Bachelor Informatica

University of Amsterdam

Breadth First Search in OpenCL: a Performance vs. Portability trade-off?

Rick Watertor

June 17, 2019

Supervisor(s): Ana-Lucia Varbanescu

Abstract

CPUs and GPUs are fundamentally different parallel architectures, aiming to speed-up workload performance in different ways. These differences reflect into fundamental programming differences, which often lead to platform-specific programs for the same functionality.

Despite these fundamental differences, attempts have been made to provide *portable* languages - i.e., languages that support both platforms with a single code base. However, one of the concerns regarding the use of portable programming models is their potential performance loss against native programming models [1], [2]. Additionally, with portability comes the question of whether the programming model is *performance portable*.

In this work, we focus on the performance portability of OpenCL, one of the earliest programming models proposed in the context of multi-core (CPUs) and many-core (GPUs) platforms, within the field of graph processing. Our goal is to answer three questions: 1) How does the performance in OpenCL compare to native implementations? 2) How do various algorithmic modifications change the relative performance of the CPU and GPU?, and 3) Is there a trade-off within OpenCL, too (i.e., can we gain performance by specializing for a given platform within OpenCL)?

To answer these questions, we propose an empirical approach, where we devise several OpenCL variants of Breadth-First Search and compare them against GPU and CPU native implementations, with the aim of closing the gap between the native and portable solutions. We further investigate and quantify the effects of specialization on different platforms by employing platform-specific optimizations, which are (more or less) likely to break the portability, but boost performance.

Our results show that there is no significant performance difference between OpenCL and CUDA. The resulting PTX codes of the kernels were equivalent, so we conclude that there seems to be no inherent reason to use CUDA over OpenCL. We have also identified a clear difference in performance between algorithms on the CPU and the GPU, which means that there are ways to express an algorithm that can favour one platform over another. Favouring a platform in terms of performance is by our definition not performance portable. Finally, we defined the *fair* specializations to be those that do not negatively affect any other platforms and we defined *anti-social* specializations as those that do negatively affect other platforms, and saw that *zero-copy* is a fair specialization, while optimizing for work-group sizes and kernel workload are anti-social specializations.

We conclude that even though OpenCL seems to perform well, care has to be taken when writing algorithms in a platform agnostic manner. A programmer cannot expect their code to be performance portable just by writing it in OpenCL, as the approach and implementation of the algorithm can have a significant influence on performance.

Contents

CHAPTER 1

Introduction

1.1 Context

Graphs are an immensely versatile representation of many real-life structures: they can represent roads, social networks, biology networks, co-authorship networks and many more. Graph analysis can thus be used in many scenarios to interpret these data, and infer new knowledge. For example, in 2017, a group of Dutch scientists used graph processing to uncover offshore financial centers [3].

Graphs are getting increasingly large and complex, requiring increasingly more computational power to process them. However, as the development of processors has hit several walls (i.e., the memory, ILP, and power walls), better performance can no longer be gained by simply waiting for new architectures [4]. Rather, fundamentally different approaches have to be taken to reach better performance.

One of these approaches is the use of parallel architectures, where multiple cores are used in parallel for data processing. However, this paradigm shift - also known as the multicore revolution - requires applications to be parallel, and requires applications to make efficient use of parallel architecture. This is no trivial task for many application domains; graph processing is one of the most challenging application domains in terms of efficient parallelization, as demonstrated by Lumsdaine et al. in 2007 [5]. The main challenges in devising efficient parallel graph processing algorithms are the inherent properties of graph processing workloads: a low compute-to-communication ratio, an irregular workload that challenges traditional parallelism due to load balancing issues, and irregular memory accesses which challenge the performance of memory hierarchies. With potential programmability challenges to exploit parallelism on top of these graph processing challenges, the task seems even more difficult.

Moreover, the performance of graph processing algorithms highly depends on the input graph. Algorithms finely tuned for graphs with few branches will likely perform worse on graphs with many branches, and vice-versa. Similarly, in parallelizing graph processing algorithms, performance also greatly depends on both the method of parallelization and the hardware platform the algorithm runs on. For example, a GPU can process many more nodes simultaneously than a CPU, but a CPU generally has less memory transfer overhead and better tolerance for work-imbalance.

Caution should be taken with directly comparing the two types of hardware, however, since CPUs and GPUs are fundamentally different in architecture, and therefore clock speed, memory latency and modus operandi. With this fundamental hardware difference comes a fundamental difference in programming CPUs and GPUs. Frameworks and languages for multiprocessing exist for both types of processing units. For example, NVIDIA GPUs can use CUDA, OpenCL, and OpenACC, while for other GPUs (ARM, AMD) most commonly supported is OpenCL; for CPUs, the most common models include OpenMP, p-threads, and OpenMPI, but many more exist.

Despite the fundamental difference in architecture, attempts have been made to make *portable* languages. In particular, OpenCL [6], OpenMP [7] (recent), and OpenACC [8] focus on portability. OpenCL is very similar to CUDA (in terms of level of abstraction and main parallelism

constructs) but also supports other computing platforms, like CPUs and FPGAs. OpenMP and OpenACC are both pragma based, and also support CPU and GPU platforms.

In this work, we focus on OpenCL, the solution with the lowest level of abstraction among these portable programming models. OpenCL is based on a standard that abstracts the hardware platform away behind portable APIs and allows *the same program* to run on multiple (types of) platforms. The device, or devices, used to run the program are chosen at runtime, and multiple devices can be used to perform actions in parallel [6].

1.2 Research Questions

While OpenCL is *functionally portable* by virtue of its standard [6], this does not mean that it is *performance portable*. A concern regarding the use of portable programming models is their potential performance loss compared to native programming models. OpenCL has been found comparable in performance to CUDA under fair comparison [1], as well as competitive on the CPU compared to OpenMP [2]. However, while OpenCL may perform similar to CUDA on the GPU and OpenMP on the CPU, these comparisons were done in isolation. The program could be fully specialized towards GPUs and perform terribly on the CPUs. In that case, the program would not be *performance portable*.

Therefore, the question this thesis attempts to answer is: *How does OpenCL perform under portability and specialization in the context of graph processing?*

Three research questions will help in answering the main question:

1. How does the performance in OpenCL compare to native implementations?
2. How do various algorithmic modifications change the relative performance of the CPU and GPU?
3. Is there a trade-off within OpenCL, too? That is, can we gain performance by specializing for a given platform within OpenCL?

To answer these questions, we propose an empirical approach, where we devise several OpenCL variants of Breadth-First Search and compare them against GPU and CPU native implementations, with the aim of closing the gap between the native and portable solutions. We further investigate and quantify the effects of specialization on different platforms by employing platform-specific optimizations, which are likely to break the portability but boost performance.

1.3 Outline

This thesis lies at the intersection of graph processing, parallel programming, and portable programming models. Thus, in Chapter 2, we provide a brief introduction in all these fields. Next, in Chapter 3, we give a detailed explanation about the methodology followed in this research. We present, in Chapter 4, our BFS case-study, diving into the details of the selected algorithms, modifications, and specializations. Our experiments and results, which use the set of implementations defined in Chapter 4, are described and analyzed in Chapter 5. In Chapter 6, we highlight relevant work, and compare in detail our findings against relevant previous conclusions from literature. Finally, in Chapter 7, we present our conclusions, we provide answers to key research questions, and highlight potential directions of future work to combat some of the limitations of the current study.

CHAPTER 2

Background

This chapter starts by providing a gentle introduction to the topic of performance portability (in Section 2.1) and its quantification over multiple platforms. We further describe the OpenCL programming model (in Section 2.2), and relevant aspects of the GPU and CPU architectures (in Section 2.3). Finally, we present a brief overview of graph processing in general and breadth-first search in particular (in Section 2.4).

2.1 Performance Portability

How well a portable implementation maintains its performance across platforms is known as *performance portability*. A measure to quantify performance portability has been defined with the following formula [9]:

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

The key parameters in the performance portability formula are: H as the set of platforms, a as the implementation, p as the problem to be solved, and $e_i(a,p)$ as efficiency. Either *architectural efficiency* or *application efficiency* can be used as $e_i(a,p)$. *Architectural efficiency* represents how well the program performs compared to the theoretical limits of the platform, and therefore, it is a measure of how well the program uses the capabilities of the platform. *Application efficiency* represents how well the performance of the program and the best performing version of the algorithm on the same platform compare.

2.1.1 Application efficiency

Application efficiency of an implementation a solving a problem p, denoted as $e_i(a,p)$, is defined as the execution time of the best known implementation on platform i, b_i, divided by the execution time of the implementation a:

$$e_i(a,p) = \frac{T_i(b_i,p)}{T_i(a,p)}$$

Due to the different execution models for kernels running on CPUs and GPUs (shared memory vs. offloading), there are different ways to define *execution time*. Generally, on CPUs, execution time does not include any other operations than computation, as the data is assumed available and ready to use. For GPUs, the problem is more nuanced: data has to be transferred from the host to the device, and the eventual result has to be transferred back from the device to the host. We call these host-to-device (H2D) and device-to-host (D2H) times respectively. The H2D and D2H times are a form of overhead caused by the use of a GPU.

We note that the impact of H2D and D2H can be significant: if a kernel runs faster on the GPU than on the CPU, the H2D and D2H overhead for the GPU can still make the overall

execution time on the GPU slower. Thus, we will make a clear distinction between kernel time and execution time. Specifically, we define the execution time of the kernel, that is, without any preprocessing, problem loading, H2D and D2H times, as *kernel time*. This is the time actually spent solving the problem on the device.

Additionally, each of these times can differ significantly over different runs (e.g., due to caching or OS overhead). Two ways of measuring execution time are common: measuring the *best* execution time over a number of runs, or measuring the *average* execution time over a number of runs. We choose to use the average execution time as this will better represent the common case, as long as the variance is limited.

2.2 OpenCL

2.2.1 Programming Model

OpenCL [6] is a standard for programming multi- and many-core systems. OpenCL explicitly defines a platform model and a memory model, which serve as its "hardware", and an API to program these models. Portability is achieved by mapping the OpenCL platform (and the API calls) onto different hardware platforms (and real machine instructions). This mapping is different for different hardware architectures, and it is the responsibility of the platform vendor to provide an efficient mapping, and a compiler able to generate a well-performing hardware-specific version from OpenCL code. This means, in theory, OpenCL code is hardware-agnostic, and all hardware specific constructs and/or performance optimizations must be left to the compiler. Compared with hardware-specific programming models (e.g., CUDA for GPUs), OpenCL enables a lot less access to low-level constructs and optimizations, thus the role of the compiler in achieving high-performance implementations is much more significant. OpenCL does still support special-purpose functionality through *extensions*, but these are only optionally supported by the hardware and can therefore break portability [10].

An OpenCL program consists of *host code* and *(device) kernel code*. The host is the driver of the application and can copy data, invoke kernels, and synchronize kernels. Actions on the host device are submitted through the OpenCL API in the form of commands into a command queue. Once the host launches a kernel, OpenCL maps the to-be-processed items onto the processing elements on the compute devices. There can be multiple compute devices, containing multiple compute units, which further contain multiple processing elements.

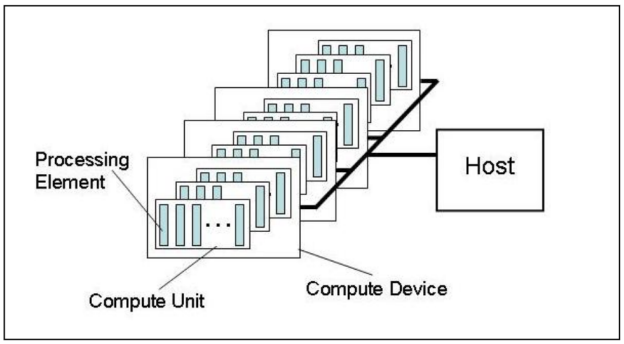


Figure 2.1: The OpenCL platform model. One host drives one or more devices containing one or more compute units. Each compute unit can process one or more processing elements. Source: the OpenCL specification [6].

A compute device can be a CPU, GPU, FPGA, or any other device supported by OpenCL. How the compute units and processing elements are separated over the devices is device-dependent. The compute units schedule and control the processing elements. Each processing element processes a piece of the data, and all processing elements execute the same instruction. These instructions are defined by the device kernel. The kernels are compiled down into the targeted

device-specific code at runtime. This process is where the portability of OpenCL emerges.

Kernels are submitted with a global work size and a local work size. The global work size determines the number of kernel instances to spawn. Each instance is called a *work-item*. The local work size determines how many work-items are grouped into one *work-group*. Each work-group is executed on a single compute unit, concurrently executing the work-items on its processing elements.

2.2.2 Memory Representation

There are different memory regions in OpenCL: *Host Memory*, which is the memory on the host processor, the 'normal' memory C and C++ programs use, and the *Device Memory*, which is the memory on the device. Given that the device can be backed by different architectures, OpenCL provides an abstraction of the memory model, which can be seen in Figure 2.2.

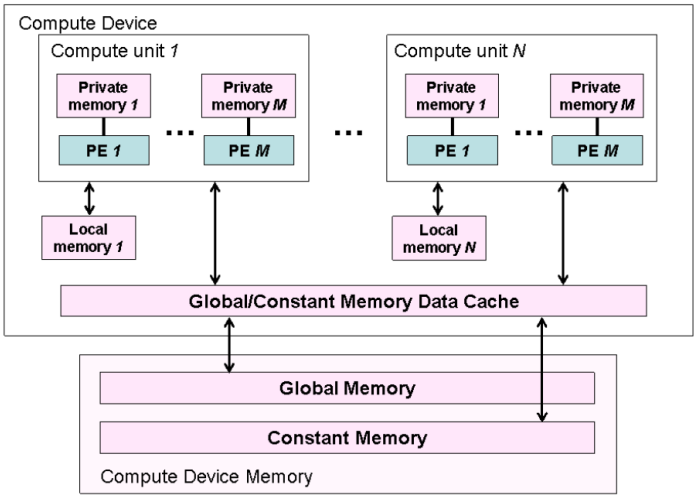


Figure 2.2: The Device Memory model of OpenCL. Global and Constant Memory is stored in the devices' main memory and accessible to all compute units. Local Memory is only available to the compute unit (work-group), and Private Memory is only available to the processing element (work-item). Source: the OpenCL specification [6].

The Device Memory consists of *Global Memory*, *Constant Memory*, *Local Memory* and *Private Memory*. *Global Memory* is accessible from all work-items in all work-groups. *Constant Memory* is a read-only Global Memory for the device. It can be read from all work-items in all work-groups, but it can only be instantiated by the host. *Local Memory* is memory local to each work-group: only work-items in that work-group can access Local Memory. *Private Memory* is the memory associated with one work-item. For example, local variables are allocated in Private Memory, and can only be accessed from the work-item itself.

Since a single device can have multiple compute elements processing data in parallel, memory accesses are prone to clash or overlap. This can easily lead to race conditions and/or cause inconsistencies for computation. For this purpose, OpenCL provides two types of synchronization primitives: *fences* and *atomics*.

Fences are a type of instruction where all running processing threads wait until all other threads have reached the fence. Moreover, a fence ensures that all actions before that fence have completed on all threads. Fences can be applied at work-group level or at global level. The former waits until all threads in the work-group have reached the fence, while the latter waits on all threads.

In some cases, a thread has to read a value, operate on it and then store it. Due to the separation of each step, this can introduce race conditions for parallel threads. The race conditions lead to non-deterministic and often incorrect results.

Atomic operations enforce the read-modify-write instructions to be performed in an indivisible manner, without interruption by other threads. Effectively, at any time, at most one thread will have access to the shared object: this behaviour eliminates race conditions but introduces serialization among parallel threads. While atomics are good for memory consistency, overusing them can cause excessive waiting.

2.3 Platforms

2.3.1 GPUs

Graphics Processing Units (GPUs) are platforms designed to compute a large number of simple pixel transformations in parallel. Due to their massive parallelism, they have been used more and more for regular parallel processing. Due to their original purpose, their architecture, execution model (offloading), and parallelism model (SIMT), are different from those of the regular CPU.

Architecture

The GPU architectures we are looking specifically at are from the CUDA architecture. The CUDA architecture consists of one or more Streaming Multiprocessors (SMs). Each SM can execute hundreds of threads concurrently using the SIMT parallelism model. A SM can process multiple thread blocks concurrently, which in turn can execute multiple threads concurrently. This allows for a high level of parallelism. The concept of work-groups and work-items in OpenCL map on these constructs: a work-group is equivalent to a thread block, and a thread executes one work-item.

The memory model of a GPU is similar to the memory model of OpenCL - the naming conventions are different per vendor, but the concepts of Global Memory, Constant Memory, Local Memory and Private Memory persist. As an example, the NVIDIA CUDA memory model can be seen in Figure 2.3.

On the GPU, Global Memory and Constant Memory are stored in the DRAM. Constant Memory also resides in a specialized region within the Global Memory, but it is faster because it is both read-only and cached. Local Memory is mapped to *Shared Memory* for NVIDIA GPUs (i.e., a special hardware memory region per SM). Shared memory is allocated per thread block: all threads in the same block have access to this memory. It is faster than global memory because it resides closer to the processing units. Private memory is called *Local Memory* for NVIDIA GPUs, which is the memory allocated to each thread. Local Memory resides in the registers of the processing element, which are located the closest to the processing elements, and are therefore the fastest type of memory. Some memory items, such as arrays and large structs do not fit in the registers, and are instead stored in a Local Memory region within the Global Memory [11]. Finally, GPUs also have a *Texture Memory*, which resides in the Global Memory, but is accessed through a specialized read-only cache.

For the remainder of this work, we use OpenCL terminology for each memory region.

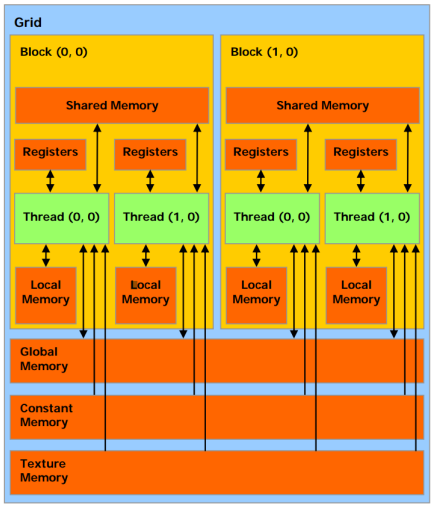


Figure 2.3: The CUDA memory model. Global, constant and texture memory are accessible from all threads (work-items) in all blocks (work-groups). Shared memory is only accessible from within the block. Local memory is only accessible by the thread. Finally, the registers also serve as 'local memory' for memory items that fit. Source: the CUDA Programming Guide [11].

Offloading

Being unable to run as stand-alone processing units, GPUs work as accelerators alongside a host (typically, a regular CPU). Thus, they execute code as dictated by the CPU, through a process called *offloading*. Offloading is the process of moving the data from the CPU onto the GPU, performing computation on the GPU, and moving the results back to the CPU. The computation on the GPU is dictated by a compute *kernel*, which is the routine that is executed on multiple parallel cores of the GPU.

Parallelism model: SIMT

A GPU implements a *Single Instruction, Multiple Threads* architecture, where multiple threads execute the same instruction at the same time (an "extension" of SIMD [12]). Groups of 32 threads, called *warps*[1], execute the same instruction on all threads. In case threads must diverge (for example, due to branches), the different branches are executed sequentially, thus losing performance (i.e., a *diverging warp* takes as long to execute as all the executed paths combined).

In short, GPUs are a source of performance through massive data parallelism, but they suffer from offloading overhead, and their lock-step SIMT execution can lead to severe penalties when code diverges in multiple branches. Thus, GPU acceleration is non-trivial for irregular problems with low parallelism.

2.3.2 CPUs

Multi-Processing

In CPUs, concurrency comes in the form of multi-threading, where multiple threads are scheduled to increase the utilization of the available resources. For multi-core CPUs, these threads can be scheduled on multiple cores to gain parallelism, where multiple threads can run at the same time on different cores, similar to how GPU threads can run at the same time.

A strong difference between CPUs and GPUs is that CPUs have far fewer cores than GPUs. On the other hand, CPU cores generally run at a higher clock speed than GPU cores. Additionally, CPUs can make use of branch prediction and prefetching mechanisms, as well as

[1]These are called warps only on NVIDIA GPUs. The same concept is called a *wavefront* on AMD GPUs, and contains 64 threads instead of 32.

multiple caching layers. These mechanisms make CPUs more flexible for irregular code. CPUs are optimized for latency, while GPUs are optimized for throughput.

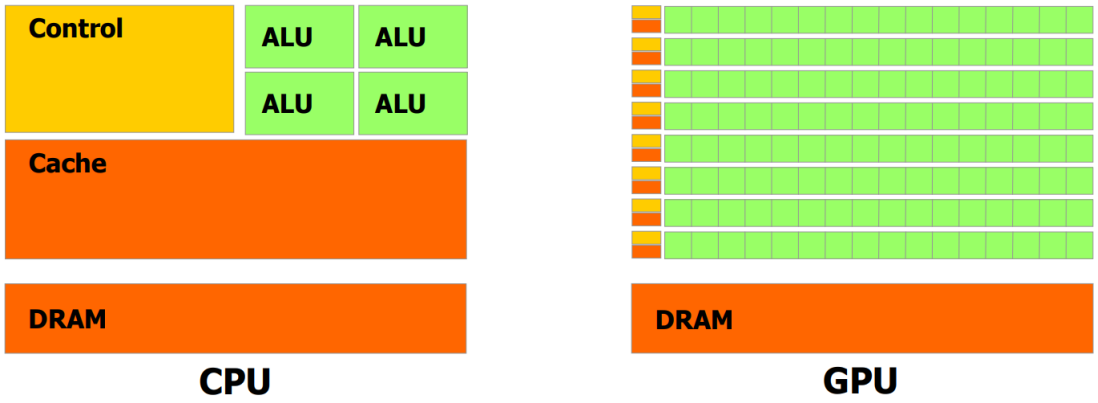


Figure 2.4: Schematic overview of CPUs and GPUs. The CPU has a much larger control unit and cache than the GPU, while the GPU has many more computational units (ALUs) than the CPU. Source: the CUDA Programming Guide [11].

Vectorization

The CPU is backed by a SIMD model through the use of *vectorization*. Vectorization allows a CPU to apply a single instruction to multiple data-items at the same time. The amount of data-items it can hold depends on the size of the register. While the size of the registers has been increasing, the number of items it can process concurrently is still extremely small compared to GPUs.

Memory Model

CPUs have a multi-layer memory hierarchy: levels closer to the CPU are faster, but smaller, than those further away. For example, at the lowest level, there is a hard drive, where memory is persistently stored. This data can be loaded into Main Memory (often known as RAM), which is a much faster memory. From RAM to the processor, there are several layers of caches (typically three: L3, L2, and L1) with increasing speed and decreasing size (from MB to KB, typically). Finally, at the highest level in the hierarchy, there are the registers, hosted inside the CPU. Depending on the CPU, caches can be shared over cores, but these are never programmatically accessible. The only programmatically available memory regions are the registers, Main Memory and the Hard Disk, while the caches are managed automatically by the CPU.

Given this tight memory model of the CPU, most OpenCL memory regions map to the Main Memory. The exception is Private Memory, which maps to registers. These mappings are not true for all architectures: for example, the Texas Instruments OpenCL implementation maps Local Memory directly onto the L2 caches [13].

2.4 Graph Processing

2.4.1 Graphs

Graphs are an abstract data structure composed of *nodes* (or vertices) and *edges*. Nodes represent entities and edges represent the connections between the entities. Examples of graphs are social networks (where nodes are the humans and edges are their relationship) or road networks (where nodes are cities and edges are the roads between the cities). Graph processing refers to the algorithms or applications that process the information in the graph, and often require its traversal, and the application of certain operations per node and/or per edge.

Graphs have various properties. The *size* of a graph is the number of edges the graph contains. The *order* of a graph is the number of nodes it contains. A graph can be directed or undirected. A directed graph has edges that are one-way. Some examples of directed graphs are web graphs (where hyperlinks are the one-way edges), road nets (where you can only drive in one direction) and scientific reference networks (where papers refer to other papers).

For undirected graphs, the *degree* of a node is defined as the number of edges connected to the node. For directed graphs, the *in-degree* and *out-degree* of a node are defined as the number of incoming and outgoing edges respectively.

There are three ways of measuring distances. First, the *distance* between two nodes is defined as the number of edges that it takes to traverse along the shortest path connecting them. Second, the *eccentricity* of a node is defined as the maximum distance from that node to any other node in the graph. Finally, the *diameter* of a graph is the maximum distance between *any* pair of nodes in the graph. A graph is *connected* if the diameter is finite, thus, there is a path from all nodes to any other. It is *disconnected* if there is any pair of nodes that do not have a path connecting them.

The inherent properties of graphs make parallelizing graph processing algorithms difficult [5]: they cause irregular workloads, irregular memory accesses, and a low compute-to-communication ratio. The irregular workload is caused by the unstructured data of graphs. The irregular structure makes it difficult to properly load-balance the problem data. The irregular memory accesses cause poor locality, which challenges the performance of memory hierarchies. Lastly, graph algorithms are generally based on exploring the graph rather than performing complex computation. This means a low compute-to-communication ratio which, combined with the poor locality, causes memory fetches to be the main bottleneck.

2.4.2 Breadth-First Search (BFS)

Breadth-first search is an algorithm which traverses a graph in a structured order. From a given *source node* to start with, it visits the nodes in order of their *depth-level*. The *depth-level* is the shortest distance of the node to the source node. Thus, the algorithm starts at the source node, visits all its children, moves onto their children, and so on. BFS should never visit a node twice.

BFS can be used in many scenarios. Some examples include single source shortest path (SSSP) for unweighted graphs, packet broadcasting, and general traversal of graphs.

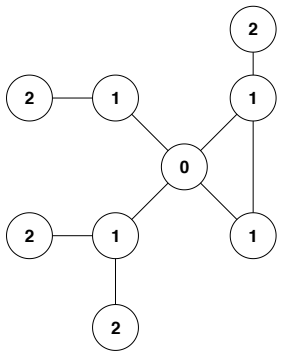


Figure 2.5: Breadth-first search. Starting at a source node 0, breadth-first search then traverses every child of the node layer by layer. Nodes are not visited twice.

Directional BFS

Algorithmically, there are many ways to solve the BFS problem. We look at two: *top-down* (also known as vertex-push) and *bottom-up* (also known as vertex-pull).

Top-down BFS defines a *frontier*, containing all nodes to check. From the frontier, it visits all the unvisited neighbours and marks them as part of the new frontier. This way, each step the frontier *pushes* the frontier status onto its neighbours, until the algorithm terminates.

Bottom-up BFS instead looks at all unvisited nodes. For each unvisited node, it checks if any of its neighbours are in the frontier, and if so, it *pulls* the frontier status onto itself. A benefit of bottom-up BFS is that all nodes only write to themselves. This means that there is no write contention, which allows for massive parallelism. An additional benefit for going bottom-up is that once a vertex finds a neighbour in the frontier, it no longer has to check any other edges. Thus, when the number of edges in the frontier is large, bottom-up can skip a large number of edge checks, which top-down cannot skip. The drawback of going bottom-up every iteration is that when the frontier is small, substantially more checks are performed than for top-down BFS. In short, for small frontiers, top-down BFS is beneficial, while for large frontiers bottom-up BFS is beneficial.

Direction-optimizing BFS

A crucial observation has been made by Beamer et al. [14] that in small-world graphs, there are iterations where the number of edges in the frontier is very large. As described, in these cases, it is more efficient to perform a *bottom-up* iteration instead of a *top-down* iteration.

A dynamic approach is the solution proposed by Beamer et al., where top-down and bottom-up are switched based on a heuristic. The direction-optimizing heuristic consists of two formulas, each with one parameter. The algorithm will switch to bottom-up when the number of edges in the frontier m_f is larger than the number of unvisited edges m_u divided by a tuning parameter α:

$$m_f > \frac{m_u}{\alpha}$$

The algorithm will switch back to top-down when the number of nodes in the frontier n_f is smaller than the total number of nodes n divided by a tuning parameter β:

$$n_f < \frac{n}{\beta}$$

It is possible to tune the parameters α and β to maximize performance. However, in this study, we will choose a static α and β to prevent additional parameters influencing the comparisons.

CHAPTER 3

Method

In this chapter, we define a methodology to test whether a series of modifications can be appropriately represented in the OpenCL programming model. We specifically identify how well OpenCL performs compared to native programming models, and we look at how the relative performance of these modifications vary over GPU and CPU. Finally, we also look at ways to specialize for performance on a target platform within OpenCL. We define the steps to take and the method to evaluate the results. Then, in Chapter 4, we pick the algorithms to apply the methodology on, select benchmark suites as reference implementations and select the approaches to be implemented.

3.1 Terminology

We define a *problem* to be the overall problem we are trying to solve. This can be, for example, sorting a list, or traversing a graph. The *application* attempts to solve the problem by employing *algorithms*. For example, bubble-sort and breadth-first search are algorithms. An algorithm can have multiple *approaches*, which attempt to improve the algorithm in various ways, such as making the algorithm parallelized or vectorized. On the bottom layer there can be multiple *implementations* of an approach, which defines the eventual code a programmer may write, in any programming language they prefer.

3.2 Approach

In order to identify the trade-off between performance and portability in OpenCL, we choose one or more problems to solve. For each problem, we choose one or more algorithms. For each algorithm, we devise or find a basic implementation, on which we will apply various modifications. We choose these modifications from theory and reference implementations. We propose the following taxonomy of modifications:

- Algorithmic modifications - modifications that change the algorithmic *approach*.
- Specialized modifications - modifications that aim to improve performance on a specific target platform. These are made on the programming model level and not the algorithmic level. We further distinguish three cases here:
  - Fair specializations
  - Anti-social specializations
  - Illegal specializations

The fair specializations are specializations that only affect the targeted platform, and do not adversely affect any other platform. For example, applying a modification for the CPU that is compiled away for GPU implementations are fair, as the GPU is not affected at all.

Anti-social specializations are specializations that have an adverse effect on other platforms. While they may improve the performance of a platform, they reduce the performance of another. Anti-social specializations may still be relevant, because the performance gains on a platform may be higher than the losses on another, or in cases where the targeted platform may be of more interest to the end-user because of availability.

Finally, any specializations that break functionality on another platform altogether are *illegal*, as this behaviour is not desired from a portability point of view.

After selecting the modifications to apply, we start with the basic implementation and apply a set of modifications, resulting in several versions of the algorithm. These versions need to be evaluated in terms of performance and performance portability. Our evaluation is empirical: we select a dataset (comprising of a selection of graphs) and a set of hardware platforms (CPUs and GPUs) to perform the empirical evaluation, and measure the raw data (i.e., execution time) for all data- and platform-combinations. We analyze the results guided by our three research questions, as seen in the following paragraphs.

3.2.1 OpenCL vs CUDA

One of the concerns regarding the use of portable programming models is their potential performance loss against native programming models [1], [2]. In this work, research question 1, "*how does performance in OpenCL compare to native implementations?*, addresses this issue: if we can quantify the performance difference between OpenCL and CUDA for our specific use case, we can use this information to asses the "handicap" that OpenCL has over the native CUDA model. If it turns out that there is no (significant) difference between OpenCL and CUDA, there is no need to worry about any systematic performance penalty induced by the programming model; in this case, the two models are *performance equivalent* (for the given case-study).

We quantify the performance disparity between OpenCL and CUDA by defining *programming model efficiency* as a new type of efficiency that directly compares the performance of equivalent implementations that use different programming models. Given a problem p, an application a, a programming model under test i, and a best-known programming model i_a:

$$e_i(a,p) = \frac{T_{i_a}(a,p)}{T_i(a,p)}$$

Or, in words: the efficiency of our programming model i solving problem p is defined by its execution time, compared to the execution time of the *best performing programming model* (on a given architecture) executing the same application a to solve problem p, *using the same approach.* That is, we need equivalent implementations (i.e., a direct port of OpenCL to/from CUDA) for a correct estimation of the programming model efficiency.

We note that keeping the application constant may be a non-trivial task for programming models that differ significantly. What is important is that the preprocessing, postprocessing, algorithm code and graph representation (data structure) should stay equivalent on both platforms. The profiling tools should also profile the equivalent regions of the two programming models. For example, a profiler that captures the execution time of the code region where OpenCL copies data to the GPU, should be represented in an equivalent way in the CUDA programming model. Equivalent algorithm code means that the exact same algorithmic approach should be taken and that the code should be as equivalent as possible. We define the kernels in a generic manner, which can be used as a template for equivalent algorithmic code.

Comparisons of OpenCL and a CPU programming model (e.g., OpenMP or pthreads) can also be made. However, due to the above constraints, such comparisons are made difficult. For example, OpenMP has a much higher abstraction model, which makes it hard to compare and measure code regions and implement equivalent code. While not impossible to make a fair comparison, we have not done so in this thesis, and left this as future work.

3.2.2 Algorithmic Modifications

To answer research question 2: *"how do various algorithmic modifications change the relative performance of the CPU and GPU?"*, we again measure the execution time of each implementation and dataset.

We do some additional analysis on these results to uncover quirks within our implementations. We can make several hypotheses based on the dataset and implementations. We expect the implementation to execute in a certain way, but with the irregularities of graph processing, the results may not always reflect that. We try to identify whether our hypotheses are confirmed or broken due to either the approach and implementation or that it is a fundamental issue with the programming model. This analysis is done in an empirical manner. Based on the hypotheses we set up, we show the relevant results, reason about them and set up further analysis to find *why*. Finally, we classify whether the quirk is part of the algorithmic approach or the programming model.

After we have identified the particularities of each implementation, and have possibly uncovered some insights into the programming model, we express the results in application efficiency in order to find the performance portability of the implementations. We do not use architectural efficiency, as it is very difficult to judge with the different types of graph inputs and subtleties of the platforms. Application efficiency can be measured by comparing the execution time of the algorithm with the *best performing implementation* of the algorithm.

With most algorithms, but especially with graph processing algorithms, defining a *best* reference is rather troublesome. Since graph inputs greatly affect how an algorithm will perform, some algorithms are better suited to some problems than others. For application efficiency in isolation, the reference matters very little, as the same relative results can be reproduced, as long as the same reference is used. However, for the eventual *performance portability* the choice of best performing implementation matters, as the reference algorithm may be unable to run on all platforms. Having a weak algorithm on one platform and a strong algorithm on the other could skew the results.

As mentioned, finding the right reference implementation is not trivial. We choose our references based on the results they report and assume that as long as they are within a good margin of performance compared to other *state-of-the-art* implementations, the eventual performance portability is not significantly skewed. We also prefer well-known frameworks and implementations, which makes it easier to position our results relative to others.

We use application efficiency to compare different implementations on the same platform. We use the performance portability metric as a measure of how performance portable an implementation is.

3.2.3 Specializations

In order to answer research question 3, "*Is there a trade-off within OpenCL too?*", we use the specialized implementations. We compare the difference in performance gained by the specializations for each platform and determine whether the specialization was fair, anti-social or illegal, by comparing its performance gain or loss across platforms.

If there is a performance gain on both the CPU and the GPU, we classify the specialization as a *fair* specialization. If there is a performance gain on the CPU and the performance of the GPU is not affected, or vice versa, we also classify the specialization as a *fair* specialization. If the specialization negatively affects one or more platforms, we define the specialization as an *anti-social* specialization. Finally, if the specialization breaks portability, we classify the specialization as an *illegal* specialization.

CHAPTER 4

Case-study analysis: BFS

In this chapter, we present a case-study for our methodology for studying the performance vs. portability trade-off in OpenCL for graph processing. Specifically, we focus on the BFS application and discuss the reference implementation, the different modifications and specializations, ultimately presenting the set of implementations that will be used for our performance analysis (Chapter 5).

4.1 Selected Algorithm

We use Breadth-First Search as the main algorithm. It has a diverse execution pattern in that it often starts with a small amount of work which dynamically grows and shrinks as the algorithm progresses. The amount of work depends very highly on the structure of the graph.

Our basic unmodified top-down BFS implementation is based on the Rodinia BFS OpenCL implementation, and each step of the algorithm makes use of the top-down kernel. The top-down kernel is applied per node and checks whether the node is part of the frontier. If so, all unvisited neighbours are added to the new frontier and are marked as visited. After all kernels have finished executing, the frontiers are swapped. The entire algorithm terminates once no nodes are added to the new frontier.

Algorithm 1 The top-down kernel that uses a bitmap data structure. For each node in the frontier, it marks all unvisited neighbours as visited, and places them in the new frontier bitmap.

```
1:  function Top-Down(node, frontier, visited, new_frontier, cost, done)
2:      if node ∈ frontier then
3:          frontier ← frontier \ node
4:          for neighbour ∈ neighbours of node do
5:              if neighbour ∉ visited then
6:                  cost[neighbour] ← cost[node] + 1
7:                  new_frontier ← new_frontier ∪ neighbour
8:                  visited ← visited ∪ neighbour
9:                  done ← false
10:     return done
```

With this approach, race conditions can exist. The kernel sets values for a neighbour, which can be a neighbour of other nodes in the frontier. However, the execution order does not matter here. All nodes in the frontier have the same cost, and the other operations are setting constant values. Therefore, regardless of the execution order, the result is correct. The race condition is not problematic.

4.2 Selected Algorithmic Modifications

4.2.1 Naive Direction-optimizing

Direction-optimizing BFS is based on the notion that when faced with a large number of edges in the frontier, it is more efficient to perform *bottom-up* BFS. This naive implementation of direction-optimizing BFS keeps mostly the same top-down approach, but adds a bottom-up variant and dynamically uses that instead. It is naive in the sense that it does not swap data structures. It uses a bitmap for both top-down and bottom-up.

Algorithm 2 The bottom-up kernel. For each unvisited node, it checks if any of its neighbours are in the frontier. If so, this node is marked as to be updated, and the kernel terminates.

```
1: function BOTTOM-UP(node, frontier, visited, updating, cost)
2:     if node ∉ visited then
3:         for neighbour ∈ neighbours of node do
4:             if neighbour ∈ frontier then
5:                 cost[node] ← cost[neighbour] + 1
6:                 updating ← updating ∪ node
7:                 break
```

However, the updating of the bitmaps is delegated to a new kernel called Update. To dynamically switch between top-down and bottom-up, the top-down and bottom-up kernels need to track the number of vertices in the frontier and the number of edges in the frontier. Since there is a race condition in the top-down kernel, the statistics could end up being counted twice, unless the visited bitmap is checked atomically. Since the visited bitmap is checked regularly, it is likely more efficient to use a secondary Update kernel instead.

The Update kernel still uses atomics to update the statistics. This is to prevent race conditions on the statistics, as they are being concurrently updated.

Algorithm 3 The Update kernel. It marks all to-be-updated nodes as visited and places them in the new frontier bitmap. It also computes the number of edges in the frontier and the number of vertices in the frontier using atomics. These are required to dynamically switch between the top-down and bottom-up kernels.

```
1: function UPDATE(node, frontier, visited, updating, amt_frontier_edges, frontier_size, done)
2:     if node ∈ updating then
3:         frontier ← frontier ∪ node
4:         visited ← visited ∪ node
5:         updating ← updating \ node
6:
7:         atomic_add(amt_frontier_edges, |node's neighbours|)
8:         atomic_inc(frontier_size)
9:         done ← false
10:    return done
```

On the host side, the statistics are used to switch between top-down and bottom-up BFS. This switch is based on two parameters: ALPHA and BETA. If the number of edges to check in the frontier is growing, and the number is larger than $no\_of\_unexplored\_edges/ALPHA$, the algorithm switches to bottom-up. If the frontier is shrinking and the number of vertices in the frontier is smaller than $no\_of\_nodes/BETA$, it switches back to top-down. Therefore, if there is a large number of edges to be checked in the frontier, the algorithm switches to bottom-up, and once the frontier is too small, it switches back to top-down.

We use a fixed ALPHA of 15 and a fixed BETA of 18. While the most efficient parameters can be found for each graph, it is out of scope for this research.

4.2.2 Data Structure

Beamer et al. state that using the wrong data structure for direction-optimizing BFS incurs a bigger loss than converting between data structures [14]. Since our basic implementation of BFS does not use a queue for top-down BFS, we choose a modification that uses a queue-based data structure for top-down BFS instead. The queue-based top-down BFS only has to process as many work-items as there are nodes in the queue. For architectures that can process fewer concurrent work-items, this may be beneficial.

Atomics are used to keep the queue free from duplicates. While the Update kernel can be used as well in this implementation, it is required for the Update kernel to process all nodes. The potential benefit of having fewer work-items would be negated by reintroducing the Update kernel.

Algorithm 4 The top-down kernel using a queue data structure. Not all nodes are traversed anymore, reducing the number of work-items. Instead, it requires atomics to ensure the node is inserted into the queue (new_frontier) only once.

```
1: function TOP-DOWN(thread_id, frontier, visited, new_frontier, cost, done)
2:     node ← frontier[thread_id]
3:     for neighbour ∈ neighbours of node do
4:         if atomic_exchange(visited[neighbour], 1) = 0 then
5:             cost[neighbour] ← cost[node] + 1
6:             new_frontier ← new_frontier ∪ neighbour
7:             done ← false
8:     return done
```

4.2.3 Direction-optimizing and Data Structure

To compare direction-optimizing BFS with and without the proper data structures, we implement a direction-optimizing variant that uses the proper data structure. The combined data structure and direction-optimizing implementation uses the queue data structure for top-down BFS and the bitmap data structure for bottom-up BFS.

The queue data structure is beneficial for small top-down frontiers. The direction-optimizing implementation ensures that top-down is only used for small frontier sizes. This means that the queue can remain small, resulting in a small number of work-items. The bottom-up kernel has to check whether any of the neighbours are in the frontier. To check this in constant time, a bitmap structure is used. The bitmap structure also does not require atomics to operate, resulting in more efficient parallelism for the bottom-up cycles.

4.3 Selected Specializations

We choose three specializations to implement in OpenCL: *Zero-copy*, *Optimal work-group size* and *Optimal per-kernel work-item size*. We implement these specializations atop the basic top-down Breadth-First Search implementation, in order to answer the final research question.

4.3.1 Zero-copy

Zero-copy is the act of not copying data. When the graph is loaded on the host device, it is copied over to the kernel execution device. When the execution device is the GPU, this copy is actually mandatory, because the GPU cannot access the same memory as the CPU. However, when the execution device is the CPU, it already has access to the data, as the host is the same CPU. Therefore, for the CPU we do not need to copy any data, which we call *zero-copy*. This may significantly save on H2D and D2H time, as these can be instantly accessed instead. We hypothesize that *zero-copy* is a *fair* specialization targeted for the CPU. We hypothesize it is a fair specialization because the GPU will not be affected by the change, as no differences are introduced there: the CPU simply skips the copying process.

4.3.2 Optimal Work-group Size

CPUs and GPUs have significantly different numbers of cores. A CPU has few cores, while most GPUs can process hundreds of threads in parallel. Threads are scheduled in work-groups, of which the size can be set programmatically. Since the number of cores differs significantly, we hypothesize that the optimal size of the work-groups differs as well. Therefore we hypothesize this specialization to be a *anti-social* specialization. If the optimal work-group sizes are different for the CPU and the GPU, it is required to prefer either platform, which is anti-social in nature.

4.3.3 Optimal workload per kernel

Instead of launching threads in differing sizes, we can also choose to have each thread perform more work. Without this specialization, each kernel performs operations on exactly one vertex before it terminates. However, we can choose to process an arbitrary number of vertices per kernel. These executions will be serialized, as a thread cannot execute multiple instructions at the same time. However, this may lead to a better load-balancing strategy on some platforms. Since some vertices have a high amount of neighbours, and others have very few, giving each thread more vertices to process may lead to a higher chance of having a similar quantity of edges to process.

Similar to the *optimal work-group size* specialization, the optimal per-kernel work-item size may also differ from platform to platform. Therefore we hypothesize this specialization to be anti-social as well.

CHAPTER 5

Performance Evaluation

In this chapter we present an in-depth empirical analysis of our BFS case-study. Our experiments focus on measuring the performance of our various OpenCL versions on different platforms. Moreover, our analysis focuses on the interpretation of these results by comparison against performance results achieved by native programming models, as well as results obtained by state-of-the-art high-performance implementations. Finally, we also present a quantitative analysis of our kernel's performance portability.

5.1 Experimental Setup

In this section we describe the three components of our experimental setup: the hardware platforms, the dataset, and the reference benchmarks used for computing application efficiency.

5.1.1 Hardware and Software Platforms

We run all implementations on both a GPU and a CPU. The system facilitating these runs is the DAS5 [15]. The CPU is a dual 8-core Intel Xeon E5-2630v3 processor, and the GPU is an NVIDIA GeForce GTX TITAN X. More details about the hardware can be found in Table 5.1.

Type	GPU	CPU
Name	NVIDIA GeForce GTX TITAN X	Intel Xeon CPU E5-2630 v3
Clock Frequency	1076 MHz	2400 MHz
Sockets	1	2
Cores per Socket	3072	8 (16 with HT)
Max. work-group size	1024	8192
Host CPU	Intel Xeon CPU E5-2630 v3	Intel Xeon CPU E5-2630 v3

Table 5.1: The hardware used to run the experiments with.

Both devices use OpenCL version 1.2. The GPU runs on CUDA 10.0. We compile all host code with the g++ 6.3.0 compiler parameterized by the -O3 flag.

5.1.2 Dataset

As dataset, we use a mix of synthetic and real-world graphs. For the synthetic graphs, we use a selection of Graph500 graphs. Graph500 graphs all have similar properties but increase in scale. Generally, a higher step on the scale takes twice as long to execute as there are twice the number of vertices. We choose the scale sizes 10, 15, 20, 22 and 23. This is a mix of smaller and larger graphs, which should give a good indication of how the implementations perform on different scales. Graph500 graphs are stochastic Kronecker graphs.

For the real-world graphs, we use a selection of the KONECT and SNAP datasets. From the KONECT collection, we choose the actor collaborations network. From the SNAP collection, we choose the graphs roadNet-CA and web-Google. With these graphs, we represent common real-world graph types, such as social networks, infrastructure networks and web graphs.

The actor collaborations network is a scale-free network. Actors are connected when they appear in the same movie. Multiple edges can exist between the same actors, but these are filtered out by preprocessing as they are excessive for BFS.

The roadNet-CA graph is a directed graph of the California road network. Road networks generally have a much lower maximum degree and therefore have a higher diameter. Where the scale-free graphs we picked have a diameter in the tens of nodes, this large road network has a diameter of 555.

The web-Google network is a directed network of hyperlinks on the internal Google website.

Graph	Vertices	Edges	Eccentricity	Source
Synthetic				
graph500-10	1025	10551	5	Graph500 [16]
graph500-15	32768	441406	7	Graph500 [16]
graph500-20	104857	15700394	7	Graph500 [16]
graph500-22	4194305	64155735	7	Graph500 [16]
graph500-23	8388609	129333677	7	Graph500 [16]
Real-World				
actor-collaborations	382220	33115812	10	KONECT [17], [18]
web-Google	916429	5105039	16	SNAP [19]
roadNet-CA	1971282	5533214	555	SNAP [19]

Table 5.2: A list of the graphs used as dataset. The eccentricity of a graph is the maximum graph distance for a given vertex, thus the number of iterations the BFS algorithm has to perform. Since all tests run from the 0 vertex, we report the eccentricity of the 0 vertex.

5.1.3 Benchmarks

In order to compare all algorithmic modifications and specialized modifications to a fixed point, and position them in the field of research, we compare the performance of each implementation to known benchmarks in the field of graph processing. We choose one for the GPU and one for the CPU.

Gunrock

Gunrock [20] is a frequently used and strongly competitive graph processing framework for CUDA on NVIDIA GPUs. It performs comparably with or better than other graph processing frameworks, especially on BFS.

We run Gunrock with command-line arguments to mimic the behaviour of our implementations. Note that the best possible parameters depend on the graph, GPU and other factors, but we trust the out-of-the-box Gunrock with the minor adjustments described below to suffice as a comparable standard.

We force the source node to 0 such that it is easier to compare the implementations. Starting from different sources could cause a difference in number of iterations, execution time, or even total work. We set the number of runs to 50, of which we report the average kernel time. We also enable idempotence, which means that Gunrock does not have to worry about visiting a node multiple times. Our unmodified and direction-optimizing implementations are also idempotent, so it is only fair to enable that in Gunrock as well. Gunrock also needs to have its direction-optimizing mode manually enabled. This is generally the fastest mode Gunrock runs in, so we enable this for all runs as well.

Therefore, the arguments we run Gunrock with are the following:

```
--src=0 --iteration-num=50 --idempotence --direction-optimized --undirected
```

GAP benchmark

For the CPU we use GAP [21]. GAP provides an implementation representative of the *state-of-the-art*, so it will suffice as a *best performing algorithm*. GAP is written in C++ with OpenMP to provide parallelism.

We run GAP with similar parameters as Gunrock and our implementation. We set the number of runs to 50 and the source node to 0. GAP can never be idempotent, as it uses atomics to ensure that their top-down queue remains valid. Our queue data structure-based implementations are also not idempotent, as they are faced with the same issue.

The argument we run GAP with are the following:

−n 50 −r 0 −s

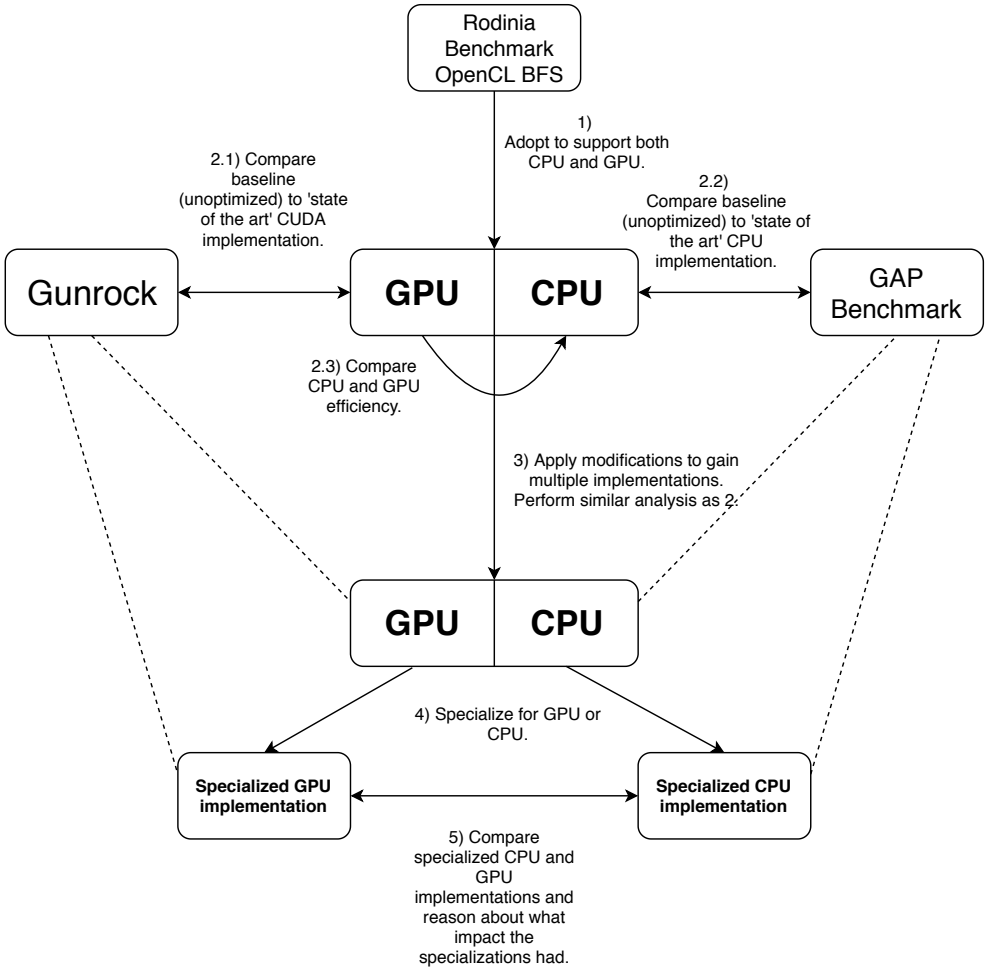


Figure 5.1: The approach of answering the research questions. We use Gunrock as the reference *best performing* algorithm on the GPU, and GAP on the CPU. We use Rodinia's approach as a base unmodified implementation of BFS in OpenCL, and implement the chosen modifications on that.

5.2 CUDA vs OpenCL

We selected two implementations to port to CUDA: the unmodified implementation and the direction optimizing + data structure implementation. The unmodified implementation gives a good baseline on how OpenCL performs compared to CUDA, while the direction optimizing + data structure implementation should yield an indication of how this relative performance evolves due to algorithmic modifications.

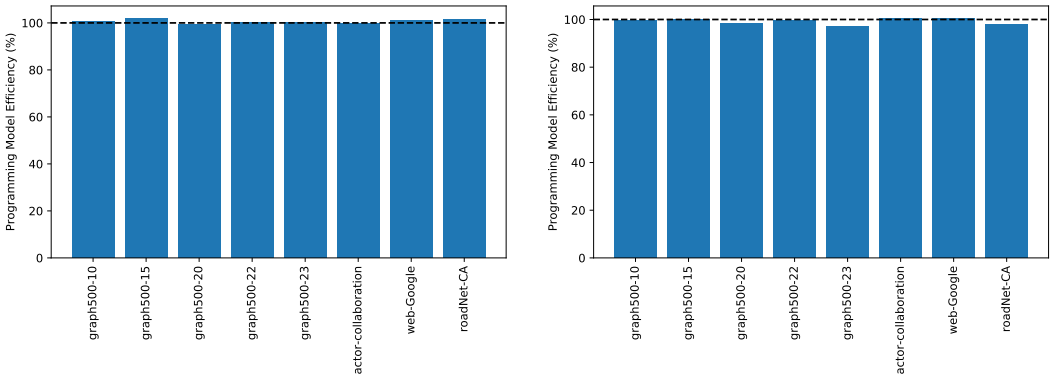
We ported the two implementations to CUDA by finding a direct complement of the OpenCL API calls in CUDA. Both have similar programming models, so most translations are direct and do not lose meaning. Table 5.3 shows a series of direct translations we used to port OpenCL to CUDA.

OpenCL	CUDA
clCreateBuffer	cudaMalloc
clEnqueueReadBuffer	cudaMemcpy
clEnqueueWriteBuffer	cudaMemcpy
clEnqueueNDRangeKernel	...<<<..., ...>>>(...)[1]
clWaitForEvents	cudaEventSynchronize
clGetEventProfilingInfo	cudaEventElapsedTime

Table 5.3: Equivalent or similar API calls in OpenCL and CUDA.

Our hypothesis is that the OpenCL implementations will perform worse than the CUDA ports, as CUDA has a more specific model to their architecture. OpenCL has to map abstractions onto the NVIDIA hardware, where it has to make certain assumptions. While the OpenCL programming model is very comparable to CUDA, differences in scheduling, the memory model, and execution order can have a major impact on performance, especially when faced with atomics and synchronization.

We run these implementations and ports on the entire dataset, and, for each graph, we calculate the *programming model efficiency*. The results are shown in Figure 5.2.



(a) Unmodified BFS

(b) Direction-optimizing + Data structure

Figure 5.2: A comparison of CUDA and OpenCL. The CUDA implementations are directly ported from the OpenCL implementations. Results are expressed as programming model efficiency, calculated with kernel times for the OpenCL and CUDA implementations over 50 runs.

Contrary to our hypothesis, Figure 5.2 shows that the OpenCL and CUDA implementations perform equally well, except for a small amount of noise. To ensure this (somewhat unexpected) behaviour is not a fluke due to our dataset selection, we looked at the *PTX code*[2] outputted by the nvcc compiler and the run-time OpenCL compiler. For the most part, the generated instructions were the same. We only noticed three types of (minor) differences between OpenCL and CUDA: (1) OpenCL targets a higher computing capability, and enables a texturing mode that seems irrelevant for our application. (2) CUDA refers to parameters as values, and whenever the address is needed, it converts them to pointers; OpenCL immediately notices that the parameters are pointers, and therefore declares them as such, so that any conversion does not have to be done later. As a side-effect, the OpenCL code requires fewer registers, as it does not need to

[1]Kernel calls in CUDA are not an API call, but rather part of the language. Work dimensions are defined within the <<< and >>>, while arguments are passed through the function call parameters. In OpenCL, clEnqueueNDRangeKernel takes the work dimensions and all arguments are set through an API call (clSetKernelArg).

[2]PTX code is the equivalent of assembly code for GPUs.

store the result of the conversion anywhere. (3) Lastly, the way OpenCL and CUDA compute the thread id is different: OpenCL adds a value from the %envreg3 register to the thread id, while CUDA does not. We assume this value to be always 0, as functionality does not seem to differ.

In the OpenCL implementations that use atomics, variables are marked *volatile* as required by the OpenCL specification. In the CUDA implementations, this is not needed. The volatile keyword tells the compiler that the value of the variable can change at any time, and therefore compilers generally do not perform any significant optimizations on the variable. Here the PTX codes also are equal[3]. This also shows that the use of volatile has no negative effect on the OpenCL implementation. The PTX code differences can be seen in Table 5.4.

CUDA	**OpenCL**
.target sm_30	.target sm_50, texmode_independent
.param .u64 BFS_1_param_0	.param .u64 .ptr .global .align 4 BFS_1_param_0
...	...
cvta.to.global.u64 %rd15, %rd10;	-
mov.u32 %r9, %ntid.x;	mov.b32 %r9, %envreg3;
mov.u32 %r10, %ctaid.x;	mov.u32 %r10, %ctaid.x;
mov.u32 %r11, %tid.x;	mov.u32 %r11, %ntid.x;
mad.lo.s32 %r1, %r9, %r10, %r11;	mad.lo.s32 %r12, %r10, %r11, %r9;
-	mov.u32 %r13, %tid.x;
-	add.s32 %r1, %r12, %r13;

Table 5.4: The differences between the PTX code of CUDA and OpenCL. Minor parameter name changes were made for clarity.

In summary, our analysis of OpenCL and CUDA reveals insignificant differences between our OpenCL and CUDA implementations. From this, we can conclude that, for our application, the use of OpenCL does not cause any overhead, and therefore OpenCL can be a promising programming model for GPU graph processing.

5.3 Algorithmic Modifications

To understand how the different algorithmic modifications influence the GPU and CPUs relative performance, we measured the average kernel execution time of each implementation on the entire dataset.

We analyze the data in three ways. First, we find "quirks" in our implementations that may be *approach*-induced or *platform*-induced, thus providing insight into how our implementations behave compared to what we expect them to do, and giving an indication of how performance portable the constructs that we used in the implementations are. Second, we determine the application efficiency of each implementation for each graph. This allows us to directly compare the different implementations, and should also enable a performance comparison of the GPU and CPU. Third, we represent the application efficiency of the GPU and CPU as Performance Portability, which gives us a hard metric on whether an implementation is performance portable. Armed with the orientation in these following sections, the performance portability should allow us to conclude which types of OpenCL constructs are performance portable, and which ones may not be.

5.3.1 Implementation Quirks

We expect our OpenCL implementations to behave in a certain way, from past experience (see our related work analysis, in Chapter 6) and theory. Specifically, we have three hypotheses. (1)

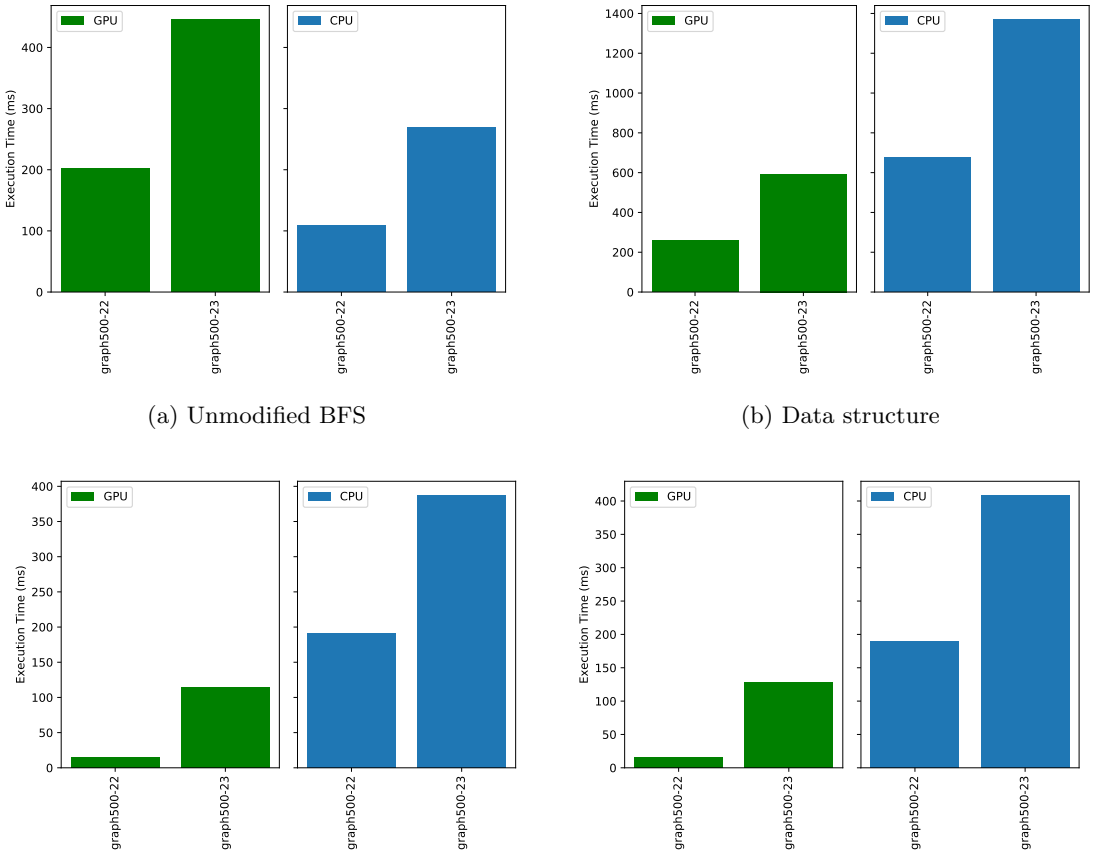
[3]The only difference on the PTX lines involving the volatile variable is that it is marked as volatile. There is no sign of different ordering of instructions, or different instructions being used altogether in the CUDA implementations.

Graph500-23 will require double the execution time compared to the graph500-22 graph because the number of vertices and edges are doubled, while the topological structure is the same. In other words, it is double the work, while not differing in execution patterns or algorithmic requirements. Moreover, this should hold for all algorithmic implementations. (2) RoadNet-CA has a high diameter and a low average degree. Therefore, the number of edges in the frontier is never very high. RoadNet-CA is therefore not likely to benefit from direction-optimizing, which may even hurt its performance. (3) The CPU will benefit more from the queue data structure than the GPU. This should happen because the GPU can process more items in parallel than the CPU, and thus having unnecessary items scheduled has less impact on the GPU than the CPU. The queue also uses atomics, and high contention between atomics will also reduce performance. The GPU is more likely to have contention than the CPU due to the larger number of threads running in parallel. Therefore, we expect the impact of the data structure modification to be different on the two target platforms.

We note that this section only includes a *representative subset* of our results, sufficient to prove our points and illustrate our main findings. The omitted results are similar in nature, and are therefore omitted for brevity. They can be found at `https://github.com/Rickyboy320/BFS_OpenCL`.

(1) Graph500 scales 22 and 23

We first analyse the execution times of all implementations for graph500-22 and graph500-23. We note the hypothesis once again: *All implementations require double the execution time when executing graph500-23 compared to the graph500-22.*



(a) Unmodified BFS

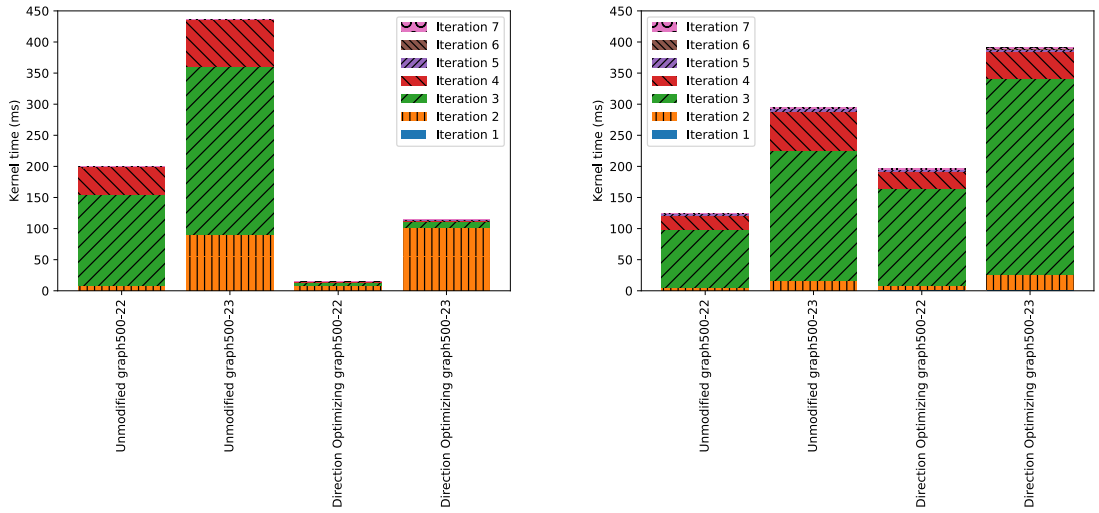
(b) Data structure

(c) Direction-optimizing

(d) Direction-optimizing + Data structure

Figure 5.3: Execution times of all implementations on the graph500-22 and graph500-23 graphs. Note that the y-axis has different scaling. In green we represent the GPU, in blue we represent the CPU.

Figures 5.3a and 5.3b show that, for the unmodified BFS and the data structure implementation, graph500-23 indeed takes double the execution time as graph500-22, on both the GPU and CPU. However, in Figure 5.3c, and 5.3d we see that when direction-optimizing BFS is introduced, graph500-23 takes significantly longer than double the execution time on the GPU. The CPU is unaffected in this difference between unmodified and direction-optimizing, which displays an interesting disparity of GPU and CPU. As the diameters of the two graphs are equal, this behaviour is not caused by the BFS traversing more levels for graph500-23 than graph500-22.



(a) GPU

(b) CPU

Figure 5.4: A breakdown of the execution times of the iterations for unmodified and direction-optimizing BFS on the GPU and CPU.

To identify why graph500-23 takes more than twice the execution time as graph500-22 on the direction-optimizing implementations, we time each iteration of the kernel to find a bottleneck. The results of this analysis are shown in Figure 5.4. This breakdown shows that for the GPU, the second top-down iteration of the direction-optimizing implementation is 10 times slower on graph500-23 than on graph500-22. The remaining iterations take about double the execution time or less, as expected, resulting in the about 5x gap in the overall execution time.

The same 10x spike happens for unmodified BFS, but the third iteration has a much larger execution time. The third iteration overshadows the difference in the second iteration, which hides the same problem on the unmodified implementation. A similar issue is happening on the CPU: the execution times are overshadowed by the large execution time of the update kernel on the third iteration.

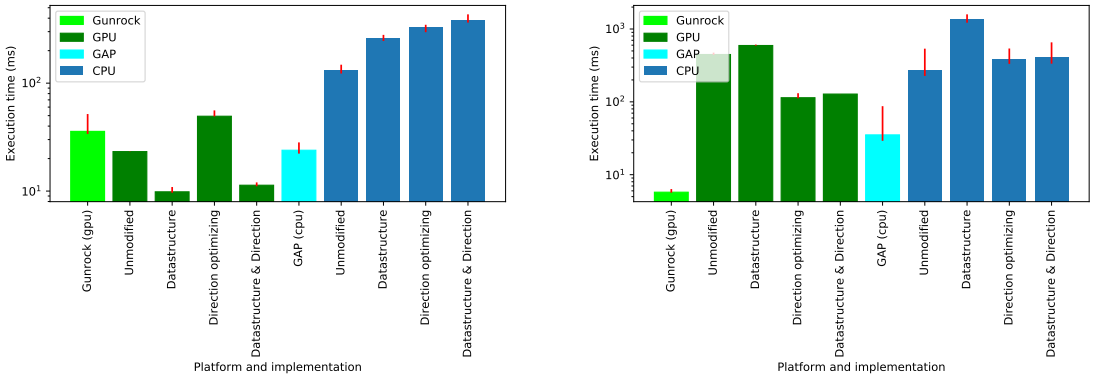
This 10x spike is an artefact of using a top-down iteration for the graph500-23 graph. The second iteration of graph500-23 would have benefited from a bottom-up iteration, but our parameters for the direction-optimizing algorithm forced a top-down iteration. There is a large number of edges to be explored in this iteration, for which the top-down implementation is ill-suited.

In short, the execution time of direction-optimizing implementation on the GPU is superlinear on graph500-23, due to the second top-down iteration not being a bottom-up iteration. We classify this behaviour as an *approach*-induced quirk. We note that with a better fit of parameters - i.e., setting the parameters of this optimization to be *graph-specific* - this quirk would disappear, resulting in even better execution times. However, estimating the best direction-optimizing parameters is out of scope for this research.

In Figure 5.4b, the iteration breakdown for the CPU, we see the differences between the second iteration times less clearly. The differences between the second top-down iterations are only up to 5x, and that behaviour remains masked under the large third iteration, even with direction-optimizing. This is peculiar, as the third iteration of direction-optimizing is almost non-existent on the GPU. We discuss the cause of this behaviour at the end of this section, in *(3) CPU performance.*

(2) Performance analysis for RoadNet-CA

We further proceed to investigate the impact of our direction-optimizing BFS on performance for a high-diameter, low-degree graph. Our expectation is that these graph properties will render this optimization inefficient.



(a) RoadNet-CA

(b) Graph500-23

Figure 5.5: Execution time of the different implementations on two graphs. In green we represent the GPU and in blue we represent the CPU. In lime we represent Gunrock and in aqua we represent GAP. Note that the y-axis are logarithmic and in differing scale for both figures.

For the GPU on the RoadNet-CA graph, Figure 5.5a confirms our second hypothesis that a high-diameter, low-degree network is hurt by our direction-optimizing implementation. Note that Gunrock is also slower than three out of four implementations on this graph. Gunrock runs a direction-optimizing variant, which appears to be ill-suited for this type of graph. The data structure modification seems to bring a significant performance gain on this graph, which even carries over onto the direction-optimizing + data structure implementation.

These observations are all direct consequences of the algorithms. The direction-optimizing version has to do extra work to track the heuristics, while it never switches to bottom-up BFS. Thus, it never gains anything from the modification. The direction-optimizing + data structure implementation has the same performance gains as the data structure implementation but loses a bit of performance due to the computation of the heuristics. The data structure modification is beneficial when faced with a high diameter graph.

Interestingly, the CPU loses performance when faced with the data structure modification, where the GPU increases significantly in performance. We will again see why at the end of this section. Also note that GAP, which implements the direction-optimizing approach as well, still significantly outperforms the unmodified and data structure CPU implementations.

In contrast, Figure 5.5b shows significant performance gains, for graph500-23, when running on the GPU with the direction-optimizing modification. The CPU loses performance *again* when using direction-optimizing.

All these results indicate that different algorithmic approaches can have different performance effects on the GPU and CPU, which can be further greatly influenced by the type of input graphs the traversal runs on.

(3) CPU performance

The performance results for both the RoadNet-CA and graph500-23 graphs show that the CPU is negatively impacted by *all* algorithmic modifications. This implies that our third hypothesis, that the queue data structure is more beneficial for the CPU than for the GPU, is wrong. To uncover why, we further investigate the CPU in more detail.

A major difference between the different BFS versions is their use of atomics. The data structure implementation uses atomics to ensure values are not entered twice into the queue,

while the direction-optimizing implementations use atomics to track the heuristics to dynamically switch between top-down and bottom-up.

In order to see whether the atomics have any significant impact on the GPU or CPU, we design the following experiment. We modify the direction-optimizing version such that it switches on a specific iteration. That way, atomics do not need to be used to keep track of the heuristics, and the performance difference between the two versions (i.e., the atomic-using direction-optimizing implementation and the atomic-free implementation) will illustrate the penalty of atomics for this version of the algorithm.

We built four versions to compare the impact of the atomics and heuristics on the overall performance. The four versions are: v1: full atomics and heuristics, v2: no atomics, v3: no edge heuristics, and v4: no edge heuristics and no atomics.

v1 is equivalent to the direction-optimizing implementation. v2 still performs the same computations, but does not use atomics, and therefore the heuristics are wrong. This is the best comparison between using atomics and not using atomics, as the only difference between this version and v1 is the removal of atomics. v3 removes edge heuristics, but vertex heuristics are still computed atomically. v4 removes the computation of all heuristics altogether.

Version	Kernel time GPU (ms)	Kernel time CPU (ms)
Unmodified BFS	202.415	107.255
v1) Direction-optimizing (full)	15.843	192.904
v2) No-atomics	14.871	167.835
v3) No-edge heuristics	13.659	108.374
v4) No heuristics and atomics	13.634	59.644

Table 5.5: Kernel time comparison for different combinations of atomics and heuristics on the GPU and CPU.

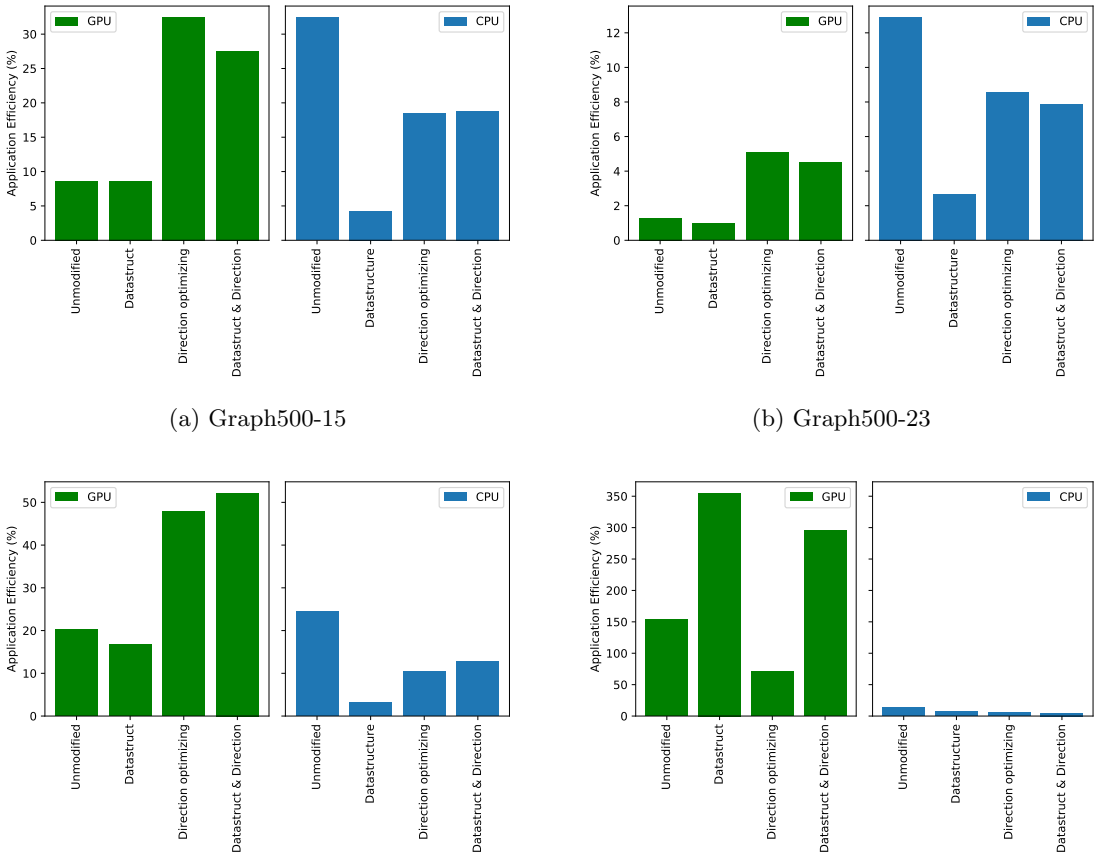
Table 5.5 shows that atomics have a significant impact on the CPU, but the heuristics are even worse. By disabling the atomics (v1 vs v2), the execution time decreases with 25ms, but it is still slower than the unmodified version. Removing the edge heuristics has a more significant impact than removing the atomics: the execution time for v3 decreases by about 85ms over v1. Removing all heuristics reduces the execution time even more, to 59ms, which is significantly faster than the unmodified version. This means that if it was not for the atomics and heuristics bottleneck, the CPU could have benefited significantly from the direction-optimizing modification as well.

On the GPU, the impact of heuristics and atomics are less significant. Removing atomics reduces the execution time by 1ms only. The use of atomics is slightly less impactful than the computing of the edge heuristics, which takes just over 2ms. Removing the vertex heuristics has no significant effect. More significant is the use of direction-optimizing over unmodified BFS. It seems that the heuristics and atomics are not a major bottleneck on the GPU.

In conclusion, the CPU's performance is severely limited by the heuristics computation required by the direction-optimizing modification; the impact of these heuristics is significantly worse than on the GPU; also, the OpenCL atomics seem to more significantly impact the performance on the CPU than the performance on the GPU. We classify the CPU-GPU performance difference because of heuristics and atomics as a *platform*-induced quirk.

5.3.2 Application Efficiency

For each graph in Figure 5.6, we can draw similar conclusions. Specifically, application efficiency increases and decreases wildly per graph. Sometimes the direction-optimizing implementation reaches a higher application efficiency than the direction-optimizing + data structure implementation, and sometimes the reverse is true. One consistent pattern does occur: on the GPU, the data structure modification reduces application efficiency, while the direction-optimizing and direction-optimizing + data structure implementations increase application efficiency. On the CPU, we see that all modifications reduce application efficiency, but the data structure modification more so than the direction-optimizing modification. The only outlier in the data is the



(a) Graph500-15

(b) Graph500-23

(c) Actor-collaborations

(d) RoadNet-CA

Figure 5.6: Application efficiency of the different implementations on part of the dataset. In green (left side of the figures), we represent the application efficiency of the GPU implementations, with Gunrock as a reference algorithm. In blue (right side of the figures) we represent the application efficiency of the CPU, with GAP as a reference algorithm. For visibility reasons, the y-axes differ in scale for each figure.

RoadNet-CA graph, which we have discussed extensively in section 5.3.1. Finally, overall, we find that direction-optimizing is on most graphs a favourable optimization for the GPU, but causes excessive overhead on the CPU due to its heuristics and atomics.

5.3.3 Performance Portability

We have identified bottlenecks for the various graph types, algorithms and platforms. We have already seen that the different algorithms have a significant impact on the relative performance of the GPU and CPU. However, to identify whether this has an impact on the performance portability metric, we convert the measured application efficiency into performance portability.

Figure 5.7 shows the performance portability for all implementations on all graphs. Comparing the data structure implementation to all the others, we can see that the data structure implementation is never the most performance portable, even on the roadNet-CA graph, where we have shown that the GPU significantly benefits from this implementation.

Comparing the unmodified and the direction-optimizing implementations, we can see that the performance portability for the synthetic graphs is higher on the direction-optimizing implementations, while the real-world graphs have higher performance portability on the unmodified implementation. The performance portability of the direction-optimizing and direction-optimizing + data structure implementations are relatively balanced.

Most graphs on the GPU greatly benefit from the direction-optimizing modification in terms

of execution time, and this is also reflected as a higher performance portability. However, the negative effect of the direction-optimizing modification on the CPU does not seem to be reflected significantly enough. While the performance portability metric can give a sense of what is the best implementation on *average*, it is not directly clear what impact the implementation will have on the platforms, for which we need the *application efficiency*.

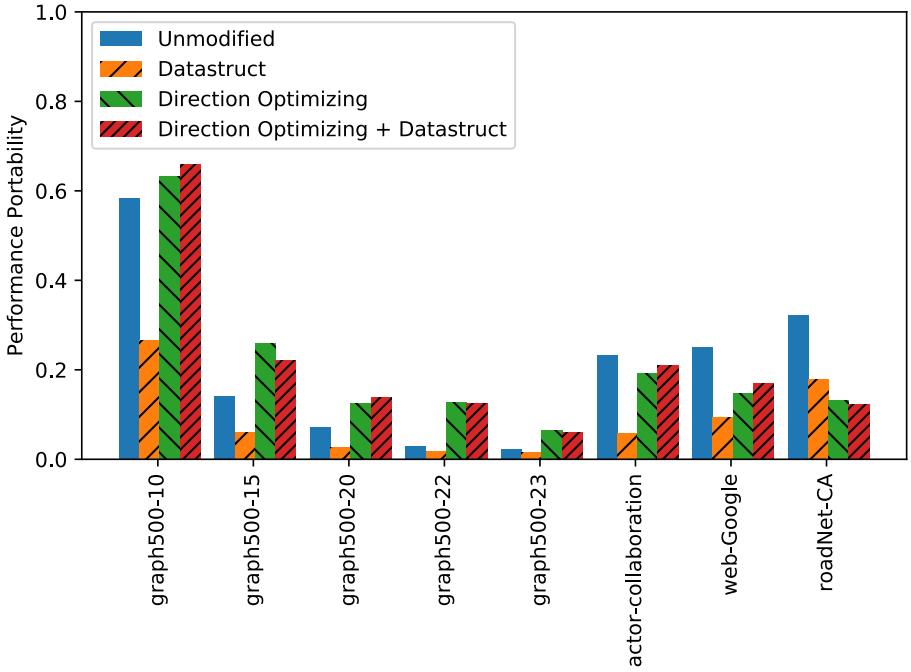


Figure 5.7: The measured performance portability of all implementations on all graphs.

We have seen that different implementations have different effects on the GPU and CPU. A consistent quirk with all algorithmic modifications was that the CPU suffers from atomics and heuristics. We have therefore shown that a program in OpenCL is not necessarily *write once, run anywhere efficiently*. Care has to be taken that the constructs used in the implementations are performance portable, which the (global memory) heuristics and atomics do not seem to be.

5.4 Specializations: GPU vs CPU

Lastly, we want to see if there are certain settings in OpenCL that can favor one platform over another. We have chosen three specializations: 1) *Zero-copy*, 2) *optimal work-group sizes*, and 3) *optimal workload per kernel*. We have defined three categories of specializations: *fair* specializations, *anti-social* specializations and *illegal* specializations. The *fair* specializations are those that favor a target platform, but do not significantly affect any others, while the *anti-social* specializations are those that favor one platform, but hurt other platforms. Finally, the *illegal* specializations are those that break portability altogether. We will categorize each of our specializations into these three categories.

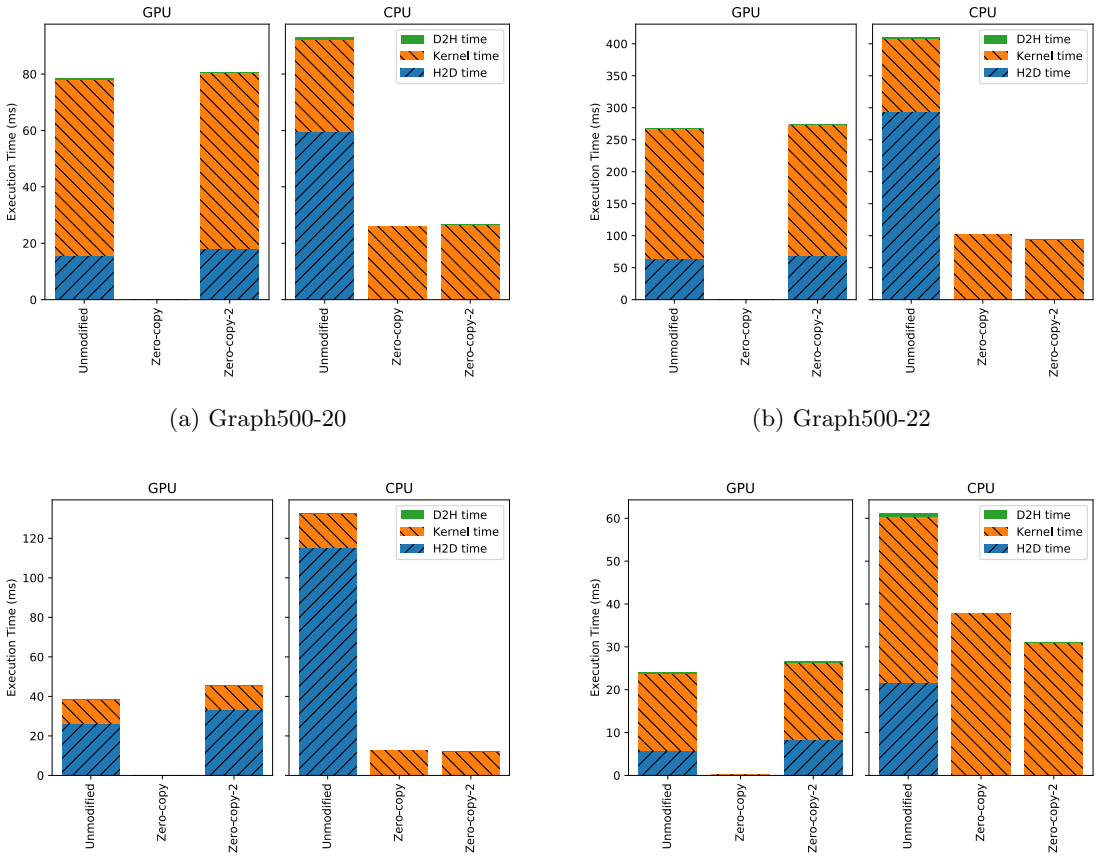
5.4.1 Zero-copy

Zero-copy refers to the idea of removing unnecessary data copying when the data is already available (and thus, setting a simple pointer to the right region is sufficient). When using OpenCL on the CPU, the data is already present in the host memory, so there is no need to copy it to another location in the same memory. Therefore, we want to circumvent this step on the CPU, and see whether any performance boost is possible. We hypothesize the zero-copy specialization to be a *fair* specialization: as long as the GPU copies over the data, still, there

should be no difference in the performance on that platform, and only the CPU will be positively affected.

As described by Jie Shen et al. [22], there are two ways of implementing zero-copy. One uses the CL_MEM_ALLOC_HOST_PTR flag to create an uninitialized zero-copy memory region. In this region both the host and the device can write and read, given that the region is mapped and unmapped, respectively. Since we already have the input data available, we instead opted to go for the second type of zero-copy: using the CL_MEM_USE_HOST_PTR flag. When creating a buffer with this flag, the memory can be directly used by the kernel, referencing the same memory region as the pointer we already had. However, depending on the device, this memory can still be *cached* on the device, and therefore, for some device types, it will still result in a copy operation. This will hold true for at least all GPUs, as they do not have direct access to the host memory.

We designed two experiments. First, we wanted to determine what would happen if all transferring of data was removed, since the buffer declared with CL_MEM_USE_HOST_PTR references the same memory regions. Since this seemed to break the algorithm on the GPU, we also designed a secondary experiment where the *runtime* copies were intact, but with CL_MEM_USE_HOST_PTR allocated buffers. The results for four graphs in our dataset can be seen in Figure 5.8.



(a) Graph500-20

(b) Graph500-22

(c) Actor-collaboration

(d) Web-Google

Figure 5.8: The effect of zero-copy on four different graphs. Zero-copy is the version where all transfers were removed (which breaks the GPU). Zero-copy-2 is the version where all *runtime* copies are intact, but it uses the CL_MEM_USE_HOST_PTR flag for the buffer.

We can see that by applying the first technique, where all data transfers were removed, the GPU fails to function. In the documentation of OpenCL, it is stated that implementations can choose to cache memory on the device that is defined with the CL_MEM_USE_HOST_PTR flag. For the GPU, this is the case, thus effectively resulting in two memory regions; the host memory and the device memory. When we are not copying the data back from the GPU to the

CPU, the algorithm terminates after one step, with an incorrect result. On the CPU, the effects are impressive: the entire memory transfer times are eliminated, resulting in great performance improvements on all graphs.

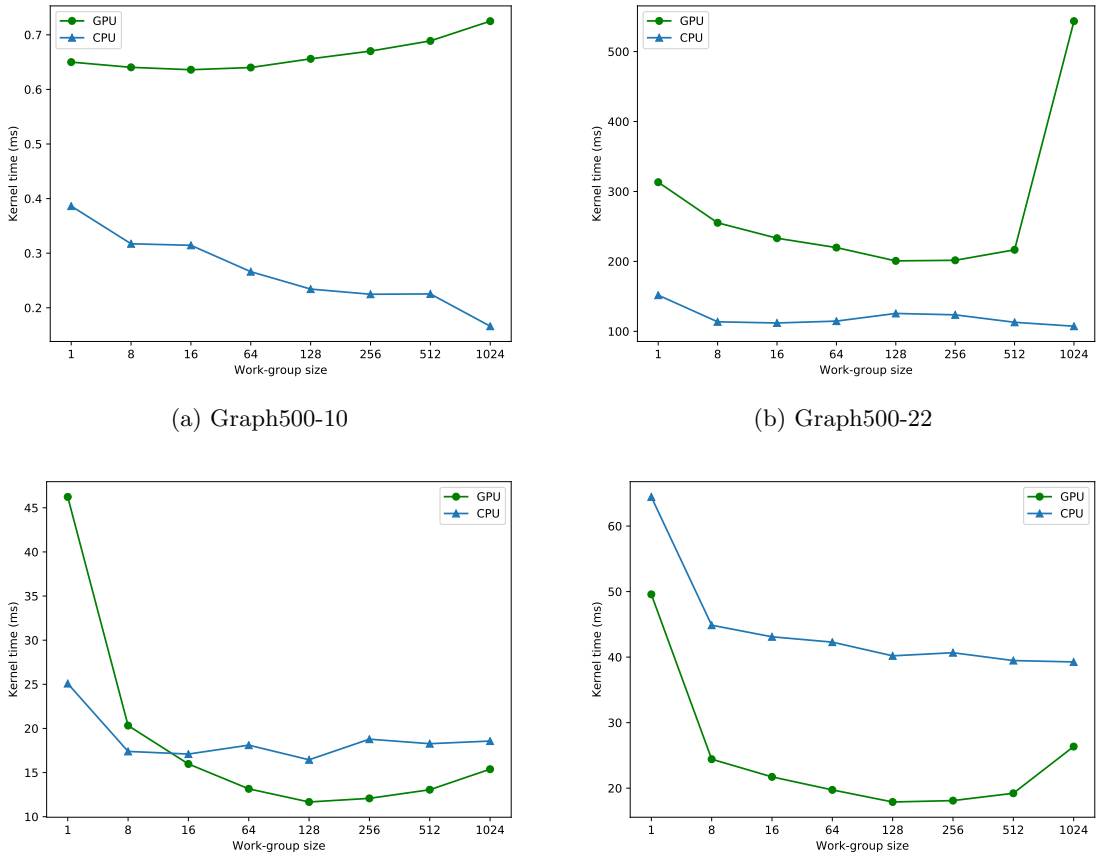
This first technique is an *illegal* specialization: it breaks the program on the GPU, but improves performance on the CPU. To recover portability, we applied the second technique (zero-copy-2), where runtime transfers remain intact. This recovers portability, while keeping the same benefits on the CPU.

We note some variability in execution time, introduced by noise. For example, in Figure 5.8d, the kernel time seems different for zero-copy and zero-copy-2, while no changes to the kernel were made. Similarly, on the GPU, there seems to be a slight increase in H2D time. Since the buffers were allocated differently, the measuring of H2D times also had to change slightly. This change in measurement could have led to these minor differences in H2D times. Regardless, the differences on the GPU are so minor that we can call zero-copy-2 a *fair* specialization.

In conclusion, zero-copy seems to have a significant benefit on the CPU, while it has, if implemented correctly, no negative effect on the GPU. Therefore zero-copy is a *fair* specialization. What is interesting about the first technique is that it shows that it indeed is possible to program OpenCL in a way that breaks portability, and therefore, performance portability.

5.4.2 Optimal Work-group Sizes

Due to the different architectures of GPUs and CPUs, we hypothesize that the optimal work-group sizes differ. To test this, we designed an experiment where we run the unmodified version on different work-group sizes and measure its kernel time. We represent the results in Figure 5.9.



(a) Graph500-10

(b) Graph500-22

(c) Actor-collaboration

(d) Web-Google

Figure 5.9: The performance impact of varying the work-group sizes, presented for 4 different graphs in our dataset.

The four graphs in Figure 5.9 display very different patterns, indicating we have a different optimal work-group size per graph. For example, for Graph500-10 (Figure 5.9a), the best-performing work-group size of the CPU is 1024, while the optimal size for the GPU is 16. For Graph500-22 (Figure 5.9b), the best configurations for the CPU are for 8, 16 or 1024 work-items per group, while for the GPU, 128 or 256 work-items per work-group should be used. For actor-collaborations (Figure 5.9c), the best configuration, for both the GPU and CPU, is to use a work-group size of 128. Lastly, for web-Google (Figure 5.9d), the optimum again lies at 1024 for the CPU, and at 128 for the GPU.

In short, there is no single work-group size that is most efficient for all graphs, nor is there one work-group size that is always optimal for both the CPU and GPU.

For the smaller graphs (graph500-10 and web-Google), we notice that the CPU benefits from a larger work-group size. We hypothesize that this is because of caching or load-balancing. We designed two additional experiments to pinpoint which one it is. To exaggerate the kernel execution over starting the program, loading the graph in memory, and displaying the result, we increased the number of executions to 50000.

We measured the caching behaviour of the 1 and 1024 work-group sizes. We represent the results in Table 5.6.

	Cache-misses (size 1)	Cache-misses (size 1024)
L1	3.78%	3.854%
LLC	20.27%	18.28%

Table 5.6: The rate of cache-misses on the L1-cache and the LLC (last-level cache) for the two work-group sizes on graph500-10.

The cache-miss difference of size 1 and size 1024 are well within the range of noise, so it seems that caching is not the culprit. We therefore also measured the execution time on each CPU core. The results of this experiment are shown in Figure 5.10.

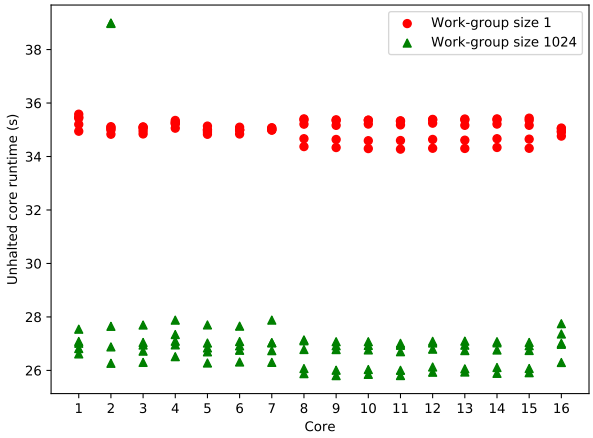


Figure 5.10: Unhalted core runtime for all cores on the CPU running 50000 runs of BFS. Represented as 5 measurements for each core.

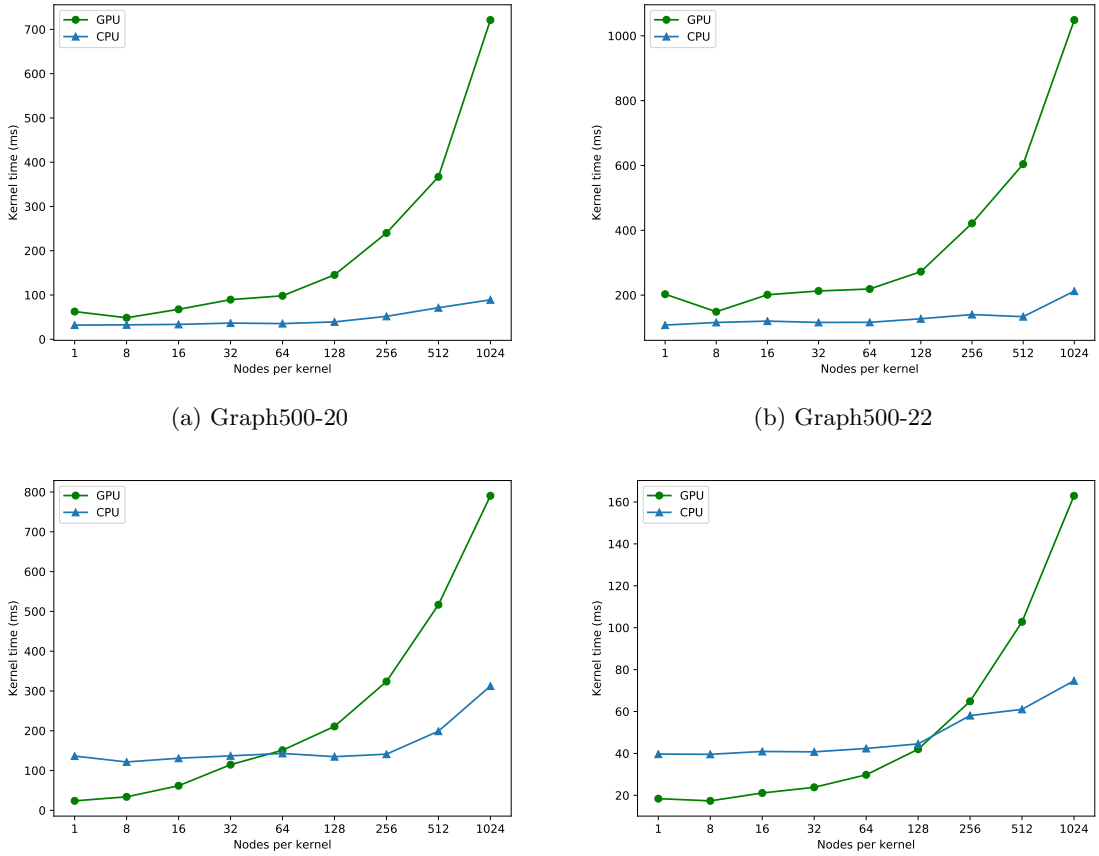
The amount of time that each core is running is relatively consistent. There is one outlier for core 2 on the work-group size 1024, but in the general case there does not seem to be any reason to believe there is any significant load-imbalance going on.

Neither caching nor load-balancing causes the difference in performance. Therefore, it seems the combination of mapping OpenCL work-items and work-groups to hardware, as well as potential scheduling approaches, are causing the CPU to prefer larger work-group sizes. More analysis is needed to pinpoint exactly what the problem is. This analysis is, due to timing constraints, left as future work.

5.4.3 Optimal workload per kernel

Scheduling a higher workload (i.e., multiple nodes) per kernel allows for a smaller number of kernel calls, and therefore work-groups. If faced with any scheduling overhead, this increased workload would reduce this overhead, at the cost of serializing a number of items. We hypothesize that the GPU and CPU benefit from different optimal items that are serialized. In particular, we hypothesize that the CPU benefits from a larger workload per work-item than the GPU.

To test this hypothesis, we run an experiment on the unmodified implementation, where each kernel executes a number of items in series. The results for four graphs are shown in Figure 5.11.



(a) Graph500-20

(b) Graph500-22

(c) RoadNet-CA

(d) Web-Google

Figure 5.11: Effect of increasing the workload per kernel on four graphs in our dataset.

Overall, in Figure 5.11, we can see that the GPU indeed suffers from a larger workload (i.e., number of graph-nodes to be processed) per kernel. In Figures 5.11a, 5.11b, and 5.11d we see that the GPU slightly benefits from 8 nodes per kernel. The CPU remains relatively stable until the number of nodes per kernel exceeds 128, where the performance starts declining. Overall, it seems that the number of nodes per kernel is better left as small as possible, with the actual graph also having a (minor) impact on this decision.

The CPU is only slightly affected by this specialization, and negatively if the items per kernel exceed 128. The GPU is negatively affected if the number of items per kernel exceeds 8. While the optima seem to lie at different points, the difference is small, and the optimum generally lies at either 1 item per kernel or 8 items per kernel. Even if the impact is very small, the optimum differs for the CPU and GPU, which makes this an *anti-social* specialization. To gain a very minor performance boost it may be useful to tune for this parameter, but for the graphs that we have investigated, it seems that a lower number of item per kernel is more efficient.

5.4.4 Summary

In summary, we note that our three hypotheses were correct. Zero-copy is indeed a fair specialization when implemented correctly. If implemented incorrectly, it can break portability with the GPU and other devices that still perform copying. Second, it is worth tuning for the optimal work-group size. The optima of the work-group size tend to differ per graph, and for the CPU and the GPU, so it is an *anti-social* specialization. Third, it may be worth tuning for an optimal number of items per kernel, but in our experiments only 1 or 8 items seemed worthwhile. Once the number of items per kernel exceeds 128, the CPU starts significantly declining. The GPU already starts declining in performance once the number of items exceeds 8. The optima for both seem to lie at different points, but the performance gain is minor. Regardless, it is therefore classified as an *anti-social* specialization.

CHAPTER 6

Related Work

In this chapter we survey relevant research that has influenced our current approach. Specifically, we discuss previous performance portability studies, where different types of applications, programming models, and success criteria are used.

In 2011, Jianbin Fang et al. [1] showed that their Breadth-First Search in OpenCL performs worse than the same algorithm implemented in CUDA: the OpenCL version only reaches about 80% of the performance of their CUDA implementation. These BFS implementations are from the Rodinia benchmark, the same code we based our implementations on. They noted that the difference in performance comes from the overhead of calling kernel functions. Since BFS has to call kernels multiple times, the overhead is more exaggerated. Our results, however, showed that there is no significant difference in performance on OpenCL and CUDA for BFS. Even the roadNet-CA graph, which starts a kernel 555 times, showed no significant difference. We have also shown that the OpenCL and CUDA compilers generate the exact same PTX code for our BFS kernels, except for some minor differences. This equivalence resulted in a comparable performance between the two programming models.

Puya Memarzia and Farshad Khunjush [23] showed that for image processing kernels, CUDA performs better than OpenCL by 5% on the GTX 570 and by 7% on the GTX 670. This is similar to what we have shown, where our kernels show no performance difference larger than 5%, neither in favour of CUDA nor OpenCL. They also showed there was a significant performance loss by using local memory over constant memory in CUDA. We have experienced similar behaviour: due to an oversight, our CUDA implementation did not use constant memory, while our OpenCL implementation did; this "accident" caused a significant performance gain for the OpenCL implementations compared to the CUDA ones (we have rectified this imbalance for the reported results). This anecdote shows that even minor changes in code can have a significant impact on the (performance) comparison of two programming models, and that a fair comparison must ensure equivalent code at the level of the entire implementation, not only the kernel.

The works of Jianbin Fang et al. [1], and Kazuhiko Komatsu et al. [24] state that the OpenCL compilers were too immature to appropriately optimize kernels using techniques like loop unrolling. Our Breadth-First Search kernels do not seem to benefit from loop-unrolling, or any other optimizations that OpenCL cannot represent. Based on our analysis, BFS is a *simple enough* kernel, where OpenCL does not lose performance. We cannot confirm or infirm any evolution/change for the OpenCL compilers with respect to their ability to perform kernel optimizations - our case-study is too simple to expose such behaviour.

Rob V. van Nieuwpoort and John W. Romein [25] looked at employing Many-Core hardware for the purpose of correlating radio astronomy signals. They also implemented several versions using OpenCL. They note that indeed, OpenCL is functionally portable, but not entirely performance portable. To gain optimal performance, they had to *specialize the code* by changing kernel/algorithm parameters, as well as using different amounts of threads for the CPU and GPU.

Peng Du et al. [26] found that OpenCL is also not fully performance portable for different GPU architectures. Instead, they showed that an auto-tuner that chooses the most optimal al-

gorithm and parameters can deliver acceptable performance. To enable auto-tuning, the authors implemented a parameterized, specialized version of the code. Using this version, the auto-tuner can tweak many parameters, including the optimal work-group size (threads per work-group) and an optimal workload per kernel (blocks each thread computes). We also found optimal work-group sizes to significantly affect performance, and (manually) tuned it for our case; indeed, an auto-tuner is a more productive solution.

Joo Hwan Lee et al. [27] performed an extensive OpenCL performance study on Multi-Core CPUs. They specifically investigate API overhead, thread scheduling, memory allocation and transfer, and vectorization. Their study on API overhead found a high overhead due to context management and JIT compilation. Furthermore, their study on thread scheduling found a performance improvement on CPUs when increasing the workload per kernel, while increasing the work-group size reduced performance. We found that, in the context of graph processing, this is precisely reversed: a smaller workload per kernel generally seemed the optimal configuration, while a larger work-group could lead to performance gains. For the memory transfer, they found that mapping and unmapping memory is more efficient on the CPU than doing an explicit memory transfer. We have shown that on the CPU it is possible to not perform any memory transfer at all (zero-copy) without negatively impacting the GPU. Finally, they discuss vectorization in OpenCL, and show that vectorization in OpenCL and OpenMP reach comparable performance. We have not specifically looked at vectorization, but it is an interesting direction for BFS on the CPU: as shown by Gao Tao et al. [28], there are approaches for (direction-optimizing) BFS which make it vectorizable. They show that by utilizing vectorization, they can reach significant increases in speed over straightforward direction-optimizing.

In summary, our research was inspired by previous work in the context of OpenCL performance portability, but is unique in that it is applied on a data-intensive, data-dependent graph processing workload, for which it provides a quantitative analysis of performance portability using a recognized metric [9].

CHAPTER 7

Conclusions

Graphs are getting increasingly large and complex, requiring a shift to parallel architectures to continue processing them efficiently. The complex interaction between input graph properties, parallel architectures features, and parallel graph processing algorithms makes the performance of many graph processing workloads difficult to predict. Thus, selecting the right algorithm and/or parallel architecture for deploying graph processing can be a difficult challenge. As portable programming models become more mature, their ability to implement platform-agnostic parallel programs is of interest for graph processing, in the hope the same program can be deployed on different architecture and the best performing one can be empirically selected. However, this scenario is only possible if these programming models offer performance portability across platforms.

In this context, the goal of our research was to determine whether the use of portable programming models for graph processing provides a portability vs. performance trade-off. To this end, we devised an empirical method to assess whether such a trade-off exists. We investigated BFS as a case-study, and focused on various aspects of OpenCL, such as its capability to perform compared to CUDA, the effect of algorithmic modifications on the performance of the CPU and GPU, and the effect of specializations on the performance and portability.

7.1 Main Findings

Our research was guided by three research questions. Our findings are as follows.

RQ1: How does the performance in OpenCL compare to native implementations? We have shown that there is no significant difference between OpenCL and CUDA. The resulting PTX codes of the kernels were equivalent, except for a few minor differences in the declaration of pointers and thread indices, which resulted in equivalent performance. Therefore, for the simple case of Breadth-First Search, OpenCL does not lose any performance, and there seems to be no inherent reason to use CUDA over OpenCL. It might even be beneficial to use OpenCL for BFS as it can support multiple platforms.

RQ2: How do various algorithmic adaptions change relative performance of the CPU and GPU? For our second research question, we have identified a clear difference between the relative performance of algorithms on the CPU and the GPU. In particular, our implementation of the direction-optimizing approach presented by Beamer et al. [14] has significant performance gains on the NVIDIA Titan X GPU, while it loses performance on the Intel Xeon CPU when compared to the unmodified implementation (based on the Rodinia benchmark). The relative performance of CPUs and GPUs are therefore still greatly affected by the chosen implementation. We have greatly improved the performance on the CPU by eliminating the (atomically calculated) heuristics for the direction-optimizing implementation. We showed that using atomics has a more

significant negative impact on the performance of the CPU than the GPU. Even more so, entirely removing the heuristics had a greater impact than only removing the atomics, which shows a great performance bottleneck by the writing to global memory. In other words, there are ways to express an algorithm, such as with the use of atomics or simply inefficient implementations, that can favour one platform over another. Favouring a platform in terms of performance by causing harm to another is by our definition not performance portable, which means that the programmer has to keep in mind the platforms the program is executing on when implementing algorithms in OpenCL.

RQ3: Can we gain performance by specializing for a given platform within OpenCL? We defined the *fair* specializations to be those that do not negatively affect any other platforms. We defined *anti-social* specializations as those that do negatively affect other platforms. Finally, we defined *illegal* specializations as the specializations that break portability altogether. We investigated three specializations: zero-copy, optimal work-group size, and optimal workload per kernel. We found that zero-copy is a *fair* specialization if implemented properly. If implemented improperly, it can cause GPU portability to break (resulting in an *illegal* specialization). Secondly, we found that the optimal work-group size differs for graphs and platform. Significant performance can be gained by tuning for the right work-group size, but the optima for the CPU and GPU differ. Therefore, tuning for the optimal work-group size is an *anti-social* specialization. Lastly, we looked at the optimal workload per kernel, where multiple vertices are handled by one kernel. We found that in general, increasing the amount of workload per kernel reduced the performance significantly. Here again, the optima for the CPU and GPU differ slightly, making this specialization an *anti-social* specialization.

Overall, we conclude that OpenCL is a portable programming model that can perform similarly to CUDA for simple applications like Breadth-First Search. Even though the programming model seems to perform well, care has to be taken when writing algorithms in a platform agnostic manner. A programmer cannot expect their code to be performance portable just by writing it in OpenCL, as the approach and implementation of the algorithm can have a significant influence on performance. Therefore, to achieve optimal performance, the underlying architectures have to be kept in mind. Finally, the programmer may also need to make fair specializations to boost their performance even more for one or more platforms. They can also opt for anti-social specializations, but only with care if they value performance portability.

7.2 Future Work

This research is extensible in many dimensions. We have used Breadth-First Search as a case study, but many more different algorithms, graph processing or not, can be studied in the same way. For example, within the field of graph processing, this study can be expanded to Pagerank, SSSP and Graph Colouring. Similarly, there are different approaches for Breadth-First Search that may display other interesting properties of performance portability in OpenCL. For example, one study came up with vectorizable kernels for BFS [28], which can possibly boost performance by applying finer-grain parallelism.

We have only investigated a small number of specializations in this work. There are likely many more techniques to target a specific platform's performance. A detailed investigation of different programming techniques and parameters in OpenCL and their respective performance on multiple platforms may be an interesting direction of research. This taxonomy of platform-targeted specializations would help programmers and researchers identify how to get the most out of their OpenCL applications or how to maximize portability. In other words, an extensive classification of specializations will help in choosing between performance and portability.

Last, but not least, it is also interesting to identify how the implementations perform on more platforms. We have only shown two: an NVIDIA Titan X GPU and an Intel Xeon CPU, but there are many more GPU and CPU vendors, as well as other types of architectures, such as FPGAs, which may pose an even more interesting portability trade-off.

Bibliography

[1] J. Fang, A. L. Varbanescu, and H. J. Sips, "A comprehensive performance comparison of cuda and opencl", *2011 International Conference on Parallel Processing*, pp. 216–225, 2011.

[2] J. Shen, J. Fang, H. Sips, and A. L. Varbanescu, "An application-centric evaluation of opencl on multi-core cpus", *Parallel Computing*, vol. 39, no. 12, pp. 834–850, 2013.

[3] J. Garcia-Bernardo, J. Fichtner, F. W. Takes, and E. M. Heemskerk, "Uncovering offshore financial centers: Conduits and sinks in the global corporate ownership network", *Scientific Reports*, vol. 7, no. 1, p. 6246, 2017.

[4] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software", *Dr. Dobbs journal*, vol. 30, no. 3, pp. 202–210, 2005.

[5] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing", *Parallel Processing Letters*, vol. 17, no. 01, pp. 5–20, 2007.

[6] *The opencl specification*, `https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf`.

[7] *Openmp application programming interface*, `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf`.

[8] *The openacc application programming interface*, `https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.7.pdf`.

[9] S. J. Pennycook, J. D. Sewall, and V. Lee, "A metric for performance portability", *arXiv preprint arXiv:1611.07409*, 2016.

[10] *The opencl extension specification*, `https://www.khronos.org/registry/OpenCL/specs/opencl-1.2-extensions.pdf`.

[11] *Nvidia cuda programming guide*, `http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf`.

[12] M. J. Flynn, "Some computer organizations and their effectiveness", *IEEE transactions on computers*, vol. 100, no. 9, pp. 948–960, 1972.

[13] *Texas instruments opencl users' guide*, `https://downloads.ti.com/mctools/esd/docs/opencl/memory/memory-model.html`.

[14] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search", *Scientific Programming*, vol. 21, no. 3-4, pp. 137–148, 2013.

[15] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff, "A medium-scale distributed system for computer science research: Infrastructure for the long term", *Computer*, vol. 49, no. 5, pp. 54–63, 2016.

[16] Graph 500 Steering Committee, *Graph500 benchmark specification*, `https://graph500.org/?page_id=12`.

[17] J. Kunegis, "KONECT – The Koblenz Network Collection", in *Proc. Int. Conf. on World Wide Web Companion*, 2013, pp. 1343–1350. [Online]. Available: `http://userpages.uni-koblenz.de/~kunegis/paper/kunegis-koblenz-network-collection.pdf`.

[18] *Actor collaborations network dataset – KONECT*, Oct. 2016. [Online]. Available: `http://konect.uni-koblenz.de/networks/actor-collaboration`.

[19] J. Leskovec and A. Krevl, *SNAP Datasets: Stanford large network dataset collection*, `http://snap.stanford.edu/data`, Jun. 2014.

[20] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu", in *ACM SIGPLAN Notices*, ACM, vol. 51, 2016, p. 11.

[21] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite", *arXiv preprint arXiv:1508.03619*, 2015.

[22] J. Shen, J. Fang, H. Sips, and A. L. Varbanescu, "Performance gaps between openmp and opencl for multi-core cpus", in *2012 41st International Conference on Parallel Processing Workshops*, IEEE, 2012, pp. 116–125.

[23] P. Memarzia and F. Khunjush, "An in-depth study on the performance impact of cuda, opencl, and ptx code", 2015.

[24] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi, "Evaluating performance and portability of opencl programs", in *The fifth international workshop on automatic performance tuning*, vol. 66, 2010, p. 1.

[25] R. V. van Nieuwpoort and J. W. Romein, "Correlating radio astronomy signals with many-core hardware", *International Journal of Parallel Programming*, vol. 39, no. 1, pp. 88–114, 2011.

[26] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming", *Parallel Computing*, vol. 38, no. 8, pp. 391–407, 2012.

[27] J. H. Lee, N. Nigania, H. Kim, K. Patel, and H. Kim, "Opencl performance evaluation on modern multicore cpus", *Scientific Programming*, vol. 2015, p. 4, 2015.

[28] T. Gao, Y. Lu, B. Zhang, and G. Suo, "Using the intel many integrated core to accelerate graph traversal", *The International Journal of High Performance Computing Applications*, vol. 28, no. 3, pp. 255–266, 2014.