSYCL
module load hipsycl/0.8.1-prerelease
make 2080ti
./lbm <paramfile> <obstaclefile>

# OPENCL
module load cuda/10.1
make
./lbm <paramfile> <obstaclefile>
```

A.1.6 AMD Radeon VII

```
# HIPSYCL
module load hipsycl/0.8.1-prerelease
make radeonvii
./lbm <paramfile> <obstaclefile>
```

```
# OPENCL
module load cuda/10.1
make
./lbm <paramfile> <obstaclefile>
```

A.2 Neutral

A.2.1 Intel Xeon 6126

```
# COMPUTECPP
module load computecpp/2.0.0
module load intel/opencl/experimental/2020.10.3.0.04
source /nfs/software/x86_64/inteloneapi-beta/2021.1.8/setvars.sh
make computecpp
./neutral <param_file>

# KOKKOS
module load kokkos/2.8.00/skx
module unload intel/parallel_studio/2019.4
source /nfs/software/x86_64/inteloneapi-beta/2021.1.8/setvars.sh
make COMPILER=INTEL TARGET=CPU
./neutral.kokkos <param_file>

# OPENMP
source /nfs/software/x86_64/inteloneapi-beta/2021.1.8/setvars.sh
make
./neutral.omp3 <param_file>
```

A.2.2 Intel i7-6770HQ

```
# COMPUTECPP
module load computecpp/2.0.0
module load intel/opencl/18.1
imodule load khronos/opencl/icd-loader
module load gcc/7.4.0
source /nfs/software/x86_64/inteloneapi-beta/2021.1.8/setvars.sh
make computecpp
./neutral <param_file>

# OPENMP
source /nfs/software/x86_64/inteloneapi-beta/2021.1.8/setvars.sh
make
./neutral.omp3 <param_file>
```

A.2.3 AMD Ryzen 2700

```
# COMPUTECPP
module load computecpp/2.0.0
module load intel/opencl/18.1
imodule load khronos/opencl/icd-loader
module load gcc/7.4.0
source /nfs/software/x86_64/inteloneapi-beta/2021.1.8/setvars.sh
```

```
make computecpp
./neutral <param_file>

# OPENMP
source /nfs/software/x86_64/inteloneapi-beta/2021.1.8/setvars.sh
make
./neutral.omp3 <param_file>
```

A.2.4 Intel Iris Pro 580

```
# COMPUTECPP
module load computecpp/2.0.0
module load khronos/opencl/icd-loader
module load gcc/7.4.0
make computecpp
./neutral <param_file>
```