Imperial College London
Department of Computing

Inter-workgroup Barrier Synchronisation on Graphics Processing Units

Tyler Rey Sorensen

May 2018

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London

Declaration

This thesis and the work it presents are my own except where otherwise acknowledged.

Tyler Rey Sorensen

Abstract

GPUs are parallel devices that are able to run thousands of independent threads concurrently. Traditional GPU programs are data-parallel, requiring little to no communication, i.e. synchronisation, between threads. However, classical concurrency in the context of CPUs often exploits synchronisation idioms that are not supported on GPUs. By studying such idioms on GPUs, with an aim to facilitate them in a portable way, a wider and more generic space of GPU applications can be made possible.

While the breadth of this thesis extends to many aspects of GPU systems, the common thread throughout is the *global barrier*: an execution barrier that synchronises all threads executing a GPU application. The idea of such a barrier might seem straightforward, however this investigation reveals many challenges and insights. In particular, this thesis includes the following studies:

- *Execution models*: while a general global barrier can deadlock due to starvation on GPUs, it is shown that the scheduling guarantees of current GPUs can be used to dynamically create an execution environment that allows for a safe and portable global barrier across a subset of the GPU threads.

- *Application optimisations:* a set GPU optimisations are examined that are tailored for graph applications, including one optimisation enabled by the global barrier. It is shown that these optimisations can provided substantial performance improvements, e.g. the barrier optimisation achieves over a 10× speedup on AMD and Intel GPUs. The performance portability of these optimisations is investigated, as their utility varies across input, application, and architecture.

- *Multitasking*: because many GPUs do not support preemption, long-running GPU compute tasks (e.g. applications that use the global barrier) may block other GPU functions, including graphics. A simple cooperative multitasking scheme is proposed that allows graphics tasks to meet their deadlines with reasonable overheads.

Acknowledgements

First and foremost I thank my adviser, Ally Donaldson. These four years have shaped me not only as a researcher, but also as a person. Research is difficult mentally, emotionally and socially. I could not have learned from a better mentor than Ally in all of these aspects. I am thankful that Ally trusted me and gave me the freedom to work on the ideas that eventually turned into this thesis. My favourite emails to write to Ally would start with "I have an idea...". In a few weeks, Ally would patiently help turn my rough thoughts into an actionable plan. He always pushed to do my best possible work, while also allowing me to develop my own creative voice. I will always look up to his scientific integrity, driven by genuine curiosity and an aim to better the wider scientific community. Although the programme was extremely difficult, working with Ally has been one of the most fulfilling experiences of my life.

I thank all my wonderful collaborators and mentors who have taught me so much about science and have been excellent examples of how to navigate this journey. In particular, I thank John Wickerson, Hugues Evrard, Nathan Chong, Mark Batty, and Sreepathi Pai. I thank my internship mentors who gave me valuable industrial experience: Vinod Grover, Todd Mytkowicz, and Madan Musuvathi. I thank Margaret Martonosi for allowing me to join her group; our short time working together has already been full of amazing opportunities and support.

I thank Ganesh Gopalakrishnan for patiently introducing me to research while I was an undergraduate at University of Utah. He is a role model not only as a scientist, but as a mentor and a human. He is a moral lighthouse that I constantly look to. I am especially grateful to him and Zvonimir Rakamarić for allowing me to spend a few months recharging at University of Utah during my PhD.

I thank the new friends I made during my my time at Imperial College. In particular, I thank Kareem Khazem for being a constant and reliable source of support, advice, and fun. I'm sure East Acton is happy to be rid of us! I thank Carsten Fuhs for making Friday pub night academic enough for me to justify attending. I thank Heidy Khlaaf for paving a way for me. I thank Marija Selakovic (and Jovan) for immediately making me

feel like family. I thank everyone else that I was able to interact with in the Department of Computing and especially the Multicore and Software Reliability groups.

I thank Richard Bornat, James Brotherston and David Pym. They selflessly helped me during an extremely difficult time and helped me find my footing at Imperial College. This thesis would not have been finished without their support during that time.

I thank my parents, Ken and Valerie Sorensen, for the principles they instilled in me and for all their support and patience. I wasn't the most cooperative teenager growing up, but the one thing I knew they wouldn't tolerate was poor performance in school. Since the first time I had homework in grade school, my parents would ensure that it was completed to the best of my ability. I still carry these expectations with me, along with the confidence that has grown out of them. This journey has been full of rash decisions on my end, but their support gave me the courage to dive in head first. Finally, I will always look up to them for their integrity and work ethic.

I thank the rest of my family: my brothers, Jason and Adam, and my sister, Annalynn. It wasn't easy being so far away for so long. Although I couldn't be there for all the birthdays and camping trips, I love you guys.

I thank my oldest and dearest friends, with whom I grew up with and still am lucky enough to see occasionally. Something about small town central Utah creates some of the kindest and most genuine people I have ever met. No matter how much time or distance separates us, we can always pick up like it was yesterday. In particular, I thank: Scott and Mardi McDonald for making their spare bedroom my home away from home; Dallin Brown for always being up for an adventure; Clay and Stephanie Nielsen for holding down the hometown; and Jordan Peterson for hanging out in the deep end.

I thank Tony Field and Albert Cohen for providing an engaging final examination. Their feedback and discussions provided a wider context for me to see this work in. For the same reasons I thank all the anonymous reviewers that provided feedback for my submissions. I know reviewing can feel like a thankless task, but it has always improved my work.

I thank all the universities that have supported me and gave me a place to call home: Snow College, University of Utah, Imperial College London, and Princeton University.

> *We can always do better. I'm going to keep trying if you guys keep trying. Let's keep rockin' and rollin', man.* [And97]

Contents

List of Tables

List of Figures

1 Introduction

This thesis covers a range of topics related to GPU computing, namely: GPU execution models, optimisations for GPU applications, and multitasking on GPUs. However, the motivation behind the work has *always* had a simple foundation: explore uncharted territories of concurrent interactions on GPUs to gain the understanding required to build synchronisation constructs that, in turn, enable new ways for applications to take advantage of GPU acceleration.

The synchronisation construct that this thesis focuses on is a *global barrier*, which aligns the computation of all threads executing a program. Such a construct is widely available as a primitive in other parallel programming frameworks, yet it is not provided as a portable primitive on GPUs. Given this, It is only natural for developers to (1) desire a GPU global barrier in order to leverage intuitions and applications from other common parallel frameworks, and (2) wonder for what reasons such a barrier is not provided.

As this thesis shows, the implementation, application use cases, and multitasking consequences of a global barrier on GPUs present many interesting problems and yield insights that we believe are fundamental to parallel programming. Although the global barrier is the unifying thread of this thesis, the studies presented have a wider breadth, with interesting results in many areas of GPU computing.

The outline of this this chapter is as follows: first, the issues investigated in this thesis are illustrated using a short story about two curious undergraduate students attempting to write GPU programs (Section 1.1). Afterwards, the thesis contributions are more formally presented, organised by chapter (Section 1.2). Following this, a list of publications by the author is given, including both publications directly included in this thesis and other publications, the involvment in which has had a substantial influence on the works presented here (Section 1.3). The chapter concludes with formal acknowledgments of contributions that have directly influenced the work in this thesis (Section 1.4).

1.1 A short story

Inspired by the introduction of Marino et al.'s paper about the difficulties of weak memory consistency models [MMM$^+$15] as they might appear to an undergraduate computer science student.

1.1.1 Classic concurrency on GPUs?

Scott has just finished his second year as a computer science undergraduate student. His favourite course this term was a module on concurrency where he learned about Pthreads [NBF96], OpenMP [Ope15], and Java threads [PGB$^+$05]. He was intrigued by the fantastic speedups that parallelism could offer on today's machines, but he also understood that parallel programming is difficult, having spent many late nights debugging his coursework.

Over the summer holiday, Scott wanted to practice parallel programming. Being a curious student, Scott wanted to try programming a GPU. While GPUs weren't covered in the module, Scott was very aware of these devices as they were popular in high-profile areas of computing such as machine-learning, crypto-currency, and video games (of course!). Additionally, he had read that GPUs could be programmed in a manner similar to that which he had learned in his module, using languages such as OpenCL [Khr15]. Scott was excited to find that his laptop had an Intel HD5500 GPU equipped with OpenCL-capable drivers.

1.1.2 A GPU global barrier

After working through the usual GPU tutorials of reductions and matrix multiplication, Scott wanted to try applying material he had learned in his course. He was particularly fond of *barrier synchronisation*, which aligned the computation of all the threads running the program. By helping to tame the large amount of scheduling non-determinism of concurrent programs, applications that used this bulk synchronisation construct were easier for him to reason about. Most of the languages he had studied in his course provided barriers as primitives, but OpenCL did not, at least not a *global barrier* across all the threads executing the program. He knew however, from his favourite textbook [HS08, ch. 17], that he could implement his own barrier using atomic read-modify-write instructions, which were provided in OpenCL.

Before long, Scott had a barrier implementation in OpenCL and a short driver program to test the barrier. However, his excitement was short-lived; every time he tried to execute the barrier program, his computer froze and he had to restart it! Frustrated, but not defeated, Scott started searching the internet to understand what was happening. He quickly found a reference to a research paper discussing barriers on GPUs [XF10], where he learned that he should only run his barrier program with as many *workgroups* (OpenCL thread groups) as the GPU had processors, or compute units.

This solution seemed simple enough. Using the OpenCL device query functions [Khr15, ch. 4.2], Scott learned that his HD5500 had 24 compute units. However, even when the program was restricted to only run with this number of workgroups, the barrier program still froze the computer! Scott's frustration turned into determination and he continued debugging, trying everything he could think of. Eventually, late into the night, Scott found that the barrier program would work if executed with up to three workgroups. However, executing the program with four or more workgroups caused the computer to freeze. Exhausted, but feeling triumphant, Scott wanted to tell someone about his success. He knew that his concurrency classmate, Mardi, had a machine built for gaming with a high-end AMD R9 Fury GPU. He sent her the OpenCL barrier program to try.

The next morning, Scott saw a message from Mardi. She had successfully run the barrier program on her AMD GPU! However, in his exhausted state, he had forgotten to tell her to limit the workgroup count, and yet, somehow the barrier driver program still ran without issue for her using the full number of compute units on her AMD GPU.

▶ **Key idea:** A standard barrier implementation may deadlock due to starvation on current GPUs. Prior work [XF10, GSO12] has shown that such barriers can execute successfully if the number of workgroups that synchronise is limited to the number of workgroups that can run in parallel on the GPU. However, this number is difficult to determine statically. Depending on the architecture, it may or may not correspond to the number of documented processors (compute units) on the GPU. To complicate matters further, the number of parallel executing workgroups is also affected by the shared memory and register usage of the program.

Scott found this experience disconcerting. None of the other parallel programming models that he had studied before limited the number of threads that he could synchronise successfully with a barrier. Even more strange was that his GPU behaved differently from Mardi's in terms of the relation to the reported number of processors and how many workgroups could synchronise using the barrier. However, these concerns could be put aside for now: he was excited to find a GPU application that he could accelerate using this barrier.

1.1.3 Using the GPU barrier

Scott began looking for GPU applications in which his barrier might be useful. He noticed that many common GPU applications, like matrix multiplication, were *embarrassingly parallel*, i.e. there was little to no communication between threads. Because of this, it wasn't a clear that a barrier could be useful in such applications. After some searching, Scott found several recent works describing graph computations on GPUs [CBRS13, PP16, BNP12], e.g. breadth first search (BFS). He had learned about these algorithms in his early courses and decided to investigate.

While looking at a GPU implementation of BFS [BNP12], Scott noticed that the program as a whole was not just a single GPU program, but rather it was a series of small GPU programs launched iteratively from the CPU. The reason for this, Scott learned, was that global synchronisation was required between each iteration of the GPU program, and this was achieved via repeat GPU program launches [Khr15, ch. 3.2.4]. He was excited to realise that his global barrier could provide the same global synchronisation without having to repeatedly relaunch the GPU program. If the barrier overhead was less than the overhead of repeat program launches, then there could be some performance benefits!

It wasn't difficult to apply the barrier optimisation to the BFS application, and soon Scott was able to do some timing experiments. He was able to find the road-network of New York in a graph representation, which he used as an input to his application. He was thrilled when he saw that his barrier-optimised BFS application outperformed the original iterative application by 3.5×! Again, wanting to share his success, he sent his barrier-optimised application to Mardi. He knew Mardi was working on analysing social networks, which can be represented as graphs similar to his road-networks, so perhaps the application would be useful for her.

The next day he had a new message from Mardi, this one not as positive as her last. The barrier-optimised BFS application on her social network graphs was not any faster than iterative application on her AMD GPU! Again, Scott found this disconcerting: even after all the work to get a functional barrier, it did not provide a performance improvement for a different GPU and use case.

▶ **Key idea:** Performance portability is known to be difficult (e.g. see [SRD17]) and graph applications on GPUs are no exception. Due to the irregular parallelism of such applications, the utility of optimisations depends not only on architectural features, but also application input. A global barrier optimisation that achieves impressive speedups in one instance may not have a performance effect in a different instance; and, in fact, can even cause a slowdown.

1.1.4 Barriers freezing graphics

In addition to the barrier-optimised BFS application not speeding up Mardi's social network graph computation, she also complained that her entire computer would have short periods of temporary freezes when running the application. These freezes were intrusive enough that she could not work on her computer while the application ran in the background, as she was used to doing with the original iterative application. Scott hadn't noticed the temporary freezes: he had been too focused on the task at hand that he wasn't trying to use his computer for anything else while running the timing experiments.

When he started programming again he noticed that Mardi was right; his computer would temporarily freeze during the barrier-optimised BFS application. Even though the analysis only took about a second to run, the UI disruption was absolutely noticeable. This GPU barrier, which started as an innocent exploration of core ideas from his concurrency course, was having issues at every point!

▶ **Key idea:** Many current GPUs do not support preemptive multi-tasking. If long-running compute tasks (e.g. a barrier-optimised BFS) are run on a GPU that is also responsible for graphical UI, then the UI may become unresponsive until the compute task finishes executing.

1.1.5 Looking forward

It was now almost the end of summer holiday and the term would starting soon. Scott and Mardi felt refreshed and prepared from their extracurricular programming exercises. They spent the next day planning their fall term schedules. In particular, Scott noticed a course titled "Formal Programming Reasoning", in which maths and logic were used to describe the behaviours of programs. He thought back to the strange differences of behaviour between his Intel GPU and Mardi's AMD GPU when executing the global barrier. Could there be any sort of maths to describe what they observed? He couldn't resist signing up!

▶ **Key idea:** The problematic barrier behaviours that cause machine freezes can be formally reasoned about using temporal logic. These freezes occur because current GPUs do not provide fair scheduling guarantees. However, because limited variants of the barrier do execute successfully, GPU schedulers are not completely unfair. Formalising the exact fairness guarantees provided (or assumed) on GPUs allows precise program reasoning not just for barriers, but other important blocking synchronisation idioms, such as mutexes.

After finalising their course schedule, Scott and Mardi talked about their future plans after they graduated. Scott said he wanted a tech job where he got free food and massages! While that all sounded very nice, Mardi wasn't so sure. She was considering applying for grad school...

1.2 Contributions

The contributions of this thesis correspond to investigations of the issues faced by Scott, Mardi, and likely many other GPU developers[1] when experimenting with a GPU global barrier. While these barriers are not supported *officially* by any major GPU programming model, it is hoped that the contributions in this thesis will be useful as these programming models evolve. The original contributions of this thesis are as follows:

- Chapter 3 presents the occupancy-bound execution (OBE) model: an abstract description of GPU scheduling, providing fair execution to any thread that has previously executed an instruction. These guarantees can be used to dynamically create an environment where it is safe to execute a barrier across a subset of the threads. An experimental campaign shows that a wide range of GPUs from different vendors (7 GPUs across 4 vendors) honour OBE guarantees.

  Using insights from this chapter, Scott and Mardi could implement a global barrier that worked on both of their GPUs, without considering architectural details.

[1] see:
`https://stackoverflow.com/questions/7703443/inter-block-barrier-on-cuda`
and
`https://stackoverflow.com/questions/34476631/opencl-and-gpu-global-synchronization`

- Chapter 4 examines a domain-specific language (DSL) and an optimising compiler for GPU graph applications. The work explores the functional generalisation of these optimisations to target a wide range GPUs. In particular, the OBE guarantees (as defined in Chapter 3) are used to enable an optimisation that requires a global barrier in a portable way. Experimental results show that performance profiles (across 6 GPUs spanning 4 vendors) can be achieved that resemble the original Nvidia results. The performance portability of the optimisations is investigated as the runtime effect of optimisations varies across GPUs, applications, and inputs.

  Using insights from this chapter, Scott and Mardi could understand that the performance effect of the global barrier optimisation is highly sensitive to different application graph inputs, but not as much to different GPUs. If Mardi would have tried the same input on her AMD GPU, she would have seen similar speedups as the Intel GPU. The DSL compiler presented in this chapter would allow Scott and Mardi to automatically and rapidly try many different optimisation settings on their respective GPUs.

- Chapter 5 is motivated by the observation that long-running GPU compute applications cause graphical applications to become temporarily unresponsive if both applications leverage the same GPU. To address this, a cooperative multi-tasking framework is presented where GPU compute applications and interactive tasks, e.g. graphics, can efficiently share resources. The primary primitive in this framework is a new global barrier, called a resizing barrier, in which applications surrender and reclaim compute resources at barrier calls. A prototype scheduler is presented and it is shown that synthetic graphics tasks are able to meet their interactive deadlines while multi-tasking with long-running GPU compute applications.

  Using insights from this chapter, Scott and Mardi could understand why their machines would temporarily freeze when executing a barrier-optimised application and understand how researchers are thinking about addressing the problem.

- Chapter 6 presents the notion of semi-fairness: a theoretical basis to reason about scheduling guarantees, such as the ones provided by OBE. Semi-fairness formally defines fairness properties through a temporal logic formula, based on weak fairness, that is parameterised by a *thread fairness criterion*, a predicate that enables fairness per-thread at certain points of an execution. Using this framework, the fairness properties of several GPU schedulers are formally defined and the behaviours of common synchronisation idioms are analysed under each scheduler.

Using insights from this chapter, Scott and Mardi could apply the material from their formal reasoning course to understand precisely the scheduling guarantees that are officially provided, or commonly assumed, for GPUs. This would allow them to formally reason about the next blocking synchronisation idiom that they wish to execute on a GPU.

1.3 Publications

The material presented in this thesis has either been published in conference articles or is currently under submission. Organised by chapter, these publications are as follows:

- The description of occupancy-bound execution and the discovery protocol, presented in Chapter 3, is based on material originally published in the 2016 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16) [SDB$^+$16]. The work has an associated approved artifact, which can be found at:

  `https://github.com/mc-imperial/gpu_discovery_barrier`

  A companion experience-report, detailing the difficulties of running the associated experimental campaign across 7 GPUs, was published in the 4th International Workshop on OpenCL (IWOCL'16) [SD16b].

- The portable GPU graph application framework, presented in Chapter 4, is currently under submission. The citation references the current draft [SPD18].

- The GPU cooperative multi-tasking scheme, presented in Chapter 5, is based on material originally published in the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'17) [SED17a]. The work received a distinguished paper award.

- The formal description of scheduler fairness guarantees, presented in Chapter 6, is based on material originally published in the 29th International Conference on Concurrency Theory (CONCUR'18) [SED18].

The author has additionally been involved in several other published works, all with a common theme of testing or specifying fine-grained interactions of threads in concurrent systems. These additional works have helped shape the research direction of this thesis:

- *The Semantics of Transactions and Weak Memory in x86, Power, ARMv8, and C++*: this work specifies the semantics of programs that use transactional memory on systems that have relaxed memory models; published in the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI'18) [CSW18]. The work received a distinguished paper award.

- *Automatically Comparing Memory Consistency Models*: this work presents a framework that given two different memory consistency models, can synthesise programs that have different behaviours under the provided memory models; published in the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17) [WBSC17].

- *Exposing Errors Related to Weak Memory in GPU Applications*: this work presents a black-box testing methodology that is able to effectively reveal bugs related to weak memory in GPU applications; published in the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI'16) [SD16a].

- *GPU Concurrency: Weak Behaviours and Programming Assumptions*: this work describes the synthesis of weak memory unit tests and a testing framework that is able to effectively run such tests on GPUs. The results show that many GPUs applications assumed an unsound memory consistency model; published in the 20th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15) [ABD+15].

1.4 Formal acknowledgements

I gratefully acknowledge and thank my collaborators throughout my PhD. While the majority of the work presented here was done by me, each co-author of the publications discussed in this thesis helped to develop the scientific narrative (this includes the prose in some cases) that will be presented. Specifically: Alastair F. Donaldson, Mark Batty, Ganesh Gopalakrishnan and Zvonimir Rakamarić contributed to the work presented in Chapter 3. Alastair F. Donaldson and Sreepathi Pai contributed to material presented in Chapter 4. Alastair F. Donaldson and Hugues Evrard contributed to material presented in Chapters 5 and 6.

In terms of technical contributions, I acknowledge and thank Sreepathi Pai for doing substantial work in modifying the IRGL compiler to generate OpenCL code, discussed in Chapter 4. I acknowledge and thank Alastair F. Donaldson for developing the *kernel* *merge* tool described in Chapter 5, which combines two GPU kernels into one *megakernel* and also adds hooks to to interact with the prototype scheduler. I acknowledge and thank Hugues Evrard for (1) adapting the two work-stealing applications discussed in Chapter 5 to use cooperative kernels and for producing Figure 5.1 in Chapter 5; and (2) generating labelled transition systems which influenced Figures 6.2b, 6.3b and 6.4 in Chapter 6.

While not directly appearing in this thesis, I thank and acknowledge: Mark Batty for producing a library abstraction proof for the memory consistency properties of the global barrier shown in Chapter 3 (as seen in [SDB$^+$16]); Alastair F. Donaldson for writing operational semantics for the cooperative kernel extensions in Chapter 5 (as seen in [SED17b]); and Hugues Evrard for leading model-checking efforts in the early stages of the work presented in Chapter 6. These three contributions substantially improved their respective publications.

2 Background

In this chapter, an overview of GPU hardware and programming models is given along with some related context. Section 2.1 provides a brief history of GPUs, from their beginnings being used exclusively to accelerate graphics rendering in video games, to their current status as programmable accelerators used in many domains. Next, details about GPU hardware and the programming models used to write programs that execute on such hardware are given in Sections 2.2 and 2.3, respectively. Finally, the GPUs used throughout this thesis and their technical profiles are presented in Section 2.4.

2.1 A brief history

A brief history of GPUs is presented here, with an emphasis on the development of general purpose computing on GPUs and some examples of their current usages. Unless otherwise mentioned, this history is derived from Chapter 1 of the textbook *CUDA by Example* [SK10] and the TechSpot article *The History of the Modern Graphics Processor* [Sin13].

The idea of using special-purpose hardware to accelerate graphics rendering has been around since the early 1950s. It is believed that a flight simulator developed by MIT in 1951 was the first successful instantiation of this idea. Through the next 30 years, specialised hardware was developed to render 2D graphics, including the Television Interface Adaptor in 1977, which was a special purpose chip designed for video and audio output of the Atari video game system. In the early 1990s, GUI driven operating systems, e.g. Microsoft Windows, imposed the requirement that home computers must be equipped with a 2D graphics card. Along with these operating systems, APIs for programming such graphics cards were developed, most notably a cross-vendor collaboration called OpenGL in 1992 and Microsoft's DirectX in 1994.

In 1996, 3Dfx released the Voodoo GPU, the first consumer GPU to accelerate 3D graphics. Other companies, such as Nvidia and ATI (now a part of AMD), began developing such cards in fierce competition. In 1999, Nvidia officially used the term *graphics processing unit* for the first time with the release of the GeForce 256, stating [NVI99]:

> The technical definition of a GPU is “a single chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second.”

while also humbly adding:

> The GPU changes everything you have ever seen or experienced on your PC.

The next breakthrough came in 2001. Version 8.0 of the DirectX standard required hardware that allowed *programmable* vertex and pixel shaders. The first GPUs to satisfy this requirment were the Nvidia GeForce 3 series. Now graphics developers had the ability to customise the computations on the GPU. Given the high throughput potential of GPUs, this new flexibility also attracted developers from other domains. Using graphics APIs, developers could execute general applications in a roundabout way by (1) presenting their application as a graphics application and then (2)interpreting the GPU graphical output as general output.

Nvidia then moved to accommodate more general purpose programming on their GPUs. The GeForce 8800 GTX GPU, released in 2006, supported the new CUDA programming model. CUDA allowed for C-like programs to be developed and executed on Nvidia GPU hardware. To support this new programming model, the new GPU hardware was designed to conform to the IEEE floating point standard, and featured a unified shader core that combined the vertex and pixel shaders. However, other vendors, e.g. AMD, Intel, and Qualcomm, were also interested in developing a framework for general purpose computations on their respective GPUs. These vendors organised through the Khronos Group, a non-profit organisation whose aim is the development of open APIs for acceleration across a range of devices. In 2008, Apple began a Khronos working group for a new and portable GPU compute language, which would eventually became OpenCL. The Mac OS X Snow Leopard operating system supported the first version of OpenCL in 2009 [Wik18]. Interestingly, Apple deprecated support of OpenCL and OpenGL in 2018 to support their new GPU language, Metal.[1]

These general purpose languages and frameworks continue to evolve. CUDA is currently on version 9.1 (January 2018) [Nvi18a], and OpenCL is on version 2.2 (May 2017) [Khr17]. Each version adds new features, allowing more ways for applications to take advantage of GPU acceleration. These frameworks enjoy significant usage both in research and in

[1] See:
`https://appleinsider.com/articles/18/06/04/opengl-opencl-deprecated-in-favor-of-metal-2-in-macos-1014-mojave`

industry. For example, the website `https://hgpu.org/`, an aggregate site for GPU research, reports that as of 2017 more than 14,000 and 63,000 academic papers have been published using OpenCL and CUDA, respectively. In an industrial setting, clusters of GPUs are widely used for machine-learning applications. For example, Facebook reports 90% scaling efficiency using a cluster of 256 GPUs [GDG$^+$17]. Consumers are also using general purpose GPU computation: the online tech publication Tom's Hardware reported in February of 2018 that there was a shortage of consumer GPUs, and as such, the prices had doubled or tripled. The high demand is attributed to GPU acceleration for cryptocurrency applications.[2]

GPUs architectures, their programming models, and the applications accelerated by GPUs continue to grow and evolve. This thesis explores ways in which the programming model might evolve, and shows that there is potential for new types of application acceleration using the proposed extensions.

2.2 GPU hardware

Many vendors currently produce GPUs, including Nvidia, AMD, Intel, ARM, and Qualcomm. Because GPUs from different vendors have varying architectural features, describing a generic common GPU architecture is difficult. However, several common characteristics pertinent to this thesis are described now and concrete details for three different GPU architectures will be given when possible. Specifically, the three architectures explored are: (1) the Nvidia Volta architecture [NVI17], as it is the most recent Nvidia architecture; (2) the AMD GNC architecture [AMD12], as it is the architecture of the two AMD chips explored in this thesis (see Section 2.4); and (3) the Intel Gen9 architecture [Int15a], as two of the Intel GPUs used in this thesis belong to this architecture, Iris and Hd5500. Additionally, two ARM GPUs are used in this thesis, but authoritative architecture specifications for either Midgard (the architecture of the ARM GPUs used in this work) or Bifost (the most current ARM archtecture) could not be found. Because OpenCL is a portable GPU programming framework, terminology from OpenCL will be used when possible.

[2] See:
`https://www.tomshardware.com/news/ethereum-effect-graphics-card-prices,34928.html`

Execution hierarchy The base unit of execution on a GPU is called a *processing unit* (PU), a scalar processor capable of executing an instruction set, including integer and floating-point computations.

Groups of processing units are partitioned into same-sized disjoint *SIMD execution groups* (SEGs).[3] Different architectures have different sizes and architectural names for SEGs. AMD calls such groups *wavefronts* and they have a fixed size of 64. Nvidia calls such groups *warps* and they have a fixed size of 32. SEG sizes on Intel GPUs vary between 8 and 16, adapted to the resource requirements of a program.

SEGs are partitioned into same-sized disjoint *compute units* (CUs). AMD architectures have 4 SEGs per CU. Nvidia uses the terminology *streaming multiprocessor* (SM) to refer to CUs and each SM contains 4 SEGs. Intel uses the term *execution units* (EUs) to refer to a compute unit; because Intel SEGs are variable sized, it is unclear from the documentation how many SEGs each EU contains.

Memory hierarchy The base memory unit of a GPU is a *register*, which exists in a register file. A register file is either local to an SEG or a CU. On AMD, each SEG has a register file 64 KB. On Intel, each CU has a register file of 28 KB. On Nvidia, each CU has a register file of 256 KB.

Each CU has a two caches: one for CU-local memory and another as an L1 cache for an inter-CU region of main memory. On AMD, the CU-local memory is 64 KB and the L1 cache is 16 KB. On Intel, there are not CU-local caches, but instead, eight CUs are grouped into a *slice*, which has a cache of 512 KB. The slice cache is automatically partitioned between CU-local memory and L1 cache depending on the resource requirements of a program. On Nvidia, the CU-local memory and L1 cache is a shared memory unit, as such the partition can be manually configured, sharing 96 KB.

Finally, a GPU has a main memory shared across all CUs. AMD has an L2 cache, whose size various across GPU models, shared across CUs to accelerate accesses to main memory. Intel shares main memory with the CPU main memory and does not document an L2 cache. Nvidia has an L2 cache of size 6144 KB. The main memory size is typically large, being the primary storage location for large amounts of data to compute. For example, the AMD R9 Fury GPU has 4 GB and the Nvidia Volta V100 GPU has up to 32 GB of main memory.

[3] This terminology is specific to this thesis, as there does not appear to be unified name for such hardware components.

2.3 GPU programming: OpenCL

GPU programming models enable development of programs that are accelerated on GPU hardware. There are several GPU programming models that exist today, each with benefits and drawbacks. CUDA [Nvi18a] is the Nvidia proprietary programming model. The benefits of CUDA are that it is well supported by Nvidia and quick to evolve. The drawback of CUDA is that it is only supported on Nvidia GPUs. OpenCL [Khr16a] is a portable GPU programming model similar to CUDA and developed by a Khronos working group with members across many hardware and software vendors (e.g. Nvidia, Intel, AMD). The benefits of OpenCL are that it is portable across vendors and is reasonably well supported.[4] The drawbacks of OpenCL is that it is slow to evolve, perhaps to accommodate the wide range of target devices that may be required to implement it. OpenCL often seems to adopt CUDA features, but lags behind by several years, especially for availability on consumer devices. Heterogeneous System Architecture (HSA) [HSA17] aims to be a portable programming model and framework, but has not appeared to pick up much practical support outside of AMD. All of these models exist at a low-level of abstraction, providing the opportunity for hand-written, detailed architectural optimisations. However, this low-level also means that program development may be difficult, requiring domain expertise and being vulnerable to C-like bugs (e.g. out-of-bounds memory errors).

The different GPU programming models generally share common characteristics, i.e. programming constructs to exploit the GPU hardware hierarchy. The high-level ideas presented in this thesis are not tied to any particular GPU programming model and generally use only the common features. However, for cohesion and to provide portable experimental artifacts, this thesis largely uses OpenCL throughout. Specifically, version 2.0 of OpenCL is used, as it provides most of the features required by the constructs explored in this thesis, and enjoys support from current Intel and AMD GPUs (see Section 2.4).

An OpenCL GPU program consists of two parts:

- a *host* program, written in C/++ and executed on the CPU. The host program interfaces with the GPU, e.g. initialising GPU memory and launching GPU programs, through the OpenCL API.

[4] OpenCL is not limited to GPUs, and can be executed on CPUs and FPGAs. However, GPUs are the main target of this thesis, and thus the focus will be limited to OpenCL execution on GPUs.

- a *device* program, often called a *kernel*, and executed on the GPU. This program is written in OpenCL C, which is based on C99. It is executed in a *single instruction, multiple threaded* (SIMT) manner, whereby all threads execute the same program but may query unique thread identifiers to access distinct data and follow varying control paths.

OpenCL C supports a hierarchical programming abstraction with components that map naturally to GPU architectural features. At the base level there are *threads*,[5] which execute on a GPU's PUs. Threads are organised into disjoint, equal-sized, *subgroups*. Threads in the same subgroup are often executed on the same SEG. Subgroups are organised into disjoint, equal-sized, *workgroups*. Typically subgroups in the same workgroup are executed on the same CU, which can execute several subgroups concurrently. A kernel is executed by one or more workgroups. The number of threads per workgroup, and the number of workgroups executing a kernel are specified by the host upon kernel launch. Most GPU vendors support a maximum workgroup size of 256, with the exception of Nvidia, which supports a maximum workgroup size of 1024. The maximum number of workgroups that can execute a kernel is not practically constrained, as threads across workgroups do not enjoy the same locality exploiting primitives that intra-workgroup threads have. Additionally, because there are no requirements for workgroups to execute concurrently, if the device is oversubscribed, workgroups can simply execute in waves.

The relative location of threads in the hierarchy dictates the extent to which they can exploit locality of resource when communicating, or otherwise interacting, with one another. Threads in the same subgroup enjoy the most locality and threads in different workgroups enjoy the least.

Memory spaces OpenCL exposes four memory spaces, each with varying locality, and hence performance, properties:

- **Private Memory.** This is the most localised memory, being thread-local. Private memory is provided in the register file, or an overflow memory region. Accesses to private memory are typically very fast due to its spacial locality to the PU, however, it is not useful for inter-thread interactions due to its limited scope. Recall that register files are typically local to SEGs; thus, subgroup primitives (as provided in OpenCL 2.1) may allow intra-subgroup threads to efficiently share memory in this region. Private memory does not persist across kernel invocations.

[5] In strict OpenCL terminology, threads are called *workitems* this thesis opts to use *threads* due to the conceptual similarities and historical prevalence of the word.

- **Local memory.**[6] This memory is local to a workgroup and typically implemented in the CU-local cache. Local memory is considered efficient to use for intra-workgroup interactions, but is not shared across workgroups. Like private memory, local memory does not persist across kernel invocations.

- **Global memory.** This memory is shared across all the threads on the device and typically implemented on the GPUs main memory. Compared to private and local memory, accesses to globally memory are considered to be slow, but it is large and can facilitate inter-workgroup interactions. Unlike private and local memory, global memory *does* persist across kernel invocations.

- **Shared virtual memory (SVM).** This memory is shared between host and device and can be used for fine-grained GPU-CPU interactions. SVM is not yet well supported on current GPUs; in the cases where support is provided, accesses to this memory region are considered to be very slow. In this thesis, SVM is used only in Chapter 5 and was only reliably provided on the Intel GPUs (see Section 2.4).

Workgroup barrier GPU programming models typically provide an intra-workgroup *barrier* instruction, which aligns the computation of threads within a workgroup and ensures up-to-date memory values can be read across these threads. For OpenCL, this construct is provided through the `barrier` instruction [Khr16a, p. 99].[7] This instruction must be called in a *workgroup-uniform* location; that is, the barrier instruction must be encountered by all threads within a workgroup or none of them. This constraint applies to to loop iterations as well: all threads within a workgroup must encounter the same barrier instance on the same loop iteration.

A thread that executes an instance of a barrier instruction must wait at the barrier instruction until all other threads in the workgroup have reached the same barrier instance. At that point, often called the barrier *release*, all threads can continue execution. Furthermore, all memory writes in workgroup-shared locations (i.e. local, global and shared virtual memory) will be visible to all other intra-workgroup threads after the barrier release.[8] This provides a simple execution and memory model for intra-workgroup cooperation and com-

[6] A potentially confusing name given that *local* is not qualified. The situation is not improved in CUDA, in which this memory region is referred to as *shared*.

[7] This instruction was officially renamed to `work_group_barrier` in OpenCL 2.0, but `barrier` remains supported; the latter is used in this thesis for brevity.

[8] In OpenCL, the `barrier` instruction takes additional arguments denoting which memory regions to synchronise, local or global (or both). In this thesis, it is assumed both memory regions are synchronised.

Table 2.1: OpenCL execution environment functions.

Function	**Description**
`get_local_id`	a workgroup local id, unique and contiguous within workgroup
`get_group_id`	a workgroup id, the same value for all threads in a workgroup
`get_global_id`	a global id, unique and contiguous for all threads
`get_local_size`	the number of threads per workgroup
`get_num_groups`	the number of workgroups executing the kernel
`get_global_size`	the number of threads, across all workgroups

munication. A similar instruction, `sub_group_barrier`, aligns intra-subgroup threads in a similar manner.

There currently does not exist native way to barrier-synchronise threads *across* workgroups during kernel execution. Current methods for such synchronisation either involve: (1) ending and relaunching the kernel, which invalidates non-persistent memory and requires interaction with the host, or (2) using ad hoc and non-standardised software inter-workgroup barrier implementations. Both of these methods are discussed in Section 3.2. This thesis explores the semantic requirements to implement such a barrier in a safe and portable way.

Execution environment The OpenCL *execution environment* provides information about the ids and number of threads executing a kernel, accessed through the functions detailed in Table 2.1 (called workitem built-in functions in OpenCL [Khr16a, pp. 69-70]). The execution environment is *static*: the number of threads and workgroups is fixed on kernel launch. Threads can query the execution environment to access contiguous unique data in a data-parallel program. Other GPU programming models, e.g. CUDA and HSA, provide similar execution environment functions. Chapters 3 and 5 propose new execution environment functions that return information about a subset of OpenCL threads for which relative forward progress is guaranteed.

OpenCL kernels may be launched with multi-dimensional workgroup and *NDrange* (the number of workgroups to execute a kernel) sizes. The execution environment functions would then take an integer argument to specify the dimension of the id to be returned. Similar to multi-dimensional arrays, multi-dimensional workgroups and NDranges can conceptually be thought of a re-organisation of a large single dimensional structure. Thus, in this thesis only single dimensional workgroup and NDranges are considered.

```
 1  kernel void vector_add(global float *A,
 2                         global float *B,
 3                         global float *C) {
 4
 5    local float scratchpad_A[256];
 6
 7    if (get_local_id() == 0) {
 8      for (int local_index = 0;
 9           local_index < get_local_size();
10           local_index++) {
11        int global_index = get_global_id() + local_index;
12        scratchpad_A[local_index] = A[global_index];
13      }
14    }
15
16    barrier();
17
18    C[get_global_id()] = scratchpad_A[get_local_id()]
19                       + B[get_global_id()];
20
21  }
```

Figure 2.1: A vector addition OpenCL kernel using local memory and workgroup synchronisation.

An example To illustrate how an OpenCL kernel is written, Figure 2.1 shows an example of a vector addition OpenCL kernel. This example uses local memory and workgroup synchronisation in a contrived way to illustrate their uses. An actual implementation would likely only use global memory, as vector elements are not used in a way that justifies the movement to local memory.

A kernel is a `void` function annotated with the `kernel` keyword, as seen in line 1. This kernel function is executed by a number of threads and workgroups specified by the host at kernel launch time. Threads can then branch to different code paths, or access different memory locations, based on the execution environment functions, as will be discussed below.

The host is responsible for allocating and initialising global memory in buffers through the OpenCL API. The host can then set the buffers as kernel arguments; global memory arguments are annotated with the `global` keyword. In this example, the host must provide three global float arrays, `A`, `B`, and `C` (lines 1, 2, and 3). The kernel will then compute `A` plus `B` element-wise and store the result in `C`. One element of the array will be processed per thread. After kernel computation, the host will be able to copy `C` back to host memory and obtain the result.

This kernel uses a local memory region to cache the `A` array, declared on line 5 using the `local` keyword. The local memory array has only 256 elements, one per workgroup thread. Thus, host is responsible for not launching more than 256 threads per workgroup. In practice, such numbers are often defined via macros, or through a variable-sized local array that can be specified at kernel launch. Additionally, because the size of the arrays are not passed in or checked, the host is responsible for launching the kernel with a number of threads equal to the number of items in the float arrays.

On line 7, one representative thread per workgroup (with local id of 0) will branch to cache a subset of the `A` array into the local memory scratchpad. The for-loop of line 8 copies one element for each thread of the workgroup into local memory. Because local memory s workgroup-local, the scratchpad is indexed from 0 to the workgroup size (255). The global array is shared across workgroup so its index is computed as an offset of the representative's global id. Threads in the same workgroup wait for the representative to populate the local memory cache at the `barrier` instruction of line 16. Recall that the `barrier` instruction synchronises all threads in the same workgroup.

Finally, lines 18 and 19 compute the vector addition. The `C` and `B` array are accessed by each thread using their unique global ids. The `A` array, cached in local memory, is indexed using the workgroup local id.

2.3.1 OpenCL feature classes

OpenCL has evolved rapidly: since the announcement of version 1.0 in October 2009, there have been five major versions of OpenCL, the most recent, version 2.2, was announced in May 2017. Each new version introduces a variety of new features that enable new ways for threads to interact, possibly exploiting the architectural features at different levels of the GPU hierarchy. Additionally, there are features that are not officially supported by OpenCL but that have been shown empirically to be supported by a range of current GPUs.

Because the version of OpenCL supported by current GPUs varies across vendors (see Section 2.4 for examples), an OpenCL application that uses cutting-edge features introduced in recent versions will not be able to run directly on platforms whose version support is lagging. To account for this, in this thesis OpenCL features are described via a series of backward-compatible *feature classes*, essentially corresponding to different versions of OpenCL. These feature classes are used in Section 4.3 to describe the level of support required for the different optimisations required by a graph application DSL. The feature classes are as follows:

OCL 1.x (support for OpenCL 1.0–1.2) This lowest-common-denominator class assumes only the concurrency support provided by OpenCL 1.0, which remains unchanged in versions 1.1 and 1.2. All threads executing a kernel can access the device's *global memory*. Threads can perform vanilla reads and writes from this memory along with a variety of read-modify-write instructions (e.g. compare-and-swap). Threads in the same workgroup can communicate through the faster *local memory*. Additionally, threads in the same workgroup can synchronise locally via the `barrier` instruction.

OCL 2.0 (support for OpenCL 2.0) OpenCL 2.0 provides a detailed *memory consistency model*, similar to that of C++, which allows fine-grained inter-thread communication in a well-defined manner through special atomic instructions and variables. Atomic instructions can be syntactically annotated with the level of the OpenCL hierarchy that the interacting threads share; this allows implementations to take advantage of architectural locality [Khr15, pp. 45-53]. Such a memory consistency model requires hardware support for cache flushing and invalidation to implement synchronisation semantics. More details about the OpenCL memory model are provided in Section 2.3.2.

OCL 2.1 (support for OpenCL 2.1) OpenCL 2.1 allows programs to exploit the SIMD execution groups through *subgroups*. New primitive functions are provided that enable threads in the same subgroup to efficiently synchronise, share thread-local variables (through the register file), and perform higher-level functions across subgroups, such as reductions and prefix sums. Additionally, there are primitives that allow thread-local predicates to be evaluated across a subgroup, either as a disjunction or conjunction [Khr16b, pp. 133-140].[9]

OCL FP (assumption of forward progress) The OpenCL standard does not provide any independent forward progress guarantees between threads in different workgroups. This means that in principle, implementation of common synchronisation primitives, such as inter-workgroup execution barriers, are prone to unfair executions where threads are blocked indefinitely. However, Chapter 3, as well as prior work [GSO12, XF10], shows that many current GPUs empirically provide forward progress under the *occupancy-bound*

[9] The documentation exists as a 2.0 extension that was made core in 2.1.

execution model, which states: concurrently executing threads will continue to be concurrently executed. Such guarantees require that the OpenCL framework, e.g. through the driver, continues to schedule workgroups in a fair way after they have been scheduled initially. This feature class assumes support for the occupancy-bound execution model.

2.3.2 OpenCL 2.0 memory consistency model

The OpenCL 2.0 memory model, based on that of C++11 [ISO12], employs a *catch-fire* semantics, where races on regular variables lead to undefined behaviour. Atomic variables, and corresponding memory accesses, are provided to give semantics to code that would otherwise be racy. Synchronisation between threads can be achieved by associating a *memory order* with each atomic variable access. The memory orders relevant to this work are *release* (applied to store operations) and *acquire* (applied to load operations). An acquire load that reads a value written by a release store in a different thread creates a *happens-before* edge from the store operation to the load operation, i.e. the operations *synchronise*. The formal synchronisation properties required for this work are provided in Section 3.5.2. The final memory order relevant to this work is *relaxed* (applied to store or load operations), which allows concurrent accesses, but provides no synchronisation.

In OpenCL (and unlike C++11), an atomic access has an associated *memory scope* annotation, specifying a level of the OpenCL execution hierarchy. This declares the intent to concurrently access the variable only within this level of the hierarchy, so that synchronisation is only provided within the given scope. The scope annotations relevant to this work are *workgroup* and *device*, allowing synchronisation between threads only in the same workgroup, and between arbitrary threads executing a kernel, respectively.

2.4 GPUs used in this work

Table 2.2 lists the GPUs used in empirical investigations throughout this thesis. A total of 11 GPUs spanning four major vendors, Nvidia, AMD, Intel and ARM, were used. While the number of CUs is reported to give an idea of the amount of parallelism of the GPU, Chapter 3 shows that compute units are not a completely reliable measure of this. All ARM and Intel GPUs are integrated; all Nvidia GPUs are discrete; for AMD, the R9 is discrete and the R7 is integrated. The two ARM GPUs are the same model, differing only by the number of compute units. Two Intel GPUs (Iris and Hd5500) have subgroup sizes that change depending on the amount of resources used by a compiled kernel; sizes of 8 and 16 have been observed. The other Intel chip (Hd520) was not used in any study

Table 2.2: The GPUs considered in this work, a short name used throughout the thesis (Short name), the number of compute units (#CUs), the subgroup size (SG size), the supported OpenCL version (OCL), and the chapter(s) where each GPU is used for empirical studies (Chapter).

Vendor	Chip	Short name	#CUs	SG size	OCL	Chapter
Nvidia	Quadro M4000	M4000	13	32	1.2	4
	GTX 1080	Gtx1080	20	32	1.2	4
	GTX 980	Gtx980	16	32	1.2	3
	Quadro K5200	K5200	12	32	1.2	3
Intel	HD520	Hd520	24	?	2.0	5
	HD5500	Hd5500	27	8,16	2.0	3,4,5
	Iris 6100	Iris	47	8,16	2.0	3,4,5
AMD	Radeon R9 Fury	R9	28	64	2.0	3
	Radeon R7	R7	8	64	2.0	3,4
ARM	Mali-T628	Mali-4	4	1	1.2	3,4
	Mali-T628	Mali-2	2	1	1.2	3

requiring subgroups and thus the subgroup size was never queried. The ARM GPUs do not have SIMD execution groups, but it is semantically sound to model their subgroup size as one.

The justification for why particular GPUs are used in the different chapters of this thesis is as follows:

- Chapter 3 excludes M4000 and Gtx1080 (Kepler and Maxwell architectures, respectively) because these GPUs were provided by Sreepathi Pai at UT Austin and were not available at the time of this work. Chapter 3 Additionally excludes Hd520 because it was in Alastair F. Donaldson's personal laptop and two other Intel GPUs were available.

- Chapter 4 excludes K5200 and Gtx980 as two newer Nvidia GPUs were provided by Sreepathi Pai. R7 was excluded because it started showing unpredictable behaviour shortly after the work of Chapter 3 and was deemed to be unreliable; e.g., execution times of the same kernel varied by several orders of magnitude. Mali-2 was excluded due to its similarity to Mali-4 and the extremely long execution time of the experimental campaign on ARM (97 hours).

- Chapter 5 uses only Intel GPUs as they provided the only reliable support of shared virtual memory needed by the prototype scheduler. Alastair Donaldson kindly allowed the use of his personal laptop, with Hd520, so that one more GPU could be used in this study.
- Chapter 6 is entirely theoretical without an experimental study.

Memory model implementation The chips considered (Table 2.2) all support the OpenCL 2.0 memory model with the exception of Nvidia and ARM chips. Because all experimental studies in this thesis require memory consistency guarantees, custom implementations of the OpenCL 2.0 atomic operations were developed for these chips. The Nvidia implementation is based on previous empirical testing of these chips from previous publications [ABD$^+$15] and [WBSC17], both of which include the thesis author as a publication author. Specifically, inline PTX (Nvidia's low level intermediate language) is used to provide Nvidia specific memory fences. The ARM implementation uses OpenCL 1.1 memory fence instructions. While the ARM implementations come with no proof of correctness or rigorous testing, the fence placement is conservative and no issues were encountered. These implementations should be seen as temporary until OpenCL 2.0 is more widely supported.

2.5 Summary

GPUs began as dedicated hardware to enable graphics rendering. Through the years, their features and computational flexibility increased to the point of being able to execute general massively parallel C-like programs. Although GPUs from different vendors have different architectural parameters, programming models such as OpenCL allow GPU programs to be written in a portable manner. These programming models evolve and adapt to support new features; this thesis explores a currently unsupported idiom: the inter-workgroup barrier.

3 A Portable Global Barrier

3.1 Personal context

I undertook this work at the beginning of my PhD. The original aim was to continue the work of my master's thesis about testing weak memory models on GPUs [Sor14]. In these memory model tests, a kernel consisted of a set-up phase, where memory locations were distributed to threads in various ways, followed by the actual memory model test (or litmus test). One of the heuristics that made the tests more effective was using a global barrier between the set-up phase and the litmus test; this temporally aligned the threads, causing more interesting behaviours to be observed.

In the earlier work, our focus was restricted to Nvidia GPUs. I was interested in making this work portable across GPUs from different vendors. The immediate difficulty was making the global barrier work on different GPUs. While Nvidia provides a convenient occupancy calculator, this is not available for any other vendor. After fighting with many deadlocks and seeing no clear path to a simple and portable occupancy calculator, it became clear that a portable GPU global barrier was interesting in its own right. This chapter presents a solution to providing a portable global barrier for GPUs.

3.2 Motivation

Execution barrier synchronisation, where a thread waits at a barrier until all threads reach the barrier, is a popular method for inter-thread communication because (1) it simultaneously aligns the computations of threads, and (2) it makes pre-barrier memory updates available to all threads, implementing a well-specified and simple memory consistency model.

Unfortunately, existing GPUs and their associated programming models do not support inter-workgroup barriers. This is because a GPU kernel can be configured with many more workgroups than the underlying hardware can execute concurrently. Execution of large numbers of workgroups is achieved in an "occupancy-bound" fashion, by delaying the scheduling of some workgroups until others have executed to completion. This is pragmatic design decision made by vendors, as traditional GPU kernels have not contained blocking synchronisation idioms, and preemption at the workgroup level would be difficult to achieve efficiently. Namely, the large local memory and register files would need to rapidly be stashed and restored. As a consequence, traditional fair scheduling guarantees associated with CPU threads are *not* provided between workgroups executing a kernel [GSO12].

As an example, consider the kernel in Figure 3.1 executed with one thread per workgroup. This kernel accepts a pointer to a global variable, `flag` (initialised to 0), and two (not equal) integers, `A` and `B`. The thread with workgroup id `B` writes `1` to `flag`, while the thread with workgroup id `A` spins waiting for workgroup `B` to write to `flag`. However, the GPU execution model is licensed to postpone execution of workgroup `B` until execution of workgroup `A` has completed. In this scenario, the kernel will deadlock due to starvation.

This idiom of one workgroup waiting on another is at the heart of an inter-workgroup barrier. When executing a barrier instance, each workgroup will wait on all other workgroups to reach the same barrier instance. If a single workgroup is blocked by the occupancy-bound execution model, then the barrier execution will deadlock.

This behaviour is easy to observe on current GPUs. A traditional CPU software barrier [HS08, ch. 17], ported to synchronise across workgroups in OpenCL, leads to deadlock on an AMD Radeon R7 GPU when used in a kernel launched with more than 4096 threads (with 256 threads per workgroup). The deadlock occurs because the number of threads exceeds the *occupancy* of the GPU for the kernel realisation, i.e. the number of workgroups that can concurrently execute the binary version of the kernel on the GPU.

Current approaches Existing GPU applications that require inter-workgroup barrier synchronisation rely on either the *multi-kernel* or the *occupancy assumption* method.

In the multi-kernel method, an application is manually split into multiple kernels, with a transition from one kernel to another each time an inter-workgroup barrier is required.

```
kernel void unsafe(global int *flag, int A, int B) {
 if (get_group_id(0) == A)
  while(*flag != 1);

 if (get_group_id(0) == B)
  *flag = 1;
}
```

Figure 3.1: A non-portable kernel that potentially deadlocks.

The transfer of control from the GPU to the CPU host between kernels provides the barrier semantics implicitly [Khr15, ch. 3.2.4]. This method is portable, making no assumptions about concurrent execution of workgroups. However, there are three immediate drawbacks to this method:

1. interaction between the GPU and CPU can be expensive, due to the overhead associated with kernel launches. Indeed, Chapter 4 measures the kernel launch overhead for many of the chips of Table 2.2 and shows that it is a significant bottleneck in many applications;

2. because the contents of registers and local memory do not persist across kernel calls, in the multi-kernel method these local resources cannot be reused across kernel calls;

3. from a software engineering standpoint, requiring multiple kernel launches may not be the most natural or maintainable structure for the code.

The alternative *occupancy assumption* approach (studied in related work [GSO12, XF10]), employs a traditional software execution barrier. Starvation, and hence deadlock, is avoided through *a priori* knowledge about the number of workgroups that can be scheduled concurrently for a particular kernel and GPU. This avoids the efficiency problems of the multi-kernel approach, but is *non-portable*, since workgroup occupancy varies dramatically between (1) GPU architectures, depending on hardware resources, e.g. the number of compute units, and (2) GPU kernels, depending on the resources that a workgroup requires to execute the kernel, e.g. registers and local memory. The kernel resources requirements are in part determined by the compiler, and thus may vary even between compiler versions.

In a similar vein, OpenCL provides *nested parallelism* [Khr15, pp. 32–33] (or *dynamic parallelism* in CUDA [Nvi18a, app. D]) where GPU threads can themselves launch a new kernel. While this feature may be useful for applications with irregular parallelism examined in this work, the high level idiom of nested parallelism, i.e. fork and join, is different enough from barrier synchronisation, i.e. synchronisation of executing threads, that it is not considered further in this chapter.

▶ **Remark** (Performance of nested parallelism). In addition to the different concurrency models of nested parallelism and global synchronisation, prior work by Pai and Pingali showed that the performance of nested parallelism on modern Nvidia GPUs is significantly lower than global synchronisation. Thus, global synchronisation appears to be the more pragmatic direction of research [PP16, Table 1].

Memory consistency An inter-workgroup barrier must also provide memory ordering properties: threads must observe up-to-date memory values during post barrier execution, and data-races between accesses separated by the barrier must be forbidden. The barrier must be implemented using sufficient synchronisation constructs, such as atomic operations and memory fences, to ensure these properties.

3.2.1 Chapter contributions

The overall contribution of this chapter is to show that portable inter-workgroup barrier synchronisation can be successfully achieved, if both the execution model *and* the memory consistency model are considered.

The heart of the contribution is an *occupancy discovery* protocol that provides a (safe) estimate of the number of occupant workgroups dynamically at the beginning of a kernel execution. The protocol can then be used to set up an execution environment such that the remainder of the kernel is only executed by the workgroups found to be occupant. Because the number of workgroups that execute the kernel is dynamic, depending on the discovered occupancy, this execution environment requires that kernels are agnostic to the number of executing workgroups. This is discussed further in Section 3.4.

Because occupant workgroups exhibit traditional fair scheduling guarantees, they can reliably participate in an inter-workgroup barrier. In this context, Section 3.5 describes the extension of an existing GPU barrier implementation from [XF10] to use the atomic operations of OpenCL 2.0. The memory ordering properties of these instructions are well-defined and allow for the barrier implementation to be formally reasoned about. In particular, the analysis includes (1) a formal specification of the memory ordering properties to be provided by an abstract inter-workgroup barrier, and (2) an illustration that the concrete barrier implementation honours the specification.

To assess portability, the discovery protocol and global barrier are evaluated across eight GPUs spanning four vendors (discussed in Section 2.4). First, the recall (i.e. the percentage of instances found) of the occupancy estimate returned by the discovery protocol is assessed. Using heuristics, it is shown that the recall is almost 100%, i.e. the occupancy estimate almost perfectly matches the occupancy bound. The Pannotia [CBRS13] and Lonestar-GPU [BNP12] benchmarks are then examined in Sections 3.6.2 and 3.6.3. These benchmarks currently achieve inter-workgroup synchronisation using the multi-kernel and occupancy assumption methods, respectively. All relevant applications from each suite are adapted to use the protocol/barrier combination and the runtimes from the adapted applications are compared with the original implementations. In all cases, the discovery protocol barrier combination enables portable execution as expected.

The performance effect of moving from multi-kernel to global barrier synchronisation varies between GPU and application. However, a reliable speedup is observed in some cases (e.g. a geomean speedup of 1.36× is observed on Iris). On the other hand, the slowdown associated with moving from an occupancy assumption barrier to a portable barrier is shown to be reasonable. This allows portable versions of the Lonestar-GPU applications to run (for the first time) on GPUs produced by vendors other than Nvidia.

To summarise, the main contributions of this chapter are:

- The presentation of the *occupancy discovery protocol*, which dynamically computes a safe estimate of the workgroup occupancy for a given GPU and kernel (Section 3.4).
- The adaptation of an existing inter-workgroup barrier [XF10] to exploit the dynamic workgroup occupancy discovered by the discovery protocol. Using OpenCL 2.0 atomic operations, it is shown that the barrier meets its intuitive memory ordering specification (Section 3.5).
- An evaluation of the discovery protocol on eight GPUs spanning four vendors (Table 2.2) showing that the approach is able to achieve near-perfect occupancy estimates (Section 3.6.1).
- An evaluation of the performance benefits of using the protocol/barrier combination against the multi-kernel and occupancy assumption methods for inter-workgroup synchronisation. Results vary across GPUs and applications, but in some cases the protocol/barrier application variants provide a substantial speedup, up to 2.34× (Sections 3.6.2 and 3.6.3).

Related publications The material presented in this chapter is based on work published in the 2016 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16) [SDB$^+$16]. The work has an associated approved artifact, which can be found at:

`https://github.com/mc-imperial/gpu_discovery_barrier`

A companion experience-report, detailing the difficulties of running the experimental campaign, was published in the International Workshop on OpenCL (IWOCL'16) [SD16b].

3.3 Occupancy-bound execution model

OpenCL does not currently specify a formal execution model for inter-workgroup interactions, and the behaviour of programs with such interactions is cautioned against in the standard [Khr15, p. 31]: *"A conforming implementation may choose to serialize the workgroups so a correct algorithm cannot assume that workgroups will execute in parallel. There is no safe and portable way to synchronize across the independent execution of workgroups since once in the work-pool, they can execute in any order."*

In previous work [GSO12, XF10], it is suggested, and supported by empirical evidence, that GPUs provide an *occupancy-bound* execution model for inter-workgroup interactions. Here, a more formal definition of the occupancy-bound execution model is given. This definition is limited to the execution of a single kernel and also assumes that non-compute functionality, e.g. graphics, is disabled by kernel execution. This assumption is consistent with empirical observations during this work, e.g. the OS graphics layer became unresponsive during kernel execution.

▶ **Looking forward:** In Chapter 5, co-scheduling between interactive tasks, e.g. graphics, and compute tasks on the same GPU is explored through a small set of new programming primitives for OpenCL.

A workgroup is *occupant* if (1) it has executed at least one instruction using the GPU's resources, and (2) it has not yet completed kernel execution. The occupancy-bound execution model requires the following conditions, related to occupant workgroups, to hold:

- **No indefinite preemption**: An occupant workgroup is guaranteed to eventually be scheduled for further execution on the GPU, regardless of the behaviour of other workgroups. Consequently, two workgroups that are simultaneously occupant can participate in blocking communications with each other that require fair scheduling, e.g. spin-locks.

- **Utilisation**: There is a constant $N > 0$, the *occupancy bound*, such that if $m < N$ workgroups are occupant, and there exist $k > 0$ workgroups that have not yet commenced execution, then one of these k workgroups will eventually become occupant. Additionally, no more than N workgroups can be occupant. Consequently, a blocking communication that requires up to N workgroups to be simultaneously occupant (but makes no assumptions about the order in which they become occupant) can be used without fear of starvation-induced deadlock.

```
kernel void native_openCL(...) {
  // A thread will do at most one element of work based on global id
  int id = get_global_id();
  if (id < data_size) {
    do_work(id)
  }
}
```

(a)

```
kernel void persistent_thread(...) {
  // A thread may do multiple elements of work based on the
  // number of participating groups
  for (int id = get_global_id(); id < data_size;
       id += get_global_size()) {
            do_work(id)
  }
}
```

(b)

Figure 3.2: Illustration of code transformation required when going from (a) the native OpenCL programming style to the (b) persistent thread programming style where kernels are parameterised by N, the number of executing workgroups .

The occupancy bound N depends on the available hardware resources. This clearly depends on the GPU architecture, but also the resource requirements of the target kernel. A kernel that uses a large amount of local memory or registers will have a high resource requirement, which may lower N. Because the resources used by a kernel may depend on details of compilation, e.g. register allocation, N also depends on the OpenCL compiler version.

The persistent thread model The persistent thread model is a GPU programming model that allows kernels to exploit fair scheduling guarantees provided by the occupancy-bound execution model. To use the persistent thread model, programmers must ensure that they launch kernels with at most N workgroups, sometimes referred to as the *maximal launch* [GSO12]. Under this restriction, all workgroups will eventually be occupant, so idioms that require fair scheduling across workgroups, e.g. inter-workgroup barriers, can be used. Applications that use the persistent thread programming model currently use occupancy assumption methods to determine N. A main contribution of this chapter is a method for programmatically determining a safe estimate for N.

One property of the persistent thread programming model is that programs are written in a way that is agnostic to the number of executing workgroups. This is because N can vary across chips or even compiler settings, and persistent thread programs aim to be portable as long as N is provided. The transformation of traditional OpenCL programs into a persistent thread style program is typically straightforward. An example is illustrated in Figure 3.2. The code of Figure 3.2a illustrates the native OpenCL programming model, where threads compute at most one item of work depending on their id. The code of Figure 3.2b shows how the code of Figure 3.2a is adapted to use the persistent thread programming style. That is, each thread performs a dynamic amount of computation based on the number of the global size, which is determined by N. Throughout the rest of this chapter, it is assumed that programs are given in the persistent thread programming style, and thus are agnostic to the number of executing workgroups.

Occupancy bound One empirical validation of the occupancy-bound execution model is the existence of an occupancy bound N such that an inter-workgroup barrier succeeds when executed with N workgroups but deadlocks when executed with $N + 1$ workgroups. Such occupancy bounds are found across a range of GPUs in Section 3.6.1. Many instances of previous work, which exploit the persistent thread model, also acknowledge this bound [WCL$^+$15, GSO12, TPO10, CT08, XF10, HHB$^+$14, OCY$^+$15, MZ16, MYB16, BNP12], adding to the empirical evidence that this execution model is supported on current GPUs.

3.4 Occupancy discovery

Here, the *occupancy discovery* protocol for dynamically determining a safe estimate of the occupancy bound for a given GPU and kernel is presented. Recall that the occupancy-bound execution model guarantees existence of an occupancy bound N for a given GPU and kernel such that N workgroups can be simultaneously occupant during execution, and are guaranteed to be fairly scheduled. Given that N is unknown, the aim is to dynamically discover an estimate n with $0 < n \leq N$, i.e. a lower bound for N. The n discovered workgroups may then proceed to successfully complete computation that requires forward progress between workgroups using a persistent thread style programming model.

To achieve this, all workgroups execute the discovery protocol at the start of the kernel, prior to any other computation. In the protocol, each workgroup executes a routine that returns a value indicating whether the calling workgroup is *participating.* A workgroup is participating if and only if the workgroup commences execution of the protocol before any

```
lock(mutex);
if (poll_open){
 M[get_group_id()] = count;
 count++;
 unlock(mutex);                        polling phase
} else {
 unlock(mutex);
 return NON_PARTICIPATING;
}

lock(mutex);
if (poll_open) {
 poll_open = false;               closing phase
}
unlock(mutex);
return PARTICIPATING;
```

Figure 3.3: Occupancy discovery protocol, executed by a single representative thread per workgroup.

workgroup has finished executing the protocol. By definition, participating workgroups are simultaneously occupant. The occupancy-bound execution model thus guarantees that participating workgroups are fairly scheduled, and the total number of participating workgroups is a lower bound for N.

Workgroups found to be non-participating immediately exit; workgroups found to be participating continue the kernel computation. Thus computation occurring after the discovery protocol, called the *main* kernel computation, can assume fair scheduling of workgroups.

Under this scheme, the native workitem built-in functions (see Section 2.3) may no longer provide contiguous unique values for threads executing the main kernel computation. For example, if the discovered participating workgroups do not have contiguous native ids then they cannot easily partition data in data-parallel programs. To overcome this, the occupancy discovery protocol constructs a new, dynamically determined, execution environment with replacements for the workitem built-in functions that do satisfy the contiguous unique property for participating workgroups.

3.4.1 Implementation

Figure 3.3 shows the discovery protocol implementation. There are four variables located in the global memory region: `count`, an integer to record the number of participating groups (initially 0); `poll_open`, a boolean recording whether the poll is open (initially true); mutex, an object that provides mutual exclusion through lock and unlock functions (initially unlocked); and M, an array that records intermediate values of count (initial values irrelevant). All protocol variables are required to be initialised prior to the protocol execution, e.g. by a separate kernel or via OpenCL host-to-device memory copies.

The protocol is split into two phases, a *polling* phase (lines 1-8) and *closing* phase (lines 11-16). Both phases are protected using mutex. For now, the implementation of mutex along with the corresponding locking functions are left abstract, only supplying mutual exclusion in locked regions. Concrete mutex implementations are discussed in Section 3.6.1. The protocol is executed by only one representative thread per workgroup.

In the polling phase, a thread checks whether the poll is open (line 2). If so, the thread marks its workgroup as *participating* by recording the current value of count in array M, at an index according to the thread's workgroup id, and incrementing count (lines 3-4); the thread then moves on to the closing phase. On the other hand, if the poll is closed, the thread simply exits the protocol, returning a flag value to indicate that the its workgroup is non-participating (line 8).

A thread that successfully completes the polling phase, i.e. observed an open poll, continues to the closing phase (line 11). If the poll is still open, the thread closes the poll (lines 12-13). Only the first thread to enter the closing phase performs this action; future threads observe a closed poll. Either way, the thread returns a participating flag (line 16).

Because thread scheduling is non-deterministic, a single thread might execute the polling phase *and* closing phase prior to any other thread commencing execution of the protocol. This would lead to an estimated occupancy bound of 1. Clearly it would be preferable to discover a tighter lower bound, especially if the true occupancy bound N is large. Experimentally it is shown that a suitable choice of mutex implementation leads to tight estimates of N in practice (Section 5.6).

It may be possible to add heuristics to this protocol to improve the recall of discovered workgroups. For example, it seems natural that a pause inserted between the polling and closing phase may increase the recall as this would provide occupant threads more time to poll before the poll is closed. In this work, such heuristics are not considered for two reasons: (1) it is shown that with the right mutex implementation, a very high recall is achieved without any additional heuristics (Section 5.6), and (2) because OpenCL does not provide any portable pause or sleep function, the pause would have to be implemented as some kind of busy spin. The busy spin would have to be tuned per GPU in order to minimise overhead while maximising recall. In Section 3.6.4, pilot experiments applying the occupancy discovery protocol to CPU implementations of OpenCL are discussed, in which this spinning heuristic is employed.

Table 3.1: Occupancy discovery execution environment functions.

Function	**Derivation**
`p_get_num_groups()`	`count`
`p_get_group_id()`	`M[get_group_id()]`

`p_get_global_id()`	`(p_get_group_id() * get_local_size())`
	`+ get_local_id()`

`p_get_global_size()`	`count * get_local_size()`

Execution environment construction The variables `M` and `count` are used to construct a new execution environment in which only participating workgroups take part in computation. Because `count` is initialised to zero and incremented once by each participating workgroup, after the protocol execution `count` contains the number of participating workgroups. Additionally, the value of `count` observed by a participating workgroup prior to incrementing (line 3) is recorded in array `M` at an index corresponding to the id of the workgroup, providing a unique *participating workgroup id*, such that the sequence of participating workgroup ids is contiguous and unique.

Table 3.1 defines the new execution environment functions. A thread can query: the number of participating workgroups (`p_get_num_groups`), its participating workgroup id (`p_get_group_id`), a contiguous unique id for participating threads (`p_get_global_id`), and the total number of participating threads (`p_get_global_size`). Each function is analogous to a native OpenCL function (the function without the `p_` prefix), but the new functions consider only participating workgroups.

Execution environment constraints As participating workgroups are determined dynamically, the main kernel computation must be agnostic to the number of executing workgroups at launch time (i.e. as in the persistent thread programming model). This is in contrast to the native OpenCL execution environment, where the number of executing workgroups is fixed on kernel launch.

In the native OpenCL execution environment a kernel can be launched with enough workgroups such that each thread computes at most one piece of data. A kernel that uses the occupancy discovery execution environment must be able to dynamically adapt the per-thread computation based on the number of participating groups, which can be queried during kernel execution. The kernels adapted to use occupancy discovery in the experiments (Section 5.6) required only simple transformations to satisfy this constraint, which is similar to the program transformations required for the persistent thread model [GSO12].

In fact, the persistent thread program of Figure 3.2b is easily adapted to the participating thread model by: (1) executing the discovery protocol before any kernel computation; and (2) replacing `get_global_id()` and `get_global_size()` with their participating group mirrors: `p_get_global_id()` and `p_get_global_size()`, respectively.

3.4.2 Properties and correctness

Here, it is argued that the occupancy discovery protocol satisfies certain key properties required by client applications. Line numbers refer to Figure 3.3.

At least one participating workgroup To ensure that main kernel computation occurs, at least one workgroup must be identified as participating.

The protocol satisfies this requirement because the poll is initialised to open. This means that at least the first thread to enter the polling phase will mark itself as participating and go on to finish the protocol, returning `PARTICIPATING` at line 16.

Consistent participating count To provide participating threads with valid execution environment values, the protocol must ensure that, upon completion, all threads view the same value in `count` and that this value is equal to the number of workgroups that return `PARTICIPATING`.

Every thread returning `PARTICIPATING` increments `count` exactly once (in a critical section), as there are no loops in the protocol and the only path to returning `PARTICIPATING` (line 16) requires incrementing `count` (line 4). Furthermore, the value in `count` does not change once any thread returns from the protocol. There are two possible return points: participating (line 16) and non-participating (line 8). At both return points, the poll must be closed. In the case of a participating return, the poll has either been closed by the returning thread or by an earlier thread (lines 12 and 13). In the non-participating case, the poll was observed to be closed (line 2). Once the poll is closed, it remains closed (there is no place in the protocol that re-opens the poll). If the poll is closed, `count` cannot be modified.

Because all threads that return `PARTICIPATING` increment `count` once, and `count` does not change after a thread returns, all threads that return `PARTICIPATING` must observe `count` to contain the total number of threads that ultimately return `PARTICIPATING`.

Participating workgroups are simultaneously occupant The main purpose of the protocol is to create an execution environment that guarantees fair scheduling between workgroups, under the assumption of the occupancy-bound execution model. It is now argued that workgroups identified as participating are indeed simultaneously occupant.

This property is shown by counterexample. Let $\mathbb{P}$ be the set of workgroups that return `PARTICIPATING`. Because simultaneous occupancy concerns multiple workgroups, suppose that $\mathbb{P}$ contains at least two workgroups. Assume that there exist workgroups $w, v \in \mathbb{P}$ that are *not* simultaneously occupant. Without loss of generality, assume that w finishes execution before v starts execution.

In order for w to finish execution, w must have executed the discovery protocol, returning `PARTICIPATING` (line 16). In order for w to have reached this line, the poll must be closed, either by w or a different participating workgroup (lines 12 and 13). Now v eventually starts execution and begins executing the discovery protocol. However, because w has finished execution (and consequently the poll is closed), v must observe a closed poll at line 2. Thus, v must return `NON_PARTICIPATING` (line 8). Therefore v is not a participating group, a contradiction.

▶ **Looking forward:** Notice that the definition of the occupancy-bound execution model does not consider thread or workgroup ids. In Chapter 6, a variant of the occupancy-bound execution model is explored where thread ids are considered. In particular, workgroups are scheduled in their id order. This new execution model enables an optimised version of the discovery protocol which allows the native workgroup id and global id functions to be used.

3.5 Inter-workgroup barrier

In this section, a global barrier that can be used to synchronise participating workgroups is described. First, an overview of the global barrier presented by Xiao and Feng [XF10] is given (abbreviated as the XF barrier from here on), as this barrier is used as a basis for the implementation in this work. Then, two changes to the XF barrier are presented, which allow the barrier to claim additional portability and correctness properties: (1) the addition of atomic instructions necessary for formal memory ordering properties and data race freedom, and (2) using an execution environment that ensures fair scheduling between workgroups.

```
1  if (get_local_id() + 1 < get_num_groups()) {
2   while (!flag[get_local_id() + 1]);
3  }
4
5  barrier();
6
7  if (get_local_id() + 1 < get_num_groups()) {
8   flag[get_local_id() + 1] = 0;
9  }
```

(a)

```
1  barrier();
2
3  if (get_local_id() == 0) {
4   flag[get_group_id()] = 1;
5   while (flag[get_group_id()] == 1);
6  }
7
8  barrier();
```

(b)

Figure 3.4: XF inter-workgroup execution barrier, one master workgroup executes (a), while all other workgroups execute (b).

3.5.1 The XF barrier

Figure 3.4 illustrates a variant of the XF software execution barrier [XF10], a GPU inter-workgroup barrier provided in the CUB CUDA library [Nvi]. Originally implemented in Nvidia's CUDA language, the XF barrier has been shown to offer high performance, exhibiting a low level of memory contention and avoiding read-modify-write instructions. The variant shown is ported to OpenCL and removes a redundant intra-workgroup barrier.

The XF barrier uses a *master/slave* model, where one workgroup is selected to be the master, executing the code of Figure 3.4a, and the remaining workgroups are slaves, executing the code of Figure 3.4b. Function `barrier()` denotes an intra-workgroup barrier operation.

The slave workgroups start with an intra-workgroup barrier, to ensure that all threads in the local workgroup have arrived at the inter-workgroup barrier (slave line 1). A representative thread in the workgroup (with local id `0`) writes 1 to the workgroup's index in `flag`, an array of flags at least as large as `get_num_groups()`, to indicate that the workgroup has arrived at the inter-workgroup barrier (slave line 4). The representative thread then spins (slave line 5), waiting for the master workgroup to release the barrier.

The remaining threads in the workgroup wait at the final workgroup barrier instruction (slave line 8) for the representative.

Each thread in the master workgroup takes responsibility for managing one slave workgroup: the workgroup with group id equal to the thread's local id plus one; one is added to the local id because the group with id 0 is the master workgroup and does not need to be managed. Each master thread spins until the workgroup it is managing has arrived at the barrier (master line 2). The master workgroup then performs a workgroup barrier (master line 5). Given that master threads manage all other (slave) workgroups, the completion of this workgroup barrier denotes that all threads across all workgroups have arrived at the barrier. Each master thread now releases the workgroup it is managing by setting that workgroup's flag to 0 (line 8).

The code of Figure 3.4 assumes that there are at least as many threads per workgroup as there are workgroups, but is easily adapted to cater for a larger number of workgroups by having each thread in the master workgroup manipulate the flags of more than one workgroup.

The XF barrier fails to directly provide *portable* inter-workgroup synchronisation for two reasons. First, because progress between workgroups is not guaranteed, the barrier is prone to deadlock due to starvation. Running the XF barrier on the Chapter 3 chips of Table 2.2 with 1024 workgroups (each with 256 threads) causes deadlock for *every* GPU studied in this chapter. Reducing the number of workgroups to 128 results in deadlock for all chips except GTX980 and R9. Reducing the workgroup count to 2 avoids deadlock in all cases. The XF barrier was originally evaluated on GPUs where the number of concurrently executing workgroups was known *a priori* so that deadlock could be avoided [XF10]. Secondly, as shown in prior work by the thesis author, some GPUs have been shown to have relaxed memory models, in which memory operations may appear to execute out of order [SD16a, ABD$^+$15]. The original XF barrier implementation, presented in CUDA (which lacks a rigorous memory model) does not formally take account of memory ordering properties.

3.5.2 OpenCL 2.0 memory model primer

To ensure that the barrier induces sufficient synchronisation, the memory consistency model must be considered. OpenCL 2.0 provides a formal memory model that allows rigorous reasoning about inter-thread communication guarantees. Because this is a language level memory model, any device that correctly supports the OpenCL 2.0 memory

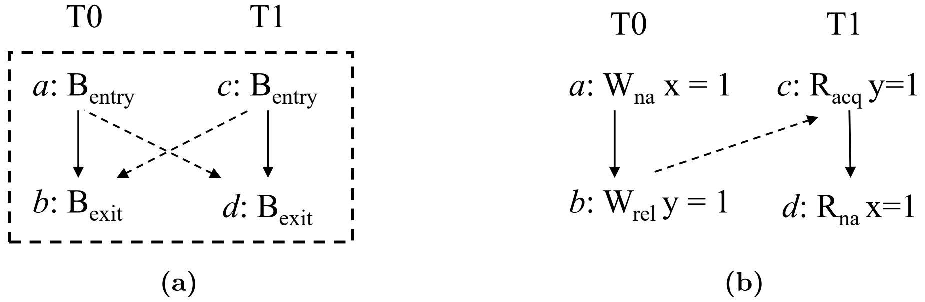


Figure 3.5: Two OpenCL synchronisation patterns used by the XF inter-workgroup barrier.

model (i.e. by compiling the language constructs to assembly level fences) will enjoy the synchronisation properties shown here.

The OpenCL memory model defines the behaviour of concurrent memory accesses, including the atomic accesses and workgroup barriers used in the inter-workgroup barrier. The memory model is *axiomatic*: program behaviours are represented as sets of executions, each a graph of the memory events of one path of control flow with relations representing ordering in the execution.

The memory model has two phases. The first finds *consistent executions* by filtering prospective program executions: imposing a set of constraints on *happens-before*, hb, a relation on events that collects together thread-local program order and inter-thread synchronisation. The second looks for *data-races* in the consistent executions, defined as an absence of happens-before between conflicting non-atomic accesses to a single variable. If even a single race exists in a single consistent execution, the entire program is given undefined behaviour. Otherwise, the consistent executions represent the program's behaviour.

The global barrier implementation relies on happens-before edges created between threads as demonstrated by the dashed arrows in the two consistent execution shapes of Figure 3.5.

Figure 3.5a presents an execution with intra-workgroup barrier synchronisation. Intuitively, a given barrier call gives rise to barrier entry (B_{entry}) and exit (B_{entry}) events that span the threads of a workgroup. A barrier entry and exit event on the same thread are ordered by *program order*. Program order induces happens-before between events on the same thread, drawn as solid vertical arrows. For each intra-workgroup barrier call, its associated events are collected into a barrier instance, signified by the surrounding dashed box. Each barrier entry synchronises with every exit in the same instance. These synchronisation edges are called a *barrier web*.

In the example of Figure 3.5b, called *message passing*, thread T0 writes to x with a non-atomic write (W_{na}) and then writes to y with an atomic write annotated as a release at the device-scope (W_{rel}). Thread T1 then reads from y using an atomic read annotated as an acquire at device-scope (R_{acq}) and then reads from x with a non-atomic read (R_{na}). In the absence of synchronisation, the non-atomic accesses to x would form a race. However, for threads in either the same or different workgroups, a device-scoped acquire read such as c that reads from a device-scoped release write such as b, results in the creation of a happens-before edge, drawn as a dashed arrow in Figure 3.5b. Happens-before is transitive, and the a-to-d edge avoids a data race on x, and forces d to read from a.

3.5.3 Implementation

Recall the original XF barrier implementation shown in Figure 3.4. As discussed in Section 3.5.1, the XF barrier has an *arrival* phase, where the master waits for every slave to announce its presence at the barrier, and a *departure* phase where the master releases the slaves from the barrier. Concurrent non-atomic accesses to the flag array would lead to data races in OpenCL. As a consequence, the XF barrier, as presented, has undefined behaviour. Moreover, the purpose of each phase is to *synchronise*, first from the slave threads to the master, and then from the master to the slave threads. Non-atomic accesses do not provide this sort of synchronisation.

To realise the intended behaviour of the XF barrier in OpenCL 2.0, the original implementation is augmented with atomic instructions. Specifically, `flag` is declared as an array of atomic integers and all accesses are annotated as release and acquire device-scoped atomics, avoiding races and inducing synchronisation. More precisely, slave line 4 and master line 8 become device-scoped release stores, and master line 2 and slave line 5 become device-scoped acquire loads.

Because the barrier covers only participating groups, its thread id functions are replaced to use the execution environment provided by the discovery protocol (see Table 3.1): `get_num_groups` becomes `p_get_num_groups` (master lines 1 and 7), returning the number of participating groups, and `get_group_id` becomes `p_get_group_id` (slave lines 4 and 5).

3.5.4 Barrier specification

Here, a generic barrier specification, parameterised by an OpenCL scope, is presented and integrated with the formal OpenCL memory model of Batty et al. [BDW16]. The OpenCL workgroup barrier primitive behaves according to the specification instantiated with workgroup scope, closely following OpenCL [Khr15, p. 53], whereas the inter-workgroup barrier behaves according to the specification instantiated with device scope. This symmetry suggests that the device-scoped barrier specification should be natural to OpenCL programmers familiar with the workgroup barrier.

OpenCL programs are required to be free from *barrier divergence* [Khr16a, p. 99]. Barrier divergence occurs when either (a) two workitems in the same workgroup reach syntactically distinct intra-workgroup barrier statements, or (b) an intra-workgroup barrier statement appears in a nest of loops, and two workitems reach the barrier statement having executed different numbers of iterations for at least one enclosing loop. See Collingbourne et al. for a formal definition [CDKQ13].

Kernels that exhibit barrier divergence have undefined behaviour, thus the focus of the approach is restricted to kernels that are divergence-free. It is assumed that two properties follow from barrier divergence-freedom: all barrier instances cover all threads at the workgroup scope and all participating threads at device scope, and no barrier instance links barrier events from within the XF barrier to those outside.

The specification updated with the global barrier provides two additions to the formal model of Batty et al.: new machinery to generate prospective executions with barrier events from programs, and new happens-before edges in the memory model.

Memory-model barrier specification The execution of a dynamic instance of a barrier gives rise to two program-ordered events on each thread in its scope (workgroup or device): a barrier entry event (B_{entry}) followed by a barrier exit event (B_{exit}). For all threads executing the barrier instance, the B_{entry} of one thread happens-before the B_{exit} of all other threads. These new hb edges arising from a single barrier instance are precisely the edges in the barrier web of Figure 3.5a. The dashed box surrounding the web is called an *instance-box* — in future drawings, the web is elided and only the box is drawn. Intuitively, the web ensures that any access preceding a barrier happens-before any access following any barrier in the same instance, avoiding races between prior and following accesses.

3.5.5 Correctness of the inter-workgroup barrier

Here it is argued that the global barrier implementation meets its specification in terms of synchronisation. A more formal proof following the *library abstraction* method of Batty et al. [BDG13], is given in the publication corresponding to this chapter [SDB$^+$16].

Progress guarantee Correctness of the barrier relies on the progress guarantees assumed from the occupancy-bound execution model and the participating group execution environment produced by the discovery protocol. Threads in the barrier implementation use spin loops to wait for writes by other threads (master line 2 and slave line 5 in Figure 3.4). If they were to repeatedly read from older writes, the barrier would hang. To avoid this, an infinite sequence of happens-before-ordered reads failing to see a write from another thread is prohibited.

Abstraction of the inter-workgroup barrier Considering the augmented XF barrier as the implementation and the abstract device-level barrier, described in Section 3.5.4, as the specification, it is now argued that the implementation induces the synchronisation required by the specification.

First, the implementation is shown to be free from data-races. This is straightforward as the augmented XF barrier contains only device-scoped atomic operations when accessing inter-workgroup shared memory. By definition, these accesses cannot produce a data-race.

Second, the synchronisation induced by the barrier is shown to match the specification. The crux of this argument involves looking at a single barrier call across all threads. There are four cases to be considered for call-return pairs: same-workgroup, master-slave, slave-master, and slave-slave, where in the latter three cases the master and slave are in different workgroups. In the specification, the barrier web covers all of the threads belonging to participating workgroups identified by the discovery protocol, and creates synchronisation between every pair of threads. For each of the four cases, it must be shown that the augmented XF barrier replicates this synchronisation.

The *axiomatisation* of the events that make up an execution of the barrier across three workgroups (simplified with only two threads each) is shown in Figure 3.6: this event graphs coincides with the various paths of control flow through the XF barrier. The spin-loops and intra-workgroup barriers constrain the execution such that there is essentially only one execution allowed. Other allowed executions would simply include failed read events at the spin-loop locations.

It is left to show that the implementation produces synchronisation in each of the four cases. This is argued by identifying synchronisation of the two varieties presented in Figure 3.5 in the execution of the augmented XF barrier, and recalling that happens-before is transitive. Thus, showing that there is a `hb` path from the top of a thread in Figure 3.6 to the bottom of another thread would establish a barrier-web `hb` between the two threads. The four cases are:

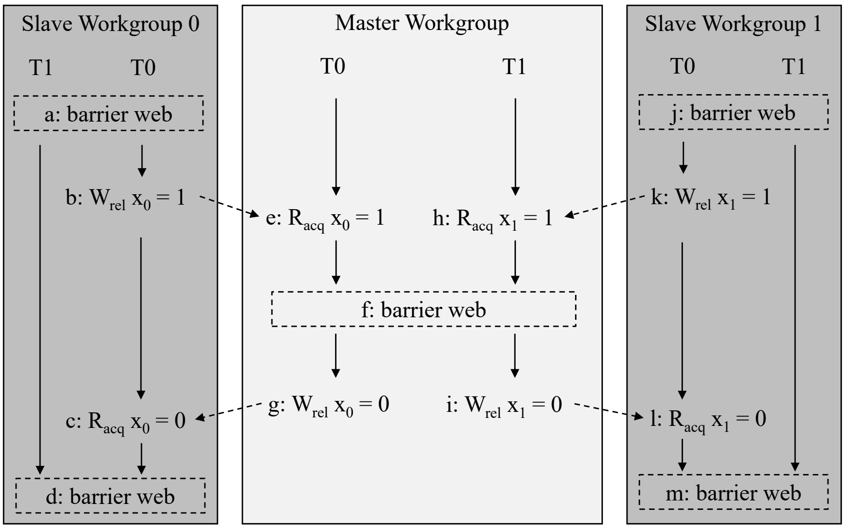


Figure 3.6: Idiomatic execution of XF barrier.

- *same-workgroup*: whether the call and return reside on the master workgroup or a slave, in Figure 3.6 there is an instance box across the workgroup. The happens-before edge is ensured by the barrier web of a workgroup-scoped barrier as in Figure 3.5a.

- *slave-master*: as the master breaks out of its loop, it must read the slave's write of the flag array. The device-level release and acquire synchronise as in Figure 3.5b, creating a happens-before edge from one thread on the slave to one on the master. In Figure 3.6, the preceding barrier on the slave, and the following one on the master then complete the happens-before edge between the slave's call and the master's return.

- *master-slave*: similarly the slave breaks out of its loop by the reading of the master's write creating synchronisation that is extended by the barriers on the master and slave.

- *slave-slave*: in this case, a slave synchronises with the master, and then the master synchronises with a slave. The prior, intervening, and following barriers complete the call-to-return happens-before edge.

3.6 Empirical evaluation

Here, an empirical evaluation of the methods described in this chapter is presented across the Chapter 3 GPUs of Table 2.2. First, the results of microbenchmarks measuring the recall of the occupancy estimate provided by the discovery protocol with respect to the occupancy bound are shown (Section 3.6.1). Then several benchmarks that use the *multi-kernel* method for inter-workgroup synchronisation are examined and adapted to use the discovery protocol/barrier combination. The runtime of the two approaches are then compared (Section 3.6.2). In a similar vein, applications written using the non-portable *occupancy assumption* approach are examined and adapted to be *portable* using the discovery protocol. The overhead caused by the discovery protocol and associated execution environment are presented (Section 3.6.3). Finally, pilot experiments where the protocol is tested on CPU implementations of OpenCL are reported on (Section 3.6.4).

To preface these experimental results, it is noted that even though the occupancy-bound execution model is not officially endorsed by OpenCL, the experiments indicate that the model is supported by all the GPUs evaluated in this chapter (Table 2.2). Specifically, the use of the inter-workgroup barrier never led to a deadlock and a concrete value N for the occupancy bound was always observable. It appears that the occupancy-bound execution model is a useful *de facto* property of devices that support OpenCL, so we believe there is a case for incorporating the model officially in OpenCL, perhaps as an extension.

3.6.1 Recall of discovery protocol

As discussed in Section 3.4, the occupancy discovery protocol identifies a *subset* estimate of co-occupant workgroups for a given GPU and kernel. It is desirable for the discovered occupancy to be as close as possible, and ideally equal, to the occupancy bound for the GPU and kernel. To assess this, microbenchmarks are used to experimentally evaluate the recall of the occupancy discovered by the protocol, experimenting with two concrete mutex implementations and four kernel configurations.

Microbenchmarks The occupancy bound for a given GPU and kernel depends on the resources used by the kernel. To thoroughly evaluate the recall of the discovery protocol, a set of kernels with a variety of resource-usage characteristics are required. Two resources are considered here: *local memory*, and *threads per workgroup.* Register pressure can also affect the occupancy bound of a kernel [PTG13], but register allocation is a function of the compiler, and as such fine-grained control over this resource is unavailable.

The microbenchmarks are based on a kernel that calls the discovery protocol to identify participating groups, which then execute one inter-workgroup barrier. The single barrier synchronisation in the microbenchmark can be configured to use participating workgroups, or all workgroups. This allows for an unsafe search for the occupancy bound. This kernel is parameterised by (1) the amount of local memory that is allocated, and (2) the number of threads per workgroup. A single microbenchmark is an instance of the kernel with a particular choice for these parameters.

Mutex implementations In Section 3.4, the mutex implementation used by the discovery protocol was left abstract. Two mutex implementations are evaluated here: a *spin-lock* and a *ticket-lock*. Both have straightforward implementations using OpenCL 2.0 atomic operations.

The *spin-lock* [Sol09, p. 269] is implemented via a flag variable that can be *locked* or *unlocked*. To lock the mutex, a thread enters a spin loop that uses an atomic test-and-set operation to write *locked* to the flag and obtain the previous flag value. The thread exits the loop when the previous flag value is *unlocked*, indicating that the thread has successfully acquired the lock. A thread unlocks the mutex by writing *unlocked* to the flag.

The *ticket-lock* [Sol09, p. 276] mutex uses two counters: a *ticket value* and a *servicing value*. To lock the mutex, a thread first atomically increments the ticket value, obtaining the old value as a *ticket*. The thread then polls the servicing value until it matches the thread's ticket, in which case the thread has acquired the lock. A thread unlocks the mutex by incrementing the servicing value. Unlike the spin-lock, where contending threads may obtain the mutex in any order, the ticket-lock is *fair*: threads obtain the mutex in the order in which they request the mutex.

Experimental setup For a given GPU, The OpenCL framework is used to identify the maximum number of bytes of local memory that can be allocated (L), and the maximum number of threads per workgroup (W); these values vary between devices. Four microbenchmark instances are then considered for the GPU, with allocated local memory set to either 1 or L bytes, and either 1 or W threads per work group. These instances capture extreme points of the resource parameter space. Each microbenchmark is executed 50 times per chip and mutex implementation, recording the mean and standard deviation of the number of discovered workgroups.

To determine the occupancy bound of a microbenchmark, the discovery protocol is disabled. The microbenchmark then attempts to execute the inter-workgroup barrier

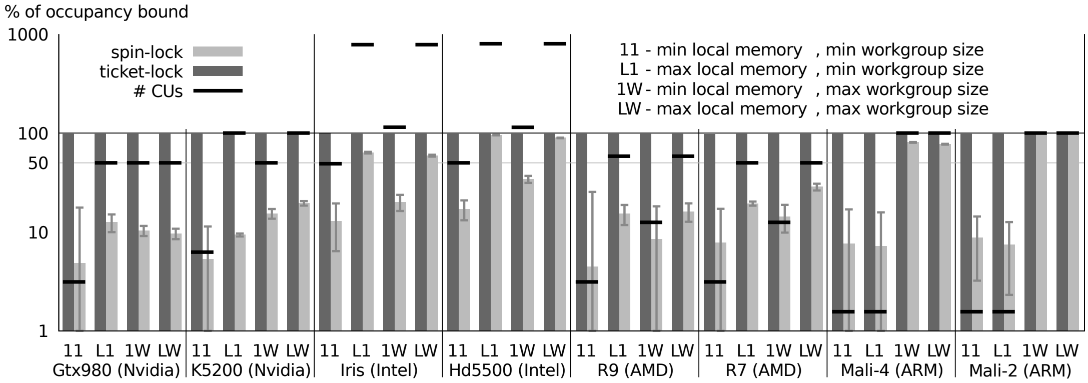


Figure 3.7: Discovered occupancy and number of compute units compared to the occupancy bound.

(unsafely) across all workgroups, searching for a value N (the occupancy bound) such that the unsafe inter-workgroup barrier succeeds for N workgroups, but hangs with $N+1$ workgroups.

Recall results Figure 3.7 shows the recall of occupancy discovery for both mutex implementations, as a percentage of the occupancy bound (y-axis, plotted using a log scale to account for the low recall of the spin-lock). For each GPU results are shown for four microbenchmarks; label xy (with $x \in \{1, L\}$ and $y \in \{1, W\}$) indicates the resource parameters associated with a microbenchmark. Light and dark grey bars show recall for the spin-lock and ticket-lock mutexes, respectively. Standard deviation whiskers are shown for the spin-lock results, and omitted for the ticket-lock results, which exhibited negligible deviation. The black horizontal bars show the number of compute units (CUs) as reported by the OpenCL framework per chip (as a percentage of the occupancy bound). Table 3.2 shows the average concrete occupancy numbers for each chip, benchmark and mutex.

The results show that with the ticket-lock mutex, the protocol almost always provides 100% recall (discovering the occupancy bound), showing suboptimal recall of still more than 95% in two cases: R7 and Iris for the 11 microbenchmark. However, the 11 configuration would not likely be used in practice as only using one thread per workgroup would severely underutilise hardware resources (e.g. SIMD processing elements). In contrast, the spin-lock performs poorly, both in the mean discovered occupancy (often less than 50% of the occupancy bound) and consistency across runs, as shown by the high standard deviation.

Table 3.2: Occupancy for each chip and microbenchmark for occupancy bound (OB), and the average discovered occupancy using the ticket-lock (TL) and spin-lock (SL).

Chip	#CUs	OB or mutex	11	L1	1W	LW
		OB	512.0	32.0	32.0	32.0
GTX980	16	TL	512.0	32.0	32.0	32.0
		SL	24.9	4.0	3.3	3.1
		OB	192.0	12.0	24.0	12.0
K5200	12	TL	192.0	12.0	24.0	12.0
		SL	10.1	1.1	3.7	2.3
		OB	96.0	6.0	41.0	6.0
IRIS	47	TL	94.9	6.0	41.0	6.0
		SL	12.4	3.8	8.2	3.5
		OB	48.0	3.0	21.0	3.0
HD5500	24	TL	48.0	3.0	21.0	3.0
		SL	8.1	2.9	7.2	2.7
		OB	896.0	48.0	224.0	48.0
R9	28	TL	896.0	48.0	224.0	48.0
		SL	39.9	7.3	19.4	7.7
		OB	256.0	16.0	64.0	16.0
R7	8	TL	250.3	16.0	64.0	16.0
		SL	20.0	3.1	9.2	4.6
		OB	256.0	256.0	4.0	4.0
MALI-4	4	TL	256.0	256.0	4.0	4.0
		SL	19.6	18.3	3.2	3.1
		OB	128.0	128.0	2.0	2.0
MALI-2	2	TL	128.0	128.0	2.0	2.0
		SL	11.2	9.5	2.0	2.0

The high accuracy of the ticket-lock-based discovery protocol can be attributed to the fairness provided by this mutex implementation, which provides a high likelihood that many workgroups will enter the poll before any workgroup closes the poll (see Figure 3.3). In contrast, with the unfair spin-lock, there is a higher likelihood that a workgroup will execute the polling and closing phases in quick succession, closing the poll before many other workgroups enter the polling phase.

The black horizontal bars show that the number of CUs (reported by OpenCL) provide neither an accurate nor safe estimate of occupancy. For example, on GTX980, R9 and R7, the number of CUs is always lower than the occupancy bound, even for the extreme *LW* case where the maximum amount of local memory is allocated and the maximum number of threads per workgroup are requested. Thus, using CUs to estimate occupancy would under-utilise GPU resources. For Mali-4, Mali-2, and K5200, the number of CUs corresponds to the occupancy bound only with high resource parameters.

It is surprising to see that on Iris and Hd5500 (Intel), the number of CUs can be *higher* than the occupancy bound. In these cases, using the number of CUs as an occupancy estimate would cause an inter-workgroup barrier to deadlock. The reason behind this, as reported by Mrozek and Zdanowicz [MZ16], is that Intel reports the number of *execution units* (EU) as the number of CUs and it may require more than one execution unit to run a workgroup.

Mrozek and Zdanowicz also provide an Intel-specific formula for computing the thread occupancy (as opposed to workgroup occupancy) for kernels that (1) do not use any local memory and (2) are launched with large workgroup sizes. These constraints correspond to the microbenchmark 1W. The occupancy formula states that the number of occupant threads is found by multiplying three values together: the number of EUs (the analogue of CUs in this specific Intel case), the threads per EU (obtained through device documentation [Int15b]), and the SIMD size (queried through the OpenCL API). For example, on Hd5500 this gives $24 \times 7 \times 32 = 5376$ as the number of occupant threads. This is exactly the number of occupant threads found for Hd5500 on microbenchmark 1W; recall that to get the number of threads, the number of workgroups is multiplied by the number of threads per workgroup, in this case $256 \times 21 = 5376$.

No formula is given for when a kernel uses local memory or small workgroup sizes, although a reading of the documentation suggests some formula may exist based on the amount of local memory allocated per group of EUs. However, such a formula would (1) be difficult to deduce from the documentation, (2) have no guarantees of safety, and (3) only be valid until a new graphics architecture is released.

Timing results To measure the runtime cost of the discovery protocol, timing measurements were also performed. That is, protocol timings were measured per chip as a function of the occupancy bound. To vary the occupancy bound, a series of microbenchmarks were considered, instantiated with increasing resource usage that leads to a decreasing occupancy bound. The workgroup size was used as the resource and values from one up to the maximum workgroup size (in multiples of eight) were considered. Each microbenchmark was run 20 times and both the average discovered occupancy and the average time to run the discovery protocol was recorded. These microbenchmarks do not contain an execution of the inter-workgroup barrier as only the protocol time is of interest here.

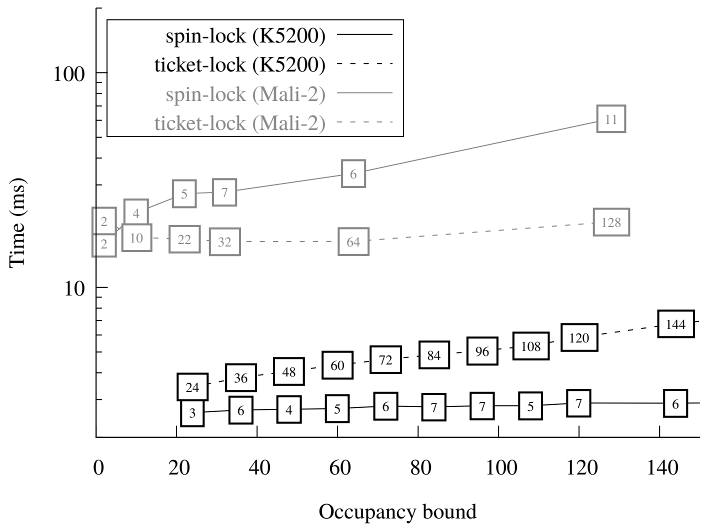


Figure 3.8: Protocol timing for K5200 and Mali-2.

Discovery protocol timing results for two representative chips, K5200 and Mali-2, are shown in Figure 3.8. For these two chips, K5200 outperforms Mali-2 in all cases; however, comparing the differing mutex strategies is more interesting here. The data labels are the average discovered occupancy for each point. There are more data points for the K5200 as it supports a larger maximum workgroup size and hence has more values for resource parameters.

For chips K5200 (shown in Figure 3.8), Gtx980, R9, and R7 the protocol using the ticket-lock is less performant than using spin-lock. This is consistent with what is generally reported for fair vs. unfair mutexes [Sol09, ch. 8]. Conversely, for chips Mali-2 (shown in Figure 3.8), Mali-4, Iris and Hd5500, the protocol using the ticket-lock is more performant than using spin-lock. This may be due to inefficient RMW operations inside the spin-lock loop. For all chips, the protocol runtime increases with the occupancy bound, although the extent varies across chips.

Regardless of performance, the low recall of the spin-lock disqualifies it for use in the protocol (Section 3.6.1). Future work might consider augmenting the spin-lock, e.g. through busy waiting, to achieve a higher recall. For most chips, the protocol executed in 5–10ms depending on the occupancy bound. The exceptions are the embedded ARM chips, which took 20–100ms to execute the protocol (for either mutex).

3.6.2 Comparison with the multi-kernel approach

Applications that originally used the multi-kernel paradigm, but which can be naturally expressed as a single kernel with inter-workgroup barriers, are now considered. In these applications, several kernels are called from a host-side loop that exits when an application-specific *stopping criterion* is met. The kernels compute data that must be copied back to the host at the end of each loop iteration in order for the stopping criterion to be evaluated. An inter-workgroup barrier allows the computation to be expressed as one large kernel, with the host-side loop migrated to run on the GPU. This requires fewer kernel launches and removes the host/device data movement between the previous kernel launches.

For such applications, The performance of the application expressed in its original multi-kernel form, vs. as a single kernel using an inter-workgroup barrier is compared.

Multi-kernel applications The Pannotia OpenCL applications benchmark the performance of graph algorithms containing irregular parallelism patterns [CBRS13]. Four of the applications that utilise the multi-kernel idiom are considered: SSSP (single source shortest path), MIS (maximal independent set), COLOR (graph colouring), and BC (betweenness centrality). The remaining applications in Pannotia either do not use the multi-kernel idiom, or use multi-dimensional execution environments, where thread ids are assigned in multi-dimensional structure rather than a flat, linear structure, which the discovery protocol does not currently handle. The discovery protocol and inter-workgroup barrier are used to write a single-kernel version of each application.

The Pannotia applications are reported to have different performance characteristics depending on the input data sets to which they are applied [CBRS13]. Given this, each application is benchmarked with all provided data sets. Each application comes with two data sets, with the exception of SSSP, which has only one. In Chapter 4, similar applications are explored using more data sets. This discrepancy in the available data-sets is because the Pannotia applications take an input graph format different from the applications examined in Chapter 4, and a unifying parser had not been developed at this point.

Experimental setup Because the number of threads per workgroup can have an impact on runtime and maximum occupancy [TGEL11], a tuning phase was first run to determine a good number of threads per workgroup. For each GPU, application, and input data-set, the application was run repeatedly with power-of-two workgroup sizes, ranging from 32 to the maximum workgroup size supported by the GPU. Preliminary testing suggested that workgroup sizes below 32 did not perform well. Each combination was executed

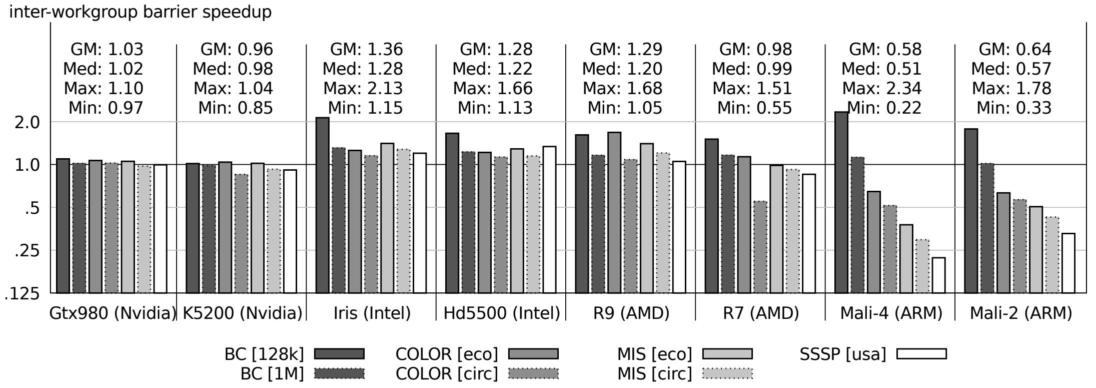


Figure 3.9: Runtime comparison of multi-kernel paradigm vs. inter-workgroup barrier.

10 times and the workgroup size that provided the fastest average runtime was recorded. The original multi-kernel application and the adapted discovery protocol applications were tuned independently.

For both the multi-kernel and discovery protocol implementations, the application/data-set combinations were then executed 20 times with the workgroup size found during the tuning phase, and the average runtime (excluding file IO for data-sets) was recorded. Application runtime was significantly longer for the ARM chips, which are designed to maximise energy-efficiency; the number of iterations for these chips was thus halved.

Results For each GPU, application and input, Figure 3.9 contains a bar plotting the average speedup associated with executing the single-kernel version of the application enabled by the inter-workgroup barrier compared with the original multi-kernel version. The shade and border of a bar indicate the associated application and input, respectively. For each GPU, the figure also shows the geometric mean (GM), median, maximum, and minimum speedups taken over all applications and inputs. All speedup values are given in Table 3.4.

The results are varied across chips and applications. For Nvidia GPUs (Gtx980 and K5200), the inter-workgroup barrier and multi-kernel variants have similar runtimes. For Intel chips (Iris and Hd5500), the inter-workgroup barrier always provides a non-negligible speedup, with mean speedups of 1.36× and 1.28×, respectively. The AMD results differ between chips: the inter-workgroup barrier always improves runtime on R9, while on R7 the following was observed: three speedups, three slowdowns, and one case where performance is unaffected.

On ARM, most results show that using the inter-workgroup barrier worsens runtime substantially, except for the BC application on the `128k` data-set, which shows a large improvement. This is likely due to the performance trade-offs of kernel launch overhead and global synchronisation, which is application and input specific. That is, kernels with long execution times amortise the cost of kernel launch as the kernel execution time becomes the bottleneck. The BC application on the `128k` data-set has the shortest kernel execution time and benefits from the barrier (also seen on the Intel and AMD GPUs). The other applications have longer kernel execution time, and thus the synchronisation overhead of the global barrier can effect performance.

The performance of the barrier is sensitive to the input for an application. For example, the inter-workgroup barrier on R7 accelerates the COLOR application for the `eco` data-set, but slows this application down for the `circ` data-set.

▶ **Looking forward:** In Chapter 4, a detailed analysis of the performance considerations of moving from multi-kernel to global barrier synchronisation is presented. It is shown that the runtime consequence of this transformation depends on certain properties of chips (e.g. kernel launch overhead), and application inputs (e.g. the average degree of graph nodes). Properties of the Pannotia MIS benchmark are discussed explicitly in Section 4.6.1.

3.6.3 Portability vs. specialisation

Attention is now turned to the use of the inter-workgroup barrier to create *portable* versions of applications that previously relied on *a priori* knowledge related to occupancy, and the performance price one pays for deploying a portable version of an application vs. relying on assumptions about occupancy.

For this purpose, applications from the CUDA Lonestar-GPU benchmark suite are examined. Originally, these applications were written in CUDA and thus, targeted only Nvidia GPUs. Like Pannotia, the Lonestar-GPU applications consist of graph algorithms that exhibit irregular parallelism patterns [BNP12]. Four Lonestar-GPU applications use a non-portable XF inter-workgroup barrier that (1) relies on assumptions about occupancy, and (2) does not consider formal memory model issues, in part due to the lack of an agreed formal memory model for CUDA. The Lonestar-GPU suite provides an occupancy estimation method that uses a CUDA-specific query function to assess the resources used by a kernel, but this estimate is not guaranteed to be safe, and is not portable.

The relevant Lonestar-GPU applications are: MST (minimum spanning tree), DMR (delaunay mesh refinement), SSSP (single source shortest path) and BFS (breadth first search). The Lonestar-GPU and Pannotia SSSP applications are fundamentally different, the former using task queues to manage the workload, and the latter using using linear algebra methods. Much like Pannotia, multiple data sets are provided for each application. The SSSP and BFS applications have three input data-sets, MST and DMR have two.

Portable and specialised OpenCL applications These four applications were ported to OpenCL, replacing the barrier implementation with the memory model-aware barrier, and the non-portable occupancy estimation function with the portable occupancy discovery protocol. These versions of the applications should be portable (at least in terms of their barrier behaviour) across OpenCL-conformant platforms that honour the occupancy-bound execution model.

However, it is speculated that there may be specific OpenCL platforms for which a developer might wish to trade portability for performance, using the memory model-aware barrier, but exploiting *a priori* knowledge about occupancy to avoid the overhead of running the discovery protocol. To investigate this trade-off, non-portable specialised variants of the four applications for each of the GPU platforms were created. These variants store pre-computed occupancy bound data, and launch kernels with maximum thread counts derived from this occupancy data. As well as avoiding running the discovery protocol, these specialised variants can use the native OpenCL thread id functions (see Section 3.4), which may, for example, allow compilers to make more aggressive optimisations.

Portability constraints and issues Numerous hurdles during the process of porting the Lonestar-GPU applications to OpenCL were encountered and written up as a separate experience report [SD16b]. These issues include OpenCL compiler bugs, OpenCL driver issues, and program bugs that only manifested on certain chips. Many of these issues were reported and confirmed with industry representatives. In particular, the following benchmarks are omitted from the study: DMR on non-Nvidia chips due to floating point issues, SSSP with the `usa` dataset on Intel due to memory requirements, and SSSP with the `usa` dataset on ARM or R7 due to prohibitively long runtimes (over 45 minutes per execution). These issues are all *independent* of the inter-workgroup barrier and stem from portability issues in GPU programming languages.

Additionally, some porting judgement was required in cases where CUDA constructs have no direct OpenCL analogue; for instance 1D texture memory is not supported for OpenCL 1.1 (the OpenCL version supported by Nvidia), and warp-aware primitives (e.g.

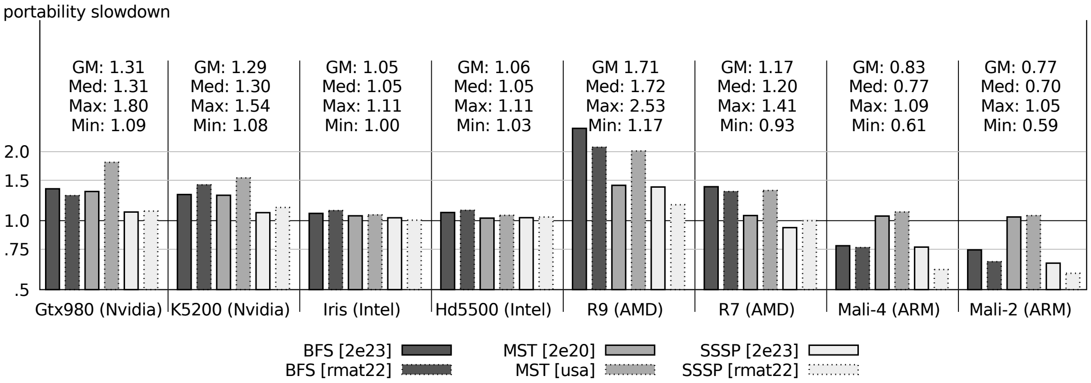


Figure 3.10: Portability slowdown for inter-workgroup barriers.

warp shuffle) are not provided until OpenCL 2.1.[1] For these issues, global memory was used instead of texture memory. Warp-aware idioms were re-written to be use workgroup locality in place of warps.

Experimental setup The same tuning process described for the Pannotia benchmarks was used to find a suitable workgroup size per GPU for each application and input data-set combination. To determine the occupancy bound for the specialised applications, specialised variants for each chip, application and input combination were created; these applications have the occupancy bound hard coded inside the application. The occupancy bound, N, is validated if the application terminates with N workgroups but not $N + 1$. Much like in the microbenchmark experiments (Section 3.6.1), N is found through a trial-and-error binary search.

Each chip, application, input combination was run for 20 iterations using both the portable and specialised variants. The workgroup size found in the tuning phase for both variants was used, and the specialised variants use the validated occupancy bound. The average runtime (excluding file IO for data-sets) was recorded. The runtime of these applications is such that all iterations were run on the ARM chips, rather than halved as in Section 3.6.2.

Results Results showing the slowdown caused by portability are shown in Figure 3.10 (concrete slowdown numbers per chip and application are given in Table 3.4). For consistency across chips, only two data-sets on the BFS, MST and SSSP applications are

[1] Section 4.4 discusses how subgroup support can be provided across several GPUs using chip-specific features.

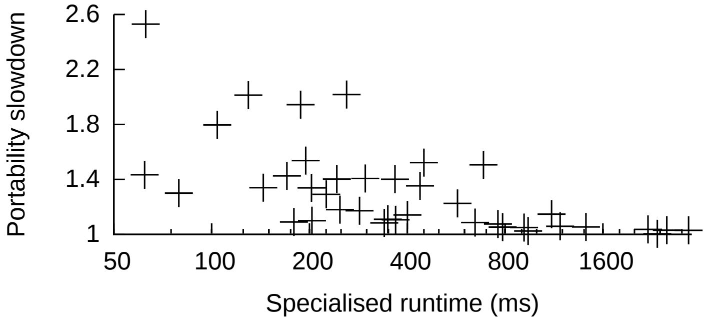


Figure 3.11: Portability overhead as runtime increases.

shown. Because this graph measures slowdown, bars that are taller than the 1.0 mark indicate that the performance was degraded by using the portable constructs. Portability on Nvidia chips (for the OpenCL application variants) costs a mean of 1.3× slowdown compared with relying on occupancy assumptions. For Intel chips, the cost of portability is low, with a slowdown of 1.11× at worst. The results for AMD are split: R9 suffers the worst slowdowns across all the GPUs considered, with a mean slowdown of 1.71×, while slowdowns associated with R7 are more modest, with a mean slowdown of 1.17× and even one speedup (SSSP on `2e23`).

Interestingly, portability provides a *speedup* for the BFS and SSSP applications on the ARM GPUs. This is because occupancy on the ARM GPUs is extremely sensitive to register pressure. In the non-portable versions of the applications, native execution environment functions are used (e.g. `get_global_id()`). The compiler maps these functions to registers, increasing register pressure and reducing the occupancy. In the portable version, the discovery execution environment functions are used and these values are cached in local memory instead of registers. Thus, the occupancy is increased, allowing more parallelism and higher performance.

While these numbers suggest that portability may have a high cost, Chapter 4 presents an optimisation to the discovery protocol in which the mutex only provides mutual exclusion to participating groups. This optimisation significantly reduces the portability overhead. For example, the highest overhead is seen on R9 with the BFS[`2e23`] application, which suffers a 2.53× slowdown when the portable constructs are used. However, the optimised discovery protocol reduces the portability overhead to 1.11×. The details of this optimisation are discussed in Section 4.6.2.

The scatter plot of Figure 3.11 provides an overview over how the slow-down associated with portability relates to the overall runtime of the specialised application. Each cross

Table 3.3: Occupancy for two CPU chips; the occupancy bound (OB), and the average discovered occupancy (out of 50 runs) using the ticket-lock with different amounts of delay between the polling and closing phases.

			Delay			
CPU chip	**#CUs**	**OB**	0	10	100	1000
Intel i7-5600U	4	4	1.0	1.3	3.3	4.0
AMD A10-7850K	4	4	1.0	1.0	1.5	3.8

refers to a particular application, input data set and GPU. The x-coordinate of the cross indicates the runtime of the specialised version of the application (averaged over 20 runs), and the y-coordinate shows the slow-down resulting from switching to a portable version of the application. The results suggest that portability has a constant cost, rather than a scaling cost: the longer the runtime of the specialised application, generally the smaller the overhead associated with portability. For example, across all GPUs, applications that took longer than 400 and 800 ms for computation in their original form showed a maximum 1.6× and 1.2× portability slowdown, respectively.

Overall, these results show that using portable constructs over specialisation can cause a slowdown, varied between applications and chips. However, understanding these results should consider that specialised applications may be *fragile* not only when ported, but also for the same GPU under compiler upgrades or code modifications. Anything that affects the resources requirements of the kernel could potentially change the occupancy of the kernel and introduce deadlocks. Given these considerations, it may be best for developers to investigate the optimisation of portable constructs rather than taking the specialised approach.

It may be possible for vendors to modify their OpenCL frameworks to provide native support for the occupancy-bound execution model. For example, vendors could provide a way to launch a kernel without specifying the number of workgroups, and instead specify that only occupant workgroups should execute the kernel. It is possible that the runtime could determine occupancy bounds efficiently using low-level proprietary knowledge. It is likely then that the native execution environment functions could be used (allowing for any compiler or hardware optimisations around these native functions). Such support would likely reduce the portability slowdown of using the methods presented in this chapter.

3.6.4 OpenCL on CPUs

This work exclusively focuses on GPU platforms because today's implementations of OpenCL appear to implement a non-traditional execution model, i.e. the occupancy-bound execution model, which makes implementing a portable execution barrier non-trivial. On the other hand, execution barriers for CPU systems do not account for such execution models (e.g. see [HS08, ch. 17]). This is because many CPU concurrency frameworks allow for preemption and have a scheduler that, in practice, provides fair scheduling across all threads.

Because OpenCL can be run on some CPU systems, some pilot experiments were run using the occupancy benchmarks of Section 3.6.1 to investigate the execution models associated with CPU implementations. It was hypothesised that an occupancy bound would not exist; that is, an inter-workgroup barrier would be deadlock-free on CPUs for any (reasonable) number of workgroups.

It was surprising to observe an occupancy bound of 4 for the two CPU frameworks tested: the Intel SDK for OpenCL (build 10094, and driver version 5.2.0.10094) and AMD APP SDK (version 2004.6), running on the Intel and AMD CPUs (respectively) detailed in Table 3.3. Thus, OpenCL implementations of inter-workgroup barriers can deadlock even on CPU systems if executed with too many workgroups. Because CPUs do not natively provide this behaviour, it is hypothesised that the runtime must implement the occupancy-bound execution model on top of the native CPU scheduler; this would be straightforward, e.g. by storing workgroups in a queue.

It was also found that the recall of the discovery protocol, even using the ticket-lock, was poor on CPUs. To increase the recall, a delay between the polling and closing phase of the protocol (see Section 3.3) was inserted. The delay consists of a loop where the thread performs a lock and unlock function on the mutex. For the ticket-lock, this allows other threads to queue in front of the spinning thread to poll. The results (Table 3.3) show that a delay can increase the recall of the protocol on CPUs. In these experiments, there was no difference in the occupancy bound when varying the amount of local memory allocated (unsurprising, since CPU architectures do not feature software-managed memory) nor the number of threads per workgroup.

These experiments show that synchronisation constructs (in this study, an execution barrier) must account for the execution model of the framework in which they operate, not just the hardware on which they will be executed: CPU devices are adept at managing preemption between threads, yet the CPU OpenCL implementations tested still exhibit an occupancy-bound execution model. Defining and reasoning about execution models and the synchronisation constructs that they allow promises to be a fruitful area of research for frameworks with native support for concurrency, e.g. OpenCL, C++, and OpenMP.

▶ **Looking forward:** In Chapter 6, a formal framework for describing scheduler fairness properties, i.e. the forward-progress guarantees of the execution model, is given. Fairness properties for several execution models are given, including the occupancy-bound execution model, OpenCL, and another GPU programming model: HSA.

3.7 Related work

The basis for the work in this chapter is the persistent thread model [GSO12] and the XF software execution barrier [XF10]; the formal memory model reasoning follows techniques for C++11 concurrency abstraction reasoning [BDG13]. The empirical evaluation uses the Pannotia [CBRS13] and Lonestar-GPU benchmarks [BNP12], which were originally designed to examine performance characteristics of irregular GPU workloads.

Irregular parallelism on GPUs The need for an inter-workgroup barrier arises due to irregular parallelism on GPUs; this has been examined with various approaches in several related works. Hower et al. propose a work-stealing programming idiom for irregular computations using shared concurrent queue data structures [HHB$^+$14]. Orr et al. extend this work by proposing new primitive GPU synchronisation operations, allowing for more efficient work stealing [OCY$^+$15]. Other approaches (e.g. [MGG12b, WDP$^+$16]) use the multi-kernel method, but focus on data representations, and computations exploit low-level GPU-specifics, e.g. warp-level instructions. None of these works consider portability across GPUs from multiple vendors.

Much work has been done on accelerating blocking irregular algorithms using GPUs using the *persistent threads* programming style for long-running kernels [KVP$^+$16, DBGO14, HN07, MGG12b, VHPN09, NCKB12, STT10, MBP12, PP16, CT08, TPO10, BNP12]. These approaches rely on the occupancy-bound execution model, flooding available compute units with work, so that the GPU is unavailable for other tasks, and assuming fair scheduling between occupant workgroups.

GPU models The formal memory model reasoning is based on a formalisation of the OpenCL 2.0 memory model [BDW16]. Other notable models developed in prior work include several variants of the hierarchical-race-free model [GHH15], and a formal model of a fragment of PTX, the compiler intermediate representation for Nvidia GPUs [ABD+15].

The occupancy discovery protocol is based on enabling the persistent thread model for GPUs [GSO12]. Related works studying GPU execution models, e.g. GPUVerify [BCD+15] and GKLEE [LLS+12], do not account for relative scheduling properties of workgroups. Gaster provides an execution model for intra-workgroup interactions, and describes how barriers can be implemented at this level [Gas15].

Wu et al. present a persistent thread CU-centric programming model that exploits CU locality across workgroups [WCL+15]. To prevent over-saturating the hardware, they present a protocol in which some of the persistent threads immediately exit the kernel without performing any computation. This is similar to the discovery protocol where non-occupant workgroups immediately exit. Essentially both protocols circumvent the proprietary GPU scheduler to enforce a certain mapping between workgroups and CUs.

3.8 Summary

As GPUs and GPU applications grow in variety, so will the need for robust and portable synchronisation and programming idioms. In this Chapter, building on non-portable previous work, a GPU inter-workgroup barrier was developed, analysed, and evaluated in a portable context for the first time. To achieve this, an occupancy discovery protocol was used to estimate the number of persistent threads, i.e. threads which have traditional fair scheduling guarantees. The OpenCL 2.0 memory model was then used to create an inter-workgroup barrier with intuitive memory model properties.

The discovery protocol was evaluated through microbenchmarks, and the runtime effects of the portable barrier were compared to existing methods for inter-workgroup synchronisation on GPUs. While the experimental results show that the runtime behaviour of the barrier varies between chips, applications and even inputs, it was shown that the inter-workgroup barrier is semantically portable across a wide range of GPUs.

The discovery protocol and associated portable inter-workgroup barrier presented in this chapter form the basis of the rest of the thesis. In particular:

- Chapter 4 discusses how the portable barrier can be used in a compiler for GPU graph applications;

- Chapter 5 discusses a cooperative multitasking scheme for GPUs, in which one of the new proposed primitives is a special global barrier. The portable barrier presented here is used as a basis for the prototype scheduler presented;
- Chapter 6 discusses a formal framework, using temporal logic, to reason about blocking synchronisation on GPUs, including the discovery protocol and barrier of this chapter.

Table 3.4: Pannotia application speedups using the inter-workgroup barrier vs. multi-kernel (higher is better), and Lonestar-GPU application slowdowns for using a portable vs. non-portable barrier (lower is better).

Chip	**Barrier speedup for Pannotia**							**Portability slowdown for Lonestar-GPU**					
	BC		COLOR		MIS		SSSP	BFS		MST		SSSP	
	`128k`	`1M`	`eco`	`circ`	`eco`	`circ`	`usa`	`2e23`	`rmat22`	`2e20`	`usa`	`2e23`	`rmat22`
GTX980	1.10	1.02	1.07	1.02	1.05	0.97	0.99	1.38	1.29	1.34	1.80	1.09	1.10
K5200	1.02	0.98	1.04	0.85	1.02	0.93	0.92	1.30	1.43	1.29	1.54	1.08	1.14
IRIS	2.13	1.31	1.26	1.15	1.41	1.28	1.20	1.08	1.11	1.05	1.06	1.03	1.00
HD5500	1.66	1.22	1.22	1.13	1.29	1.15	1.34	1.09	1.11	1.03	1.05	1.03	1.04
R9	1.62	1.16	1.68	1.08	1.40	1.20	1.05	2.53	2.09	1.43	2.01	1.40	1.17
R7	1.51	1.16	1.13	0.55	0.99	0.92	0.85	1.41	1.34	1.05	1.35	0.93	1.00
MALI-4	2.34	1.12	0.65	0.51	0.38	0.30	0.22	0.78	0.76	1.05	1.09	0.77	0.61
MALI-2	1.78	1.02	0.63	0.57	0.51	0.43	0.33	0.75	0.66	1.04	1.05	0.65	0.59

4 Functional and Performance Portability for GPU Graph Applications

4.1 Personal context

While I was pleased with the functional portability of the global barrier, as presented in Chapter 3, the performance results were underwhelming. I wanted to push for a stronger case for official support of a GPU global barrier and I felt that the best way forward was to find GPU applications where a global barrier provided significant speedups across GPUs spanning many vendors.

Luckily, a perfect opportunity quickly presented itself. When I presented the work of Chapter 3 at the 2016 OOPSLA conference in Amsterdam, the talk directly after mine was given by Sreepathi Pai, who presented his work on a GPU graph application DSL and an associated optimising compiler [PP16]. One of the optimisations of his compiler was exactly transforming multi-kernel synchronisation into global barrier synchronisation, as discussed in Sections 3.6.2. His work targeted Nvidia GPUs exclusively, and as such he was able to use Nvidia's occupancy calculator to determine the occupancy bound required for a global barrier. Our work provided a way for this optimisation to become functionally portable across many GPUs.

This chapter presents the results of a collaboration that followed, in which we combined the two works to develop a portable version of the optimising compiler, of which the portable global barrier was an essential ingredient. As the results of this chapter show, we now further understand situations where a global barrier can provide significant performance improvements across a range of GPUs spanning several vendors. Additionally, the rich set of applications, inputs, and GPUs available allowed for an exploration of performance portability in the domain of graph algorithms on GPUs.

4.2 Motivation

Is a GPU in a mobile device similar to a GPU running in a data center? For certain attributes, such as the number of compute units, clock frequency and power consumption, the answer is clearly no. These differences are by design and therefore these GPUs will continue to differ on these attributes. As a result, it is unrealistic to expect a program that achieves peak performance on one GPU to achieve similar peak performance on another GPU *unchanged*. Additionally, code transformations that provide runtime differences (e.g. a speedup) on one GPU, may provide different runtime behaviour on another GPU (e.g. a slowdown). As a concrete example, Figure 3.9 from Chapter 3 shows that the runtime effects of transforming multi-kernel synchronous code to global barrier synchronous code varies significantly across different GPUs.

Over the years, programmers have become skilled at capturing differences in machines as *tunable* parameters, whose values are determined as late as possible to allow a program to adapt to the final machine on which it is being run, i.e. to make it more *performance portable*. Such parameters were initially machine-linked [WD98], but increasingly sophisticated techniques allow tuning over algorithms [ACW$^+$09], inputs [DAV$^+$15], and microarchitecture designs [CHR$^+$16] often under control of a generic autotuner or search strategy [AKV$^+$14, MSH$^+$14]. GPU programs, also, contend with a diverse set of implementations and have used similar strategies [MGG12a, SRD17, PP16] to achieve performance portability.

Debugging lack of performance portability is hard. By construction, autotuning specialises for a particular machine, application and input; thus it provides no guarantees for performance across these dimensions. Although autotuning has proved to be useful in many cases, questions remain as to how differences in machines affect performance. In particular, opportunities to identify and rectify performance issues are missed.

The goal of this chapter is to ask: if exhaustive data on program performance is available, can issues be identified that impede performance portability and can they be apportioned to microarchitecture, applications, inputs or a combination of these?

The domain for this study is GPU graph algorithms written in OpenCL. Graph algorithm kernels were studied for three reasons:

1. The irregular nature of these algorithms require global synchronisation in many cases. The work of Chapter 3 provides a recipe for providing portable global synchronisation using a global barrier, and the performance considerations of such a barrier have not been rigorously explored across many GPUs.

2. As is shown in Section 4.3, high performance graph kernels use many more OpenCL features than, for example, a high performance matrix multiplication kernel. Thus, they exercise more of the hardware implementation. For this reason, graph algorithm kernels are notoriously hard to write in a performance portable way.

3. Recent work [PP16] has produced a compiler that applies GPU optimisations to graph algorithms written in an intermediate language called IRGL and generates CUDA code. This can be used to generate many different kernel implementations of the same algorithm in a controlled fashion to explore performance portability. Retargeting this compiler to generate portable OpenCL code is a natural choice.

▶ **Example 4.1** (Performance and functional portability). On an AMD R9 Fury GPU, a maximal-independent-set (MIS) algorithm on a road network graph enjoyed a 1.13× speedup when two optimisations were applied: one involving SIMD atomic combining and another involving global barrier synchronisation. Running the same OpenCL code on an Nvidia Quadro M4000 GPU causes a dramatic 1.69× *slowdown* (0.59× speedup). The code does not even compile on an ARM Mali-T628 GPU since it does not provide support for the SIMD primitives required for the optimisations. The global barrier optimisation relies on a degree of independent forward progress that *is* empirically provided by current GPUs (as described in Chapter 3), but is not mandated by the OpenCL standard. Therefore, it might not be provided by future architectures. For future-proof portability, this optimisation might be disabled, in which case the observed speedup drops from 1.13× to 1.06× on the AMD GPU.

In general, writing *performance-portable* code for GPU architectures is difficult for three principal reasons:

1. **Architectural differences** between platforms mean that optimisations that benefit one platform may have little impact, or a negative impact, on another platform;

2. **Differences in supported language features** between platforms may harm *performance* if one opts to exploit only those language features that are supported by all implementations, and may harm *portability* if support for the latest language features is assumed;

3. **Data set diversity** can make it hard to choose good optimisation settings for irregular applications that work well across a range of interesting data sets, even if the target platform is fixed.

These challenges are not limited to GPUs, but are particularly relevant in this domain for the following reasons: (1) multiple vendors offer competing chips with significant architectural differences; (2) version support OpenCL varies widely in practice; and (3) it has been shown that the utility of optimisations for existing irregular GPU algorithms varies significantly based on the input data set [PP16].

4.2.1 Chapter contributions

This chapter presents a large experimental study that explores the 95 optimisation combinations (all possible configurations for the optimisations considered) applied to 49 (graph application, input) combinations, across 6 GPUs from 4 vendors (see Table 2.2). The rich set of performance data is queried, investigating the following seven research questions; for each research question, the associated subsection of the chapter in which it is addressed is given:

1. Are optimisations for GPU graph applications, developed in previous work and targeted to Nvidia GPUs, beneficial for GPUs from other vendors? How do the top speedups and slowdowns compare across GPUs and how does the distribution of optimisations required for top speedups vary across GPUs (Section 4.5.1)?

2. How do top speedups fare when functional portability is reduced? That is, how much performance do newer, less supported, features enable across benchmarks and chips (Section 4.5.2)?

3. Are there "portable optimisations"; that is, optimisation settings that provide better than baseline performance across the board (Section 4.5.3)?

4. Are certain optimisations always, or commonly, required for top speedups across the board? If not, what is the distribution of optimisations required for top speedups (Section 4.5.4)?

5. What are the performance consequences of abandoning portability by degrees and specialising to chips, applications or inputs across benchmarks in this domain (Section 4.5.5)?

6. Can optimisation strategies specialised per chip deliver insights into performance-critical architectural, application, or input features (Section 4.5.6)?

7. How does an optimisation policy guided by tuning on single GPU platform fare when used to make optimisation decisions for other GPU platforms (Section 4.5.7)?

Outline First, descriptions of the GPU graph algorithm optimisations that were considered in this chapter are given, including the required OpenCL features for each and, thus, the architectural features that each optimisation is sensitive to (Section 4.3). The experimental methodology is described in Section 4.4, containing a description of an analysis that creates optimisation strategies for various levels of specialisation (Section 4.4.2), and a description of the GPUs and applications used in the empirical study (Section 4.4.3 and Section 4.4.4). The results of the empirical study, organised by the above research questions are given in Section 4.5. In Section 4.6, several results from Chapter 3 are revisited given the new findings of this chapter. The chapter concludes with related work in Section 4.7.

Related publications The material presented in this chapter is based on work currently under submission. The citation references the current draft [SPD18].

4.3 Generalising optimisations to OpenCL

Recent work on GPU acceleration of graph algorithms presented four key architecture-independent graph algorithm optimisations. These were shown, via their embedding in an optimising compiler generating CUDA code, to achieve state-of-the-art performance for a number of applications running on Nvidia GPUs [PP16]. To study performance portability, this compiler, originally generating only CUDA, is retargeted to generate OpenCL.

The four optimisations are discussed in Section 4.3.1—4.3.4, in each case providing: (1) a high-level description of the optimisation; details of challenges associated with generalising the optimisation to OpenCL, and (2) details of the GPU architectural features that govern the performance potential of the optimisation. The optimisations are summarised in Table 4.1.

4.3.1 Cooperative conversion

The OpenCL execution hierarchy (see Section 2.3) can be exploited to reduce the cost of expensive, serialising operations such as atomic read-modify-write (RMW) instructions. For example, many graph algorithms track the dynamic workload through a global worklist. Each push of a thread ordinarily requires one RMW. However, threads can communicate at the subgroup or workgroup levels to combine individual pushes into a single push of multiple results that uses only one RMW. This *cooperative conversion* optimisation is abbreviated to coop-cv.

Table 4.1: List of optimisations, the OpenCL features they exploit and the architectural parameters that influence performance. Feature class refers to the OpenCL feature classes described in Section 2.3.

Optimisation	OCL features	Feature class	Architecture parameters
cooperative conversion (coop-cv)	Local memory, `sub_group_any`, `sub_group_reduce`, `barrier`, `atomic_fetch_and_add`, `popcount`	OCL 2.1	workgroup size, subgroup size, atomic read–modify–write throughput, subgroup collectives throughput
fine-grained nested parallelism (fg)	Local memory, `barrier`	OCL 1.x	local memory size, barrier throughput
subgroup nested parallelism (sg)	Local memory, `sub_group_barrier`, `sub_group_any`, `atomic_store`, `atomic_load`	OCL 2.1	subgroup size, subgroup barrier throughput, local memory size
workgroup nested parallelism (wg)	Local memory, `atomic_store`, `atomic_load`, `barrier`	OCL 2.0	workgroup size, local memory size, barrier throughput, atomic load/store throughput
iteration outlining (oitergb)	`atomic_load`, `atomic_store`	OCL FP	overheads for kernel launch and memory transfers, global memory fence throughput, workgroup scheduler behaviour
workgroup size of 256 (sz256)	`clEnqueueNDRangeKernel`	OCL 1.x	occupancy, resource limits

OpenCL generalisation Unlike in CUDA, subgroup operations must be *uniform*, i.e. they must be executed by all or none of the threads in the subgroup. Thus, subgroup-uniform branches must be generated (for example, by equalising loop trip counts across threads), and predication is used to prevent execution of code that would originally have not executed. Subgroup primitives can then be executed uniformly, performing a subgroup reduction on the predicate that caused the non-uniform branch in the original optimisation.

Cooperative conversion requires fine-grained subgroup communication, thus falling into the OCL 2.1 feature class.

Performance considerations The number of atomic RMW operations that can be avoided depends on workgroup size and subgroup size. Communication uses local memory and the appropriate barriers (workgroup/subgroup). Note that on architectures that implement subgroups with lockstep execution, subgroup barriers are free. The performance impact of this optimisation depends on the overhead of the orchestration (i.e. synchronisation and communication) vs. the cost of the global RMW operations.

4.3.2 Nested parallelism

Not to be confused with OpenCL nested parallelism (which mimics CUDA's dynamic parallelism), the *nested parallelism* optimisation tackles the classic problem of parallelising nested loops, the inner of which is usually irregular in graph algorithms. Specifically, it generates *inspectors* and *executors* that inspect the inner loop iteration space at runtime and redistribute work among the threads. The specific schemes for distributing work are based on proposals by [MGG12b] and redistribute work among threads of the workgroup (wg) or threads of the subgroup (sg). When redistributing to threads of the workgroup, the executor can choose between serialising the *outer* loop or linearising the iteration space (fine-grained or fg). Often, all three strategies wg, sg or fg must be used in combination, with wg handling high-degree nodes, sg handling medium degree nodes and fg handling the rest.

The fg variant can also be parameterised by the number of edges processed per iteration. Two possibilities were considered in this work: a single edge (denoted fg1) and eight edges (denoted fg8). The entire class of nested parallelism optimisations is abbreviated to np.

OpenCL generalisation The fg scheme of this optimisation is straightforwardly ported to OpenCL, and thus is placed in the OCL 1.x class.

The wg scheme has two OpenCL considerations: first, the work distribution requires several specialized workgroup scan operations. In CUDA, these are provided via a high-performance library, for which there is not an OpenCL equivalent. In this work, a simple OpenCL scan implementation provided in [Mer] was used. Additionally, the wg scheme requires concurrent writes to the same location in a leader-election idiom. OpenCL deems this as a data-race, rending the entire program undefined [Khr15, p. 48]. Thus, all racy accesses were identified and changed to OpenCL 2.0 atomic operations (see Section 2.3.2); each loop redistribution required six memory accesses to be changed. As a result of using atomic operations, wg is placed in the OCL 2.0 class.

The sg scheme requires subgroup orchestration similar to coop-cv. However, sg makes the additional assumption that subgroup threads execute in lock-step; this is not guaranteed in OpenCL (starting with Volta, this is not the case for Nvidia either [NVI17, p. 26]). Two subgroup barriers per loop were required to explicitly provide the lock-step synchronisation that was previously assumed. These subgroup primitives place the sg scheme in the OCL 2.1 class.

Performance considerations Because these schemes involve a large amount of inter-thread communication mediated through barriers (both workgroup and subgroup), the throughput of barriers is a critical factor. The fg scheme has the highest number of barriers, but this can be reduced by increasing the number of edges processed per thread, though this consumes more local memory. Read and write latency to local memory is important as communication and work redistribution occurs through this memory region. Finally, if there is very little load imbalance among threads (for example, due to uniform degree graphs), the overhead of these schemes may outweigh the benefits.

4.3.3 Iteration outlining

Many graph algorithms execute kernels iteratively until a fixed point has been reached. For example, in breadth-first search (BFS), the number of dependent iterations is proportional to the diameter of the graph. In the case of planar graphs, such as road networks, this may be thousands of iterations. If each kernel execution is very short, then the launch overhead dominates execution time. In iteration outlining, code that launches kernels is *outlined* to the GPU. As a result, kernel launches are turned into GPU function calls, with synchronisation between function calls provided by a global barrier, i.e. the GPU barrier construct discussed in Chapter 3, which synchronises across all threads on the GPU. This optimisation is exactly the transformation from the multi-kernel synchronisation approach to the global barrier approach discussed in Section 3.6.2. This optimisation is abbreviated to oitergb.

OpenCL generalisation The crux of this optimisation for OpenCL is a *portable* global barrier; the exact topic of Chapter 3 in this thesis. The discovery protocol and barrier presented earlier can directly be used here. To recap, the discovery protocol dynamically discovers the GPU occupancy (i.e. workgroups that are guaranteed relative forward progress) and creates a custom execution environment. Using this dynamic information, a portable global barrier can be used. Because of the forward progress requirements of this optimisation, it falls into the OCL FP class.

A possible concern of using the methods of Chapter 3 in this optimisation is that the results of Section 3.6.3 show that the portable constructs can have a significant overhead compared with a priori occupancy approaches. For example, BFS[2e23] suffers a 2.53× slowdown on R9 when the portable constructs are used. However, in this chapter, an optimised discovery protocol is used which significantly reduces the portability overhead. For example, the same application on R9 using the optimised discovery protocol only suffers 1.11× slowdown. Thus, the portability overhead is greatly reduced from what the results of Section 3.6.3 suggest. The optimisation, along with some results, are discussed in Section 4.6.2.

OpenCL 2.0 provides support for *nested parallelism* [Khr15, p. 32] where a thread can enqueue another kernel while executing on the device. In theory, this nested parallelism could be used to achieve a similar on-chip synchronisation. Two difficulties were encountered when investigating this. First, the nested parallelism model is *tail recursive*, that is, the child kernel is not guaranteed to execute until the parent finishes. This proved to be a difficult transformation for loops in the applications. Secondly, the available implementations of nested parallelism were found to be extremely unreliable, causing compiler crashes and runtime errors that could not be diagnosed, despite a best-effort following of the standard. Thus, OpenCL nested parallelism is not provided as an iteration outlining optimisation at this time.

Performance considerations The performance impact of iteration outlining on a platform is a function of the kernel launch overhead (including a memory copy), the execution time of the barrier and the execution time of the kernel. While the first two are largely architecture dependent, the last also depends on the application and input.

4.3.4 Workgroup size

The final optimisation considered is simply resizing the number of threads in the workgroup from 128 to 256. A workgroup size of 128 is used as the default as not all of the GPUs investigated support a workgroup size of 256 in all cases (see Section 4.4.5). This optimisation is a simple change to the kernel launch parameters and thus falls into the OCL 1.x class. The workgroup size is known to affect occupancy (see Chapter 3), which can affect performance. This optimisation is abbreviated to sz256.

4.3.5 The optimisation space

With the exception of fg, all optimisations can be enabled or disabled independently. In the case of fg, two variants were considered: fg1 and fg8. Thus, there are 95 total optimisation combinations, excluding the baseline, where no optimisations are enabled.

4.4 Experimental methodology

The compiler for graph algorithms, generating OpenCL code via the optimisations described in Section 4.3, allowed the opportunity to undertake a large study in performance portability for this domain. First, useful terminology is defined for describing the methodology and results (Section 4.4.1). Then an analysis is described which, given a set of empirical data, is able to produce optimisation strategies at various levels of specialisation (Section 4.4.2). Next, the GPUs considered in this chapter are described in detail (Section 4.4.3). Following this, the graph algorithms to which the optimisations were applied are discussed and along with their graph inputs (Section 4.4.4). Finally, the methods used to gather experimental data are detailed (Section 4.4.5). A discussion of the experimental findings then follows in Section 4.5.

4.4.1 Terminology

An *application* is a graph algorithm expressed in the compiler DSL; the same DSL used in [PP16]. An application accepts an *input*, which is a graph. A *chip* refers to one of the GPUs listed in Table 2.2, but also includes the runtime environment. A *benchmark* is an (application, input) tuple, and a *test* is an (application, input, chip) tuple. That is, a *benchmark* is the instantiation of an application for a given input, and a *test* is the instantiation of a benchmark for a given chip.

The compiler generates OpenCL code by applying zero-or-more *optimisations* to an application. An *optimisation configuration* records which optimisations were enabled. Most optimisations are binary (enabled or disabled), the exception of the fg optimisations, which may either be disabled, or enabled and set to a numeric value. As previously mentioned, the two values for fg considered in this chapter are one and eight, noted as fg1 or fg8, respectively. The *baseline* configuration disables all optimisations.

For a test and two optimisation configurations o_1 and o_2: o_1 yields a *confident speedup* over o_2 if the difference is statistically significant at the 95% confidence interval. A *confident slowdown* is defined analogously. Henceforth, *speedup* and *slowdown* refer to confident speedups and slowdowns unless stated otherwise. An optimisation configuration yields a speedup (slowdown) for a test if it yields a speedup (slowdown) for the test over the baseline configuration.

An *optimisation strategy* maps a test to an optimisation configuration. For example, the *baseline* strategy maps every test to the baseline configuration. Similarly, the *oracle* strategy maps a test to the configuration that led to the maximum speedup observed during the experiments.

4.4.2 Creating optimisation strategies by specialisation

An optimisation strategy is more *portable* if it uses *less* information about a test when mapping it to an optimisation configuration. The *baseline* strategy is completely portable: all optimisations are disabled regardless of the test. In contrast, the *oracle* strategy is least portable since it is the most specialised, being generated via exhaustive search.

An analysis is now described that is able to produce, from empirical data, a spectrum of optimisation strategies between baseline and oracle by incorporating more and more information about a test. A key challenge in this analysis is to soundly identify useful optimisations – those that impact positively on performance on average – under various degrees of specialisation.

This analysis uses a set of *exhaustive* test results, with runtimes from all possible optimisation configurations. The aim of the approach described here is not to determine the *best* optimisations as that is easily accomplished through exhaustive search, i.e. the oracle model. Similarly, It is also not the aim to make *predictive* models, in which models prescribe optimisations to use in different situations. Indeed, evaluation of predictive models would require partitioning data into training and evaluation sets, which is not done here. Rather, the aim is to develop *descriptive* models, in which exhaustive data is mined to understand trade-offs between portability and specialisation. Such models have value in exploring various portability dimensions and might serve as a comparison point for future predictive approaches.

At a high level, to consider whether an optimisation should be enabled for a (set of) test(s), the empirical data is split into two sets. The first set contains runtime information when the optimisation was enabled and the second when it was disabled. A statistical procedure, the Mann-Whitney U (MWU) test [MW47], is used to determine whether enabling the optimisation caused a statistically significant change in test runtimes. If so, then the median of the set of runtimes where the optimisation was enabled is examined to determine whether the optimisation caused a speedup or slowdown. In the case of a speedup, the optimisation is recommended to be enabled. If the optimisation did not cause a statistically significant change in runtimes across the tests, then the optimisation is conservatively recommended to be left disabled.

While a variety of statistical methods could be used on the list of normalised runtimes, e.g. simply checking means or medians, the MWU test has several nice properties. First, the MWU test is non-parametric and does not assume any specific distribution of values. Because empirical runtimes may contain values from different chips or applications, it cannot be assumed, for example, the runtimes have a normal distribution. Secondly, the MWU test does not consider the magnitude of value differences; only if they are greater or less than the comparison set. This is important as magnitudes of runtimes can vary greatly across chips and the aim of the analysis is not to favour one chip over another.

To specialise the optimisation configurations per chip, application, input or combination of these, the above analysis is performed, but on partitions of the data. For example, when specialising optimisations per chip, the data is partitioned into subsets, one for each chip, and the aforementioned procedure is applied to each partition. Each partition will yield an optimisation strategy specialised to that partition's chip.

The analysis is shown in detail in Algorithm 4.1, with specialisation illustrated for the per-chip case (other specialisations are similar). The approach is described top-down starting with function SPECIALISE_FOR_CHIP (line 1). This function simply iterates through the available chips in the global list CHIPS (line 3), partitions the data into a set per chip (line 4), and then launches the analysis for the partition, which returns an optimisation strategy recommended for the partition (line 5).

The function that identifies an optimisation configuration for a partition is OPTS_FOR_PARTITION (line 7). Here, the aim is to extract the effect of an individual optimisation *opt* across the entirety of the data partition. To do this, two lists A and B are constructed per optimisation (line 10). The lists are populated by iterating through all the valid optimisation configurations where *opt* is enabled. This is computed by iterating through all possible optimisation configurations, stored in ALL_OPT_CONFIGS, and filtering the configurations where *opt* is enabled using the function ENABLED (line 11).

For each optimisation configuration *os*, the mirror optimisation configuration is created where *opt* is disabled (line 13); thus, the only difference between the two optimisation configurations *os* and *dis_os* is that *opt* is enabled in the former and disabled in the latter. All test data in *partition* is then considered (label 14), checking for each test p if the runtime under the two optimisation configurations is statistically significant (i.e. if the runtimes are confidently different) via CONFIDENT (line 15). If so, then the runtime with *opt* enabled is normalised to the runtime when *opt* was disabled (line 16) and added to the

Algorithm 4.1 Finding optimisation strategies for different levels of specialisation.

```
 1: function SPECIALISE_FOR_CHIP(data)
 2:     chip_opts = MAP()
 3:     for chip in CHIPS do
 4:         chip_data = {X in data where X was run on chip}
 5:         chip_opts[chip] = OPTS_FOR_PARTITION(chip_data)
 6:     return chip_opts

 7: function OPTS_FOR_PARTITION(partition)
 8:     enabled_opts = {}
 9:     for opt in OPTS do
10:         A = [], B = []
11:         opt_configs = [os in ALL_OPT_CONFIGS if ENABLED(opt, os)]
12:         for os in opt_configs do
13:             dis_os = os[opt = disabled]
14:             for p in partition do
15:                 if CONFIDENT(p[os], p[dis_opt]) then
16:                     A.add(p[os] / p[dis_opt])
17:                     B.add(1.0)
18:         if ENABLE_OPT(A,B) then
19:             enabled_opts.add(opt)
20:     return enabled_opts

21: function ENABLE_OPT(A,B)
22:     significant = MWU(A,B)
23:     return significant and MEDIAN(A) < 1.0
```

list A. Because the value added to A was normalised, the *baseline* value of 1.0 is added to the second list B.

With data from A and B, a statistical analysis can be performed to determine whether *opt* should be enabled (lines 18 and 19). Essentially, ENABLE_OPT (line 21), takes the two lists A and B and applies the MWU test (line 22). The MWU test takes two lists of values X and Y and tests whether there is a statistically significant difference in *ranks* between the two lists. That is, it will test if one list has stochastically larger values than the other list. If the test determines this with $p < .05$, the median of A is considered: if it is less than 1.0, indicating half of the tests have sped up, then the optimisation is enabled (line 23); otherwise it is disabled.

▶ **Remark** (Ranks and confidence intervals). The procedure for finding optimisation strategies at various levels of specialisation (i.e. Algorithm 4.1) uses a *nonparametric* test, that is, it does not assume a normal distribution of values. This is important as it compares runtime data across chips, inputs and applications. However, at the finer-granularity when the chip, input and application is fixed, a confidence interval around the mean is used to determine if a speedup is confidence or not. This is valid, as the runtime data collected in these comparisons is simply from repeat runs in same environment.

4.4.3 GPU platforms tested

Table 2.2 summarises the GPU platforms used in the experimental evaluation of this chapter: 6 GPUs spanning 4 vendors. The AMD R9 and the Nvidia GPUs are discrete, all others are integrated. GPUs from the same vendor also span different architecture configurations. The Nvidia M4000 and GTX1080 belong to the Maxwell and Pascal architectures respectively. The HD5500 and IRIS are both Broadwell architecture, but at different graphics tiers (as described by Intel); tiers GT3 and GT2, respectively, where GT3 is the top tier for this architecture. They also differ in the number of compute units, the size of subgroups, and the highest OpenCL version supported. Given these differences, the GPUs considered in this chapter represent a compelling landscape over which to examine functional and performance portability.

Closing the functional gap Table 2.2 shows that none of the GPUs examined in this chapter support OpenCL 2.1.[1] Recall from Section 4.3 that the complete set of graph algorithm optimisations requires an implementation to support OpenCL 2.1 and provide forward progress guarantees, i.e. to the OCL FP feature class described in Section 2.3.1. In order to facilitate the full range of optimisations, GPU-specific workarounds were devised to provide the needed features, using facilities available in earlier OpenCL versions to model future version features as faithfully as possible, allowing all GPUs of this chapter to support the OCL FP feature class.

Section 2.4 describes how ARM and Nvidia chips were brought to OCL 2.0 from OCL 1.x by a best-effort OpenCL 2.0 memory model implementation for each chip. Indeed, recall that this functionality was also required for the experimental study in Chapter 3.

[1] In fact, we are not aware of any GPU that supports OpenCL 2.1 at the time of writing.

Table 4.2: The applications considered along with their available optimisations. Application variants considered state-of-the-art are noted with a (*). Applications with no variants have only one implementation.

App	Variants	Available opts	Citation
BFS	cx	sz256, np, coop-cv, oitergb	[MGG12b]
	topo	sz256, np, coop-cv, oitergb	[BNP12]
	tp	sz256, np, coop-cv, oitergb	[MGG12b]
	wl	sz256, np, coop-cv, oitergb	[MGG12b]
	hybrid*	sz256, np, coop-cv, oitergb	[MGG12b]
CC	*	sz256, np, coop-cv, oitergb	[SKN10]
MIS	worklist	sz256, coop-cv, oitergb	[Lub86]
	pannotia*	sz256, coop-cv, oitergb	[CBRS13]
MST	boruvka (single worklist)	sz256, coop-cv	[BNP12]
	boruvka (multi worklists)*	sz256, coop-cv	[BNP12]
PR	residual	sz256, np	[WLDP15]
	residual-wl*	sz256, np, coop-cv	[WLDP15]
	tp	sz256, np	[PP16]
SSSP	nf*	sz256, np, coop-cv, oitergb	[DBGO14]
	topo	sz256, coop-cv, oitergb	[BNP12]
	wl	sz256, np, coop-cv, oitergb	[BNP12]
TRI	*	sz256, np	[Pol16]

To bring all chips up to the OCL 2.1 feature class, support for *subgroups* is required. On AMD, the Khronos subgroup extension mirroring the OpenCL 2.1 subgroup functionality [Khr16b, pp. 133-140] is available. Intel GPUs likewise support a specialised subgroup extension that provides the functionality required by the optimisations [Ash16]. On Nvidia, inline PTX provides warp intrinsic [Nvi18e, p. 209], which are Nvidia's own subgroup implementation. The architecture of the ARM GPU does not feature subgroups [Lok11], so a subgroup size of one is used, which is a valid subgroup size. This nullifies any performance benefit that might be obtained from subgroup optimisations, but allows OpenCL kernels generated by optimisations that assume subgroups to run in a functionally-portable manner on ARM.

For OCL FP support, Chapter 3 has demonstrated that GPUs from the four vendors considered do support the occupancy-bound execution model in practice. Thus, this satisfies the requirements of OCL FP directly.

4.4.4 Benchmarks

The IRGL compiler is accompanied by 19 graph applications, of which 17 were used in this work. Delaunay Mesh Refinement (DMR) is not used, as some of its (large) support libraries are written in CUDA. SSSP with a priority worklist is not used, as it uses a high-performance key–value GPU sort from ModernGPU [Bax], for which an an equivalent OpenCL library was not found. The applications can be split into 7 high-level problems – breadth-first search (BFS), connected components (CC), maximal independent set (MIS), minimal spanning tree (MST), pagerank (PR), single-source shortest path (SSSP), and triangle counting (TRI). Each problem has multiple implementation strategies, summarised in Table 5.2, the strategies marked (*) implement the fastest algorithms, as discussed in [PP16]. The "available opts" column shows which optimisations apply to an implementation; variants with all possible combinations of these optimisations were generated for the experiments. The "citation" column points to the work where the application variant was originally presented.

It is noted that $\text{SSSP}_{\text{topo}}$ uses a similar strategy to BFS_{topo}, however, the `np` optimisation is not available for $\text{SSSP}_{\text{topo}}$. This is due to an unresolved issue in the compiler implementation, and in principle `np` should be applicable to $\text{SSSP}_{\text{topo}}$. More engineering effort is planned for the compiler, which should address this issue.

Each application is accompanied by a checker that can be used to validate executions. BFS, CC, MST, SSSP, and TRI applications must produce bit-exact results every time. MIS applications, while non-deterministic, must produce solutions that satisfy key invariants that were checked by a script provided with IRGL. Finally, although PR applications exhibit floating point variance, their relative differences from a high confidence oracle must be within range as checked using the `numdiff` utility.

Three graph inputs from three important classes were used in this work: a road network graph of New York (`usa.ny`), a uniformly random graph (`2e23`), and a RMAT-style graph (`rmat20`). The `usa.ny` road network input is a high-diameter, uniform low degree graph. The `2e23` input is a synthetic random graph with uniformly distributed edges. The input `rmat20` is also a synthetic graph, but a recursive procedure is used to construct it, so that vertex degrees exhibit a power-law distribution [CZF04]. Bi-directional versions of these graphs were constructed for problems like MIS and MST that require undirected graphs. MST is not run on `2e23` due to an undiagnosed error.

4.4.5 Gathering data

For each execution, the time it takes to execute the application/input on the GPU was recorded, as this is the primary performance characteristic of interest. As in the original applications, the time taken to load the graph inputs and the initial/final transfers of graph data to and from the GPU were ignored.

Each test was run three times. From the runtime data, the average and 95% confidence intervals [Jai91] were gathered. About 20% of the collected data have confidence intervals that exceed 15% of the average. These appear to be random outliers, but nevertheless are included in the analysis directly. More runs would likely reduce the noise, however, as mentioned below, the total experimental runtime is high and more runs would not be feasible.

As is common in such large experimental runs, several failures during the experimental campaign were experienced. First, a small number of runs (typically less than five out of ten thousand) failed without producing any output. In these cases, the experiments were simply re-run and the issue did not occur in the repeated runs.

Second, on ARM, 130 benchmarks/optimisation configurations pairs failed due to the device not being able to support a workgroup size of 256. This is valid OpenCL behaviour, although it was only observed on this device. In these cases, the sz256 optimisation is considered unavailable. Third, a variety of vendor compiler issues were encountered, similar to issues described in the work for Chapter 3 and presented in [SD16b]. For these issues, workarounds were developed in which parts of the code were transformed in a semantically equivalent way. Typically, the transformed code uses variables that the compiler cannot know statically, e.g. kernel arguments. For example, it was found that `while(true)` loops that used an explicit `break` condition commonly caused compiler issues. A simple transformation changing the loop to `while(one != 0)` where `one` is a kernel argument that the host always sets to `1` resolved many of the issues.

In total, across all platforms and benchmarks, the experimental run takes approximately 237 hours. The platform that accounts for the largest amount of this runtime is Mali-4, taking 97 hours as the GPU and CPU are smaller and slower devices. Specifically, Mali-4 has a clock rate of 533 MHz and has 4 compute units. Out of the chips used in this chapter, the GPU with the next lowest clock rate is the M4000 with a clock rate of 772 MHz, but it has 13 compute units. The GPU with the lowest occupancy is the Hd5500 with an occupancy of 3 using maximum resources (see Table 3.2); however it has a clock rate of 950 MHz.

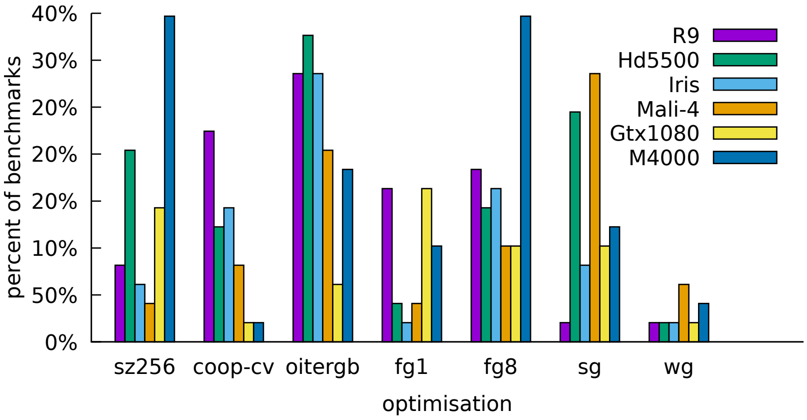


Figure 4.1: Summary of optimisations used per-chip to obtain the top speedups.

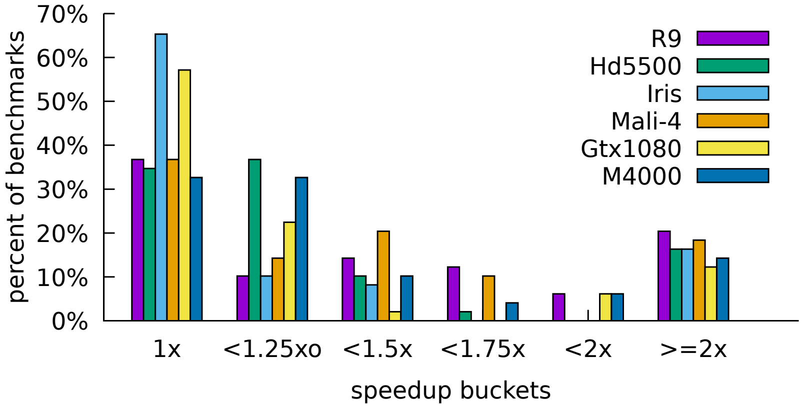


Figure 4.2: Summary of top speedups for all benchmarks, split by chip.

4.5 Results and discussion

The research questions of the chapter introduction are now explored by mining the experimental data.

4.5.1 Top speedups across architectures

Research question *Are optimisations for GPU graph applications, developed in previous work and targeted to Nvidia GPUs, beneficial for GPUs from other vendors? How do the top speedups and slowdowns compare across GPUs and how does the distribution of optimisations required for top speedups vary across GPUs?*

Table 4.3: The maximum speedup and slowdown per chip and associated application. The input for each is `usa.ny`.

	R9	**Hd5500**	**Iris**	**Mali-4**	**Gtx1080**	**M4000**
max speedup	14.61×	16.61×	13.25×	3.95×	5.10×	3.54×
app	$\text{BFS}_{\text{hybrid}}$	SSSP_{nf}	BFS_{tp}	BFS_{tp}	SSSP_{nf}	SSSP_{nf}
max slowdown	6.89×	22.15×	18.70×	15.21×	7.99×	10.00×
app	SSSP_{wl}	PR_{tp}	PR_{wl}	SSSP_{wl}	PR_{wl}	PR_{tp}

First, the effectiveness of the optimisations was assessed, previously tested only on Nvidia GPUs [PP16], across a range of chips from multiple vendors. Figure 4.1 shows a histogram of optimisations that were necessary for the top speedup for a benchmark per chip. By "necessary", it is meant that if the optimisation is disabled, then the benchmark suffers a slowdown from the top observed speedup. The figure shows that each optimisation is necessary to achieve a top speedup for at least one benchmark across all chips. Thus, these optimisations are not specific to Nvidia GPUs.

Second, these optimisations are shown to also lead to generous speedups across chips. The value of top speedups achieved are distributed similarly across chips as shown in the histogram of Figure 4.2. The "1×" bucket counts benchmarks for which no (confident) speedup was obtained; the "$< S\times$" bins count the number of benchmarks for which the best speedup was less than $S\times$ but greater than or equal to the maximum speedup associated with the previous bin; the final bin counts the benchmarks for which a top speedup of at least 2× was achieved.

For most chips, the percentage of benchmarks that could not be sped up is less than 37%. However, speedups appear to be more difficult to obtain for Iris and Gtx1080, for which no speedups were observed for 65% and 57% of benchmarks, respectively. Notably, these "difficult" chips are from *different* vendors, and other chips from the *same* vendors (Hd5500 and M4000) exhibit significantly fewer results in this category (i.e. applications that did not exhibit speedups).

The other extreme of the histogram, the ">= 2×" bin, shows that larger speedups were observed for between 12% and 21% of benchmarks across all chips. Thus, while not all chips were able to obtain speedups on the same percentage of benchmarks, all chips were able to observe more than a 2× speedup on a similar percentage of benchmarks.

Table 4.3 shows the maximum speedup and slowdown per chip and the associated benchmark; the relevant input turns out to be `usa.ny` in all cases. R9, Iris and Hd5500 achieve speedups of more than 10× in the best case; the speedups for Mali-4, Gtx1080

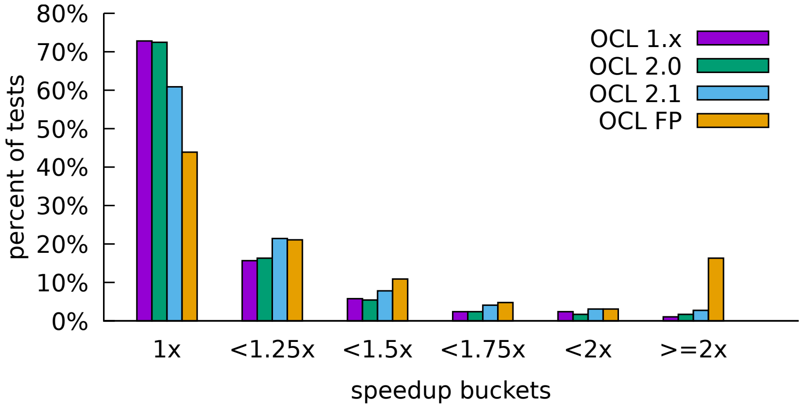


Figure 4.3: Summary of top speedups over all tests, grouped by feature class.

and M4000 are more modest, never exceeding 6×. The only commonly-enabled optimisations for these top speedups is oitergb; the reasons for this are investigated through microbenchmarking in Section 4.5.5. With respect to maximum slowdowns, all chips can suffer at least a 7× slowdown in the worst case. Thus, optimisation choices can have a significant effect on the runtime of benchmarks across chips.

4.5.2 Top speedups for feature classes

Research question *How do top speedups fare when functional portability is reduced? That is, how much performance do newer, less supported, features enable across benchmarks and chips?*

The focus is now turned to assessing the extent to which the availability of the latest, or even possible next-generation, features of GPU programming models impacts on performance across the benchmarks and chips. The histogram of Figure 4.3 shows top speedups across all tests when optimisations are limited to their minimum required feature class (see Table 4.1). For example, values corresponding to OCL 1.x are limited to the fg and sz256 optimisations and OCL FP values are free to use any optimisation.

These results show clearly that recent and next-generation programming model features matter for performance in this domain. More than 70% of tests show no speedups when optimisations are limited to the OCL 1.x and OCL 2.0 feature classes; and 94% of tests in this class have less than a 1.5× associated speedup. Optimisations from the OCL 2.1 class, which adds subgroup support, improve the situation, reducing the tests for which no speedup is achieved to 61%. The majority of the largest speedups, that is speedups of 2× or larger, feature the oitergb optimisation, which is unique to the OCL FP class.

Table 4.4: The top and bottom five universal optimisation strategies ranked according to the number of tests slowed down.

Enabled optimisations	Slowdowns	Speedups	Geomean speedup
fg8	36	60	1.01
fg	37	58	0.98
wg, fg8	38	61	1.00
wg, fg	41	55	0.96
sg	41	56	1.00
wg, coop-cv, oitergb	157	44	0.72
sz256, wg, oitergb	167	44	0.61
sz256, wg	173	23	0.60
sz256, wg coop-cv, oitergb	189	35	0.54
sz256, wg coop-cv	195	22	0.53

In summary, support for cutting edge OpenCL features leads to significant speedups across the test set. These results offer a compelling motivation for vendors to efficiently support these language-level abstractions. Additionally, the forward progress guarantees required by OCL FP are not officially supported by any vendor; again, these results provide a compelling case for official support to be considered, e.g. via an OpenCL extension.

4.5.3 Applying optimisations universally

Research question *Are there "portable optimisations"; that is, optimisation settings that provide better than baseline performance across the board?*

Having established that optimisations are portable, it is now explored whether any optimisation configuration produces speedups *universally*, i.e. across all tests. Such a configuration would yield an appealingly simple optimisation strategy. To answer this, all 95 optimisation configurations were enumerated, where at least one optimisation is enabled. For each test, the runtime for every optimisation configuration is compared with the runtime of the baseline. If an optimisation o in an optimisation configuration is not applicable to an application (see Table 5.2), then the runtime of the test using the optimisation configuration without o enabled is used.

Every optimisation configuration yields a slowdown for at least one test. Table 4.4 shows the top and bottom optimisation configurations, ranked by the number of tests slowed down by the configuration. The configuration that causes the fewest slowdowns (36; 12% of all tests) applies fg8 in isolation. However, this "least harmful" optimisation can be seen as low-risk and low-reward: it is in the OCL 1.x feature class, which provides very

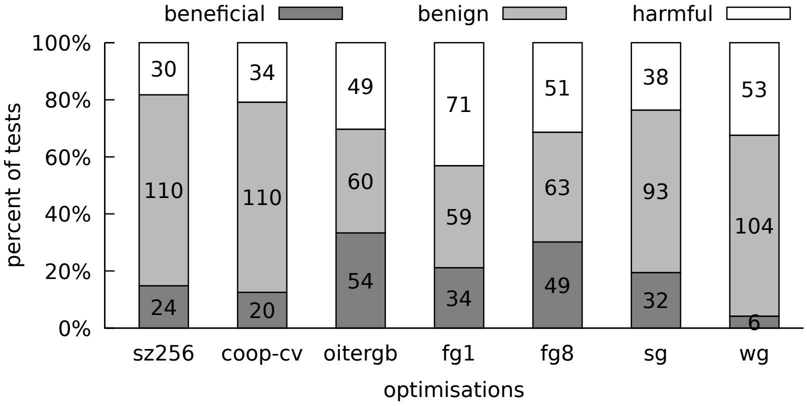


Figure 4.4: Percent of tests for which each optimisation is beneficial, benign, or harmful.

limited speedups in general (see Section 4.5.2), and applying this optimisation universally provides only a 1.01× geomean speedup, compared with a 1.5× geomean speedup for the oracle strategy.

At the other extreme, the combination of sz256, wg, and coop-cv causes the largest number of slowdowns (195; 66% of all tests), leading to a 0.53× geomean speedup overall (i.e. nearly a 2× slowdown). However, this configuration should not be discounted entirely: it yields a 1.7× speedup for the (GTX1080, sssp-nf, `rmat20`) test, and a maximum observed speedup for 2.3% of all tests. Thus, while this configuration appears largely harmful, there are a small number of cases where it is beneficial and even optimal.

These results show that achieving performance portability universally in this domain is very difficult: *every* optimisation configuration leads to a slowdown for at least one test.

4.5.4 Optimisations in top speedups

Research question *Are certain optimisations always, or commonly, required for top speedups across the board? If not, what is the distribution of optimisations required for top speedups?*

Since no optimisation combination avoids slowdowns completely, optimisations that contribute to top speedups are now examined. For each test, the optimisation configuration C that led to the top observed speedup is found. This top speedup is then compared with the speedups observed when optimisations are individually toggled.

If disabling an optimisation o that is enabled in C leads to a slowdown over C, o is marked as *beneficial* for the test. If enabling an optimisation o that is disabled in C leads to a slowdown over C, o is marked as *harmful* for the test. An optimisation that is neither beneficial nor harmful for a test is marked as *benign*.

Figure 4.4 summarises the extent to which the optimisations are beneficial, benign, or harmful, across all tests; cases where an optimisation is not applicable for a given test (see Table 5.2) were excluded. Nearly all optimisations are harmful for at least 18% of tests, and most optimisations are harmful in more cases than they are beneficial. The most harmful optimisation is fg1, harmful for 43% of tests; the most beneficial are oitergb and fg8, beneficial for 33% and 30% of tests, respectively; the least beneficial is wg, benefiting just 4% of tests.

Indeed, all optimisations that are present in top speedup configurations for some tests cause slowdowns in top speedup configurations of other tests. Thus, there is no optimisation that can be safely enabled (or disabled) in a majority of cases to achieve top speedups.

4.5.5 Portable optimisations

Research question *What are the performance consequences of abandoning portability by degrees and specialising to chips, applications or inputs across benchmarks in this domain?*

The results of Section 4.5.4 show that the effects of optimisations vary substantially across tests, thus deriving optimisation strategies from the runtime data is challenging. The results of applying the analysis of Section 4.4.2 to the runtime data for various degrees of portability is now presented. This allows an investigation into the trade-offs whereby performance improves as portability constraints are reduced. High-level results relating to portability vs. specialisation are shown for the test set, considering three dimensions of specialisation: chip, application, and input.

To specialise over a dimension d, the tests are partitioned into distinct subsets where all tests in a subset have the same value for d. Each subset is then assigned an optimisation configuration using the analysis Section 4.4.2. In the present analysis, all possible dimensions were considered: *chip*, *app(lication)*, *input*, *chip/app*, *app/input*, *chip/input*. The *global* strategy employs no specialisation (and hence no partitions). For the *oracle* strategy that is specialised to a test (i.e. all three basic dimensions), a simple search through the experimental data for the maximum speedup was performed.

The effects of specialisation The discussion starts with Figure 4.5, which shows for each optimisation strategy the percentage of tests that exhibited a speedup, slowdown or

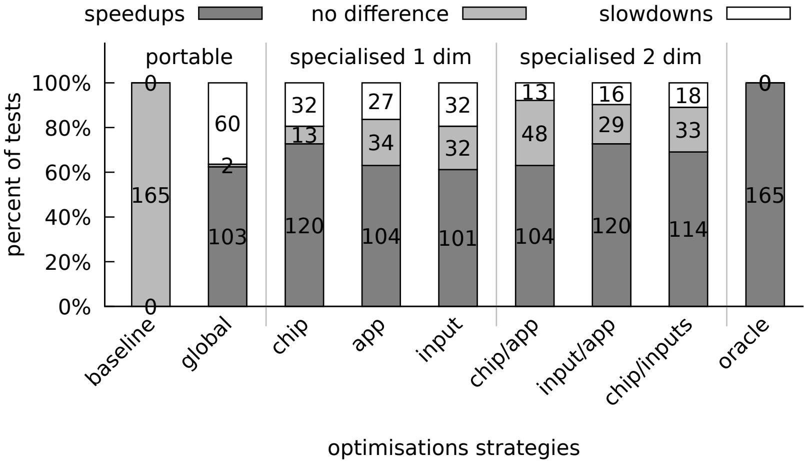


Figure 4.5: The percentage of tests for each optimisation strategy that provided a speedup, no difference or slowdown. Concrete test counts are given on the bars.

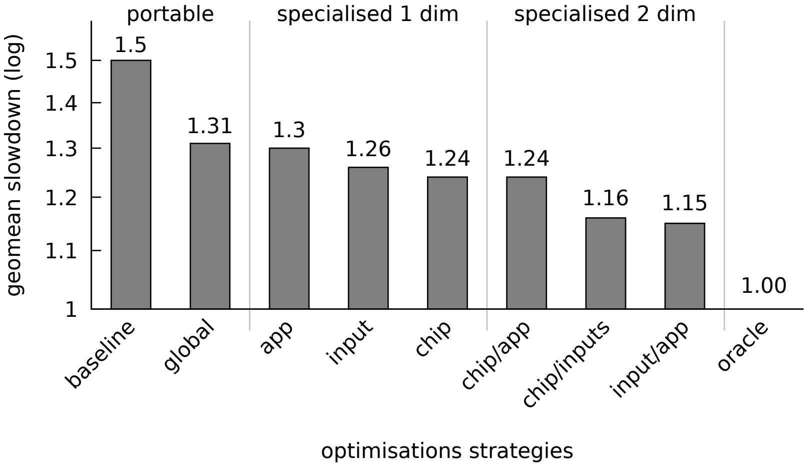


Figure 4.6: The geomean slowdown compared to the oracle across all tests for different optimisation strategies. Concrete test counts are given on the bars.

no significant change under the optimisation strategy. In this chart, the 43% of tests for which confident speedups were not observed are excluded (see Figure 4.3). As a result, the baseline strategy shows no difference on all tests and the oracle strategy shows speedups on all tests.

The completely portable strategy (global) provides a speedup on 62% of tests and a slowdown on 18% of tests. Each additional dimension of specialisation halves the number of slowdowns. The count of speedups roughly remains in the same order of magnitude between a global optimisation strategy up to (but not including) the oracle strategy.

While Figure 4.5 shows the number of speedups and slowdowns, it does not measure the magnitude of the runtime differences between optimisation strategies. To illustrate runtime magnitude differences, the geomean under the oracle strategy and the geomean under each optimisation strategy across all tests was computed. Figure 4.6 shows for each strategy the strategy geomean normalised to the oracle geomean across all tests. Thus, the value shows the average slowdown per test under the strategy against the best observed. The baseline strategy shows a geomean of 1.5× slowdown, while the oracle strategy shows no slowdown.

The following observations can be made from Figures. 4.5 and 4.6: the optimal single dimension to specialise across for speedups is *chip*, which provides 120 speedups as opposed to 104 and 101 for *app* and *input* respectively. Additionally, the geomean slowdown is 1.24× as opposed to 1.3× and 1.26× for *app* and *input*. Specialising for application gives the fewest slowdowns but also the largest mean slowdown.

The optimal two dimensions to specialise across for speedups is inputs and applications, with 120 tests showing a speedup with a geomean slowdown of 1.15×. While the single dimension chip specialisation gives the same number of speedups, it has twice as many slowdowns (16 vs. 32). The optimal two dimensions for the fewest slowdowns is applications and architecture, with 13 slowdowns. Much like the single dimension, this optimisation strategy also has the largest geomean slowdown at 1.24×. Interestingly, this is the same geomean slowdown as the optimisation strategy only across chips. This suggests that the chip and application dimensions do not synergise well.

4.5.6 Optimisations and features of chips, applications and inputs

Research question *Can optimisation strategies specialised per chip deliver insights into performance-critical architectural, application, or input features?*

Here the optimisation strategies produced by the analysis are explored. In particular, the global strategy and the single dimension (chip, input, application) specialisation strategies are considered, shown in Table 4.5. The optimisations are studied to reveal details about the domain of specialisation, e.g. why the strategies for the R9 and Iris chips enable coop-cv while the other chips do not. Optimisations for combined dimensions (e.g. *chip* and *input*) are not shown as the optimisations can be seen as the combination of information from the two constituent dimensions.

First, universal properties of Table 4.5 are briefly discussed. In particular, no strategy shown ever enables the sz256, fg1 or wg optimisation. Thus, the information available at these dimensions of specialisation is not enough for the analysis to ever enable these optimisations. For wg, this observation is consistent with earlier results. That is, Figure 4.4 shows that wg is only required in 6 tests to achieve the top speedup; the lowest of any optimisation. This is not the case for sz256 or fg1 and it has not yet been diagnosed why these optimisations are never enabled from the analysis. Nevertheless, again as shown by Figure 4.4, sz256, fg1, and sz256 are required in some cases to observe some top speedups; however, the analysis here does not shed light on these situations.

Global optimisations

As seen in Table 4.5, the global strategy enables the optimisations oitergb, fg8 and sg. This result is consistent with Figure 4.4; namely, these three optimisations have the highest percentage of tests for which they are beneficial. The exception is that the fg1 optimisation has a larger percentage than sg, however fg1 also has a large percentage of tests for which it is harmful; this is likely why the analysis did not enable fg1. While it is shown in Section 4.5.3 that no global strategy avoids slowdowns completely, Figure 4.5 shows that this global strategy gives a speedup on over half of the tests. Figure 4.6 shows that while this strategy still suffers a considerable average slowdown over the oracle strategy (1.31×), the global strategy gives a considerable mean speedup over the baseline strategy. Thus, while the analysis can produce specialised models, it appears to be capable of discovering effective global strategies as well.

Chip-specific optimisations

Here, the following observations of the chip-specialised strategy Table 4.5 are explored: (1) the two Nvidia chips are the only two chips that disable oitergb; (2) Iris and R9 uniquely enable coop-cv; and (3) Mali-4 enables sg even though it does not have subgroups. The observations are each examined using microbenchmarks, revealing interesting performance features of the chips.

Table 4.5: The optimisations enabled by the analysis for the following levels of portability: global, per-chip, per-app, and per-input.

Strategy	**sz256**	**coop-cv**	**oitergb**	**fg1**	**fg8**	**sg**	**wg**
global			✓		✓	✓	
chip							
R9		✓	✓		✓	✓	
Hd5500			✓		✓	✓	
Iris		✓	✓		✓	✓	
Mali-4			✓			✓	
Gtx1080					✓	✓	
M4000					✓	✓	
input							
`usa.ny`			✓		✓		
`rmat20`					✓	✓	
`2e23`					✓	✓	
app							
BFS_{cx}		✓			✓	✓	
BFS_{topo}			✓				
BFS_{tp}		✓			✓	✓	
BFS_{wl}			✓		✓	✓	
$\text{BFS}_{\text{hybrid}}$					✓	✓	
CC					✓	✓	
$\text{MIS}_{\text{worklist}}$							
$\text{MIS}_{\text{pannotia}}$							
$\text{MST}_{\text{single-wl}}$							
$\text{MST}_{\text{multi-wl}}$							
$\text{PR}_{\text{residual}}$					✓	✓	
$\text{PR}_{\text{residual-wl}}$					✓	✓	
PR_{tp}					✓	✓	
SSSP_{nf}			✓		✓	✓	
$\text{SSSP}_{\text{topo}}$			✓				
SSSP_{wl}			✓		✓	✓	
TRI					✓	✓	
Strategy	**sz256**	**coop-cv**	**oitergb**	**fg1**	**fg8**	**sg**	**wg**

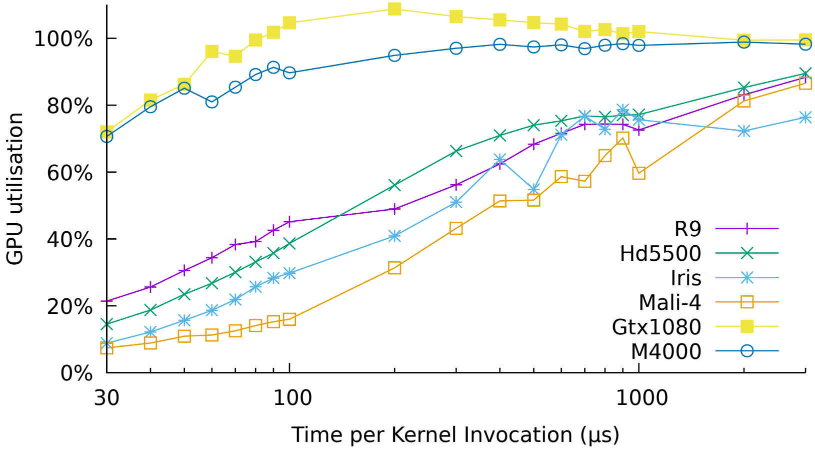


Figure 4.7: Results of the kernel launch frequency microbenchmark per chip.

Overhead of kernel launches and memory copies The oitergb optimisation is enabled by all chips except those from Nvidia. To establish a causal relationship, a microbenchmark (similar to the one used in previous work to motivate the optimisation on Nvidia architectures [PP16]) was developed and run across the GPUs of this chapter. Essentially, the microbenchmark launches a constant-time kernel a fixed number of times (10, 000), interleaving the launches with a memory copy of a single integer from the GPU to the CPU. The memory copy between kernel launches mimics the loop dependence that oitergb moves to the GPU. The constant-time kernels establish the exact utilisation of the GPU, so timing the entire procedure reveals the overhead of launching these kernels and of the memory copies that oitergb is designed to reduce. OpenCL does not provide device timers like CUDA's `clock64`, so a calibration loop is used to approximate constant-time kernels and therefore the results are somewhat noisy.

Figure 4.7 shows the utilisation of the GPU as the kernel execution time is varied. For a given kernel time, Nvidia chips have relatively higher utilisation compared to other chips, implying that they have the lowest launch and memory copy latencies. The kernel launch and memory copy overheads are sufficiently higher for all other chips that they must include oitergb for performance. Note that oitergb is used by some benchmarks on Nvidia GPUs (Figure 4.1), but for fewer benchmarks than other chips.

Subgroup RMW combining The coop-cv optimisation is enabled for R9 and Iris (but not the other Intel chip, Hd5500). The most common form of this optimisation aggregates atomic RMW instructions within a subgroup. To investigate why this optimi-

Table 4.6: Microbenchmark results for subgroup atomic combining and workgroup memory divergence.

	R9	**Hd5500**	**Iris**	**Mali-4**	**Gtx1080**	**M4000**
sg-combine	**22.10**	.98	**7.95**	1.06	.88	.97
m-divergence	1.04	1.07	1.08	**6.45**	1.27	1.08

sation is only turned on for a few architectures, an OpenCL microbenchmark was written to measure the time for N `atomic_fetch_and_add` invocations on a single memory location (here, $N = 20,000$). In a separate microbenchmark, all atomics in the subgroup are combined into one atomic (mimicking coop-cv), thus potentially improving throughput by the size of the subgroup. The *sg-combine* row of Table 4.6 shows the speedup of this coop-cv microbenchmark version over the original.

The speedups from R9 and Iris, the two chips for which the analysis suggest the coop-cv optimisation, are notably higher than values for the other chips. The overhead of subgroup communication for combining causes the speedups to be a fraction of the subgroup size – R9 has a subgroup size of 64, but sees only a 22× speedup. Iris uses a subgroup size of 16 for these kernels, and delivers about half of that as speedup.

The Mali-4 chip has a subgroup size of 1, and does not show speedup as expected. The Nvidia chips do not exhibit speedup, but investigation has shown that the CUDA compiler already performs this optimisation natively.[2] It is suspected this is also the case for the Nvidia OpenCL compiler. Stranger is that coop-cv is enabled for Iris but not Hd5500, both from Intel. The compiled GPU assembly is not available in a documented form for these chips and thus, OpenCL compiler optimisations cannot be explored. It is speculated that coop-cv occurs in the OpenCL compiler similar to Nvidia GPUs, but it is not clear why it does not occur on Iris.

Intra-workgroup memory divergence Since the Mali-4 has a subgroup size of 1, it is intriguing to investigate why the sg optimisation is enabled for it. Recall that sg improves load balance by redistributing work over all threads from the same subgroup. By careful elimination, it was eventually discovered that the *workgroup barriers* that are a part of the sg optimisation were the source of the speedup. Previous work [LLK^{+}14] has found that semantically unnecessary workgroup barriers can improve performance in GPU kernel execution by limiting the memory divergence of threads in the same workgroup.

[2] From personal communication with Sreepathi Pai.

Table 4.7: Kernel launches and worklist sizes for different inputs when running BFS_{wl}.

Input	Kernel launches	Average worklist size
`usa.ny`	621	606.5
`rmat20`	14	65155.2
`2e23`	19	466083.4

Again, a microbenchmark was developed to test this by having two kernels read and write to a large array using strided accesses indexed by thread id. In one of the kernels, a semantically unnecessary barrier is introduced into the loop, so that threads in a workgroup are never more than one iteration away from each other. The speedup of the kernel with barriers over the kernel without barriers across all chips is shown in the *m-divergence* row of Table 4.6.

While all chips appear to benefit from the barrier, the clear outlier is the MALI-4, on which adding the gratuitous barrier leads to an impressive 6.45× speedup. Thus, the MALI-4 appears to be extremely sensitive to intra-workgroup memory divergence, Figure 4.1 shows that sg is required for top speedups on MALI-4 more than any other chip. Thus, while initially confounding, these findings suggests a new optimisation may be required to protect against memory divergence on all GPUs.

Input-specific optimisations

The different optimisation configurations for the input strategy from Table 4.5 are now examined. Namely, that `usa.ny` enables oitergb and disables sg.

Recall from Sections 4.3.3 and 4.5.6 that oitergb is beneficial in the presence of many short kernels being launched iteratively. Graphs with a high diameter, such as `usa.ny`, tend to lead to many short kernels. The graphs `2e23` and `rmat20` have lower diameters and thus have the opposite property: a small number of longer running kernels. For example, Table 4.7 shows the number of iterations as well as the average size of the input worklist for each input when running the BFS_{wl} application. The number of items in the worklist can be used as an estimate of the time each kernel launch takes. Notice that `usa.ny` causes at least 30× more iterations than the other two inputs. Additionally, on average there is at least 100× less work per kernel. Because of this property, `usa.ny` is able to benefit from oitergb regardless of chip or application.

The sg optimisation, disabled for `usa.ny`, is reasoned about similarly. The distribution optimisation of sg balances the worklist items. If there are not many items in the worklist, then the overhead of the distribution orchestration outweighs the benefits. Again, Table 4.7

Table 4.8: Average worklist sizes for three variants of BFS across different inputs.

App	`usa.ny`	`rmat20`	`2e23`
$\mathrm{BFS}_{\mathrm{wl}}$	606.5	65155.2	466083.4
$\mathrm{BFS}_{\mathrm{cx}}$	3698.5	590010.9	1766063.3
$\mathrm{BFS}_{\mathrm{tp}}$	3661.4	627411.1	1864174.7

shows that the average number of items in the worklist for `usa.ny` is much smaller than the other two inputs, and is small enough that sg is disabled.

Application-specific optimisations

To complete the discussion, a few observations on application-specific optimisations shown in Table 4.5 are explored. It should be borne in mind that not all optimisations are applicable to all applications (Table 5.2), thus optimisations that are not applicable for an application will never be enabled for that application. Four high-level observations about application specific optimisations can be made.

The first observation is that coop-cv is enabled only for $\mathrm{BFS}_{\mathrm{cx}}$ and $\mathrm{BFS}_{\mathrm{tp}}$. The reason for this is that these two variants of BFS employ an *optimistic* worklist strategy, where all candidate nodes in a frontier are pushed to the worklist. The other variants, e.g. $\mathrm{BFS}_{\mathrm{wl}}$, only push nodes that have not yet been examined. The optimistic strategy can be beneficial due to streaming memory accesses as their labels do not have to be checked before pushing. However, the optimistic strategy also leads to many more worklist pushes, and thus atomic RMW operations. This can be seen by the average worklist sizes presented in Table 4.8; notice how the tp and cx variants have rougly 5× the number of items as the wl variant. Because of this, aggregating atomics is especially valuable in these applications.

The second observation is that for all applications for which np is applicable, only $\mathrm{BFS}_{\mathrm{topo}}$ disables fg8 and sg. This is because the topo applications use a strategy in which the input worklist logically contains all nodes in the graph. These nodes are partitioned across threads. If a thread finds that a node is not important for the current iteration, the thread is able to simply exit. However, if an np optimisation is used, then all threads must continue executing, including participating in barrier synchronisation. Because fg8 and sg only distribute work across across subgroups, there may be many subgroups without meaningful work. These threads then simply add overhead, making fg8 and sg unattractive for this strategy. Because $\mathrm{SSSP}_{\mathrm{topo}}$ implements a similar strategy, it is speculated that the analysis would also disable fg8 and sg for this application as well. However, as discussed

Table 4.9: Number of kernel launches and average time (ms) per kernel for HD5500.

App	`usa.ny`		`rmat20`		`2e23`	
	launches	avg. time	launches	avg. time	launches	avg. time
SSSP_{nf}	2407	.03	60	13.77	80	31.25
$\text{MIS}_{\text{pannotia}}$	17	1.65	8	25.00	17	165.47

in Section 4.4.4, the np optimisations are not available for $\text{SSSP}_{\text{topo}}$ due to an unresolved issue in the compiler implementation.

The third observation is that oitergb is mixed across the applications. While not all cases are examined here, the two applications with the highest and lowest *common language effect size*, i.e. the percent of observations for which the optimisation caused a performance improvement, for oitergb in the MWU analysis. The application with the highest common language effect size is SSSP_{nf}, in which oitergb caused a speedup in 89% of 481 total samples. The application with the lowest common language effect size is $\text{MIS}_{\text{pannotia}}$, in which oitergb caused a speedup in only 20% of 44 samples. There are fewer samples for $\text{MIS}_{\text{pannotia}}$ because the np optimisations are not available.

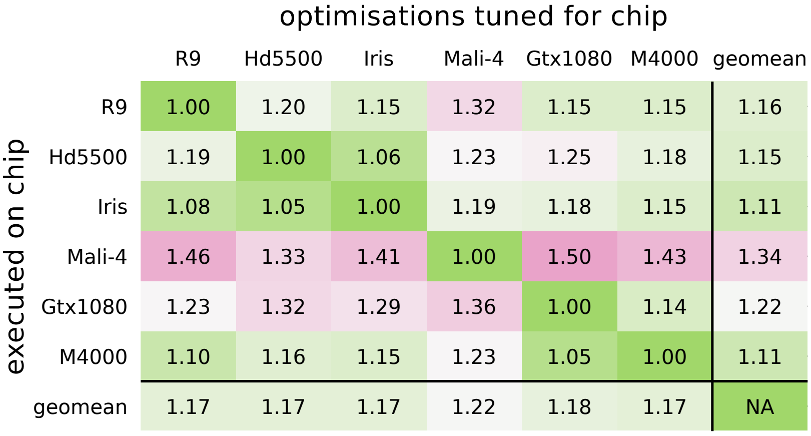
As discussed throughout, oitergb benefits cases where there are many short kernel launches. Table 4.9 shows the number of kernel launches for both applications, as well as the average time per kernel on HD5500. Notice that SSSP_{nf} always has at least 4× the number of kernel launches for each input compared with $\text{MIS}_{\text{pannotia}}$. Additionally, the average kernel time is roughly 55×, 2×, and 5× less for SSSP_{nf} than for $\text{MIS}_{\text{pannotia}}$ for the three inputs. As a result, SSSP_{nf} is a better candidate for oitergb than $\text{MIS}_{\text{pannotia}}$.

The final observation is that MIS and MST applications do not enable *any* optimisations. Thus, these applications likely do not have chip or input independent features that optimisations can exploit.

4.5.7 Optimisation policies

Research question *How does an optimisation policy guided by tuning on single GPU platform fare when used to make optimisation decisions for other GPU platforms?*

The effect of taking an optimisation strategy tuned for a given chip and applying it to all chips is now discussed. Such situations may arise if optimisations strategies are specialised for one chip, e.g. a chip that is readily available to a developer, and then ported new chip, e.g. a chip that is unavailable to a developer. For this, a new optimisation strategy per chip C is generated, which given a benchmark b, returns the optimisation configuration that achieved the highest speedups for C on b. These strategies were generated through an



	R9	Hd5500	Iris	Mali-4	Gtx1080	M4000	geomean
R9	1.00	1.20	1.15	1.32	1.15	1.15	1.16
Hd5500	1.19	1.00	1.06	1.23	1.25	1.18	1.15
Iris	1.08	1.05	1.00	1.19	1.18	1.15	1.11
Mali-4	1.46	1.33	1.41	1.00	1.50	1.43	1.34
Gtx1080	1.23	1.32	1.29	1.36	1.00	1.14	1.22
M4000	1.10	1.16	1.15	1.23	1.05	1.00	1.11
geomean	1.17	1.17	1.17	1.22	1.18	1.17	NA

Figure 4.8: Heatmap of the geomean slowdown compared to the oracle of executing an optimisation configuration tuned for one chip on all other chips.

exhaustive search through the data, similar to the oracle strategy. For all pairs (C, C') of chips, the runtimes for C using optimisation strategies tuned for C are compared against the runtimes using optimisation strategies tuned for C'. As a metric, the geomean slowdown over the oracle strategy is used (similar to Figure 4.6). That is, what is the average slowdown across all benchmarks restricted to chip C' if using an optimisation strategy tuned for chip C.

Figure 4.8 shows a heatmap of these results, where the rows correspond to chips running the benchmarks and the columns correspond to optimisation strategies tuned for individual chips. The diagonal is 1.00 as there are no slowdowns for a chip using the optimisation strategy tuned for that chip. The values of the bottom row shows the geomean across all values in the column associated with an optimisation strategy. Smaller values indicate a given optimisation strategy is more portable, causing fewer slowdowns across chips. The numbers of the far right column show the geomean slowdown across the row associated with the chip. Smaller values indicate a given chip suffers fewer slowdowns under optimisation strategies tailored for different chips.

No chip optimised strategy is completely portable to another chip. However, the two Intel chips (Iris and Hd5500) behave similarly to each other. In contrast, Nvidia's Gtx1080 (a newer architecture) slows down when the optimal configuration from the M4000 (an older architecture) is used. However, the M4000 works well with the Gtx1080 configuration. These counter-intuitive generational differences are concerning, for they limit the extent to which knowledge gained on one GPU is less transferable to another. Interestingly, the IRIS behaves well under the R9 strategy (1.08×), but the reverse is not true (1.15×).

MALI-4 provides the highest geomean slowdowns, possibly given its architectural differences. That is, an optimisation strategy tailored for MALI-4 slows downs benchmarks on other chips. The MALI-4 also suffers the most slowdowns when optimisation strategies for other GPUs are applied.

4.6 Revisiting results from Chapter 3

The discussion of results in this chapter raise several questions about the results of the previous chapter. Here, two separate issues are discussed that may appear to be contentious across chapters, but upon further investigation, are explainable.

4.6.1 Multi-kernel vs. oitergb

Section 3.6.2 discusses performance benefits of modifying applications that use iterative kernel launches against using a portable global barrier, which is exactly the oitergb optimisation presented in this chapter. The results of Section 3.6.2 show that performance improvements are modest, with only two tests out of 56 (about 3%) showing a speedup of 2× or more. However, in this chapter, Figure 4.3 shows that adding the oitergb optimisation leads to 16% of the tests showing a speedup of 2× or more (as it is the only optimisation in the OCL FP feature class).

The reason for this disconnect is that the Pannotia benchmarks are not as optimised as the IRGL benchmarks, and as such, the kernel execution times are not short enough to yield as much benefit as was observed in this chapter. As a concrete example, consider $\text{MIS}_{\text{pannotia}}$, the only benchmark common to the results of this chapter and Section 3.6.2. It is shown in Table 4.9 that $\text{MIS}_{\text{pannotia}}$ has many fewer (and longer) kernel invocations than another application that benefits greatly from oitergb. As another example, the Pannotia SSSP implementation uses a sparse matrix vector multiplication approach that examines every node in every kernel launch. The SSSP approaches in this chapter examine fewer nodes per iteration and thus have shorter kernel execution times.

As an additional piece of supporting evidence, consider the oitergb microbenchmark results of Figure 4.7. Notice that the rightmost data point shows that IRIS has the lowest GPU utilisation for the longest running kernels. Thus, for longer running kernels, IRIS would be expected to benefit the most. This is exactly what the results of the previous

Table 4.10: Average runtime (ms) over 20 iterations of the discovery protocol and the portability overhead of the BFS application of Section 3.6.3 using two mutex strategies.

Mutex	**Protocol time (ms)**	**BFS[`2e23`]**	**BFS[`rmat22`]**
original	136.25	2.53×	2.09×
self-destructing	2.21	1.11×	1.09×

chapter show in Section 3.9. For the long-running Pannotia benchmarks, Iris has the largest geomean speedup when using the global barrier.

These results reinforce the conclusion that global barrier optimisations are best used to avoid many short kernel launches.

4.6.2 Cost of portability

In this chapter, the discovery protocol and global barrier approach of the previous chapter were used to implement the oitergb optimisation. However, Figure 3.10 shows that the overhead of these portable constructs can be quite severe when compared to a barrier approach that uses a priori occupancy knowledge. In the worst case, R9 suffers a geomean slowdown of 1.71× when using a portable approach. This raises the question as to whether the oitergb implementation in this chapter suffers similar severe slowdowns due to the portable implementation.

Upon deeper investigation into the results of Figure 3.10, the cause of the large slowdowns was identified to be the execution of the discovery protocol. In particular, the slowdown was greatest among the chips with the higher occupancies, i.e. R9 and Nvidia chips. This is because many workgroups are contending for the discovery protocol mutex. While a fair mutex is critical for a high recall, it has low throughput in the presence of high contention, as the lock is restricted to be acquired in the order that it was requested.

For the oitergb implementation in this chapter, this issue was addressed by implementing a self-destruct mechanism to the discovery protocol mutex. The idea behind this optimisation is that the mutex is no longer required after the poll is closed; threads will simply exit (see Figure 3.3). Thus, the thread that closes the poll can also destroy the mutex. In turn, this alleviates lock contention. In this scheme, the `poll_open` flag must be declared atomic to avoid OpenCL data-races.

Table 4.10 shows timing results, restricted to R9 as it suffered the most extreme slowdowns, of: (1) running the discovery protocol in isolation with and without the self-destructing mutex; and (2) the new portability overhead of the two most severe slowdowns of Figure 3.10 – BFS with `2e23` and `rmat22`. Clearly, the exploding mutex reduces the cost of the discovery protocol significantly, and as a result, the portability overhead of the applications becomes much smaller. Thus, while there likely is some cost associated with the discovery protocol and custom execution environment, the portability costs of the oitergb optimisation presented in this chapter are likely much less than Figure 3.10 might suggest.

4.7 Related work

A study [VVdL+15] of the portability of three OpenCL graph algorithms using a CPU and two Nvidia GPUs concluded that the effects of inputs swamped out any benefits gained by optimising them for specific OpenCL platforms and that maintaining a single (functionally) portable version across platforms was more productive. The results of this chapter support the finding that inputs play a significant role in performance, but show that specialising for input is better than not optimising at all and specialising across other dimensions using a compiler can deliver even more performance.

Merrill et al. [MGG12a] construct a performance-portable library of parallel primitives containing reductions, scans, sorting algorithms, etc., by encoding tunable parameters such as number of items per load, the number of threads per thread block, etc., in the CUDA/C++ type system. Their system is evaluated on three Nvidia GPUs and reaches similar conclusions about the lack of a globally applicable optimisations for the problems they consider.

Other work [ZSC13] has studied the performance portability of OpenCL on more traditional GPU problems, such as SGEMM, SpMV and FFT.

In particular, Price and McIntosh-Smith [PMS17] study performance portability of a GPU Jacobi solver over Nvidia, AMD and Intel chips (including two CPU Intel chips). The parameter space in their work is sufficiently large (13 parameters, some with many options) that they do not perform an exhaustive search, but rather use an autotuning method for several hours. A heat map, similar to Figure 4.8, is used to show the difficulties of portability, especially across vendors. To address these issues, they employ a bi-level autotuning optimisation approach, using a metric of the worst performing test. That is, they consider one optimisation configuration o better than another o' for a set of benchmarks b, if the largest slowdown applying o' is less than the largest slowdown applying o over b. This approach would not work immediately in the domain of this chapter, as Section 4.5.3 shows that all possible optimisations cause at least one slowdown. Thus, this approach would simply converge to the completely unoptimised configuration. Using a metric other than the largest slowdown may be possible, but many candidates are immediately problematic. For example, if the number of slowdowns is used as a metric, then the first optimisation configuration of Table 4.4 would be found, which, as discussed, is a low-risk, low-reward configuration. Additionally, if the geomean was used as a metric, then the approach may favour certain chips, as Table 4.3 shows, some chips provide higher speedups than others. Thus, the choice of the analysis presented in this chapter uses a non-parametric, rank based test which appears to overcome these limitations.

It has been shown that the layout of byte code, stack frames, and the heap can introduce performance measurement bias on CPU systems. Previous work has used layout randomisation techniques across runs to remove such biases [CB13, MDHS09]. It is unclear if GPU systems suffer similar measurement biases and proprietary OpenCL runtimes make it difficult to implement similar randomisation techniques. Randomisation might improve the quality of the data collected for each test, but otherwise the methods of this chapter would be unchanged. The choice to use a non-parametric statistical method in this chapter was motivated by previous work that shows experimental data from computer systems is largely not normally distributed, making techniques like ANOVA inapplicable, and proposes the use of quantile regression [dOFD$^+$13]. The MWU test is similarly non-parametric, but quantile analysis was not necessary for the application domain in this chapter.

Muralidharan et al. [MRH$^+$16] propose a technique for autotuning across architectures without retraining for the target architecture. Using a set of performance data (e.g. performance counters) obtained from 6 different Nvidia GPUs, and using target architecture features at runtime, they use an SVM to predict the most appropriate variants. The results of this chapter aim to construct *descriptive* models as opposed to predictive models, and treats GPUs, applications and inputs as black boxes. Appropriately suited to the poor state of portable OpenCL profilers, the approach in this chapter requires only the ability to run and time a program.

4.8 Summary

Enabled by the functionally portable global barrier of Chapter 3, this chapter has demonstrated the difficulty of achieving performance portability of OpenCL graph algorithms across a diverse set of GPUs. The results have shown that universally beneficial (or harmful) optimisations do not exist. A data analysis was presented that can consume exhaustive experimental data and yield optimisation strategies that can be tailored for portability by various degrees of specialisation over chips, applications and inputs. The geomean gap between an oracular optimisation strategy and a strategy that specialise for application and input (but is oblivious to chip) is 16%. Using strategies customised to each dimension, several performance bottlenecks were identified. Along each dimension, the following were shown to be key features for performance:

- *chip features:* kernel launch latency, lack of atomic RMW combining, and memory divergence;
- *input features:* the diameter of the graph;
- *application features:* the number of nodes pushed to a frontier based worklist and the number and average runtime of individual kernels;

Overall, for the domain of GPU graph algorithms, this chapter has provided bounds for performance under various portability constraints.

While the performance evaluation of the global barrier was inconclusive in Chapter 3, this chapter has explored in detail a domain of GPU algorithms where there is a clear benefit for the global barrier. We believe this study strengthens the argument that official support for an OpenCL global barrier should be considered. Next, Chapter 5 addresses concerns encountered from industry about the interaction between interactive GPU applications, e.g. graphics, and applications that use a global barrier.

5 Cooperative GPU Multitasking

5.1 Personal context

The obvious limitation to the global barrier work of Chapter 3 and 4 is that the occupancy-bound execution model on which it relies is not officially supported by OpenCL, nor by any vendor. We wanted to understand whether the OpenCL committee (and the associated vendors) would be open to considering the execution model given that we had shown that it has some pragmatic utility. To that end, I presented the global barrier of Chapter 3 at ARM (in Cambridge, UK), AMD (in Redmond, WA) and at the International Workshop on OpenCL (in Vienna, AT). Additionally, we had email correspondence with Lee Howes, who had worked on GPUs at Qualcomm and is an editor of the OpenCL specification [Khr15], and Andrew Richards at Codeplay.

The take-away from all these conversations was that while today's GPUs empirically support the occupancy-bound execution model, no vendor wants to commit to the model. That is, vendors want to keep the possibility open for development that would break the progress guarantees of the model. In particular, we understood that: (1) a completely fair preemptive model, e.g. like that of mainstream CPUs, would likely be too expensive due to the large states of workgroups, but (2) unfair preemption, where an occupant workgroup is preempted with no guarantees of returning execution relative to other occupant workgroups, may be useful for cases of multitasking with high priority tasks (e.g. graphics) or energy throttling, especially on mobile GPUs.

The work presented in this chapter provides new programming constructs that aim to address the concerns from industry, while still offering enough progress guarantees to use blocking idioms, including global barrier synchronisation.

5.2 Motivation

Many interesting parallel algorithms are *irregular*: the amount of work to be processed is unknown ahead of time and may change dynamically in a workload-dependent manner. There is growing interest in accelerating such algorithms on GPUs, with the work in

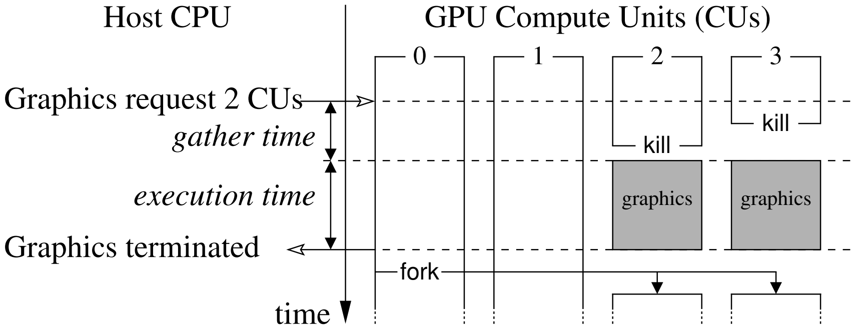


Figure 5.1: Cooperative kernels can flexibly resize to allow other tasks, e.g. graphics, run concurrently.

Chapter 4 being a concrete example. In some cases, irregular algorithms can be optimised using *blocking synchronisation* between workgroups, as seen with the global barrier used by the oitergb optimisation in Chapter 4. Another blocking idiom used in parallel irregular algorithms is *work stealing*, which requires each workgroup to maintain a queue, typically mutex-protected, to enable stealing by other workgroups.

To avoid starvation, a blocking concurrent algorithm requires *fair* scheduling of workgroups. For example, if one workgroup holds a mutex, an unfair scheduler may cause another workgroup to spin-wait forever for the mutex to be released. Similarly, an unfair scheduler can cause a workgroup to spin-wait indefinitely at a global barrier so that other workgroups do not reach the barrier.

A degree of fairness: occupancy-bound execution As discussed in Chapter 3, current GPU programming models specify almost no fairness guarantees regarding scheduling of workgroups, and current GPU schedulers are not totally fair in practice. Instead, the occupancy-bound execution model, empirically observed on many current GPUs, maps each workgroup to a hardware resource and provides relative fairness to as many workgroups can simultaneously occupy the GPU. In this chapter, the hardware resource required to execute a workgroup will be called a *compute unit*, although as noted in Section 3.6.1, sometimes a compute unit can be oversubscribed with more than one workgroup occupant.

The occupancy-bound execution model does not guarantee fair scheduling between all workgroups: if the hardware resources of a GPU are fully occupied, then a not-yet-occupant workgroup will not be scheduled until some occupant workgroup completes execution. Yet the execution model *does* provide fair scheduling between *occupant* workgroups, which are bound to separate compute units that operate in parallel. Current GPU implementations of blocking algorithms assume the occupancy-bound execution model, using either occupancy assumption or discovery methods as discussed in Chapter 3.

Resistance to occupancy-bound execution Despite its practical prevalence, none of the current GPU programming models actually mandate occupancy-bound execution. Further, there are reasons why this model is undesirable. First, occupancy-bound execution does not enable multitasking, since a workgroup effectively *owns* a compute unit until the workgroup has completed execution. The GPU cannot be used meanwhile for other tasks, even high priority interactive tasks (e.g. graphics rendering). Second, *energy throttling* is an important concern for battery-powered devices [VC13]. In the future, it may be desirable for a mobile GPU driver to power down some compute units, unfairly suspending execution of associated occupant workgroups, if the battery level is low.

Our assessment, informed by discussions with a number of industrial practitioners who have been involved in the OpenCL and/or HSA standardisation efforts, is that GPU vendors (1) will not commit to the occupancy-bound execution model they currently implement, for the above reasons, yet (2) will not guarantee fair scheduling using preemption. This is due to the high runtime cost of preempting workgroups, which requires managing thread local state (e.g. registers, program counters) for all workgroup threads (up to 1024 on Nvidia GPUs), as well as local memory (up to 64 KB on Nvidia GPUs). Vendors instead wish to retain the essence of the simple occupancy-bound model, supporting preemption only in key special cases.

As a concrete example, Nvidia's Pascal architecture is documented to support preemption [NVI16], yet the documentation provides no fairness guarantees. Indeed, on a GPU from this architecture (GTX Titan X), starvation for blocking algorithms is still observed: a kernel containing a global barrier executes successfully when run with 56 workgroups, but hangs indefinitely when run with 57 workgroups. Thus, indicating that the ability to preempt does not imply totally fair scheduling.

A way forward: cooperative kernels To summarise: blocking algorithms demand fair scheduling, but for various pragmatic reasons GPU vendors will not commit to fairness guarantees, even the limited guarantees provided by the occupancy-bound execution model. This chapter presents *cooperative kernels*, an extension proposal to the OpenCL GPU programming model that aims to resolve this impasse. The extensions and their semantics are briefly described below.

A kernel that requires fair scheduling is identified as *cooperative*, and written using two additional language primitives, offer_kill and request_fork, placed by the programmer.

Where the cooperative kernel could proceed with fewer workgroups, a workgroup can execute offer_kill, offering to sacrifice itself to the scheduler. This indicates that the workgroup would ideally continue executing, but that the scheduler may preempt the workgroup; the cooperative kernel must be prepared to deal with either scenario. Where the cooperative kernel could use additional resources, a workgroup can execute request_fork to indicate that the kernel is prepared to proceed with the existing set of workgroups, but is able to benefit from one or more additional workgroups commencing execution directly after the request_fork program point.

The use of request_fork and offer_kill creates a contract between the scheduler and the cooperative kernel. Functionally, the scheduler must guarantee that the workgroups executing a cooperative kernel are fairly scheduled, while the cooperative kernel must be robust to workgroups leaving and joining the computation in response to offer_kill and request_fork. Non-functionally, a cooperative kernel must ensure that offer_kill is executed frequently enough such that the scheduler can accommodate soft real-time constraints, e.g. allowing a smooth frame-rate for graphics. In return, the scheduler should allow the cooperative kernel to utilise hardware resources where possible, killing workgroups only when demanded by other tasks, and forking additional workgroups when possible.

Cooperative kernels allow for *cooperative multitasking*, used historically when preemption was not available or too costly. Cooperative kernels avoid the cost of arbitrary preemption as the state of a workgroup killed via offer_kill does not have to be saved. Previous cooperative multitasking systems have provided *yield* semantics, whereby a processing unit would temporarily give up its hardware resource. Cooperative kernels deviate from this design because, in the case of a global barrier, adopting yield would force the cooperative kernel to block *completely* when a single workgroup yields, stalling the kernel execution until the given workgroup resumes. Instead, offer_kill allows a kernel to make progress with a smaller number of workgroups, with workgroups potentially joining again later via request_fork.

Figure 5.1 illustrates sharing of GPU compute units between a cooperative kernel and a graphics task. Workgroups 2 and 3 of the cooperative kernel are killed at an offer_kill to make room for a graphics task. The workgroups are subsequently restored to the cooperative kernel when workgroup 0 calls request_fork. The *gather time* is the time between resources being requested and the application surrendering them via offer_kill. To satisfy soft real-time constraints, the gather time should be low; the experimental evaluation of Section 5.7.2 shows that, in practice, the gather-time for applications is acceptable for a range of graphics workloads.

The cooperative kernels model has several appealing properties:

1. By providing fair scheduling between workgroups, cooperative kernels meet the needs of blocking algorithms, including irregular algorithms.

2. The model has no impact on the development of regular (non-cooperative) compute and graphics kernels.

3. The model is backwards-compatible: offer_kill and request_fork may be ignored, and a cooperative kernel will behave exactly as a regular kernel does on current GPUs.

4. Cooperative kernels can be implemented over the occupancy-bound execution model provided by current GPUs: the prototype implementation of this chapter uses no special hardware or driver support above what is provided the OpenCL standard.

5. Cooperative kernels are *complementary* to preemption; that is, if hardware support for preemption *is* available, it can be leveraged to implement cooperative kernels efficiently, as the programmer provides "smart" preemption points.

Placing the primitives manually is straightforward for a representative set of GPU-accelerated irregular algorithms examined in this chapter. The experimental evaluation shows that the model can enable efficient multitasking of cooperative and non-cooperative tasks.

5.2.1 Chapter contributions

In summary, the main contributions of this chapter are:

- *cooperative kernels*, an extension to the OpenCL GPU programming model that supports the scheduling requirements of blocking algorithms (Section 5.4), while also addressing pragmatic concerns of vendors;

- *a prototype implementation* of cooperative kernels, including a simple scheduler, on top of OpenCL 2.0 (Section 5.5);

- *two sets of applications*, one set of applications adapted to use the cooperative kernels model, and another set of applications to be representative of interactive GPU tasks (Section 5.6);

- *an experimental evaluation* assessing the overhead and responsiveness of the cooperative kernels approach over a set of irregular algorithms (Section 5.7), including a best-effort comparison with the efficiency afforded by hardware-supported preemption available on Nvidia GPUs (Section 5.7.3).

Related publications The material presented in this chapter is based on work published in the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'17) [SED17a]. *The work received a distinguished paper award.*

5.3 Two blocking GPU idioms

The chapter begins by discussing two high-level code patterns that appear in existing GPU applications and use blocking synchronisation between workgroups. The two patterns discussed are: work stealing (Section 5.3.1) and graph traversal (Section 5.3.2).

5.3.1 Work stealing

Work stealing enables dynamic balancing of tasks across processing units. It is useful when the number of tasks to be processed is dynamic, due to one task creating an arbitrary number of new tasks. Work stealing has been explored in the context of GPUs [CT08, TPO10]. Each workgroup has a local queue from which it obtains tasks to process, and to which it stores new tasks. If its queue is empty, a workgroup tries to *steal* a task from another queue.

Figure 5.2 illustrates a work stealing kernel. Each thread receives a pointer to the task queues, in global memory, initialised by the host to contain initial tasks. A thread uses its workgroup id (line 2) as a queue id to access the relevant task queue. The `pop_or_steal`

```
1 kernel void work_stealing(global Task * queues) {
2  int queue_id = get_group_id();
3  while (more_work(queues)) {
4   Task * t = pop_or_steal(queues, queue_id);
5   if (t) {
6    process_task(t, queues, queue_id);
7   }
8  }
9 }
```

Figure 5.2: An excerpt of a work stealing algorithm in OpenCL.

```
1  kernel void graph_app(global graph *g, global nodes *n0,
2                                         global nodes *n1) {
3    int level = 0;
4    global nodes *in_nodes = n0;
5    global nodes *out_nodes = n1;
6    int tid = get_global_id();
7    int stride = get_global_size();
8    while(in_nodes.size > 0) {
9      for (int i = tid; i < in_nodes.size; i += stride) {
10       process_node(g, in_nodes[i], out_nodes, level);
11     }
12     swap(&in_nodes, &out_nodes);
13     global_barrier();
14     reset(out_nodes);
15     level++;
16     global_barrier();
17   }
18 }
```

Figure 5.3: An outline of an OpenCL graph traversal.

function (line 4) pops a task from the workgroup's queue or tries to steal a task from other queues. Although not depicted here, concurrent accesses to queues inside `more_work` and `pop_or_steal` are guarded by a mutex per queue, implemented using atomic compare and swap operations on global memory.

If a task is obtained, then the workgroup processes it (line 6), which may lead to new tasks being created and pushed to the workgroup's queue. The kernel presents two opportunities for spin-waiting: spinning to obtain a mutex, and spinning in the main kernel loop to obtain a task. Without fair scheduling, threads waiting at these points might spin indefinitely, causing the application to hang.

5.3.2 Graph traversal

Figure 5.3 illustrates a frontier-based graph traversal algorithm, representative of algorithms studied in Chapter 4 and in related work [BNP12, PP16]. The kernel is given three arguments in global memory: a graph structure, and two arrays of graph nodes. Initially, `n0` contains the starting nodes to process. Private variable `level` records the current frontier level, and `in_nodes` and `out_nodes` point to distinct arrays recording the nodes to be processed during the current and next frontier, respectively.

The application iterates as long as the current frontier contains nodes to process (line 8). At each frontier, the nodes to be processed are evenly distributed between threads through *stride-based* processing.

In this case, the stride is the total number of threads, obtained via `get_global_size`. A thread calls `process_node` to process a node given the current level, with nodes to be processed during the next frontier being pushed to `out_nodes`. After processing the frontier, the node array pointers of a thread are swapped (line 12).

At this point, the GPU threads must wait for all other threads to finish processing the frontier. To achieve this, a global barrier is used (line 13). After all threads reach this point, the output node array is reset (line 14) and the level is incremented. The threads use another global barrier to wait until the output node is reset (line 16), after which they continue to the next frontier. As discussed throughout this thesis, the global barrier is not provided as a GPU primitive. However, a robust portable global barrier can be implemented following the approach of Chapter 3.

Blocking summary The mutexes and barriers used by these two examples appear to run reliably on current GPUs for kernels that are executed with no more workgroups than can be simultaneously occupant on the GPU, e.g. as in Chapter 4. This is due to the fairness of the occupancy-bound execution model, discussed in Chapter 3, that current GPUs have been shown, experimentally, to provide. But, as discussed in Section 5.2, this model is not endorsed by language standards or vendor implementations, and may not be respected in the future.

In Section 5.4.2, it is shown how the work stealing and graph traversal examples of Figures. 5.2 and 5.3 can be updated to use the cooperative kernels programming model to resolve the scheduling issue.

5.4 Cooperative kernels

The new constructs of the cooperative kernels programming model are summarised in Table 5.1 and the semantics of the new constructs are described in more detail in Section 5.4.1. The blocking idioms presented in the previous section are used to assess programmability of these constructs (Section 5.4.2), then the nonfunctional properties required by the model are outlined (Section 5.4.3). In the extended version of the published work, operational semantics are given [SED17b]. These are not provided in this thesis as they are primarily a contribution of Alastair F. Donaldson. Section 5.5.3 discusses reasonable alternative semantics for the cooperative kernel constructs that might be considered.

Table 5.1: Overview of new cooperative kernel constructs.

Construct	**Description**
transmit	A qualifier for private variables. The values in variables qualified with transmit in thread 0 of a workgroup will be copied into the private memory of new workgroups forked with request_fork.
request_fork	Signals to the scheduler that *if* there are sufficient resources available, new workgroups can be forked and join the computation at the call of request_fork. New workgroups are assigned contiguous ids and have access to kernel arguments and variables qualified with transmit from the workgroup that called request_fork. Must be called in a workgroup-uniform location.
offer_kill	Signals to the scheduler that *if* resources are needed elsewhere, the calling workgroup can permanently leave the computation at this point. The scheduler may only kill the active workgroup with the highest id. Must be called in a workgroup-uniform location.
global_barrier	Like the global barrier discussed in Chapter 3 and 4, synchronises all active threads executing the cooperative kernel. Must be called in a kernel-uniform location.
resizing_global_barrier	Similar to global_barrier, synchronises all active threads. Additionally, workgroups may be forked or killed (similar to request_fork and offer_kill) at the point of synchronisation. Must be called in a kernel-uniform location.

5.4.1 Semantics of cooperative kernels

As with a regular OpenCL kernel, a cooperative kernel is launched by the host application, passing parameters to the kernel and specifying a desired number of threads and workgroups. Unlike in a regular kernel, the parameters to a cooperative kernel are immutable (though pointer parameters can refer to mutable data).

Cooperative kernels are written using the following extensions: transmit, a qualifier on the variables of a thread; offer_kill and request_fork, the key functions that enable cooperative scheduling; and global_barrier and resizing_global_barrier primitives for inter-workgroup synchronisation.

Transmitted variables A variable declared in a cooperative kernel can optionally be annotated with a new transmit qualifier. Annotating a variable v with transmit means that when a workgroup uses request_fork to spawn new workgroups, the workgroup should transmit its current value for v to the threads of the new workgroups. The semantics for this are detailed in the discussion of request_fork below.

Active workgroups If the host application launches a cooperative kernel requesting N workgroups, this indicates that the kernel should be executed with a *maximum* of N workgroups, and that as many workgroups as possible, up to this limit, are desired. However, the scheduler may initially schedule fewer than N workgroups, and as explained below the number of workgroups that execute the cooperative kernel can change during the lifetime of the kernel.

The number of *active workgroups*—workgroups executing the kernel—is denoted M. Active workgroups have consecutive ids in the range $[0, M - 1]$. Initially, at least one workgroup is active; if necessary the scheduler must postpone the kernel until some compute unit becomes available. For example, in Figure 5.1: at the beginning of the execution $M = 4$; while the graphics task is executing $M = 2$; after the fork $M = 4$ again.

When executed by a cooperative kernel, get_num_groups returns M, the *current* number of active workgroups. This is in contrast to get_num_groups for regular kernels, which returns the fixed number of workgroups that execute the kernel.

Fair scheduling *is* guaranteed between active workgroups; i.e. if some thread in an active workgroup is enabled, then eventually this thread is guaranteed to execute an instruction.

Semantics for offer_kill The offer_kill primitive allows the cooperative kernel to return compute units to the scheduler by offering to sacrifice workgroups. The idea is as follows: allowing the scheduler to arbitrarily and abruptly terminate execution of workgroups might be drastic, yet the kernel may contain specific program points at which a workgroup could *gracefully* leave the computation.

Similar to the OpenCL workgroup barrier primitive, offer_kill, is a workgroup-level function—it must be encountered uniformly by all threads in a workgroup.

Suppose a workgroup with id m executes offer_kill. If the workgroup has the largest id among active workgroups then it can be killed by the scheduler, except that workgroup 0 can never be killed (to avoid early termination of the kernel). More formally, if $m < M - 1$ or $M = 1$ then offer_kill is a no-op. If instead $M > 1$ and $m = M - 1$, the scheduler can choose to ignore the offer, so that offer_kill executes as a no-op, or accept the offer, so that execution of the workgroup ceases and the number of active workgroups M is atomically decremented by one. Figure 5.1 illustrates this, showing that workgroup 3 is killed before workgroup 2.

Semantics for request_fork Recall that a desired limit of N workgroups was specified when the cooperative kernel was launched, but that the number of active workgroups, M, may be smaller than N, either because (due to competing workloads) the scheduler did not provide N workgroups initially, or because the kernel has given up some workgroups via offer_kill calls. Through the request_fork primitive (also a workgroup-level function), the kernel and scheduler can collaborate to allow new workgroups to join the computation at an appropriate point and with appropriate state.

Suppose a workgroup with id $m \leq M$ executes request_fork. Then the following occurs: an integer $k \in [0, N - M]$ is chosen by the scheduler; k new workgroups are spawned with consecutive ids in the range $[M, M + k - 1]$; the active workgroup count M is atomically incremented by k.

The k new workgroups commence execution at the program point immediately following the request_fork call. The variables that describe the state of a thread are all uninitialised for the threads in the new workgroups; reading from these variables without first initialising them is an undefined behaviour. There are two exceptions to this: (1) because the parameters to a cooperative kernel are immutable, the new threads have access to these parameters as part of their local state and can safely read from them; (2) for each variable v annotated with transmit, every new thread's copy of v is initialised to the value that thread 0 in workgroup m held for v at the point of the request_fork call. In effect, thread 0 of the forking workgroup *transmits* the relevant portion of its local state to the threads of the forked workgroups.

Figure 5.1 illustrates the behaviour of request_fork. After the graphics task finishes executing, workgroup 0 calls request_fork, spawning the two new workgroups with ids 2 and 3. Workgroups 2 and 3 join the computation where workgroup 0 called request_fork.

Notice that $k = 0$ is always a valid choice for the number of workgroups to be spawned by request_fork, and is guaranteed if M is equal to the workgroup limit N.

Global barriers Because workgroups of a cooperative kernel are fairly scheduled, a global barrier primitive can be provided in this model. Two variants are specified: global_barrier and resizing_global_barrier.

The global_barrier primitive is similar to the global barrier described throughout this thesis. It is a kernel-level function, i.e. if it appears in conditional code then it must be reached by *all* threads executing the cooperative kernel. On reaching a global_barrier, a thread waits until all threads have arrived at the barrier. Once all threads have arrived, the threads may proceed past the barrier with the guarantee that all global memory accesses issued before the barrier have completed. The global_barrier primitive can be implemented similar to the approach described in Chapter 3, however the implementation must take into account a growing and shrinking number of workgroups, as request_fork and offer_kill may change the count of active workgroups.

The resizing_global_barrier primitive is also a kernel-level function. It is identical to global_barrier, except that it caters for cooperation with the scheduler: by issuing a resizing_global_barrier the programmer indicates that the cooperative kernel is prepared to proceed after the barrier with more or fewer workgroups.

When all threads have reached resizing_global_barrier, the number of active workgroups, M, is atomically set to a new value, M' say, with $0 < M' \leq N$. If $M' = M$ then the active workgroups remain unchanged. If $M' < M$, workgroups $[M', M-1]$ are killed. If $M' > M$ then $M' - M$ new workgroups join the computation after the barrier, as if they were forked from workgroup 0. In particular, the transmit-annotated local state of thread 0 in workgroup 0 is transmitted to the threads of the new workgroups.

The semantics of resizing_global_barrier can be modelled via calling request_fork and offer_kill, surrounded and separated by calls to a global_barrier, i.e.:

```
global_barrier();
if(get_group_id() == 0) request_fork();
global_barrier();
offer_kill();
global_barrier();
```

The enclosing global_barrier calls ensure that the change in number of active workgroups from M to M' occurs entirely within the resizing barrier, so that M changes atomically from a programmer's perspective. The middle global_barrier, while not semantically necessary, ensures that forking occurs before killing, so that workgroups $[0, \min(M, M') - 1]$ are left intact. That is, so that workgroups are not forked immediately after they are killed.

Because resizing_global_barrier can be implemented as above, is is not regarded *conceptually* as a primitive of the cooperative kernels model. However, Section 5.5.2 shows how a resizing barrier can be implemented more efficiently through direct interaction with the scheduler.

5.4.2 Programming with cooperative kernels

A changing workgroup count Unlike in regular OpenCL, the value returned by `get_num_groups` is not fixed during the lifetime of a cooperative kernel: it corresponds to the active group count M, which changes as workgroups execute offer_kill, and request_fork. The value returned by `get_global_size` is similarly subject to change. A cooperative

```
 1 kernel void cooperative_work_stealing(global Task * queues) {
 2   int queue_id;
 3   while (more_work(queues)) {
 4     request_fork();
 5     offer_kill();
 6     queue_id = get_group_id();
 7     Task * t = pop_or_steal(queues, queue_id);
 8     if (t) {
 9       process_task(t, queues, queue_id);
10     }
11   }
12 }
```

Figure 5.4: Cooperative kernel version of the work stealing kernel of Figure 5.4. Changes to correctly use cooperative features are highlighted.

kernel must thus be written in a manner that is robust to changes in the values returned by these functions.

In general, their volatility means that use of these functions should be avoided. However, the situation is more stable if a cooperative kernel does not call offer_kill and request_fork directly, so that only resizing_global_barrier can affect the number of active workgroups. Then, at any point during execution, the threads of a kernel are executing between some pair of resizing barrier calls, called a *resizing barrier interval* (considering the kernel entry and exit points conceptually to be special cases of resizing barriers). The active workgroup count is constant within each resizing barrier interval, so that get_num_groups and get_global_size return stable values during such intervals. As illustrated below for graph traversal, this can be exploited by algorithms that perform strided data processing.

Adapting work stealing In this pattern, there is no state to transmit since a computation is entirely parameterised by a task, which is retrieved from a global queue. Figure 5.4 shows the original work stealing code of Figure 5.2 adapted to use cooperative features. Calls to request_fork and offer_kill are added at the start of the main loop (lines 5 and 4) to let a workgroup offer itself to be killed or forked, respectively, before it processes a task. Note that a workgroup may be killed even if its associated task queue is not empty, since remaining tasks will be stolen by other workgroups. In addition, since request_fork may be the entry point of a workgroup, the queue id must now be computed after it, so the call to get_group_id is moved inside the loop after request_fork, i.e. to line 6 in the adapted

```
 1 kernel void cooperative_graph_app(global graph *g,
 2                                    global nodes *n0,
 3                                    global nodes *n1) {
 4   transmit int level = 0;
 5   transmit global nodes *in_nodes = n0;
 6   transmit global nodes *out_nodes = n1;
 7   while(in_nodes.size > 0) {
 8     int tid = get_global_id();
 9     int stride = get_global_size();
10     for (int i = tid; i < in_nodes.size; i += stride) {
11       process_node(g, in_nodes[i], out_nodes, level);
12     }
13     swap(&in_nodes, &out_nodes);
14     resizing_global_barrier();
15     reset(out_nodes);
16     level++;
17     resizing_global_barrier();
18   }
19 }
```

Figure 5.5: Cooperative kernel version of the graph traversal kernel of Figure 5.3. Changes to correctly use cooperative features are highlighted.

code. In particular, the queue id cannot be transmitted since a newly spawned workgroup should read from its own queue and not the one of the forking workgroup.

Adapting graph traversal Figure 5.5 shows a cooperative version of the graph traversal kernel of Figure 5.3 from Section 5.3. On lines 14 and 17, the original global barriers are changed into resizing barriers. Several variables are marked to be transmitted in the case of workgroups joining at the resizing barriers (lines 4, 5 and 6): `level` must be restored so that new workgroups know which frontier they are processing; `in_nodes` and `out_nodes` must be restored so that new workgroups know which of the node arrays to use for input and output. Lastly, the static work distribution of the original kernel is no longer valid in a cooperative kernel. This is because the stride (which is based on M) may change after each resizing barrier call. To fix this, the work is re-distributed after each resizing barrier call by recomputing the thread id and stride (lines 8 and 9). This example exploits the fact that the cooperative kernel does not issue offer_kill nor request_fork directly: the value of `stride` obtained from `get_global_size` at line 9 is stable until the next resizing barrier at line 14.

Patterns for irregular algorithms Section 5.6.1 describes the set of irregular GPU algorithms used in the experiments of this chapter, which largely captured the irregular blocking algorithms that were available as open source GPU kernels at the time of the work. These algorithms all employ either work stealing or operate on graph data structures. Placing the new constructs in these applications followed common, easy-to-follow patterns in each case. The work stealing algorithms have a transactional flavour and require little or no state to be carried between transactions. The point at which a workgroup is ready to process a new task is a natural place for offer_kill and request_fork, and few or no transmit annotations are required. Figure 5.5 is representative of most level-by-level graph algorithms. It is typically the case that on completing a level of the graph algorithm, the next level could be processed by more or fewer workgroups, which resizing_global_barrier facilitates. Some level-specific state must be transmitted to new workgroups, which was determined on a case-by-case basis.

5.4.3 Non-functional requirements

The semantics presented in Section 5.4.1 describe the behaviours that a developer of a cooperative kernel should be prepared for. However, the aim of cooperative kernels is to find a balance that allows *efficient* execution of algorithms requiring fair scheduling, and *responsive* multitasking, so that the GPU can be shared between cooperative kernels and other shorter tasks with soft real-time constraints. To achieve this balance, an implementation of the cooperative kernels model, and the programmer of a cooperative kernel, must strive to meet the following non-functional requirements.

The purpose of offer_kill is to let the scheduler destroy a workgroup in order to schedule higher-priority tasks. The scheduler relies on the cooperative kernel to execute offer_kill sufficiently frequently that soft real-time constraints of other workloads can be met. Using the work stealing example: a workgroup offers itself to the scheduler after processing each task. If tasks are sufficiently short then the scheduler will have ample opportunities to de-schedule workgroups. But if tasks are very long then it might be necessary to rewrite the algorithm so that tasks are shorter and more numerous, to achieve a higher rate of calls to offer_kill. Getting this non-functional requirement right is GPU- and application-dependent. In Section 5.6.3, experiments are presented that measure the response rate that would be required to co-schedule graphics rendering with a cooperative kernel, maintaining a smooth frame rate.

Recall that, on launch, the cooperative kernel requests N workgroups. The scheduler should thus aim to provide N workgroups if other constraints allow it, by accepting an offer_kill only if a compute unit is required for another task, and responding positively to request_fork calls if compute units are available.

5.5 Prototype implementation

Our vision was that cooperative kernel support will be integrated in the runtimes of future GPU implementations of OpenCL, with driver support for the required new primitives. However, to explore cooperative kernels experimentally on current GPUs, a prototype was developed that mocks up the required runtime support via a *megakernel*, and exploits the occupancy-bound execution model to ensure fair scheduling between workgroups. Though, it is emphasised that an aim of cooperative kernels is to *avoid* depending on the occupancy-bound model. The prototype exploits this model simply to allow experimentation on current GPUs whose proprietary drivers are not available to modify. The megakernel approach is described in Section 5.5.1 followed by implementation details of various scheduler components in Section 5.5.2.

5.5.1 The megakernel mock up

Instead of multitasking multiple separate kernels, a set of kernels are merged into a megakernel—a single, monolithic kernel. The megakernel is launched with as many workgroups as can be occupant concurrently. One workgroup takes the role of the scheduler,[1] and the scheduling logic is embedded as part of the megakernel. The remaining workgroups act as a pool of workers. A workgroup repeatedly queries the scheduler to be assigned a task, which corresponds to executing a cooperative or non-cooperative kernel. In the non-cooperative case, the workgroup executes the relevant kernel function uninterrupted, then awaits further work. In the cooperative case, the workgroup either starts from the kernel entry point or immediately jumps to a designated point within the kernel, depending on whether the workgroup is an initial workgroup of the kernel, or a workgroup forked via request_fork. In the latter case, the new workgroup also receives a struct containing the values of all relevant transmit-annotated variables.

[1] The scheduler requirements given in Section 5.4 are agnostic to whether the scheduling logic takes place on the CPU or GPU. To avoid expensive communication between GPU and the GPU, in this work the scheduler was implemented on the GPU.

Simplifying assumptions For ease of implementation, the prototype implementation has several limitations. Namely, the prototype:

- supports multitasking a single cooperative kernel with a single non-cooperative kernel, though the non-cooperative kernel can be invoked many times;
- requires that offer_kill, request_fork and resizing_global_barrier are called from the entry function of a cooperative kernel. This allows the use of `goto` and `return` to direct threads into and out of the kernels;
- requires that all transmit variables are declared in the root scope. This constraint simplifies sharing transmit annotated variables with newly forked workgroups.

These constraints did not prohibit any applications in the benchmarks considered for this work. A non-mock implementation of cooperative kernels would not use the megakernel approach, so the engineering effort associated with lifting this restriction in the prototype implementation was not deemed to be worthwhile.

5.5.2 Scheduler design

To enable multitasking through cooperative kernels, the runtime (in this work, the megakernel) must: (1) track the state of workgroups, i.e. whether a workgroup is waiting or computing a kernel; (2) maintain consistent context states for each kernel, e.g. tracking the number of active workgroups; and (3) provide a safe way for these states to be modified in response to request_fork and offer_kill. These issues are now discussed. Additionally, the implementation of a more efficient resizing barrier is presented. The presentation discusses how the scheduler would handle arbitrary combinations of kernels, though as noted above, the current prototype implementation is restricted to the case of two kernels.

Scheduler contexts and resource messages To dynamically manage workgroups executing cooperative kernels, the framework must track the state of each workgroup and provide a channel of communication from the scheduler workgroup to workgroups executing request_fork and offer_kill. To achieve this, a *scheduler context* structure is used, mapping a primitive workgroup id to the workgroup's status, which is either *available* or the id of the kernel that the workgroup is currently executing. The scheduler can then send workgroups executing cooperative kernels a *resource message*, commanding workgroups to exit at offer_kill, or spawn additional workgroups at request_fork. Thus, the scheduler context needs a communication channel for each cooperative kernel. In the prototype implementation, communication channels are provided using OpenCL 2.0 atomic variables in global memory.

Launching kernels and managing workgroups To launch a kernel, the host sends a data packet to the GPU scheduler consisting of a kernel to execute, kernel inputs, and a flag indicating whether the kernel is cooperative. In the prototype implementation, this host-to-device communication channel is provided using OpenCL fine-grained SVM atomic operations (see Section 2.3.2).

Upon receiving a data packet describing a kernel launch K, the scheduler must decide how to schedule K. Suppose K requests N workgroups. The scheduler queries the scheduler context. If there are at least N available workgroups, K can be scheduled and commence execution immediately. Suppose instead that there are only $N_a < N$ available workgroups, but a cooperative kernel K_c is executing. The scheduler can use K_c's channel in the scheduler context to command K_c to relinquish $N - N_a$ workgroups via executions of offer_kill. Once N workgroups become available, the scheduler then instructs N workgroups from the available workgroups to execute kernel K. If the new kernel K is itself a cooperative kernel, the scheduler would be free to provide K with fewer than N active workgroups initially.

If a cooperative kernel K_c is executing with fewer workgroups than it initially requested and there exists available workgroups, then the scheduler may decide to instruct the available workgroups to join the execution of K_c; these workgroups will join the next time a workgroup of K_c executes request_fork. To do this, the scheduler asynchronously signals K_c through the communication channel to indicate the number of workgroups that should join at the next request_fork command. When a workgroup w of K_c subsequently executes request_fork, thread 0 of w updates the kernel and scheduler contexts so that the given number of new workgroups are directed to the program point after the request_fork call. This involves searching through the workgroup pool to find available workgroups, as well as copying the values of transmit-annotated variables to the new workgroups.

An efficient resizing barrier In Section 5.4.1, the semantics of a resizing barrier were defined using calls to request_fork and offer_kill between two global_barrier calls. It is possible, however, to implement the resizing barrier using only one call to a global barrier with request_fork and offer_kill embedded inside. Such an implementation is now described.

The starting global barrier implementation follows the master/slave XF barrier, described in Chapter 3.5.1. Recall that in this implementation one workgroup (the master) collects signals from the other workgroups (the slaves) indicating that they have arrived at the barrier and that they are waiting for a reply indicating that they may leave the barrier. Once the master has received a signal from all slaves, it replies with a signal to each slave saying that they may leave.

Incorporating request_fork and offer_kill into such a barrier implementation is straightforward. Upon entering the barrier, the slaves first execute offer_kill, possibly exiting. The master then waits for M slaves (the number of active workgroups), which may decrease due to offer_kill calls by the slaves, but will not increase. Once the master observes that M slaves have arrived, it knows that all other workgroups are waiting to be released. The master executes request_fork, and the statement immediately following this request_fork is a conditional that forces newly spawned workgroups to join the slaves in waiting to be released. Finally, the master releases all the slaves: the original slaves and the new slaves that joined at request_fork.

While the implementation described above is simple, it is also sub-optimal. Workgroups execute offer_kill only once per resizing barrier call and, depending on order of arrival, it is possible that only one workgroup is killed per resizing barrier call. This prevents the scheduler from gathering workgroups quickly, which may not be sufficient for interactive tasks, e.g. graphics.

The *gather time*, i.e. the time between when the scheduler receives a kernel launch request and when enough workgroups to execute the new kernel are made available, can be reduced by providing a new query function for cooperative kernels, which returns the number of workgroups that the scheduler wants to obtain from the cooperative kernel.

A more efficient resizing barrier can now be implemented using this new scheduler hook as follows: (1) the master waits for all slaves to arrive; (2) the master calls request_fork and commands the new workgroups to be slaves; (3) the master calls query, obtaining a value W; (4) the master releases the slaves, broadcasting the value W to them; (5) workgroups with ids larger than $M - W$ spin, calling offer_kill repeatedly until the scheduler claims them—from query it is known that the scheduler will eventually do so. Results show that the barrier using query greatly reduces the gather time in practice (Section 5.7.2).

5.5.3 Alternative semantic choices

The semantics of cooperative kernels has been guided by the applications studied in this work (described in Section 5.6.1). Several cases where different and also reasonable semantic decisions are now discussed.

Killing workgroups in decreasing id order The semantics of offer_kill are such that only the active workgroup with the highest id can be killed. This has an appealing property: it means that the ids of active workgroups are contiguous, which is important for efficient strided accessing of data. The cooperative graph traversal algorithm of Figure 5.5 illustrates this: the algorithm is prepared for `get_global_size` to change after each resizing barrier call, but depends on the fact that `get_global_id` returns a contiguous range of thread ids.

A disadvantage of this decision is that it may provide sub-optimal responsiveness from the point of view of the scheduler. Suppose the scheduler requires an additional compute unit, but the active thread with the largest id is processing some computationally intensive work and will take a while to reach offer_kill. The chosen semantics mean that the scheduler cannot take advantage of the fact that another active workgroup may invoke offer_kill sooner.

Cooperative kernels that do not require contiguous thread ids (e.g. the work stealing example of Figure 5.4) might be more suited to a semantics in which workgroups can be killed in any order, but where workgroup ids, and thus thread global ids, are not guaranteed to be contiguous.

Although cooperative multitasking is not mentioned, fairness properties that consider decreasing workgroup ids also appear in the forward progress guarantees of the HSA GPU programming model [HSA17]. This model, and the associated fairness guarantees, are discussed in Chapter 6.

Keeping one workgroup alive The cooperative kernel semantics dictate that the workgroup with id 0 will not be killed if it invokes offer_kill. This avoids the possibility of the cooperative kernel terminating early due to the programmer inadvertently allowing all workgroups to be killed, and the decision to keep workgroup 0 alive fits well with the choice to kill workgroups in descending order of id.

However, there might be a use case for a cooperative kernel reaching a point where it would be acceptable for the kernel to exit, although desirable for some remaining computation to be performed if competing workloads allow it. In this case, a semantics where all workgroups can be killed via offer_kill would be appropriate, and the programmer would need to guard each offer_kill with an id check in cases where killing all workgroups would be unacceptable. For example:

```
    if(get_group_id(0) != 0) offer_kill();
```

would ensure that at least workgroup 0 is kept alive.

Transmission of partial state from a single thread Recall from the semantics of request_fork that newly forked workgroups inherit the values of transmit-annotated variables associated with thread 0 of the forking workgroup. Alternative choices here would be to have forked workgroups inherit values for *all* variables from the forking workgroup, and to have thread i in the forking workgroup provide the valuation for thread i in each spawned workgroup, rather than having thread 0 transmit the valuation to all new threads.

The decision for transmitting only selected variables is based on the observation that many of a thread's private variables are dead at the point of issuing request_fork or resizing_global_barrier, thus it would be wasteful to transmit them. A live variable analysis could instead be employed to over-approximate the variables that might be accessed by newly arriving workgroups, so that these are automatically transmitted.

In all cases, it was found that a variable that needed to be transmitted had the property of being uniform across the workgroup. That is, despite each thread having its own copy of the variable, each thread was in agreement on the variable's value. As an example, the `level`, `in_nodes` and `out_nodes` variables used in Figure 5.5 are all stored in thread-private memory, but all threads in a workgroup agree on the values of these variables at each resizing_global_barrier call. As a result, transmitting the thread 0's valuation of the annotated variables is equivalent to (and more efficient than) transmitting values on a thread-by-thread basis. A real-world example where the current semantics would not suffice has not yet been encountered.

5.6 Evaluation applications and GPUs

Here, the experience of porting irregular kernels to cooperative kernels, which were used in the evaluation, is discussed (Section 5.6.1). The GPUs used in this chapters experiments are detailed (Section 5.6.2). The section concludes with a description of how non-cooperative workloads that mimic real interactive GPU tasks, specifically graphics rendering, were developed (Section 5.6.3).

5.6.1 Cooperative applications

Table 5.2 gives an overview of the 8 blocking applications ported to cooperative kernels in this work. There are four graph algorithms from the global barrier variants of the Pannotia [CBRS13] applications (see Section 3.6.2). Two of the applications are based on the Lonestar GPU applications of Section 3.6.3. The blocking idiom common to these two sets of applications is the global barrier. The table indicates how many of the original

Table 5.2: Blocking GPU applications investigated: the number of global barriers converted into resizing barriers, the number of cooperative constructs added, the lines of code (LoC) and the number of inputs.

App	Barriers to resizing	offer_kills	request_forks	transmits	LoC	Inputs
COLOR	2 / 2	0	0	4	55	2
MIS	3 / 3	0	0	0	71	2
BC	3 / 6	0	0	3	150	2
SSSP$_{\text{pannotia}}$	3 / 3	0	0	0	42	1
BFS	2 / 2	0	0	4	185	2
SSSP$_{\text{lonestar}}$	2 / 2	0	0	4	196	2
OCTREE	0 / 0	1	1	0	213	1
GAME	0 / 0	1	1	0	308	1

■ Pannotia ▨ Lonestar GPU □ work stealing

global barriers were changed to resizing barriers, and how many variables need to be annotated with transmit. In all cases, all of the global barriers were converted to resizing global barriers, except in the BC application, which contains barriers deeper in the call stack. Recall that this case is not supported by the prototype implementation, although these barriers could in principle be converted. Because there is an SSSP application common to both Pannotia and Lonestar GPU, a subscript is used to distinguish them. The Lonestar GPU approach uses a worklist approach while the Pannotia version uses sparse matrix operations. Thus, while the two applications share the same name, they implement significantly different strategies. Although many applications of Chapter 4 use a global barrier, these applications were not available at the time that the work of this chapter was conducted. However, the Lonestar GPU BFS application is similar to many of the applications of Chapter 4.

The remaining two applications are based on work stealing idioms from [CT08]. Originally written in CUDA, a port to OpenCL was required. These two applications required the addition of one request_fork and one offer_kill at the start of the main loop, and no variables needed to be transmitted, similar to the example discussed in Section 5.4.2.

Most graph applications come with two different data sets as input, while the work stealing applications have just one. This leads to 13 application/input pairs in total.

5.6.2 GPUs used in evaluation

The prototype scheduler implementation (Section 5.5) requires two optional features of OpenCL 2.0: SVM fine-grained buffers and SVM atomics. Out of the available GPUs (see Table 2.2), four met the requirements: Hd520, Hd5500, Iris, and R7. However, AMD's

Linux drivers for OpenCL suffer from a known defect whereby long-running kernels lead to defunct processes that the OS cannot kill [SD16b]. This issue was observed on R7 using the latest driver and thus made this chip difficult to experiment on. Additionally, alarming results were observed on this platform when using SVM atomics: modifying a kernel to include a simple CPU/GPU handshake through SVM slowed down kernel execution by an order of magnitude in some cases. This behaviour, combined with the defunct processes issue, lead us to believe that the support required for cooperative kernels is not yet mature enough within AMD's drivers. Thus the experiments were run on three Intel GPUs: Hd520, Hd5500 and Iris.

5.6.3 Developing non-cooperative kernels

Enabling rendering of smooth graphics in parallel with blocking irregular algorithms is an important use case for cooperative kernels. However, because the prototype implementation is based on a megakernel that takes over the entire GPU (see Section 5.5), the native interaction cannot be assessed directly.

The following method was devised to determine OpenCL workloads that simulate the computational intensity of various graphics rendering workloads. This method makes the following assumption: the system has a GUI driven by the GPU and insufficient GPU resources will cause the GUI to momentarily glitch.[2] First, a synthetic kernel is designed that occupies all compute units of a GPU for a parameterised time period t. This kernel is invoked in an infinite loop by a host application. A maximum value for t is found such that the synthetic kernel executes without having an observable impact on the graphics rendering of the host (i.e. there are no screen glitches). Using the computed value, the application is run for X seconds, the time $Y < X$ dedicated to GPU execution during this period is measured and the number of kernel launches n that were issued is recorded. The value $X \geq 10$ was used in all experiments. The values $(X - Y)/n$ and X/n estimate the average time spent using the GPU to render the display between kernel calls (call this E) and the period at which the OS requires the GPU for display rendering (call this P), respectively.

[2] This was observed to be the case on the Intel GPUs experimented with in this chapter.

Table 5.3: Period and execution time for each rendering task.

Rendering task	Period-P (ms)	Execution time-E (ms)
light	70	3
medium	40	3
heavy	40	10

This approach was used to measure the GPU availability required for three rendering tasks:

- *light*, whereby desktop icons were smoothly emphasised under the mouse pointer;
- *medium*, whereby window dragging over the desktop was smoothly animated; and
- *heavy*, which required smooth animation of a WebGL shader in a browser. In this case, the Chrome experiments were used.[3]

For each rendering task, the results are shown in Table 5.3. For medium and heavy, the $40ms$ period coincides with the human persistence of vision. The $3ms$ execution duration of both light and medium configurations indicates that GPU computation is cheaper for basic display rendering compared with more complex rendering.

5.7 Evaluation

Here, the results of running cooperative kernels on the prototype implementation are presented. First, the overhead associated with moving to cooperative kernels when multitasking is *not* required is examined (Section 5.7.1). Then, the responsiveness and throughput of non-cooperative workloads in the presence of cooperative workloads are examined (Section 5.7.2). The section concludes with a comparison of cooperative kernels against a model of *kernel-level* preemption, which appears to be what current Nvidia GPUs provide (Section 5.7.3).

5.7.1 Overhead of cooperative kernels

Invoking the cooperative scheduling primitives incurs some overhead even if no killing, forking or resizing actually occurs, because the cooperative kernel still needs to interact with the scheduler to determine this. This overhead is assessed by measuring the slowdown

[3] See `https://www.chromeexperiments.com`

Table 5.4: Cooperative kernel slowdown without multitasking.

Chip	**Overall**		**Global barrier apps**		**Work stealing apps**	
	geomean	max	geomean	max	geomean	max
HD520	1.20	1.75*	1.18	1.75*	1.42	1.42$^\diamond$
HD5500	1.14	1.45†	1.16	1.45†	1.10	1.18$^\diamond$
IRIS	1.08	1.37‡	1.07	1.37‡	1.12	1.23$^\diamond$

*COLOR[eco], †BC[128k], ‡SSSP$_{\text{lonestar}}$[usa.ny], $^\diamond$OCTREE

in execution time between the original and cooperative versions of a kernel, forcing the scheduler to never modify the number of active workgroups in the cooperative case.

Recall that the mega kernel-based implementation merges the code of a cooperative and a non-cooperative kernel. This can reduce the occupancy for the merged kernel, e.g. due to higher register pressure. This is an artefact of the prototype implementation, and would not be a problem if cooperative primitives were implemented inside the GPU driver. Thus, both the original and cooperative versions of a kernel were launched with the reduced occupancy bound in order to meaningfully compare execution times.

Results Table 5.4 shows the geometric mean and maximum slowdown across all applications and inputs, with averages and maxima computed over 10 runs per benchmark. For the maximum slowdowns, the application and input are indicated. The slowdown is below 1.75× even in the worst case, and closer to 1.2× on average. The best-performing chip is IRIS, with an average slowdown of 1.08× and a maximum slowdown of 1.37×. The worst-performing chip is HD520 with an average slowdown of 1.2× and a maximum slowdown of 1.75×. In all cases, the worst slowdown occurs on a global barrier application, although the application/input combination differs across chips. However, on average, work stealing applications suffer larger slowdowns than barrier applications, except for the HD520 chip.

These results are encouraging, especially since the performance of the prototype could clearly be improved upon in a native implementation.

5.7.2 Multitasking via cooperative scheduling

The responsiveness of multitasking between a long-running cooperative kernel and a series of short, non-cooperative kernel launches is now assessed. Additionally, the performance impact of multitasking on the cooperative kernel is investigated.

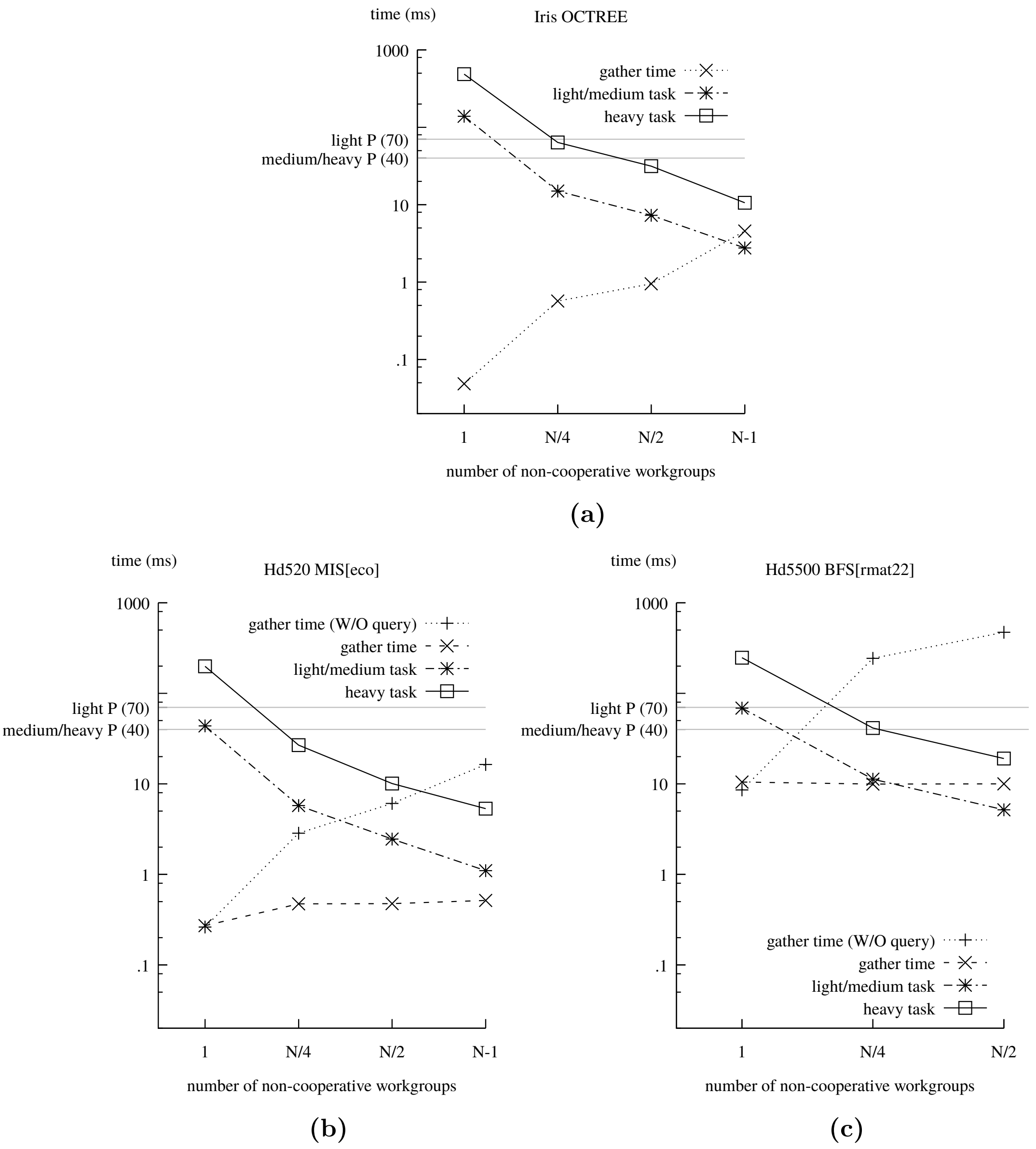
For a given cooperative kernel and its input, the kernel was launched and then a non-cooperative kernel was repeatedly scheduled. The non-cooperative kernel aims to simulate the intensity of one of the three classes of graphics rendering workloads discussed in Section 5.6.3. In these experiments, matrix multiplication was used as the non-cooperative workload, with matrix dimensions tailored to reach the appropriate execution duration. Four different cases for the number of workgroups requested by the non-cooperative kernel were considered: (1) one workgroup; (2) a quarter of the available workgroups; (3) half of the available workgroups; and (4) all-but-one of the available workgroups. For the graph algorithms, both the regular and query resizing global barriers were experimented with.

Thus, these experiments span 13 pairs of cooperative kernels and inputs, 3 classes of non-cooperative kernel workloads, 4 quantities of workgroups requested for the non-cooperative kernel, and 2 variations of resizing barriers for graph algorithms, leading to 288 configurations. Each configuration was run 10 times on each GPU in order to report averaged performance numbers. For each run, the execution time of the cooperative kernel was recorded. For each scheduling of the non-cooperative kernel during the run, the *gather time* needed by the scheduler to collect workgroups to launch the non-cooperative kernel and the non-cooperative kernel execution time was recorded. For every kernel (cooperative and non-cooperative), the results were sanity-checked by comparing their computed result with expected reference results.

To work around compiler crash and compiler hang bugs in Intel's current OpenCL drivers, some semantics-preserving changes were applied to a few of the applications. Driver bugs also meant that results could not be obtained for some configurations where the non-cooperative kernel asks for all-but-one workgroups (namely BC, BFS and $\text{SSSP}_{\text{lonestar}}$ on all chips, and COLOR on IRIS). Furthermore, the GAME application did not run at all on HD520. These driver bugs led to severe failures such as machine freezes and blue screens. Thus, the results exclude these configurations.

Responsiveness Figure 5.6 reports, on three configurations, the average gather and execution times for the non-cooperative tasks with respect to the number of workgroups allocated to it. A logarithmic scale is used for time since gather times tend to be much smaller than execution times. The horizontal grey lines indicates the desired period (P) for non-cooperative tasks. These graphs show a representative sample of the results; the full set of graphs for all configurations is provided in the extended version of the paper that this chapter is based on [SED17b].

The top graph (Figure 5.6a) illustrates results from the OCTREE work stealing application on IRIS. When the non-cooperative kernels (light, medium or heavy) uses only one workgroup, the execution time is long enough that the non-cooperative task cannot complete within the period required for a screen refresh. The gather time is very low though,



(a)

(b)

(c)

Figure 5.6: Gather time and non-cooperative task time results for three cases: (a) OCTREE on Iris; (b) MIS[eco] on Hd520; and (c) BFS[rmat22] on Hd5500.

since the scheduler needs to collect only one workgroup. The more workgroups that the non-cooperative task asks for, the faster the task computes: here the non-cooperative kernel becomes fast enough with a quarter (resp. half) of available workgroups for light and medium (resp. heavy) graphics workload. Inversely, the gather time increases since the scheduler must collect an increasing number of workgroups and thus, must wait for the workgroups executing the OCTREE application to encounter an offer_kill.

The bottom two graphs (Figures 5.6b and 5.6c) show results for graph algorithms on the other two GPUs, HD520 and HD5500 respectively. These applications use global barriers, and results are shown using both the regular and query barrier designs described in Section 5.5.2. The execution times for the non-cooperative task were averaged across all runs, including with both types of barrier. The average gather time associated with each type of barrier is shown separately. The graphs show a similar trend to the work stealing graph of Figure 5.6a: as the number of workgroups requested by the non-cooperative increases, the execution time of the non-cooperative tasks decrease and the gather time increases.[4]

A difference between the two graphs for the barrier applications is that the gather time is higher on the right graph (Figure 5.6c) than on left graph (Figure 5.6b). As discussed in the previous chapter (Section 4.4.4), the `rmat22` input is a high-degree graph, and thus executes resizing barriers infrequently for BFS. The MIS application encounters resizing barriers more frequently on the `eco` input. This is illustrated in Table 5.5, which shows how many resizing barriers are executed and the execution time on HD5500 for these two benchmarks. The average frequency of resizing barrier calls can then be computed as the benchmark execution time divided by the number of resizing barriers. The table shows that MIS[`eco`] executes resizing barriers more than 10× more frequently than BFS[`rmat22`]. The scheduler thus has fewer opportunities to collect workgroups and gather time increases.

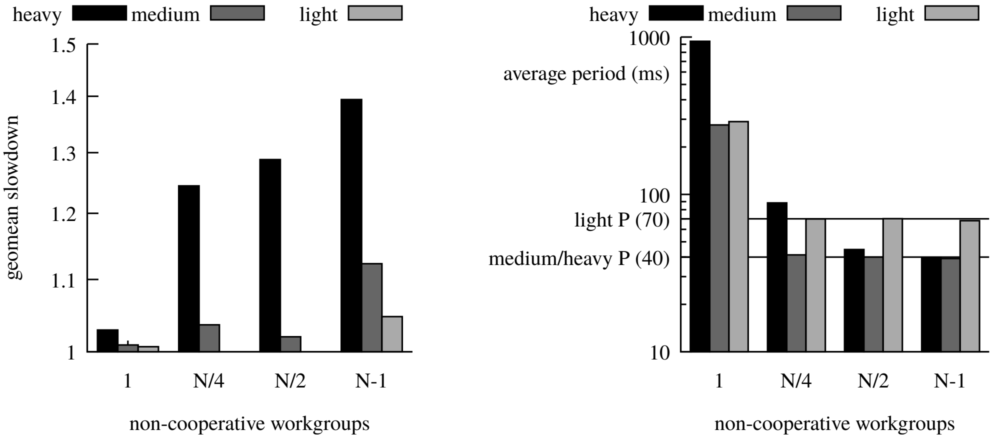
Nonetheless, on both global barrier examples, the scheduling responsiveness can benefit from the query barrier: when used, this barrier lets the scheduler collect all needed workgroups as soon as they hit a resizing barrier. As illustrated on both barrier graphs, the gather time of the query barrier is almost stable with respect to the number of workgroups that need to be collected.

Performance Figure 5.7a reports, for IRIS, the overhead brought by the scheduling of non-cooperative kernels over the cooperative kernel execution time. This is the slowdown associated with running the cooperative kernel in the presence of multitasking, vs. running

[4] The right figure has only 3 points on the x-axis: recall that BFS experienced an undiagnosed error with the final configurations, and thus there are no runs with "N-1" non-cooperative workgroups.

Table 5.5: The number of resizing barriers for the barrier applications of Figure 5.6b and 5.6c, along with their execution time (on HD5500) and average frequency of resizing barrier calls.

App[input]	resizing_global_barriers	Execution time (ms)	Avg. resizing frequency (ms)
BFS[rmat22]	30	311	10.37
MIS[eco]	3003	3263	1.09



(a)

(b)

Figure 5.7: (a) slowdown of cooperative kernel when multitasking various non-cooperative workloads, and (b) the period with which non-cooperative kernels are able to execute.

the cooperative kernel in isolation (geometric mean over all applications and inputs). Figure 5.7b shows the period at which non-cooperative kernels can be scheduled (arithmetic mean over all applications and inputs). The results for the other GPUs were similar. Results are partitioned for the three non-cooperative workloads: light, medium and heavy. The two horizontal lines Figure 5.7b correspond to the period goals of the workloads: the higher (resp. lower) line corresponds to a period of 70ms (resp. 40ms) for the light (resp. medium and heavy) workload.

The data includes some outliers that occur with benchmarks in which the resizing barriers are not called very frequently and the graphics task requires half or more workgroups. For example, a medium graphics workload for BFS on the rmat22 input has over an 8× overhead when asking for all-but-one of the workgroups. As Figure 5.7a shows, most of the benchmarks were much better-behaved than this. The massive overhead of this outlier is likely because resizing barriers were called infrequently enough that the cooperative kernel

Table 5.6: Application overhead of multitasking three graphics workloads using kernel-level preemption and using cooperative kernels.

Graphics task	Kernel-level	Cooperative	Resources
light	1.04	1.04	$N/4$
medium	1.08	1.08	$N/4$
heavy	1.33	1.39	$N/2$

was never able to fork workgroups and was thus resigned to executing the cooperative kernel with only one workgroup. Remedies for situations like this might include introducing semantically unnecessary resizing barriers to cooperative kernel in order to provide the scheduler with more opportunities to fork workgroups.

As Figure 5.7a shows, co-scheduling non-cooperative kernels that request a single workgroup leads to almost no overhead, but as Figure 5.7b shows, the period is far too high to meet the needs of any of the three non-cooperative workloads. For example, a heavy workload averages a period of 939ms, while a period of 40ms is required for a smooth graphics frame rate. As more workgroups are dedicated to non-cooperative kernels, they execute quickly enough to be scheduled at the required period. For the light and medium workloads, a quarter of the workgroups executing the non-cooperative kernel are able to meet their goal period (70ms and 40ms resp.). However, a quarter of the workgroups are not sufficient to meet the goal for the heavy workload (giving a mean period of 88ms and requiring 40ms). If half of the workgroups are allocated to the non-cooperative kernel, the heavy workload averages a period 10% over its goal (mean of 44ms). If all-but-one are allocated, the heavy workload reaches its goal period.

As expected, allocating more non-cooperative workgroups increases the execution time of the cooperative kernel. Still, heavy workloads meet their period by allocating all-but-one non-cooperative workgroups, incurring a slowdown of less than 1.4× on average. Light and medium workloads meet their period with only a negligible overhead of the cooperative kernel (less than a 1.03× slowdown on average).

Overall, these experimental findings are encouraging, especially because they provide a lower bound on potential performance of the cooperative kernel model. Implementing the model natively, with device-specific runtime support, could only improve performance and responsiveness compared with that of the megakernel-based prototype used in these experiments.

5.7.3 Comparison with kernel-level preemption

Nvidia's recent Pascal architecture provides hardware support for instruction-level preemption [NVI16, SA16]. However, this preemption occurs at the kernel granularity, and not at the finer granularity of workgroups, as described in this chapter. Intel GPUs do not provide this feature, and the OpenCL prototype of cooperative kernels cannot run on Nvidia GPUs, making a direct comparison impossible. Because empirical comparisons are not possible, a theoretical analysis of the overheads associated with sharing the GPU between graphics and compute tasks via kernel-level preemption is presented here.

Suppose a graphics workload is required to be scheduled with period P and duration D, and that a compute kernel requires time C to execute without interruption. Assuming the cost of preemption is negligible (e.g. Nvidia have reported preemption times of 0.1ms for Pascal [SA16], because of special hardware support), then the overhead associated with switching between compute and graphics every P time steps is $P/(P-D)$.

This theoretical kernel-level preemption overhead model is compared with the experimental cooperative kernel results of Figure 5.7 for each graphics workload. The reported cooperative kernel overhead is using the geomean slowdown of the configuration that allowed the deadline of the graphics task to be met. Based on the above assumptions, the cooperative kernel approach provides similar overhead for low and medium graphics workloads, however, has a higher overhead for the high workload.

The low performance of cooperative kernels for heavy workloads is because the graphics task requires half of the workgroups, crippling the cooperative kernel enough that request_fork calls are not issued as frequently. As mentioned earlier, future work may examine how to insert more resizing calls in these applications to address this. These results suggest that a hybrid preemption scheme may work well. That is, the cooperative approach works well for light and medium tasks; on the other hand, heavy graphics tasks benefit from the coarser grained, kernel-level preemption strategy. However, the preemption strategy requires appears to require specialised hardware.

5.8 Related Work

Much work has been done on accelerating irregular computations on GPUs, often relying on the occupancy-bound execution model; references to such work are given in Section 4.7. However, the occupancy-bound execution model is not guaranteed on current or future GPU platforms. As the experiments of this chapter demonstrate, the cooperative kernels model allows blocking algorithms to be upgraded to run in a manner that facilitates responsive multitasking.

GPU multitasking and scheduling Hardware support for preemption has been proposed for Nvidia GPUs, as well as *SM-draining*, whereby workgroups occupying a compute unit are allowed to complete until the SM becomes free for other tasks [TGC+14]. SM draining is limited the presence of blocking constructs, since it may not be possible to drain a blocked workgroup. A follow-up work adds the notion of SM *flushing*, where a workgroup can be re-scheduled from scratch if it has not yet committed side-effects [PPM15]. Both approaches have been evaluated using simulators, over sets of regular GPU kernels. Very recent Nvidia GPUs (i.e. the Pascal architecture) support preemption, though, as discussed in Section 5.2 and Section 5.7.3, it is not clear whether they guarantee fairness or allow tasks to share GPU resources at the workgroup level [NVI16].

A number of works have considered how best to schedule dynamic workloads on GPUs. Among these, the Whippletree approach employs a persistent megakernel to schedule multiple, dynamic tasks in a manner that aims to best utilise GPU resources [SKB+14], and the TimeGraph approach similarly aims to optimise scheduling of competing workloads, in the context of OpenGL [KLRI11]. None of these approaches tackles the problem of *fair* scheduling on GPUs, thus they do not aid in safe deployment of blocking irregular algorithms.

As mentioned throughout this thesis, CUDA and OpenCL provide the facility for a kernel to spawn further kernels, called nested parallelism in CUDA [Nvi18a, app. D] and dynamic parallelism in OpenCL [Khr15, pp. 32–33]. This can be used to implement a GPU-based scheduler, by having an initial scheduler kernel repeatedly spawn further kernels as required, according to some scheduling policy [MO16]. However, kernels that uses dynamic parallelism are still prone to unfair scheduling of workgroups, and thus does not help in deploying traditional blocking algorithms on GPUs.

Cooperative multitasking Cooperative multitasking is used by some current operating systems, especially in the domain of real-time computing. For example, FreeRTOS can be configured with either a preemptive or cooperative scheduling system [Ama17, p. 355]. RISC OS [RIS] currently only provides a cooperative scheduler. Additionally, cooperative multitasking can be implemented in high-level languages for cases in which preemptive multitasking is either too costly or not supported on legacy systems [Tar91].

In the context of GPUs, cooperative multitasking has been used to refer to the standard programming model, where the onus is on the GPU kernel to complete execution within a reasonable time budget [ACKS12, IOOH12]. This is in contrast to the cooperative kernel model, which specifically aims to support the needs of long-running GPU tasks.

5.9 Summary

This chapter has presented *cooperative kernels*, a small set of GPU programming extensions (summarised in Table 5.1) that allow long-running, blocking kernels to be fairly scheduled and, at the same time, dynamically adjust resource usage to account for external GPU requirements, e.g. multitasking and energy throttling. In particular, cooperative kernels allow global barrier synchronisation, which was shown to enable significant performance improvements for irregular applications in Chapter 4. In fact, Section 4.5.6 shows that the situations where the global barrier is most beneficial in terms of performance are exactly the situations where the resizing_global_barrier is able to reliably meet multitasking deadlines. These examples have many global barriers and short global barrier intervals, meaning that if the global barriers were changed to resizing_global_barriers, then the scheduler will frequently have the opportunity to manage resources.

Experimental results, using a megakernel-based prototype, show that the model is a good fit for current GPU-accelerated irregular algorithms. Light and medium graphics tasks are able to meet their deadlines with cooperative kernels with no more overhead than a kernel-level preemption approach, which likely requires special hardware support. The performance that could be gained through a native implementation with driver support would likely be even better. Additionally, programming with the cooperative kernel constructs is straightforward for existing blocking kernels. The placement of cooperative constructs could even be automated, e.g. they could be emitted by the optimising compiler of Chapter 4.

Another benefit of cooperative kernels is that they are *complementary* to preemption strategies. Indeed, if a GPU model had a fair scheduler enabled by preemption, the implementation could simply ignore the cooperative constructs, i.e. choosing never to kill or fork workgroups, or it might use cooperative constructs as *hints* to where more efficient preemption could occur. Such a hybrid model could overcome the shortcomings of cooperative kernels by guaranteeing deadlines, and also benefit from the strengths of cooperative kernels by efficiently preempting at cooperative hooks when possible.

6 A Formalisation of GPU Fairness Properties

6.1 Personal context

During the experimental campaigns of the previous chapters, several experiences with inter-workgroup synchronisation stood out to me. When executing a global barrier, the occupancy needed to be considered to avoid starvation. However, when executing a mutex, the occupancy did *not* need to be considered. Furthermore, a GPU scan application from the CUB library [Nvi] had a "waterfall" synchronisation pattern, in which workgroups passed values only to their immediate neighbours; we could not show starvation-freedom for this application using the occupancy-bound execution model. Clearly these idioms had different *blocking* properties. Looking through the literature, it seemed like there were many classes of *non-blocking* programs (e.g. non-locking, wait-free, obstruction free), but blocking programs always seemed to be lumped together without distinction.

Given this, I wanted to explore finer-grained classes of blocking programs and the corresponding fairness properties that schedulers need to provide in order to guarantee starvation-freedom. I was unaware of any previous attempt to formalise this area, and this was a glaring blindspot in the reasoning behind the applications we were developing. I was inspired by the formalisation efforts of weak memory models in [AMT14], where a single formal framework could be tweaked to describe the behaviours of different systems, and I hoped to lay some groundwork on something similar for fairness. This chapter presents an effort to bring some formal clarity to the fairness properties of various GPU schedulers.

6.2 Motivation

As discussed throughout this thesis, blocking synchronisation idioms, e.g. mutexes and barriers, play an important role in concurrent programming. Schedulers on traditional multi-core CPU systems typically provide fairness guarantees that are strong enough that blocking synchronisation works as expected. However, systems with *semi-fair* schedulers, e.g. GPUs, are becoming increasingly common. Such schedulers provide varying degrees of fairness, guaranteeing enough to allow some, but not all, blocking idioms. While a number of applications that use blocking idioms do run on today's GPUs (e.g. applications of Section 4 using the oitergb optimisation), reasoning about liveness properties of such applications is difficult as documentation and examples are scarce and scattered.

The aim of this chapter is to clarify fairness properties of semi-fair schedulers. To do this, a general temporal logic formula is defined, based on weak fairness and parameterised by a *thread fairness criterion*: a predicate that enables fairness per-thread at certain points of an execution. To investigate the practical implications of the formalisation, the fairness properties for three GPU schedulers are formally defined: OpenCL, HSA, and occupancy-bound execution. Then, existing GPU applications are examined and it is shown that none of these schedulers are strong enough to provide the fairness properties required by these applications.

Because these applications execute as expected on current GPUs, it hence appears that existing GPU scheduler descriptions do not entirely capture the fairness properties that are provided on current GPUs. Thus, fairness guarantees for two new schedulers are presented that aim to support existing GPU applications. The behaviour of common blocking idioms under each scheduler is examined and one of the new schedulers is shown to allow a more natural implementation of the discovery protocol of Chapter 3.

6.2.1 Schedulers

The *scheduler* of a concurrent system is responsible for the placement of virtual threads onto hardware resources. There are often insufficient resources for all threads to execute in parallel, and it is the job of the scheduler to dictate resource sharing, potentially influencing the temporal semantics of concurrent programs. For example, consider a two threaded program where thread 0 waits for thread 1 to set a flag. If the scheduler never allows thread 1 to execute then the program will hang due to starvation. Thus, to reason about liveness properties, developers must understand the *fairness* guarantees provided by the scheduler.

Current GPU programming models offer a compelling case study for scheduler semantics for three reasons: (1) some blocking idioms are known to hang due to starvation on current GPUs, e.g. as shown in Chapter 3, a global barrier will hang if executed with more workgroups than the occupancy bound; (2) other blocking idioms, e.g. mutexes, run without starvation on current GPUs; yet (3) documentation for some GPU programming models explicitly states that no guarantees are provided, while others state only minimal guarantees that are insufficient to ensure starvation-freedom even for mutexes. Because GPU schedulers are largely embedded in closed-source proprietary frameworks (e.g. drivers), this chapter does not consider concrete scheduling logic, but instead aims to derive formal fairness guarantees from prose documentation and empirical behaviours.

GPU threads in the same workgroup can synchronise efficiently via the OpenCL `barrier` instruction. Yet, despite practical use cases, there are no such intrinsics for inter-workgroup synchronisation. Instead, inter-workgroup synchronisation is achieved by building constructs, e.g. mutexes, using finer-grained primitives, such as atomic read-modify-write instructions (RMWs). However, reasoning about such constructs is difficult as inter-workgroup thread interactions are relatively unstudied, especially in relation to fairness. Thus, this chapter considers inter-workgroup interactions exclusively and the threads considered will be assumed to be in disjoint workgroups. Under this constraint, it is cumbersome to use the word *workgroup* and distracts from the theoretical aim this work. Because a workgroup can be thought of as being a "composite thread", henceforth the word *thread* is used to mean *workgroup* in this chapter.

The unfair OpenCL scheduler and non-blocking programs As noted in Chapter 3, OpenCL disallows all blocking synchronisation due to scheduling concerns, stating [Khr15, p. 31]: "A conforming implementation may choose to serialize the [threads] so a correct algorithm cannot assume that [threads] will execute in parallel. There is no safe and portable way to synchronize across the independent execution of [threads] . . . ." Such weak guarantees are acceptable for many popular GPU programs, such as matrix multiplication, as they are *non-blocking.* That is, these programs can terminate without starvation under an *unfair* scheduler, i.e. a scheduler that provides no fairness properties.

Blocking synchronisation and fair schedulers On the other hand, there are many useful *blocking* synchronisation idioms, which require fairness properties from the scheduler to ensure starvation-freedom. Three common examples of blocking idioms considered throughout this work, *barrier*, *mutex* and *producer-consumer (PC)*, are described in Table 6.1. Intuitively, a *fair* scheduler provides the guarantee that any thread that is able to execute will eventually execute. Fair schedulers are able to guarantee starvation-freedom for the idioms of Table 6.1.

Table 6.1: Blocking synchronisation idioms considered in this work.

Idiom	Description
barrier	Aligns the execution of all participating threads: a thread waits at the barrier until all threads have reached the barrier. Blocking, as a thread waiting at the barrier relies on the other threads to make enough progress to also reach the barrier.
mutex	Provides mutual exclusion for a critical section. A thread *acquires* the mutex before executing the critical section, ensuring exclusive access. Upon leaving, the mutex is *released*. Blocking, as a thread waiting to acquire relies on the thread in the critical section to eventually release the mutex.
producer-consumer (PC)	Provides a handshake between threads. A *producer* thread prepares some data and then sets a flag. A *consumer* thread waits until the flag value is observed and then reads the data. Blocking, as the consumer thread relies on the producer thread to eventually set the flag.

Table 6.2: Blocking synchronisation idioms guaranteed starvation-freedom under various schedulers.

	Fair	**HSA**	**OBE**	**Unfair (e.g. OpenCL)**
barrier	yes	no	occupancy-limited	no
mutex	yes	no	yes	no
PC	yes	one-way	occupancy-limited	no

6.2.2 Semi-fair schedulers

Two schedulers have been described so far: fair and unfair, under which starvation-freedom for blocking idioms is either always or never guaranteed, respectively. However, some GPU programming models have *semi-fair* schedulers, under which starvation-freedom is guaranteed for only some blocking idioms. Two such schedulers are now described and an informal analysis of the idioms of Table 6.1 under these schedulers is given (summarised in Table 6.2). If starvation-freedom is guaranteed for all threads executing idiom i under scheduler s then it is said that i is *allowed* under s.

Heterogeneous system architecture (HSA) Similar to OpenCL, Heterogeneous System Architecture (HSA) is a parallel programming model designed to efficiently target GPUs [HSA17]. Unlike OpenCL however, the HSA scheduler conditionally allows blocking between threads based on thread ids. Thread B can block thread A, if: "[thread] A comes after B in [thread] flattened id order" [HSA17, p. 46]. Under this scheduler: a barrier is *not* allowed, as all threads wait on all other threads regardless of id; a mutex is *not* allowed, as the ids of threads are not considered when acquiring or releasing the mutex; PC is *conditionally* allowed *if* the producer has a lower id than the consumer.

Occupancy bound execution (OBE) As presented in Chapter 3, occupancy-bound execution (OBE) is a pragmatic GPU execution model that aims to capture the guarantees that current GPUs have been shown experimentally to provide. While OBE is not officially supported, a substantial number of GPU programs (including applications of Chapter 4 using the oitergb optimisation and applications of Chapter 3) depend on the guarantees provided by this execution model. Recall that the OBE scheduler guarantees fairness among the threads that are currently *occupant* (i.e., are actively executing) on the GPU hardware resources. The fairness properties, given Section 3.3, are: "A [thread that has executed at least one instruction] is guaranteed to eventually be scheduled for further execution on the GPU." Under this scheduler: a barrier is *not* allowed, as all threads wait on all other threads regardless of whether they have been scheduled previously; a mutex *is* allowed, as a thread that has previously acquired a mutex will be fairly scheduled such that it eventually releases the mutex; PC is *not* allowed, as there is no guarantee that the producer will be scheduled fairly with respect to the consumer.

While general barrier and PC idioms are not allowed under OBE, Chapter 3 shows that constrained variants of these idioms are allowed by using the occupancy discovery protocol. Recall that the protocol, as described in Section 3.4, works by identifying a subset of threads that have been observed to take an execution step, i.e. it *discovers* a set of co-occupant threads. Barrier and PC idioms are then able to synchronise threads that have been discovered; thus OBE allows *occupancy-limited* variants of these idioms.

It is worth noticing that the two variants of PC shown in Table 6.2 (occupancy-limited and one-way) are incomparable. That is, one-way is not occupancy-limited, as the OBE scheduler makes no guarantees about threads with lower ids being scheduled before threads with higher ids. Similarly, occupancy-limited is not one-way, as the OBE scheduler allows bi-directional PC synchronisation if both threads have been observed to be co-occupant.

▶ **Remark** (CUDA). Like OpenCL, CUDA gives no scheduling guarantees, stating [Nvi18a, p. 11]: "[Threads] are required to execute independently: It must be possible to execute them in any order, in parallel or in series." Still, some CUDA programs rely on OBE or HSA guarantees (see Section 4). The recent version 9 of CUDA introduces *cooperative groups* [Nvi18a, app. C], which provide primitive barriers between programmer specified threads. Because only primitive barriers are provided, cooperative groups are not considered in this chapter. Indeed, the aim of this work is to reason about fine-grained fairness guarantees, as required by general blocking synchronisation.

6.2.3 Chapter contributions

The results of Table 6.2 raise the following points, which this chapter aims to address:

1. The temporal correctness of common blocking idioms varies under different GPU schedulers; however, no formal scheduler descriptions are known that are able to validate these observations.

2. Two GPU models, HSA and OBE, have schedulers that are incomparable. However, for each scheduler, there exist programs that rely on its scheduling guarantees. Thus, neither of these schedulers captures all of the guarantees observed on today's GPUs.

To address (1), a formula is provided, based on weak fairness, for describing the fairness guarantees of semi-fair schedulers. This formula is parameterised by a *thread fairness criterion* (TFC), a predicate over a thread and the program state, that can be tuned to provide a desired degree of fairness. This formula is evaluated by defining thread fairness criteria for the two existing semi-fair schedulers: HSA and OBE (Section 6.5).

To address (2), the claim that existing programs rely on HSA and OBE is substantiated by examining existing blocking GPU programs that run on current GPUs. It is shown that there exist programs that rely on HSA guarantees, as well as programs that rely on OBE guarantees (Section 6.6). That is, neither the HSA nor the OBE schedulers entirely capture guarantees on which existing GPUs applications rely.

To remedy this, the fairness properties of two new schedulers are presented, defined by their respective thread fairness criterion. The two new schedulers are: *HSA+OBE*, a simple combination of HSA and OBE, and *LOBE* (linear OBE), an intuitive strengthening of OBE based on contiguous thread ids. Both provide the guarantees required by current

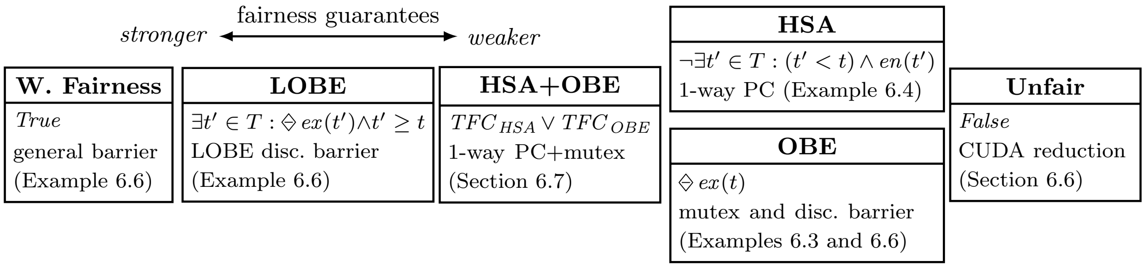


Figure 6.1: The semi-fair schedulers defined in this work from strongest to weakest.

programs (Section 6.7), however it is argued that LOBE corresponds to a more intuitive scheduler implementation. Next, an optimisation to the occupancy discovery protocol of Chapter 3 is shown, which exploits exclusive LOBE guarantees (Section 6.7.1).

The schedulers and their properties discussed in this chapter are summarised in Figure 6.1. Each scheduler is given a box in the figure. The first line in the box shows the scheduler's TFC (over a thread t), followed by the idiom(s) allowed under the scheduler and where in the chapter the idiom is analysed. Because the schedulers are ordered by strength, any idiom to the right of the schduler is allowed under the scheduler and any idiom to the left is disallowed. HSA and OBE are aligned vertically as they are incomparable.

To summarise, the chapter contributions are as follows:

- A formalisation of semi-fair schedulers using a temporal logic formula and the use this definition to describe the HSA and OBE GPU schedulers (Section 6.5).

- An analysis of blocking GPU applications showing that no existing GPU scheduler definition is strong enough to describe the guarantees required by all such programs (Section 6.6).

- Two new semi-fair schedulers that meet the requirements of current blocking GPU programs are presented: HSA+OBE and LOBE (Section 6.7). The guarantees of the LOBE scheduler are shown to provide a more natural implementation of the discovery protocol (Section 6.7.1).

Related work on programming models and GPU schedulers was discussed in Section 6.2.1, while applications that depend on specific schedulers are surveyed in Section 6.6.

Related publications The material presented in this chapter is based on material originally published in the 29th International Conference on Concurrency Theory (CONCUR'18) [SED18].

6.3 GPU program assumptions

The formal semantics in this chapter assume these program constraints, which are common to GPU applications:

1. *Termination*: programs are expected to terminate under a fair scheduler. GPU programs generally terminate, and in fact, they get killed by the OS if they execute for too long [SD16b].

2. *Static thread count*: while dynamic thread creation is recently available, e.g. through OpenCL nested parallelism [Khr15, pp. 32–33], this work addresses only static parallelism. The survey of programs in Section 6.6 did not reveal any programs that used nested parallelism.

3. *Deterministic threads*: the scheduler is the only source of nondeterminism; the computation performed by a thread depends only on the program input and the order in which threads interleave. This is the case for all GPU programs examined.

4. *Enabled threads*: all threads are *enabled*, i.e. able to be executed, at the beginning of the program and do not cease to be enabled until they terminate. While some systems contain scheduler-aware intrinsics, e.g. condition variables [Bar], GPU programming models do not. As a result, the idioms of Table 6.1 are implemented using atomic operations and busy-waiting, which do not change whether a thread is enabled or not.

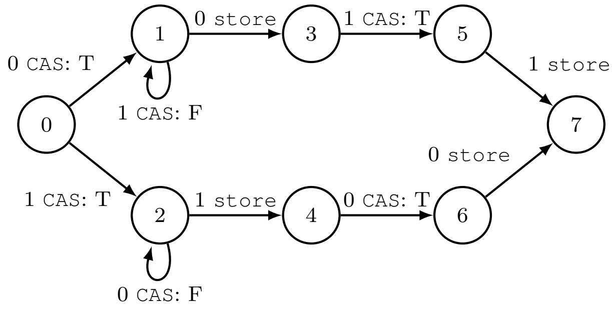
5. *Sequential consistency*: while GPUs have relaxed memory models (e.g. see [SD16a]), this work aims to understand scheduling under the interleaving model as a first step.

6.4 Formal program reasoning

A *sequential* program is a sequence of instructions and its behaviour can be reasoned about by step-wise execution of instructions. This work does not provide instruction-level semantics, but examples can be found in the literature (e.g. for GPUs, see [BCD$^+$15]). A *concurrent* program is the parallel composition of n sequential programs, for some $n > 1$. The set $T = \{0, 1, \ldots, n-1\}$ provides a unique id for each thread, often called the *tid*. The behaviour of a concurrent program is defined by all possible *interleavings* of atomic (i.e. indivisible) instructions executed by the threads. Let A be the set of available atomic instructions.

```
void thread0(mutex m) {
  // acquire
  while(!CAS(m,0,1));
  // release
  store(m,0);
}
void thread1(mutex m) {
  // acquire
  while(!CAS(m,0,1));
  // release
  store(m,0);
}
```

(a)



(b)

Figure 6.2: Two threaded mutex idiom (a) program code and (b) corresponding LTS.

For example, Figure 6.2a shows two sequential programs, `thread0` (with *tid* of 0) and `thread1` (with *tid* of 1), which both have access to a shared mutex object. The set A of atomic instructions is $\{$`CAS(m,old,new)`, `store(m,v)`$\}$, whose semantics are as follows:

- `CAS(m,old,new)`: atomically checks whether the value of `m` is equal to `old`. If so, updates the value to `new` and returns true. Otherwise returns false.

- `store(m,v)`: atomically stores the value of `v` to `m`.

Using these two instructions, Figure 6.2a implements a simple mutex idiom, in which each thread loops trying to acquire a mutex (via the `CAS` instruction), and then immediately releases the mutex (via the `store` instruction). While other mutex implementations exist, e.g. see [HS08, ch. 7.2], the blocking behaviour shown in Figure 6.2a is idiomatic to mutexes.

Labelled transition systems To reason about concurrent programs, a *labelled transition system* (LTS) is used. Formally, an LTS L is a 4-tuple $(S, I, L, \rightarrow)$ where

- S is a finite set of states, with $I \subseteq S$ the set of initial states. A state contains values for all program variables and a program counter for each thread.

- $L \subseteq T \times A$ is a set of labels. A label is a pair (t, a) consisting of a thread id $t \in T$ and an atomic instruction $a \in A$.

- $\rightarrow \subseteq S \times L \times S$ is a transition relation. For convenience, the relation $(p, (t, a), q) \in \rightarrow$ is sometimes abbreviated to $p \xrightarrow{t,a} q$, or even $p \xrightarrow{t} q$ if a is not relevant to the discussion. Leaving out the a element of the tuple is not ambiguous as only per-thread deterministic programs are considered. Dot notation is used to refer to members of the tuple; e.g., $\alpha.t$ is written to refer to the thread id component of α for an $\alpha \in \rightarrow$.

Given a concurrent program, the LTS can be constructed iteratively. A start state s is created with program initial values. For each thread $t \in T$, the next instruction $a \in A$ is executed to create state s' to explore. L is updated to include (t, a) and $(s, (t, a), s')$ is added to $\rightarrow$. This process iterates until there are no more states to explore. For example, the LTS for the program of Figure 6.2a is shown in Figure 6.2b. For ease of presentation, the state program values are omitted; labels show the thread id followed by the atomic action. If the action has a return value (e.g. `CAS`) that value is shown following the action. In the example figure, T and F indicate return values of true and false, respectively.

For a thread id $t \in T$ and state $p \in S$, t is said to be *enabled* in p if there exists a state $q \in S$ such that $p \xrightarrow{t} q$. This is captured by the predicate: $en(p, t)$. Next, p is a *terminal* state if no thread is enabled in p; that is, $\neg en(p, t)$ holds for all $t \in T$. This is captured by the predicate: $terminal(p)$. Intuitively, $en(p, t)$ states that it is possible for a thread t to take a step at state p and $terminal(p)$ states that all threads have completed execution at state p. The program constraints of Section 6.3 ensure that all threads are enabled until their termination.

Program executions and temporal logic The executions E of a concurrent program are all possible *paths* through its LTS. Formally, a path $z \in E$ is a (possibly infinite) sequence of transitions: $\alpha_0\alpha_1\ldots$, with each $\alpha_i \in \rightarrow$, where $\alpha_i.p$ and $\alpha_i.q$ refer to the p and q elements of the relation in $\rightarrow$. An execution path is constrained such that: the path starts in an initial state, i.e. $\alpha_0.p \in I$; adjacent transitions are connected, i.e. $\alpha_i.q = \alpha_{i+1}.p$; and if the path is finite, with n transitions, then it leads to a terminal state, i.e. $terminal(\alpha_{n-1}.q)$.

▶ **Remark** (Infinite paths). Because the programs in this work are assumed to terminate under fair scheduling (Section 6.3), infinite paths are discarded by the fair scheduler, but not necessarily semi-fair schedulers. Indeed, these infinite paths allow semi-fair schedulers to be distinguished.

Given a path z and a transition α_i in z, $\mathit{pre}(\alpha_i, z)$ is used to denote the transitions up to, and including, i of z, that is, $\alpha_0\alpha_1 \ldots \alpha_i$. Similarly, $\mathit{post}(\alpha_i, z)$ is used to denote the (potentially infinite) transitions of z from α_i, that is, $\alpha_i\alpha_{i+1} \ldots$. For convenience, $\mathit{en}(\alpha, t)$ is used to denote $\mathit{en}(\alpha.p, t)$ and $\mathit{terminal}(\alpha)$ is used to denote $\mathit{terminal}(\alpha.q)$. Finally, a new predicate is defined: $\mathit{ex}(\alpha, t')$, which holds if and only if $t' = \alpha.t$. Intuitively, $\mathit{ex}(\alpha, t')$ indicates that thread t' executes the transition α.

The notion of executions E over an LTS allows reasoning about *liveness* properties of programs. However, the full LTS may yield paths that realistic schedulers would exclude, illustrated in Example 6.1. Thus, fairness properties, provided by the scheduler, are modelled as a *filter* over the paths in E.

► **Example 6.1** (Mutex without fairness). The two-threaded mutex LTS given in Figure 6.2b shows that it is possible for a thread to loop indefinitely waiting to acquire the mutex if the other thread is in the critical section, as seen in states 1 or 2. Developers with experience writing concurrent programs for traditional CPUs know that on most systems, these non-terminating paths do not occur in practice!

Fairness filters and liveness properties can be expressed using *temporal logic*. For a path z and a transition α in z, temporal logic allows reasoning over $\mathit{post}(\alpha, z)$ and $\mathit{pre}(\alpha, z)$, i.e. reasoning about future and past behaviours. Following the classic definitions of fairness, linear time temporal logic (LTL), is used in this work (see, e.g. [BK08, ch. 5] for an in-depth treatment of LTL). For ease of presentation, a less common operator, $\diamondminus$, from past-time temporal logic (which has the same expressiveness as LTL [LPZ85]) is used. Temporal operators are evaluated with respect to z (a path) and α (a transition) in z. They take a formula ϕ, which is either another temporal formula or a transition predicate, ranging over $\alpha.p$, $\alpha.t$, or $\alpha.q$ (e.g. *terminal*). The three temporal operators used in this work are:

- The global operator $\Box$, which states that ϕ must hold for all $\alpha' \in \mathit{post}(\alpha, z)$.
- The future operator $\Diamond$, which states that ϕ must hold for at least one $\alpha' \in \mathit{post}(\alpha, z)$.
- The past operator $\diamondminus$, which states that ϕ must hold for at least at one $\alpha' \in \mathit{pre}(\alpha, z)$.

To show that a liveness property f holds for a program with executions E, it is sufficient to show that f holds for all pairs (z, α) such that $z \in E$ and α is the first transition in z. For example, one important liveness property is eventual termination: $\Diamond \mathit{terminal}$.

Applying this formula to the LTS of Figure 6.2b, a counter-example (i.e. a path that does not terminate) is easily found: $0 \xrightarrow{1} 2, (2 \xrightarrow{0} 2)^{\omega}$. In this path, thread 0 loops indefinitely trying to acquire the mutex. Infinite paths are expressed using ω-regular expressions [BK08, ch. 4.3].

However, many systems are able to reliably execute programs that use mutexes in a manner similar to Figure 6.2a. Such systems have *fair* schedulers, which do not allow the infinite looping paths described above. A fairness guarantee provided by a scheduler is expressed as a temporal predicate on paths and is used to filter out problematic paths before a liveness property, e.g. eventual termination, is considered.

In this work, *weak fairness* [BK08, p. 258] is considered, which is typically expressed as:

$$\forall t \in T : \Diamond\Box en(t) \implies \Box\Diamond ex(t) \tag{6.1}$$

Intuitively, weak fairness states that if a thread *is able to* execute, then it will *eventually* execute. There is also a notion of *strong fairness* that is useful in more complicated interactions, e.g. if per-thread actions are not deterministic. However, the program assumptions in this work do not allow programs containing such interactions.

▶ **Remark** (Weak fairness and program assumptions). The formula for weak fairness given in Equation 6.1 is the definition as traditionally given in the literature. However, the program assumptions given in Section 6.3 make the initial $\Diamond$ operator unnecessary. That is, the left-hand side of the implication in the original definition is satisfied when thread t is *eventually* enabled and *globally* remains enabled. The program assumptions state that all threads are initially enabled and remain enabled until termination. Thus, the $\Diamond$ predicate is trivially satisfied by every thread at the beginning of the execution. As a result, a simpler and equivalent form of weak fairness could be considered in this work:

$$\forall t \in T : \Box en(t) \implies \Box\Diamond ex(t) \tag{6.2}$$

Because the two definitions are equivalent under the program assumptions, this chapter will continue to reference the traditional definition due to its prevalence in the literature.

▶ **Example 6.2** (Mutex with weak fairness). Having defined fairness, proving termination for the LTS of Figure 6.2b can now be returned to. If the scheduler provides weak fairness, then any paths that do not satisfy the weak fairness definition can be discarded. The two problematic paths are: $0 \xrightarrow{1} 2, (2 \xrightarrow{0} 2)^\omega$ and $0 \xrightarrow{0} 1, (1 \xrightarrow{1} 1)^\omega$. Neither path satisfies weak fairness: in both cases the thread that can break the cycle is always enabled, yet it is not eventually executed once the infinite cycle begins. Thus, if executed on system which provides weak fairness, the program of Figure 6.2a is guaranteed to eventually terminate.

6.5 Formalising semi-fairness

The formalism for reasoning about fairness properties for semi-fair schedulers is now presented. Semi-fairness is parameterised by a per-thread predicate called the *thread fairness criterion*, or TFC. Intuitively, the TFC states a condition which, if satisfied by a thread t, guarantees fair execution for t.

Formally an execution is semi-fair with respect to a TFC if the following holds:

$$\forall t \in T : \Diamond\Box(en(t) \land TFC(t)) \implies \Box\Diamond ex(t) \tag{6.3}$$

The formula is similar to weak fairness (Equation 6.1), but in order for a thread t to be guaranteed eventual execution, not only must t be enabled, but the TFC for t must also hold. Semi-fairness for different schedulers, e.g. HSA and OBE, can be instantiated by using different TFCs, which in turn will yield different liveness properties for programs under these schedulers, e.g. as shown in Table 6.2.

The weaker the TFC is, the stronger the fairness condition is. Semi-fairness with the weakest TFC, i.e. true, yields classic weak fairness. Conversely, semi-fairness with the strongest TFC, i.e. false, yields no fairness.

Formalising a specific notion of semi-fairness now simply requires a TFC. This is illustrated by defining TFCs to describe the semi-fair guarantees provided by the OBE and HSA GPU schedulers, introduced informally in Section 6.2.1.

Formalising OBE semi-fairness The prose definition for the OBE scheduler fits this framework nicely, as it describes the per-thread condition for fair scheduling: once a thread has been scheduled (i.e. executed an instruction), it will continue to be fairly scheduled. This is straightforward to encode in a TFC using the $\diamondminus$ temporal logic operator (see

Section 6.4), which holds for a given predicate if that predicate has held at least once in the past. Thus the TFC for the OBE scheduler can be stated formally as follows:

$$TFC_{OBE}(t) = \diamondminus\, ex(t) \tag{6.4}$$

Formalising HSA semi-fairness A TFC for the HSA scheduler is less straightforward because the prose documentation is given in terms of relative allowed blocking behaviours, rather than in terms of thread-level fairness. Recall the definition from Section 6.2.1: thread B can block thread A if: "[thread] A comes after B in [thread] flattened id order" [HSA17, p. 46]. Searching the documentation further, another snippet phrased closer to a TFC is found, stating [HSA17, p. 28]: "[Thread] $i + j$ might start after [thread] i finishes, so it is not valid for a [thread] to wait on an instruction performed by a later [thread]." It is assumed here that j refers to any non-zero positive integer. Because these prose documentation snippets do not discuss fairness explicitly, it is difficult to directly extract a TFC. Thus, a best-effort attempt is made following this reasoning: (1) if thread i is fairly scheduled, no thread with id greater than i is guaranteed to be fairly scheduled; and (2) threads that are not enabled (i.e. they have terminated) have no need to be fairly scheduled. Using these two points, a TFC can be derived for HSA stating: a thread is guaranteed to be fairly scheduled if there does not exist another thread that has a lower id and is enabled. Formally:

$$TFC_{HSA}(t) = \neg\exists t' \in T : (t' < t) \wedge en(t') \tag{6.5}$$

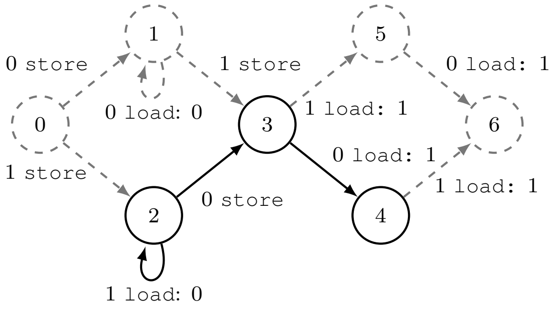
Although this TFC is somewhat removed from the prose snippets in the HSA documentation, this formal definition has value in enabling precise discussions about fairness. For example, confidence in this definition can be provided by illustrating that the idioms informally analysed in Section 6.2.1 behave as expected; see Example 6.3, 6.4 and 6.6.

▶ **Example 6.3** (Mutex with semi-fairness). Here, the mutex LTS of Figure 6.2b is analysed under OBE and HSA semi-fairness guarantees. Recall the two problematic paths (causing starvation) are: $0 \xrightarrow{0} 1, (1 \xrightarrow{1} 1)^\omega$. and $0 \xrightarrow{1} 2, (2 \xrightarrow{0} 2)^\omega$

- **OBE**: In both problematic paths, one thread t acquires the mutex, and the other thread t' spins indefinitely. However, thread t has executed an instruction (acquiring the mutex) and is thus guaranteed eventual execution under OBE; the problematic paths violate this guarantee as thread t never executes after it acquires. Therefore both paths are discarded, guaranteeing starvation-freedom for mutexes under OBE.
- **HSA**: The second problematic path: $0 \xrightarrow{1} 2, (2 \xrightarrow{0} 2)^\omega$, cannot be discarded as thread 0 waits for thread 1 to release. Thread 1 does not have the lowest id of the enabled threads, thus there is no guarantee of eventual execution. Therefore starvation-freedom for mutexes cannot be guaranteed under HSA.

```
void thread0(int x0, int x1) {
 // produce to tid 1
 store(x0,1);
 // consume from tid 1
 |while(load(x1) != 1);|
}
void thread1(int x0, int x1) {
 // produce to tid 0
 |store(x1,1);|
 // consume from tid 0
 while(load(x0) != 1);
}
```

(a)



(b)

Figure 6.3: Two threaded PC idiom (a) program code and (b) corresponding LTS. Omitting (a) lines in gray and (b) states and transitions in gray and dashed lines yields the one-way variant of this idiom.

▶ **Example 6.4** (PC with semi-fairness). Figure 6.3 illustrates a two-threaded producer-consumer (PC) program. A new atomic instruction, `load`, is used, which simply reads a value from memory (the return value is given on the LTS edges). Thread 0 produces a value via `x0` and then spins, waiting to consume a value via `x1`. Thread 1 is similar, but with the variables swapped. A subset of this program, omitting lines 4, 5, 8, and 9, shows the *one-way* PC idiom, where threads only consume from threads with lower ids, i.e., only thread 1 consumes from thread 0. The LTS for the one-way variant omits states 0, 1, 5, and 6 and the start state changes to state 2.

There are two problematic paths for the general test, in which one of the threads spins indefinitely waiting for the other thread to produce a value: $0 \xrightarrow{0} 1, (1 \xrightarrow{0} 1)^\omega$, and $0 \xrightarrow{1} 2, (2 \xrightarrow{1} 2)^\omega$. For the one-way variant, there is one problematic path: $(2 \xrightarrow{1} 2)^\omega$. This program is now analysed under OBE and HSA semi-fairness.

- **OBE**: Consider the problematic path $0 \xrightarrow{1} 2, (2 \xrightarrow{1} 2)^\omega$. Because thread 0 has not executed an instruction, OBE does not guarantee eventual execution for thread 0 and thus this path cannot be discarded. Similar reasoning shows that the problematic path for the one-way variant cannot be discarded either. Thus, neither general nor one-way producer consumer idioms are allowed under OBE.

- **HSA**: Consider the problematic path $0 \xrightarrow{0} 1, (1 \xrightarrow{0} 1)^\omega$. Because thread 1 does not have the lowest id of the enabled threads, HSA does not guarantee eventual execution for thread 1 and this path cannot be discarded. However, consider the problematic path for the one-way variant: $(2 \xrightarrow{1} 2)^\omega$. Because thread 0 has the lowest id of the enabled threads, HSA guarantees thread 0 will eventually execute, thus causing this path to be invalid. Therefore, general PC is not allowed under HSA, but one-way PC, following increasing order of thread ids, is allowed.

6.6 Inter-workgroup synchronisation in the wild

The scheduling guarantees assumed by existing GPU applications are now examined. This provides a basis for understanding (1) what scheduling guarantees are actually provided by existing GPUs, as these applications run without issues on current devices, and (2) the utility of schedulers, i.e. whether their fairness guarantees are exploited by current applications.

The exploration of GPU applications that use inter-workgroup synchronisation was performed on a best-effort basis by searching through popular works in this domain. Candidate programs were manually examined, searching for the idioms in Table 6.1, and relating them to the corresponding scheduler under which they are allowed.

OBE programs The most prevalent examples of applications that require OBE guarantees use the occupancy-limited global barrier. Such applications were studied in detail in Chapter 4. Earlier examples, such as the Xaio and Feng global barrier [XF10], which Chapter 3 references heavily, use a priori occupancy knowledge for scheduling guarantees. The Xaio and Feng work has been cited many times, mostly by works describing applications that are accelerated using the global barrier. Also, as mentioned throughout this thesis, the *persistent thread model* [GSO12] describes other use cases of OBE guarantees, including work stealing. A concrete work stealing application was originally presented in [CT08] and used as an example in Chapter 5. Thus, the OBE scheduler guarantees appear to be well-tested and useful on current GPUs.

HSA programs Only four applications that use the one-way PC idiom were found: two scan implementations, a sparse triangular solve (SpTRSV) application, and a sparse matrix vector multiplication (SpMV) application. While there are few applications in this category, it is argued that they are important, as they appear in vendor-endorsed libraries.

The two scan applications, one found in the popular Nvidia CUB GPU library [Nvi] and the second presented in [YLZ13], use a straightforward one-way PC idiom. Both scans work by computing workgroup-local scans on independent chunks of a large array. Threads compute chunks according to their thread id, e.g. thread 0 processes the first chunk. A thread t then passes its local sum to its immediate neighbour, thread $t+1$, who spins while waiting for this value. The neighbour factors in this sum and then passes an updated value to its neighbour, and so forth. A 2014 study on the performance of different GPU scan implementations found the CUB implementation to be the most efficient [Mer15].

The SpMV application, presented in [DG15], has several workgroups cooperate to calculate the result for a single row. Before any cooperation, the result must first be initialised, which is performed by the workgroup with the lowest id out of the cooperating workgroups. The other workgroups spin, waiting for the initialisation. This algorithm is implemented in the clSPARSE library [clS], a joint project between AMD and Vratis.

The SpTRSV application, presented in [LLH$^+$16], allows multiple producers to accumulate data to send to a consumer. However, in the triangular solver system, all producers will have lower ids than the relative consumers. Thus the PC idiom remains one-way.

OpenCL programs Applications that contain non-trivial inter-workgroup synchronisation and are non-blocking were also searched for. These applications would be guaranteed starvation-freedom under any scheduler, including the unfair OpenCL scheduler. The criteria for non-trivial synchronisation used is: inter-workgroup interactions that cannot be achieved by a single atomic read-modify-write (RMW) instruction. While examples of non-blocking data-structures were found (e.g. in the work-stealing applications of [Hwu11, ch. 35]), the top level loop was blocking as threads without work waited on other threads to complete work. Interestingly, we found only one application that appeared to be globally non-blocking: a reduction application in the CUDA SDK [Nvi18b], called `threadFenceReduction`, in which the final workgroup to finish local computations also does a final reduction over all other local computations.

6.7 Unified GPU semi-fairness

The exploration of applications in Section 6.6 shows that there are current applications that rely on either HSA or OBE guarantees. Because these applications run without starvation on current GPUs, it appears that current GPUs provide stronger fairness guarantees than either HSA or OBE describe. In this section, two new semi-fairness guarantees are proposed, which unify the HSA and OBE guarantees. As such, these new guarantees potentially provide a more accurate description of current GPUs schedulers.

HSA+OBE semi-fairness A straightforward approach to creating a unified fairness property from two existing semi-fair properties is to take the disjunction of the TFCs. Thus, threads guaranteed fairness under either existing scheduler are guaranteed fairness under the unified scheduler. This can be done with the HSA and OBE semi-fair schedulers to create a new unified semi-fairness condition, called HSA+OBE, i.e.,

$$TFC_{HSA+OBE}(t) = TFC_{HSA}(t) \vee TFC_{OBE}(t) \tag{6.6}$$

Thinking about the set of programs for which a scheduler guarantees starvation-freedom, let P_{HSA} be the set of programs allowed under HSA, with P_{OBE} and $P_{HSA+OBE}$ defined similarly. Note that $P_{HSA} \cup P_{OBE} \subset P_{HSA+OBE}$; that is, there are programs in $P_{HSA+OBE}$ that are neither in P_{HSA} nor P_{OBE}. For example, consider a program that uses one-way PC synchronisation and also a mutex. This program is not allowed under the OBE or HSA scheduler in isolation, but is allowed under the semi-fair scheduler defined as their disjunction. However, this idiom combination seems contrived as only four existing applications discussed in Section 6.6 exploits the one-way PC idiom (see Section 6.6) and it is not obvious that a mutex would be useful in these applications.

LOBE semi-fairness The HSA+OBE fairness guarantees may be sufficient for reasoning about existing applications, but these guarantees do not seem like they would naturally be provided by a system scheduler implementation. HSA+OBE guarantees fairness to (1) the thread with the lowest id that has not terminated (thanks to HSA) and (2) threads that have taken an execution step (thanks to OBE). For example, it might allow relative fair scheduling only between threads 0, 23, 29, and 42, if they were scheduled at least once in the past. Thus, HSA+OBE allows for "gaps", where threads with relative fairness do not have contiguous ids. Perhaps a more intuitive scheduler would guarantee that threads with relative fairness have contiguous ids.

Given these intuitions, a new semi-fair guarantee is defined, called *LOBE* (linear occupancy-bound execution). Similar to OBE, LOBE guarantees fair scheduling to any thread that has taken a step. Additionally, LOBE guarantees fair scheduling to any thread t if another thread t' (1) has taken a step, and (2) has an id greater than or equal to t (hence the word *linear*). Formally, the LOBE TFC can be written:

$$TFC_{LOBE}(t) = \exists t' \in T : \diamondsuit\, ex(t') \wedge t' \geq t \tag{6.7}$$

It is now shown that LOBE is a unified scheduler, i.e. any program allowed under HSA or OBE is allowed under LOBE. To do this, it is sufficient to show that $\mathit{TFC}_{OBE} \implies \mathit{TFC}_{LOBE}$ and $\mathit{TFC}_{HSA} \implies \mathit{TFC}_{LOBE}$. First, consider $\mathit{TFC}_{OBE} \implies \mathit{TFC}_{LOBE}$: this is trivial as the comparison check in TFC_{LOBE} includes equality, thus any thread that has taken a step is guaranteed to be fairly scheduled.

Considering now $TFC_{HSA} \implies TFC_{LOBE}$: first recall a property of executions from Section 6.4, namely that an execution either ends in a state where all threads have terminated, or it is infinite. Thus, at an arbitrary non-terminal point in an execution, some thread t must take a step. If t has the lowest id of the enabled threads, then both LOBE and HSA guarantee that t will be fairly executed. If t does not have the lowest id of the enabled threads, then LOBE guarantees that *all* threads with lower ids than t will be fairly executed, including the thread with the lowest id of the enabled threads, thus satisfying the fairness constraint of HSA.

6.7.1 LOBE discovery protocol

Because $\mathit{TFC}_{HSA+OBE}$ is defined as the disjunction of TFC_{HSA} and TFC_{OBE}, the reasoning in Section 6.7 is sufficient to show that LOBE fairness guarantees are at least as strong as HSA+OBE. A practical GPU program is now discussed for which correctness relies on the stronger guarantees provided by LOBE compared to HSA+OBE. This example shows that (1) LOBE guarantees are strictly stronger than HSA+OBE, and (2) fairness guarantees exclusive to LOBE can be useful in GPU applications.

The example is a modified version of the discovery protocol from Chapter 3, which dynamically discovers threads that are guaranteed to be co-occupant, and are thus guaranteed relative fairness by OBE. Recall that the protocol works using a virtual *poll*, in which threads have a short time window to indicate, using shared memory, that they are co-occupant. The protocol acts as a filter: discovered co-occupant threads execute a program, and undiscovered threads exit without performing any meaningful computation. Because only co-occupant threads execute the program, OBE guarantees that blocking idioms such as barriers can be used reliably.

GPU programs are often data-parallel, and threads use their ids to efficiently partition arrays; thus having contiguous ids is vital. Because OBE fairness does not consider thread ids, in order to provide contiguous ids, the discovery protocol dynamically assigns *new* ids to discovered threads (see Table 3.1 in Section 3.4). While functionally this approach is sound, there are two immediate drawbacks: (1) programs must be written using new thread ids, which can require intrusive changes, and (2) the native thread id assignment

Algorithm 6.1 Occupancy discovery protocol. Applying LOBE optimisation removes the code in $\dbox{\text{dashed boxes}}$ and adds the code in $\boxed{\text{solid boxes}}$.

```
1:  function DISCOVERY_PROTOCOL($open, count, \dbox{id\_map,}\ m$)
2:      Lock($m$)
3:      if $open \boxed{\vee (tid < count)}$ then
4:          $\dbox{id\_map[tid] \leftarrow count}$
5:          $\dbox{count \leftarrow count + 1}$
6:          $\boxed{count \leftarrow max(count, tid + 1)}$
7:          Unlock($m$)
8:      else
9:          Unlock($m$)
10:         return $False$
11:     Lock($m$)
12:     if $open$ then
13:         $open \leftarrow False$
14:     Unlock($m$)
15:     return $True$
```

on GPUs may be optimised by the driver for memory accesses in data-parallel programs; using new ids would forego these optimisations. Exploiting the scheduling guarantees of LOBE, the discovery protocol can be modified to preserve native thread ids and also ensuring contiguous ids.

▶ **Example 6.5** (Thread ids and data locality). It is possible that the protocol discovers four threads (with tids 2-5) and creates the following mapping for their new dynamic ids: $\{(5 \to 0), (2 \to 1), (3 \to 3), (4 \to 4)\}$. The GPU runtime might have natively assigned threads 2 and 3 to one processor (or a GPU compute unit) and threads 4 and 5 to another. Because these compute units often have caches, data-locality between threads on the same compute unit could offer performance benefits [WCL$^+$15]. In data-parallel programs, there is often data-locality between threads with consecutive ids. Thus, in our example mapping, the (native) threads, 2 and 5 could not exploit data locality, as their new ids are consecutive, but their native ids are not.

The discovery protocol algorithm, given in Figure 3.3, is reproduced in Algorithm 6.1. The changes made to exploit LOBE guarantees are indicated by $\dbox{\text{dashed boxes}}$ for removed code and $\boxed{\text{solid boxes}}$ for added code. Here, a quick reiteration of the original discovery protocol algorithm is given. The algorithm has two phases, both protected by the same mutex m. The first phase is the *polling phase* (lines 2-10), where threads are able to indicate that they are currently occupant (i.e. executing). The *open* shared variable is initialised to true to indicate that the poll is open. A thread first checks whether the poll is open (line 3). If so, then the thread marks itself as discovered; this involves obtaining a new id (line 4) and incrementing the number of discovered threads through the shared variable *count* (line 5). The thread can then continue to the closing phase (starting line 11). If the poll was not open, the thread indicates that it was not discovered by returning false (lines 8-10). In the closing phase, a thread checks to see if the poll is open; if so, the thread closes the poll and no other threads can be discovered at this point (lines 12-13). All threads who enter the closing phase have been discovered to be co-occupant, thus they return true (line 15). The total number of co-occupant threads will be stored in *count*.

This protocol can be optimised by exploiting fairness guarantees of LOBE. In particular, because LOBE guarantees that threads are fairly scheduled in contiguous id order, the protocol can allow a thread with a higher id to *discover* all threads with lower ids. As a result, threads are able to keep their native ids, although the number of discovered threads is still dynamic. The optimisation to the discovery protocol is simple: first the *id_map*, which originally mapped threads to their new dynamic ids is not needed (lines 1 and 4). Next, the number of discovered threads is no longer based on how many threads were observed to poll, but rather on the highest id of the discovered threads (line 6). Finally, even if the poll is closed, a thread entering the poll may have been discovered by a thread with a higher id; this is now reflected by each thread comparing its id with *count* (line 3). In Example 6.6, we show that a barrier prefaced by the LOBE optimised protocol is not allowed under HSA+OBE guarantees, and thus illustrate that LOBE fairness guarantees are strictly stronger than HSA+OBE.

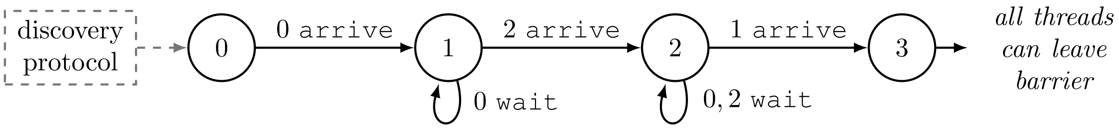


Figure 6.4: Sub-LTS of a barrier, with an optional discovery protocol preamble.

▶ **Example 6.6** (Barriers under semi-fairness). The behaviour of barriers is now analysed, with optional discovery protocols, under our semi-fair schedulers. Figure 6.4 shows a subset of an LTS for a barrier idiom that synchronises three threads with ids 0, 1, and 2. For the sake of clarity, instead of using atomic actions that correspond to GPU instructions, abstract instructions `arrive` and `wait` are used, which correspond to a thread marking its arrival and waiting at the barrier, respectively.

The sub-LTS shows one possible interleaving of threads arriving at the barrier, in the order 0, 2, 1. The final thread to arrive (thread 1) allows all threads to leave. The sub-LTS shows the various spin-waiting scenarios that can occur in a barrier at states 1 and 2. A discovery protocol can optionally be used before the barrier synchronisation.

The sub-LTS is first analysed using the LOBE optimised discovery protocol (Section 6.7.1). Recall that the LOBE discovery protocol discovers threads if it has seen a step from a thread with an equal or greater thread id. In our example with three threads, the fewest behaviours the protocol is guaranteed to have seen is a step by thread 2, denoted: $DP \xrightarrow{2} 0$.

- **HSA+OBE**: Consider the starvation path:
  $DP \xrightarrow{2} 0, 0 \xrightarrow{0} 1, 1 \xrightarrow{0} 2, (2 \xrightarrow{0} 2, 2 \xrightarrow{2} 2)^{\omega}$.
  This path cannot be disallowed by HSA+OBE as at state 2, HSA+OBE guarantees fair scheduling for the thread with the lowest id (thread 0) and any threads that have taken a step (threads 0 and 2). This path requires fair execution from thread 1 to break the starvation loop. Thus, barrier synchronisation using the LOBE discovery protocol is not allowed under HSA+OBE or any of the weaker schedulers (OBE and HSA).

- **LOBE**: The above starvation path is disallowed by LOBE, as LOBE guarantees fair execution for any thread t that has executed *and* any thread with a lower id than t. Because at state 0, the LOBE discovery protocol has observed a step from thread 2, fairness is guaranteed for threads 2, 1, and 0. Thus, barriers with LOBE discovery protocol are allowed under LOBE.

▶ **Example 6.6 cont.** Now, the general barrier (i.e. with no discovery protocol) is analysed:

- **LOBE**: The starvation path $0 \overset{0}{\to} 1, (1 \overset{0}{\to} 1)^\omega$ is not disallowed by LOBE, as LOBE cannot guarantee fair execution for any thread other than thread 0 at state 1 where the infinite starvation path begins. Thus, general barriers are not allowed under LOBE. Because LOBE is stronger than HSA+OBE, HSA and OBE, the general barrier is not allowed under these schedulers either.

Finally, the barrier using the original discovery protocol (as described in Section 6.7.1) is analysed:

- **HSA:** The starvation path $DP \xrightarrow{0,1,2} 0, 0 \overset{0}{\to} 1, (1 \overset{0}{\to} 1)^\omega$ is not disallowed by HSA, as HSA only guarantees fair execution to the lowest enabled thread (i.e. thread 0). To break this starvation loop in the sub-LTS, thread 2 would need fairness guarantees. Thus barriers using the original discovery protocol are not allowed under HSA.

- **OBE:** Because the original discovery protocol guarantees all threads have taken a step before the barrier execution (i.e. $DP \xrightarrow{0,1,2} 0$), OBE guarantees all three threads fair scheduling. Thus all starvation loops in the sub-LTS are guaranteed to be broken, and the barrier using the original discovery protocol is allowed under OBE. Because HSA+OBE and LOBE are stronger than OBE, this synchronisation idiom is also allowed under those schedulers.

6.8 Summary

Current GPU programming models provide loose scheduling fairness guarantees in English prose, if any at all. In practice, Chapter 3 has shown that GPUs feature *semi-fair* schedulers that are neither fair, nor totally unfair. The goal of this chapter is to clarify the fairness guarantees that GPU programmers can rely on, or at least the ones they assume. To this aim, a formal framework was introduced that combines the classic weak fairness with a thread fairness criterion (TFC), enabling fairness to be specified at a per-thread level. The framework is illustrated by defining the TFC for HSA (from its specification) and OBE (from its description in Chapter 3). The scheduling guarantees are used to analyse three classic concurrent programming idioms: barrier, mutex and producer-consumer.

While some existing GPU programs rely on either HSA or OBE guarantees, these two models are not comparable. Specifically, the one-way producer-consumer idiom is allowed under HSA but may starve under OBE. Conversly the occupancy-limited producer-consumer idiom is allowed under OBE but may starve under HSA. Thus, GPUs that aim to support the examined current GPU programs must support stronger guarantees that neither HSA nor OBE entirely capture. The presented framework allows for the straightforward combination of scheduling guarantees, which is used to define the HSA+OBE scheduler. Another unified scheduler, LOBE, is also defined, which offers slightly stronger fairness guarantees than HSA+OBE, but corresponds to a more intuitive scheduler implementation. The guarantees unique to LOBE are shown to be useful through a GPU protocol optimisation for which other GPU semi-fair schedulers do not guarantee starvation-freedom, but LOBE does.

The formal framework, examples, and application survey presented in this chapter lay a foundation on which we hope GPU developers and vendors can use to work toward a more clear understanding of scheduling guarantees on GPU accelerators.

7 Conclusion

The aim of this thesis was to provide an exploration of the GPU inter-workgroup barrier, and in particular to investigate the behaviour of such a barrier under various GPU semantics and situations where significant performance improvements might be enabled by such a barrier. This thesis concludes with a summary of the contributions and discussions of future work.

7.1 Contributions

This thesis has made the following original contributions in the exploration of inter-workgroup barrier synchronisation for GPUs:

- Chapter 3 introduced the *occupancy-bound execution* (OBE) model: an abstraction that captures forward progress guarantees empirically observed on many current GPUs. Using these guarantees, a *discovery protocol* was presented, which safely estimates a set of co-occupant workgroups; such workgroups are guaranteed relative forward progress. The discovery protocol, built on top of the execution model, allowed the implementation of a portable inter-workgroup barrier. Experimental results showed that, using heuristics, the discovery protocol was nearly always able to identify the maximum number of workgroups that can be co-occupant on the GPU. This scheme was successfully demonstrated on six GPUs spanning four vendors, showing, for the first time, a portable inter-workgroup barrier.

- Chapter 4 presented a set of GPU optimisations for *irregular applications* generalised to portable OpenCL. One of the optimisations required a portable global barrier, for which the scheme of Chapter 3 was used. This generalisation enabled a large empirical study, spanning 6 GPUs, 17 applications, and 3 inputs. All combinations of GPUs, applications and inputs were run under all available optimisation combinations. A methodology for evaluating performance trade-offs of portability versus specialisation was developed and used to evaluate various optimisation strategies in this domain. While the observed result trends were intuitive—more specialisation yields more performance—the methodology provides quantification to the trade-offs as well as insights into characteristics of GPUs, applications, or inputs. For example, GPUs that have a high kernel launch overhead can greatly benefit from using the optimisation enabled by the portable inter-workgroup barrier.

- Chapter 5 introduced the *cooperative kernels programming* model, developed in response to industry feedback suggesting that OBE progress guarantees may not hold for future GPUs; the main concern being that multi-tasking might create situations where occupant workgroups are unfairly preempted. The cooperative kernel model consists of three new primitives be added to the OpenCL programming model: (1) an instruction where a workgroup volunteers to be killed if its hardware resources are needed; (2) an instruction where a workgroup requests that additional workgroups can be forked if resources are available; and (3) an inter-workgroup barrier where workgroups can be killed or forked. These three instructions, via their interactions with a workgroup scheduler, can allow multi-tasking while also avoiding the high cost of workgroup preemption. Several long-running compute kernels that use work stealing and inter-workgroup barriers were shown to be easily adapted to the cooperative kernel programming model. A prototype scheduler was implemented in OpenCL and it was shown that the long-running compute kernels could be multi-tasked with three intensities of interactive tasks while meeting their deadlines.

- Chapter 6 formalised a hierarchy of progress guarantees using a temporal logic formula based on weak fairness. The formula is parameterised by a *thread fairness criterion* (TFC), which determines fairness on a per-thread basis. A TFC, and thus, progress guarantees, was formalised for each of the workgroup schedulers of the OpenCL, OBE, and HSA GPU programming models. Three common blocking idioms (mutex, barrier, and producer-consumer) were analysed under each model; each were shown to either deadlock due to starvation or successfully terminate. Under this analysis, it was shown that OBE and HSA are incomparable, yet there exist programs executing successfully on current GPUs relying on one or the other model. Thus, current GPUs appear to offer guarantees stronger than any existing description. To address this, two *unified* models were presented. The guarantees of one of the unified models, linear OBE (LOBE), was shown to enable an optimisation of the discovery protocol of Chapter 3.

7.2 Future work

Directions for future work are now discussed. Section 7.2.1 begins by presenting *immediate* avenues for future work, in which the thesis contributions could be extended in a straightforward way. Section 7.2.2 discusses the higher-level research context exposed in this work and ideas about future work that address the fundamental concepts that motivated this thesis.

7.2.1 Immediate future work

Utilising open stacks In this thesis, the OpenCL execution stack was treated as a black box; only the OpenCL API and kernel language were used. However, recently both AMD and Intel have provided open GPU stacks, including the OpenCL framework (e.g. compilers and drivers). Specifically, AMD has released ROCm[1] and Intel has released NEO.[2] Both of these open stacks provide lower-level controls to modify the OpenCL runtime. For example, it may be possible to query the occupancy bound for a given GPU kernel and natively provide a *maximal launch* [GSO12] option. Under this scheme, the program does not specify the number of workgroups, but instead the runtime provides as many workgroups as can be simultaneously occupant. Such a scheme would remove the need the for the discovery protocol, and thus, the associated application performance overheads reported in Sections 3.6.3 and 4.6.2. However, without official progress guarantees, such an effort would be relying on undocumented features and, as discussed in Chapter 5, such guarantees may not be supported on future GPUs.

Prescriptive performance models The work of Chapter 4 used *descriptive* models to explain the empirical results. However, such models require a large amount of empirical data to produce. The more common use case for an optimising compiler is to be able to determine effective optimisations without having to run a large number of experiments. This use case would require *prescriptive* performance models, in which a compiler would use static, or a small amount of dynamic, information to confidently pick an effective optimisation configuration for an application, GPU, and possibly input. Section 4.5.6 starts to lay the foundation for such models, by teasing out characteristics of applications, GPUs, and inputs for which certain optimisations appear to be effective. In particular, for the inter-workgroup barrier optimisation, the effective situations actually appear to

[1] See:
`https://rocm.github.io/`

[2] See:
`https://software.intel.com/en-us/forums/opencl/topic/758168`

be fairly straightforward. The simple micro-benchmark of Section 4.5.6 can measure the kernel launch overhead: the more overhead, the more effective the global barrier appears to be. Additionally, applications and inputs that cause a large number of short kernel launches also appear to be good candidates for inter-workgroup barrier optimisations. Such a prescriptive model could have been used in Section 3.6.2 to quickly determine that these tests were not suitable for barrier optimisations, yet many of the tests of Chapter 4 were suitable.

Complete progress models The work of Chapter 6 was constrained to inter-workgroup progress guarantees. However, semi-fair scheduling is found at other levels of the GPU hierarchy. For example, it is well-known that an intra-subgroup spin-lock will hang due to SIMD thread scheduling on pre-Volta Nvidia GPUs;[3] pilot experiments on other GPUs examined in this work show similar behaviour. However, intra-subgroup mutexes can be implemented if there is no blocking behaviour in *subgroup-divergent* blocks. Thus, a mutex in which *all* subgroup threads spin at the lock appears to work as intended on current GPUs. In fact, several applications of Chapter 4 use such a mutex without issue. Interestingly though, the mutex implementation must use `volatile` quantifiers, otherwise several of the compilers transform the convergent spin-lock into a divergent spin-lock and the application hangs. These experiences show that there is much needed formalisation in this area.

In addition to intra-subgroup interactions, intra-workgroup (and inter-subgroup) interactions should also be considered. Much like inter-workgroup scheduling, the OpenCL standard gives no scheduling guarantees at the intra-workgroup level of the GPU hierarchy, stating [Khr15, p. 31]:

> The work-items within a single work-group execute concurrently but not necessarily in parallel (i.e. they are not guaranteed to make independent forward progress).

These intra-workgroup scheduling guarantees were not explored in Chapter 6 as we were unaware of any motivating examples. Many programs use the OpenCL intra-workgroup barrier primitive instruction, which appears to mostly capture the needed blocking synchronisation at this level of the hierarchy.

[3] See: `https://stackoverflow.com/questions/2021019/implementing-a-critical-section-in-cuda`

Thus, a more complete scheduling model of GPUs should consider interactions at different levels of the hierarchy, much like how prior work on GPU memory models introduced new *scopes* and provided different semantics depending on the scope of the interaction [ABD$^+$15]. Ideally, each scope could be instantiated with a non-scoped progress model, e.g. the models presented in Chapter 6. Such a framework would be modular and provide a more complete picture of GPU scheduling.

High impact use case Although the applications studied throughout this thesis, e.g. BFS and SSSP, provide interesting challenges for GPU acceleration, it is not clear whether they have impact in high-priority domains where GPUs are vital for high performance, such as machine-learning or computer vision. The lack of a high impact use case may be one reason why the GPU language standards have not pursued official support of an inter-workgroup barrier. To encourage development of a portable impactful use case, one approach would be to provide a well-documented and approachable library of the optimised applications of Chapter 4. For example, Nvidia provides the nvGraph CUDA library [Nvi18d], which includes an API for SSSP and PR routines. Because Nvidia has dedicated resources to this library, it is likely that developers are using these routines. Thus, a portable OpenCL equivalent library may be appealing.

More concretely, previous work on Deep Recurrent Neural Networks (RNNs) has shown that a global barrier optimisation can improve performance by nearly an order of magnitude on Nvidia GPUs [DSC$^+$16]. This approach not only avoids the kernel launch overhead (as seen in Section 4.5.6), but also takes advantage of fast persistent local memory across barrier calls. None of the applications studied in this work exploit persistent local memory; there was not a clear path in the algorithms to use such a feature. The barrier-optimised RNN implementation appears to have been incorporated into CuDNN through the function `cudnnCreatePersistentRNNPlan` [Nvi18c, p. 68]. RNNs are used in state-of-the-art speech recognition systems,[4] and a portable optimised OpenCL version might enable a more diverse range of GPUs to be efficiently used for machine-learning applications.

[4] Seen firsthand during the internship at Microsoft while working on Cortana speech recognition system.

7.2.2 Fundamental future work

Even under possibly the simplest concurrent programming model, i.e. a sequentially consistent memory model and fair scheduling, bugs such as data races and deadlocks are readily found. The OBE progress model, as well as the other progress models, add complexities to the already difficult concurrency model. Although pragmatic, these complicated models might not be the ideal way forward.

Personal perspective I believe that fundamental future work should take a step back to examine our GPU programming models and find ways to simplify the model while allowing for intuitive and performance-enhancing idioms, such as the inter-workgroup barrier. This future work is not as straightforward as the ideas presented above. To carry this work in a meaningful way would require in-depth knowledge of GPU applications, GPU architecture, and programming languages. However, as GPUs (and other accelerators) become more common, how can we expect them to be programmable by developers who are not domain-experts with such complicated models?

The idea of aiming for simpler concurrent programming models is not new, and previous work can provide guidance on how to approach such a daunting task. Another domain that I am familiar with is weak memory models. In this domain, it appears that there are three schools of thought about how programming models should address weak memory:

1. The weak memory models on current architectures are difficult, but can be tackled by sufficient formalisation and exposition, e.g. see [AMT14].

2. Current memory models are weak, yet programmers should not have to deal with their complex semantics. Thus, a *contract* model is used whereby programmers follow certain rules and the architecture must provide intuitive semantics. If the rules are not followed, then program behaviour is undefined; e.g. the DRF model of [AH90].

3. An intuitive programming model is the highest priority. Architectures and compilers should work together to always ensure intuitive semantics, without the programmer required to follow any certain rules; e.g. see [MMM+15].

I believe research into applying any of these programming model approaches to forward progress concerns would be fruitful. I, however, am most interested in thinking about a *contract* model. For example, a progress contract might state: all blocking behaviours in a program must be annotated; if properly annotated, then the architecture guarantees that the program will exhibit behaviours as if run under a weakly-fair scheduler. A contract like this would require a formalisation of blocking behaviours and new annotations that can intuitively indicate such behaviours. I think that an exploration along these lines would be very interesting and I hope to pursue such work during my career.

7.3 Summary

Given the small, but growing, presence of GPU programs that contain blocking idioms, it seems likely that vendors and languages will have to officially support forward progress guarantees in some form. The longer that the community goes without official guarantees, the more programs are going to be written relying on empirical and folklore guarantees. This "wild west" environment is harmful to many central tenants of computer science, including portability, verification, and abstraction. This thesis has explored one such blocking idiom: the inter-workgroup barrier. Sufficient guarantees have been formalised and empirically shown to hold on current GPUs to support such a barrier. Situations where the barrier provides significant performance improvements have also been shown. Finally, foundations for future scheduling models have been proposed. These three investigations make a strong case to carry forward discussions with programmers and vendors about restoring order to the "wild west" by providing official documentation around progress guarantees and, hopefully, support for the inter-workgroup barrier.

Bibliography

[ABD+15] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. GPU concurrency: Weak behaviours and programming assumptions. In *ASPLOS*, pages 577–591. ACM, 2015.

[ACKS12] Jacob T Adriaens, Katherine Compton, Nam Sung Kim, and Michael J Schulte. The case for GPGPU spatial multitasking. In *HPCA*, pages 1–12, 2012.

[ACW+09] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. PetaBricks: A Language and Compiler for Algorithmic Choice. In *PLDI*, pages 38–49. ACM, 2009.

[AH90] Sarita V. Adve and Mark D. Hill. Weak ordering-a new definition. In *ISCA*, pages 2–14. ACM, 1990.

[AKV+14] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. OpenTuner: An Extensible Framework for Program Autotuning. In *PACT*, pages 303–316. ACM, 2014.

[Ama17] Amazon Web Services. The FreeRTOS reference manual: version 10.0.0 issue 1, 2017.

[AMD12] AMD. AMD GRAPHICS CORES NEXT (GCN) ARCHITECTURE, 2012. Whitepaper.

[AMT14] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014.

[And97] Paul Thomas Anderson. Boogie Nights, 1997. New Line Cinema.

[Ash16] Ben Ashbaugh. cl_intel_subgroups version 4, Aug. 2016.

[Bar] Blaise Barney. POSIX threads programming: Condition variables. `https://computing.llnl.gov/tutorials/pthreads/#ConditionVariables` (visited January 2018).

[Bax] Sean Baxter. ModernGPU. Retrieved June 2018 from `https://moderngpu.github.io/intro.html`.

[BCD$^+$15] Adam Betts, Nathan Chong, Alastair F. Donaldson, Jeroen Ketema, Shaz Qadeer, Paul Thomson, and John Wickerson. The design and implementation of a verification technique for GPU kernels. *ACM Trans. Program. Lang. Syst.*, 37(3):10, 2015.

[BDG13] Mark Batty, Mike Dodds, and Alexey Gotsman. Library abstraction for C/C++ concurrency. In *POPL*, pages 235–248. ACM, 2013.

[BDW16] Mark Batty, Alastair F. Donaldson, and John Wickerson. Overhauling SC atomics in C11 and OpenCL. In *POPL*, pages 634–648. ACM, 2016.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking.* The MIT Press, 2008.

[BNP12] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on GPUs. In *IISWC*, pages 141–151. IEEE, 2012.

[CB13] Charlie Curtsinger and Emery D. Berger. STABILIZER: Statistically Sound Performance Evaluation. In *ASPLOS*, pages 219–228. ACM, 2013.

[CBRS13] Shuai Che, Bradford M. Beckmann, Steven K. Reinhardt, and Kevin Skadron. Pannotia: Understanding irregular GPGPU graph applications. In *IISWC*, pages 185–195, 2013.

[CDKQ13] Peter Collingbourne, Alastair F. Donaldson, Jeroen Ketema, and Shaz Qadeer. Interleaving and lock-step semantics for analysis and verification of GPU kernels. In *ESOP*, pages 270–289. Springer, 2013.

[CHR$^+$16] L. W. Chang, I. E. Hajj, C. Rodrigues, J. Gmez-Luna, and W. M. Hwu. Efficient kernel synthesis for performance portable programming. In *MICRO*, pages 1–13. ACM, 2016.

[clS] clSPARSE. Retrieved June 2018 from `https://github.com/clMathLibraries/clSPARSE`.

[CSW18] Nathan Chong, Tyler Sorensen, and John Wickerson. The semantics of transactions and weak memory in x86, Power, ARMv8, and C++. In *PLDI*. ACM, 2018.

[CT08] Daniel Cederman and Philippas Tsigas. On dynamic load balancing on graphics processors. In *SIGGRAPH*, pages 57–64. Eurographics Association, 2008.

[CZF04] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In *SDM*, pages 442–446, 2004.

[DAV$^+$15] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O'Reilly, and Saman Amarasinghe. Autotuning Algorithmic Choice for Input Sensitivity. In *PLDI*, pages 379–390. ACM, 2015.

[DBGO14] Andrew Davidson, Sean Baxter, Michael Garland, and John D. Owens. Work-efficient parallel GPU methods for single-source shortest paths. In *IPDPS*, pages 349–359. IEEE Computer Society, 2014.

[DG15] M. Daga and J. L. Greathouse. Structural agnostic SpMV: Adapting CSR-adaptive for irregular matrices. In *HiPC*, pages 64–74. IEEE, 2015.

[dOFD$^+$13] Augusto Born de Oliveira, Sebastian Fischmeister, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Why you should care about quantile regression. In *ASPLOS*, pages 207–218. ACM, 2013.

[DSC$^+$16] Greg Diamos, Shubho Sengupta, Bryan Catanzaro, Mike Chrzanowski, Adam Coates, Erich Elsen, Jesse Engel, Awni Hannun, and Sanjeev Satheesh. Persistent RNNs: Stashing recurrent weights on-chip. In *Proceedings of The 33rd International Conference on Machine Learning (ICML)*, pages 2024–2033. PMLR, 2016.

[Gas15] Benedict Gaster. A look at the OpenCL 2.0 execution model. In *IWOCL*, pages 2:1–2:1. ACM, 2015.

[GDG$^+$17] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training ImageNet in 1 hour. *CoRR*, abs/1706.02677, 2017.

[GHH15] Benedict R. Gaster, Derek Hower, and Lee Howes. HRF-relaxed: Adapting HRF to the complexities of industrial heterogeneous memory models. *Trans. Archit. Code Optim. (TACO)*, 12(1):7:1–7:26, 2015.

[GSO12] Kshitij Gupta, Jeff Stuart, and John D. Owens. A study of persistent threads style GPU programming for GPGPU workloads. In *InPar*, pages 1–14, 2012.

[HHB$^+$14] Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. Heterogeneous-race-free memory models. In *ASPLOS*, pages 427–440, 2014.

[HN07] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *HiPC*, pages 197–208, 2007.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., 2008.

[HSA17] HSA Foundation. HSA programmer's reference manual: HSAIL virtual ISA and programming model, compiler writer, and object format (BRIG). (rev 1.1.1), March 2017.

[Hwu11] Wen-mei W. Hwu. *GPU Computing Gems Jade Edition.* Morgan Kaufmann, 2011.

[Int15a] Intel. The Compute Architecture of Intel Processor Graphics Gen8, 2015. Version 1.1.

[Int15b] Intel. The Compute Architecture of Intel Processor Graphics Gen9, Version 1.0, Aug. 2015.

[IOOH12] Fumihiko Ino, Akihiro Ogita, Kentaro Oita, and Kenichi Hagihara. Cooperative multitasking for GPU-accelerated grid systems. *Concurrency and Computation: Practice and Experience*, 24(1), 2012.

[ISO12] ISO/IEC. Standard for programming language C++, 2012.

[Jai91] Raj Jain. *The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling.* Wiley professional computing. Wiley, 1991.

[Khr15] Khronos Group. The OpenCL specification version: 2.0 (rev. 29), July 2015.

[Khr16a] Khronos Group. The OpenCL C specification version 2.0 (rev. 33), April 2016.

[Khr16b] Khronos OpenCL Working Group. The OpenCL extension specification, November 2016.

[Khr17] Khronos Group. The OpenCL specification version: 2.2 (rev. 2.2-3), May 2017.

[KLRI11] Shinpei Kato, Karthik Lakshmanan, Ragunathan Rajkumar, and Yutaka Ishikawa. Timegraph: GPU scheduling for real-time multi-tasking environments. In *USENIX ATC*, 2011.

[KVP$^+$16] Rashid Kaleem, Anand Venkat, Sreepathi Pai, Mary W. Hall, and Keshav Pingali. Synchronization trade-offs in GPU implementations of graph algorithms. In *IPDPS*, pages 514–523, 2016.

[LLH$^+$16] Weifeng Liu, Ang Li, Jonathan Hogg, Iain S. Duff, and Brian Vinter. A synchronization-free algorithm for parallel sparse triangular solves. In *Euro-Par*, pages 617–630. Springer, 2016.

[LLK$^+$14] S. H. Lo, C. R. Lee, Q. L. Kao, I. H. Chung, and Y. C. Chung. Improving GPU memory performance with artificial barrier synchronization. *IEEE Transactions on Parallel and Distributed Systems*, 25(9):2342–2352, 2014.

[LLS$^+$12] Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P. Rajan. GKLEE: concolic verification and test generation for GPUs. In *PPoPP*, pages 215–224. ACM, 2012.

[Lok11] Anton Lokhmotov. ARM Midgard architecture, 2011. Retrieved June 2018 from `http://www.heterogeneouscompute.org/hipeac2011Presentations/OpenCL-Midgard.pdf`.

[LPZ85] Orna Lichtenstein, Amir Pnueli, and Lenore Zuck. The glory of the past. In *Logics of Programs*, pages 196–218. Springer Berlin Heidelberg, 1985.

[Lub86] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15(4):1036–1053, 1986.

[MBP12] Mario Méndez-Lojo, Martin Burtscher, and Keshav Pingali. A GPU implementation of inclusion-based points-to analysis. In *PPoPP*, pages 107–116, 2012.

[MDHS09] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *ASPLOS*, pages 265–276. ACM, 2009.

[Mer] Bruce Merry. clogs: C++ library for sorting and searching in OpenCL applications.

[Mer15] Bruce Merry. A performance comparison of sort and scan libraries for GPUs. *Parallel Processing Letters*, 25, 2015.

[MGG12a] Duane Merrill, Michael Garland, and Andrew Grimshaw. Policy-based tuning for performance portability and library co-optimization. In *InPar*, pages 1–10, 2012.

[MGG12b] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU graph traversal. In *PPoPP*, pages 117–128. ACM, 2012.

[MMM$^+$15] Daniel Marino, Todd Millstein, Madanlal Musuvathi, Satish Narayanasamy, and Abhayendra Singh. The silently shifting semicolon. In *SNAPL*, LIPIcs, pages 177–189. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015.

[MO16] Pinar Muyan-Özçelik and John D. Owens. Multitasking real-time embedded GPU computing tasks. In *PMAM*, pages 78–87, 2016.

[MRH$^+$16] Saurav Muralidharan, Amit Roy, Mary Hall, Michael Garland, and Piyush Rai. Architecture-adaptive code variant tuning. In *ASPLOS*, pages 325–338. ACM, 2016.

[MSH$^+$14] Saurav Muralidharan, Manu Shantharam, Mary Hall, Michael Garland, and Bryan Catanzaro. Nitro: A framework for adaptive code variant tuning. In *IPDPS*, pages 501–512. IEEE Computer Society, 2014.

[MW47] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, 18(1):50–60, 1947.

[MYB16] Sepideh Maleki, Annie Yang, and Martin Burtscher. Higher-order and tuple-based massively-parallel prefix sums. In *PLDI*, pages 539–552. ACM, 2016.

[MZ16] MichałMrozek and Zbigniew Zdanowicz. GPU daemon: Road to zero cost submission. In *IWOCL*, pages 11:1–11:4. ACM, 2016.

[NBF96] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming.* O'Reilly & Associates, Inc., 1996.

[NCKB12] Sadegh Nobari, Thanh-Tung Cao, Panagiotis Karras, and Stéphane Bressan. Scalable parallel minimum spanning forest computation. In *PPoPPP*, pages 205–214, 2012.

[Nvi] Nvidia. CUB. `http://nvlabs.github.io/cub/` (visited January 2018).

[NVI99] NVIDIA. Graphics Processing Unit (GPU), 1999. archived at `https://web.archive.org/web/20160408122443/http://www.nvidia.com/object/gpu.html`.

[NVI16] NVIDIA. NVIDIA TESLA P100 GPU ARCHITECTURE, 2016. Whitepaper WP-08019-001_v01.1.

[NVI17] NVIDIA. NVIDIA TESLA V100 GPU ARCHITECTURE, 2017. Whitepaper WP-08608-001_v1.1.

[Nvi18a] Nvidia. CUDA C Programming Guide, Version 9.1, January 2018.

[Nvi18b] Nvidia. CUDA Code Samples, Version 9.1, January 2018.

[Nvi18c] Nvidia. CuDNN 7.1.2 Developer Guide, May 2018.

[Nvi18d] Nvidia. nvGraph Library User's Guide v10.0.130, May 2018.

[Nvi18e] Nvidia. Parallel thread execution ISA: Version 6.1, March 2018.

[OCY^{+}15] Marc S. Orr, Shuai Che, Ayse Yilmazer, Bradford M. Beckmann, Mark D. Hill, and David A. Wood. Synchronization using remote-scope promotion. In *ASPLOS*, pages 73–86. ACM, 2015.

[Ope15] OpenMP Architecture Review Board. OpenMP application program interface version 4.5, 2015.

[PGB^{+}05] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice.* Addison-Wesley Professional, 2005.

[PMS17] James Price and Simon McIntosh-Smith. Exploiting auto-tuning to analyze and improve performance portability on many-core architectures. In *High Performance Computing*, pages 538–556. Springer, 2017.

[Pol16] Adam Polak. Counting triangles in large graphs on GPU. In *IPDPS*, pages 740–746. IEEE Computer Society, 2016.

[PP16] Sreepathi Pai and Keshav Pingali. A compiler for throughput optimization of graph algorithms on GPUs. In *OOPSLA*, pages 1–19, 2016.

[PPM15] Jason Jong Kyu Park, Yongjun Park, and Scott A. Mahlke. Chimera: Collaborative preemption for multitasking on a shared GPU. In *ASPLOS*, pages 593–606, 2015.

[PTG13] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan. Improving GPGPU concurrency with elastic kernels. In *ASPLOS*, pages 407–418. ACM, 2013.

[RIS] RISC OS. Preemptive multitasking. Retrieved June 2018 from `http://www.riscos.info/index.php/Preemptive_multitasking`.

[SA16] Ryan Smith and Anandtech. Preemption improved: Fine-grained preemption for time-critical tasks, 2016. Retrieved June 2018 from `http://www.anandtech.com/show/10325/the-nvidia-geforce-gtx-1080-and-1070-founders-edition-review/10`.

[SD16a] Tyler Sorensen and Alastair F. Donaldson. Exposing errors related to weak memory in GPU applications. In *PLDI*, pages 100–113. ACM, 2016.

[SD16b] Tyler Sorensen and Alastair F. Donaldson. The hitchhiker's guide to cross-platform OpenCL application development. In *IWOCL*, pages 2:1–2:12, 2016.

[SDB$^+$16] Tyler Sorensen, Alastair F. Donaldson, Mark Batty, Ganesh Gopalakrishnan, and Zvonimir Rakamaric. Portable inter-workgroup barrier synchronisation for GPUs. In *OOPSLA*, pages 39–58, 2016.

[SED17a] Tyler Sorensen, Hugues Evrard, and Alastair F. Donaldson. Cooperative kernels: GPU multitasking for blocking algorithms. In *FSE*, pages 431–441, 2017.

[SED17b] Tyler Sorensen, Hugues Evrard, and Alastair F. Donaldson. Cooperative kernels: GPU multitasking for blocking algorithms (extended version). *CoRR*, 2017.

[SED18] Tyler Sorensen, Hugues Evrard, and Alastair F. Donaldson. GPU schedulers: How fair is fair enough? In *CONCUR*, pages 23:1–23:17. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018.

[Sin13] Graham Singer. The history of the modern graphics processor, 2013.

[SK10] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming.* Addison-Wesley Professional, 2010.

[SKB$^+$14] Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg. Whippletree: task-based scheduling of dynamic workloads on the GPU. *ACM Trans. Graph.*, 33(6):228:1–228:11, 2014.

[SKN10] J. Soman, K. Kishore, and P. J. Narayanan. A fast GPU algorithm for graph connectivity. In *IPDPSW*, pages 1–8, 2010.

[Sol09] Y. Solihin. *Fundamentals of Parallel Computer Architecture: Multichip and Multicore Systems.* Solihin Publishing, 2009.

[Sor14] Tyler Sorensen. Testing and exposing weak GPU memory models. Master's thesis, University of Utah, 2014.

[SPD18] Tyler Sorensen, Sreepathi Pai, and Alastair F. Donaldson. When one size doesnt fit all: Quantifying performance portability of graph applications on GPUs. 2018. *under submission.*

[SRD17] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Lift: A functional data-parallel IR for high-performance GPU code generation. CGO, pages 74–85. IEEE Press, 2017.

[STT10] Steven Solomon, Parimala Thulasiraman, and Ruppa K. Thulasiram. Exploiting parallelism in iterative irregular maxflow computations on GPU accelerators. In *HPCC*, pages 297–304, 2010.

[Tar91] Marc Tarpenning. Cooperative multitasking in C++. *Dr. Dobb's J.*, 16(4), April 1991.

[TGC$^+$14] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramírez, Nacho Navarro, and Mateo Valero. Enabling preemptive multiprogramming on GPUs. In *ISCA*, pages 193–204, 2014.

[TGEL11] Y. Torres, A. Gonzalez-Escribano, and D.R. Llanos. Understanding the impact of CUDA tuning techniques for Fermi. In *High Performance Computing and Simulation (HPCS)*, pages 631–639, 2011.

[TPO10] Stanley Tzeng, Anjul Patney, and John D. Owens. Task management for irregular-parallel workloads on the GPU. In *HPG*, pages 29–37, 2010.

[VC13] Narseo Vallina-Rodriguez and Jon Crowcroft. Energy management techniques in modern mobile handsets. *IEEE Communications Surveys and Tutorials*, 15(1):179–198, 2013.

[VHPN09] Vibhav Vineet, Pawan Harish, Suryakant Patidar, and P. J. Narayanan. Fast minimum spanning tree for large graphs on the GPU. In *HPG*, pages 167–171, 2009.

[VVdL$^+$15] Ana Lucia Varbanescu, Merijn Verstraaten, Cees de Laat, Ate Penders, Alexandru Iosup, and Henk Sips. Can portability improve performance?: An empirical study of parallel graph analytics. In *ICPE*, pages 277–287. ACM, 2015.

[WBSC17] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. Automatically comparing memory consistency models. In *POPL*, pages 190–204. ACM, 2017.

[WCL$^+$15] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey S. Vetter. Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations. In *ICS*, pages 119–130, 2015.

[WD98] R. Clint Whaley and Jack J. Dongarra. Automatically Tuned Linear Algebra Software. In *SC*, pages 1–27. IEEE Computer Society, 1998.

[WDP$^+$16] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on the GPU. PPoPP. ACM, 2016.

[Wik18] Wikipedia contributors. OpenCL — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=OpenCL&oldid=842868273`, 2018. [Online; accessed 12-June-2018].

[WLDP15] Joyce Jiyoung Whang, Andrew Lenharth, Inderjit S. Dhillon, and Keshav Pingali. Scalable data-driven pagerank: Algorithms, system issues, and lessons learned. In *Euro-Par*, pages 438–450. Springer, 2015.

[XF10] Shucai Xiao and Wu-chun Feng. Inter-block GPU communication via fast barrier synchronization. In *IPDPS*, pages 1–12, 2010.

[YLZ13] Shengen Yan, Guoping Long, and Yunquan Zhang. Streamscan: Fast scan algorithms for GPUs without global barrier synchronization. In *PPoPP*, pages 229–238. ACM, 2013.

[ZSC13] Yao Zhang, Mark Sinclair, and Andrew A. Chien. Improving Performance Portability in OpenCL Programs. In *Supercomputing*, Lecture Notes in Computer Science, pages 136–150. Springer, 2013.