

Imperial College London  
Department of Computing

**Fast static analysis for compile-time  
restructuring of application parallelism on  
Graphics Processing Units**

Nicolai Stawinoga

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



# Abstract

Parallelism is everywhere, with co-processors such as Graphics Processing Units (GPUs) accelerating the performance of applications such as training deep-learning neural networks, climate forecasting, bitcoin mining, medical imaging, or data analytics on platforms ranging from desktop computers to cloud computing and high performance clusters to mobile phones. Code optimisations enable realising the available performance of such devices, and automating these optimisations enables performance portability of software between different architectures.

In this thesis, we consider two code optimisations that can improve application performance by reducing the degree of hardware and software parallelism in a program execution: *thread coarsening*, which by merging threads reduces the number of threads launched, and *artificial occupancy reduction*, which limits the number of threads simultaneously processed by allocating superfluous resources.

We show how occupancy prediction through re-compilation can enable the selection of near-optimal coarsening factors at compile-time, by which thread coarsening can be applied in a fully automated manner without requiring auto-tuning. We demonstrate that our approach can achieve a maximum speedup of 5.08x (1.30x average) across three different NVidia GPU architectures, two modes of coarsening, different problem sizes, and for code pre-optimised to different degrees.

When trying to predict the likely effects of thread coarsening, it is important to consider the effects it might have on cache pressure. We describe how a fast static analysis based on partial symbolic execution can be implemented to identify cache line re-use in programs. We demonstrate how this heuristic approach can improve on the runtime and memory requirements of a more extensive re-use distance analysis by several orders of magnitude, causing it to be sufficiently light-weight for run-time execution. We show that the analysis is able to identify kernels that are likely to experience an increase in cache pressure after coarsening.

We explore the interaction of thread coarsening and artificial occupancy reduction, which can have negative effects on cache pressure and processor workload, respectively. We show that the two optimisation techniques can cancel these out when applied in combination, and yield a performance improvement of 8% in some cases. We investigate whether the cache line re-use analysis can identify candidates for artificial occupancy reduction.



*To my parents.*



# Acknowledgements

I would like to thank my supervisor, Dr. Tony Field, for everything he has done for me throughout my time in the Department of Computing, from giving my first lecture as an undergraduate to providing a constantly open door to his office for me as his PhD student during the last few years. His knowledge, patience and encouragement, as well as his passion for adventuring into new topics of computer science have undoubtedly contributed enormously to my own development as well as to my producing this thesis. I cannot imagine a better mentor and guide.

To Professor Paul Kelly, my second supervisor, I owe much of my knowledge about computer architecture and performance optimisations since my time as an undergraduate. For his knowledge and advice, often conveyed in deep, inspiring discussions over a casual cup of tea, I am forever grateful. Not to forget, he also introduced me to Professor J. Ramanujam, whose feedback (and name suggestion) for our cache line re-use analysis have been greatly appreciated.

I am deeply indebted to my undergraduate tutor, Professor Susan Eisenbach, for her mentorship over the years since my first year, for teaching me to teach students in their first year, and for her ability to cheerfully simplify any complex matter into merely a half sentence, which has so often provided invaluable insight. I am truly grateful to have learned so much from Susan.

I am grateful to my examiners, Professor Michael O'Boyle and Dr. Alastair Donaldson, for making my viva such a truly memorable experience. Their extensive feedback and suggestions have no doubt improved the quality of this thesis significantly.

Additionally, I would like to extend my thanks to at least a few of the people without whom this work would have likely not been possible, including Chris Pearson, Geoff Carruthers, Ben Ramadan, and countless of my friends and former colleagues at Formicary Ltd., who contributed so much to my own development, and for whose support during the transition into PhD studies I am exceedingly grateful. I wish to add my personal thanks to Franz Josef Klingen, my first computing teacher, whose early encouragement to self-study has proven truly invaluable, and to Robert von der Gracht, for guiding me through my Abitur. I am grateful for the many supportive people I met at the Department of Computing, including Professors Sophia Drossopoulou and Philippa Gardner, and my friend and class mate Dr. Will Jones, whose practical advice on completing a PhD and compiling a thesis have helped this work become reality.

Last but definitely not least, I would like to thank my family and friends for their continuous support and encouragements. While I could not possibly list names lest I forget someone, a special thanks must go to my parents, to whom this thesis is dedicated. After all, I would not be here without them.



# Contents

|          |                                                   |           |
|----------|---------------------------------------------------|-----------|
| <b>1</b> | <b>Introduction</b>                               | <b>1</b>  |
| 1.1      | Thesis structure and originality . . . . .        | 4         |
| 1.2      | Copyright declaration . . . . .                   | 4         |
| <b>2</b> | <b>GPU technologies</b>                           | <b>5</b>  |
| 2.1      | GPU Programming History . . . . .                 | 5         |
| 2.2      | The CUDA language and API . . . . .               | 9         |
| 2.3      | The OpenCL language and API . . . . .             | 12        |
| 2.4      | LLVM, SPIR, and AXTOR . . . . .                   | 15        |
| 2.5      | NVidia GPU Architectures . . . . .                | 17        |
| 2.5.1    | Thread Block Scheduling . . . . .                 | 17        |
| 2.5.2    | Streaming Multiprocessors . . . . .               | 20        |
| 2.5.3    | GPU Chipsets . . . . .                            | 27        |
| 2.5.4    | Memory Hierarchy . . . . .                        | 28        |
| 2.5.5    | Occupancy . . . . .                               | 29        |
| 2.6      | Related Work . . . . .                            | 30        |
| 2.6.1    | GPU programming and algorithms . . . . .          | 30        |
| 2.6.2    | GPU performance . . . . .                         | 30        |
| 2.6.3    | High-level general purpose languages . . . . .    | 31        |
| 2.6.4    | Domain-specific languages . . . . .               | 31        |
| 2.6.5    | Heterogeneous computing . . . . .                 | 32        |
| 2.6.6    | Auto-tuning . . . . .                             | 32        |
| 2.6.7    | Code optimisation tools and frameworks . . . . .  | 33        |
| 2.6.8    | Thread Coarsening . . . . .                       | 33        |
| 2.6.9    | Program analysis . . . . .                        | 34        |
| 2.6.10   | GPU verification and programmability . . . . .    | 35        |
| 2.6.11   | Benchmarks . . . . .                              | 35        |
| <b>3</b> | <b>Thread Coarsening</b>                          | <b>37</b> |
| 3.1      | Thread coarsening to reduce parallelism . . . . . | 37        |
| 3.2      | The code transformation . . . . .                 | 39        |
| 3.3      | Implementation . . . . .                          | 42        |

|          |                                                                                              |           |
|----------|----------------------------------------------------------------------------------------------|-----------|
| 3.4      | Thread-level coarsening . . . . .                                                            | 43        |
| 3.5      | Block-level coarsening . . . . .                                                             | 45        |
| 3.6      | Preserving block semantics . . . . .                                                         | 46        |
| <b>4</b> | <b>Predicting Optimal Coarsening Factors</b>                                                 | <b>49</b> |
| 4.1      | Coarsening Factor Selection . . . . .                                                        | 49        |
| 4.2      | An occupancy-based model for coarsening factor selection . . . . .                           | 51        |
| 4.3      | Implementation . . . . .                                                                     | 52        |
| 4.4      | Evaluation . . . . .                                                                         | 53        |
| 4.5      | Performance for varying problem sizes . . . . .                                              | 58        |
| 4.6      | Discussion . . . . .                                                                         | 61        |
| <b>5</b> | <b>Static analysis for identifying cache pressure</b>                                        | <b>63</b> |
| 5.1      | Effects of coarsening on cache pressure . . . . .                                            | 63        |
| 5.2      | Re-use distance analysis . . . . .                                                           | 65        |
| 5.3      | Polyhedral analysis . . . . .                                                                | 65        |
| 5.4      | Approximate cache line re-use analysis . . . . .                                             | 66        |
| 5.5      | Memory Access Descriptors for cache line re-use analysis . . . . .                           | 69        |
| 5.5.1    | Example 1: A simple mapping function . . . . .                                               | 69        |
| 5.5.2    | Example 2: Two-Dimensional Matrix Transpose . . . . .                                        | 72        |
| 5.5.3    | Example 3: Conditionals . . . . .                                                            | 74        |
| 5.5.4    | Example 4: Loops . . . . .                                                                   | 77        |
| 5.6      | Comparison with re-use distance analysis . . . . .                                           | 80        |
| 5.7      | Implementation . . . . .                                                                     | 81        |
| 5.8      | Evaluation . . . . .                                                                         | 82        |
| 5.8.1    | Kernels Safe to Coarsen . . . . .                                                            | 83        |
| 5.8.2    | Kernels with Cache Line Re-use . . . . .                                                     | 84        |
| 5.8.3    | Data-dependent Kernels . . . . .                                                             | 86        |
| 5.8.4    | Speedups . . . . .                                                                           | 86        |
| 5.8.5    | Coarsening at lower occupancy . . . . .                                                      | 87        |
| 5.8.6    | Discussion . . . . .                                                                         | 88        |
| <b>6</b> | <b>Artificial occupancy reduction</b>                                                        | <b>91</b> |
| 6.1      | Motivation . . . . .                                                                         | 91        |
| 6.2      | Implementing a compiler pass . . . . .                                                       | 92        |
| 6.3      | Experiments . . . . .                                                                        | 95        |
| 6.3.1    | Case study: Matrix transpose . . . . .                                                       | 96        |
| 6.3.2    | Cache line re-use analysis as an indicator for artificial occupancy re-<br>duction . . . . . | 98        |
| 6.3.3    | Coarsening and artificial occupancy reduction . . . . .                                      | 101       |
| 6.4      | Discussion . . . . .                                                                         | 104       |

|          |                                        |            |
|----------|----------------------------------------|------------|
| <b>7</b> | <b>Conclusions</b>                     | <b>107</b> |
| 7.1      | Applications and future work . . . . . | 108        |
| 7.2      | Closing remarks . . . . .              | 109        |



# List of Tables

|     |                                                                                        |     |
|-----|----------------------------------------------------------------------------------------|-----|
| 2.1 | CUDA built-in variables. . . . .                                                       | 11  |
| 2.2 | Differences in terminology between OpenCL and CUDA. . . . .                            | 13  |
| 3.1 | Implementation details for thread and block-level coarsening. . . . .                  | 46  |
| 4.1 | Properties of the test systems. . . . .                                                | 54  |
| 4.2 | Optimisation levels of reduction kernels from the NVidia OpenCL SDK. . . . .           | 55  |
| 4.3 | Kepler, thread-level coarsening. . . . .                                               | 57  |
| 4.4 | Maxwell, thread-level coarsening. . . . .                                              | 57  |
| 4.5 | Pascal, thread-level coarsening. . . . .                                               | 57  |
| 4.6 | Kepler, block-level coarsening. . . . .                                                | 58  |
| 4.7 | Maxwell, block-level coarsening. . . . .                                               | 58  |
| 4.8 | Pascal, block-level coarsening. . . . .                                                | 58  |
| 5.1 | Overview of Rodinia kernels used in our experiments. . . . .                           | 84  |
| 5.2 | Achieved speedups for Rodinia kernels. . . . .                                         | 86  |
| 6.1 | Performance of Rodinia benchmarks at artificially reduced occupancy. . . . .           | 99  |
| 6.2 | Performance of coarsened Rodinia benchmarks at artificially reduced occupancy. . . . . | 102 |



# List of Figures

|      |                                                                                       |     |
|------|---------------------------------------------------------------------------------------|-----|
| 2.1  | Intel microprocessor trends . . . . .                                                 | 6   |
| 2.2  | OpenCL work-item built-in functions and kernel execution context. . . . .             | 13  |
| 2.3  | LLVM compilation pipeline . . . . .                                                   | 16  |
| 2.4  | Thread block schedulers of the Fermi and Kepler architectures. . . . .                | 18  |
| 2.5  | Dynamic parallelism in CUDA . . . . .                                                 | 19  |
| 2.6  | Diagram of a CUDA core. . . . .                                                       | 20  |
| 2.7  | Streaming Multiprocessor (SMX) of the GK110 Kepler architecture. . . . .              | 21  |
| 2.8  | Streaming Multiprocessor (SMM) of the GM204 Maxwell architecture. . . . .             | 22  |
| 2.9  | Streaming Multiprocessor (SM) of the GP104 Pascal architecture. . . . .               | 23  |
| 2.10 | Warp scheduler of the Fermi architecture. . . . .                                     | 25  |
| 2.11 | Instruction dispatch in the Fermi architecture. . . . .                               | 26  |
| 2.12 | Full-chip block diagram of the GM204 chipset, Maxwell architecture. . . . .           | 27  |
| 2.13 | Memory hierarchy of the Kepler architecture. . . . .                                  | 28  |
| 3.1  | Effects of applying thread coarsening by a factor of 2 to a simple kernel. . . . .    | 38  |
| 3.2  | Effects on launch configuration of applying a coarsening factor of 2 . . . . .        | 39  |
| 3.3  | An LLVM Coarsening Pipeline. . . . .                                                  | 42  |
| 3.4  | Code structure of a kernel when applying thread and block-level coarsening. . . . .   | 44  |
| 4.1  | Speedups achieved by coarsening the reduction benchmarks. . . . .                     | 56  |
| 4.2  | The reduction1 benchmark with different input sizes. . . . .                          | 61  |
| 5.1  | The memcpy kernel before and after coarsening. . . . .                                | 64  |
| 5.2  | Performance effects of coarsening on the matrix transpose kernel. . . . .             | 64  |
| 5.3  | The coarsening pipeline extended by cache line re-use analysis. . . . .               | 80  |
| 5.4  | Coarsening flowchart . . . . .                                                        | 81  |
| 5.5  | Speedup and occupancy for coarsening Rodinia kernels. . . . .                         | 85  |
| 6.1  | Matrix-transpose kernel with artificially reduced occupancy. . . . .                  | 96  |
| 6.2  | Coarsened matrix-transpose kernels with artificially reduced occupancy. . . . .       | 97  |
| 6.3  | Performance impact of artificial occupancy reduction on the hotspot3D kernel. . . . . | 100 |
| 6.4  | Effects of coarsening and artificial occupancy reduction. . . . .                     | 103 |



# Introduction 1

The landscape of modern computing architectures is increasingly more parallel and diverse. Designs range from heterogeneous multi-core processors in the manner of ARM's *big.LITTLE* architecture [34], vector instruction sets as employed in the manner of Intel and AMD [26], Tensor Processing Units [42] in the manner of Google, or *Graphics Processing Units* (GPUs) in the manner of ATI or NVidia (c.f. [104]), enabling high performance in a variety of different contexts. Cross-platform languages such as OpenCL [36] or higher-level languages enable *functional portability* of code across different architectures. However, the increasing diversity of parallel architectures has given rise to the *performance portability problem* (see [25]), which is the challenge that code optimised for one particular architecture is not likely to achieve the same good performance on a different architecture.

The problem of performance portability is, in some sense, one of performance automation. If, for instance, a given program can be automatically optimised for a specific architecture, and the process can be repeated for a different platform, the program is said to be performance portable. In practice, such portability is not straightforward – different optimisations and optimisation parameters may need to be applied to achieve optimal performance, and the more architectural differences exist between targeted platforms, the more the optimisations applied may differ. Although a general solution to the problem of performance portability does not currently exist, many approaches and successful partial solutions have been presented. These include:

- *Compiling from higher-level languages*, which takes the approach of starting from a high-level, typically functional description of the program, for instance in the manner of Mainland and Morrisett [52]. The idea is that by abstracting away implementation details and in particular memory access patterns, decisions regarding implementation details can be left to the compiler. In reality, this generally results in very large optimisation spaces, while it is often not clear what the right choices should be.
- *Code generation from domain-specific languages* (DSLs), which limits the focus to one specific problem domain. While also starting from a high-level representation of the problem, the approach allows for deep knowledge of problem domain and hence the code to be generated, to be combined with detailed understanding of the targeted architectures, often resulting in very high-performing solutions (e.g., see [74, 87]).
- *Auto-tuning*, which takes a 'black box' approach to performance automation, assuming

no or limited knowledge of either code or targeted architecture [22]. As a way of automatically traversing the optimisation space, differently optimised versions of a program are repeatedly compiled, executed, and evaluated in order to establish the best performing one. Auto-tuning can be very useful in smaller, more contained contexts, but can become time-consuming when cross-evaluating combinations of optimisation parameters in larger search spaces.

- *Code libraries*, which offer manually tuned, high-performing code for specific use-cases to be incorporated in general-purpose code, and are thus characteristically similar to code generation in DSL contexts.

In this thesis we will take an integrated approach to studying software performance, seeking to develop a deep understanding of GPU micro-architectures, program behaviour, and specific performance optimisations for reducing parallelism, in order to develop an integrated understanding of their interplay. The main objective is to automate the performance tuning of a previously unseen set of general-purpose programs for a set of different NVidia GPU architectures at compile-time. We will study individual code optimisations in the context of a source-to-source transformation, leaving all remaining implementation details up to the programmer.

The first optimisation we will study is the well-known thread-coarsening optimisation, originally proposed by Volkov and Demmel [95], and implemented as semi-automatic source-to-source code transformations by Unkule et al. [92] and Magni et al. [49]. We build on an existing framework that automatically re-structures a given program in a coarsened manner. This enables us to explore in depth questions as to *when* to apply the optimisation and *how* to apply it in any given context. The underlying task is thus to gain a clear understanding of why the optimisation works the way it does for a particular program running on a specific architecture.

As target platforms we choose a selection of NVidia GPUs. While our primary goal is to automate the optimisation for any one particular platform, we will show that the chosen GPUs have sufficient architectural similarity to allow us to use the same general approach, yet have sufficient architectural differences that optimisation parameters need to be chosen differently.

Our contributions are as follows:

### *Thread coarsening*

We provide an in-depth study of thread coarsening that extends the state-of-the-art understanding of the transformation, and its effects on code structure and run-time behaviour on NVidia GPUs. We contrast two different modes of coarsening, which increase granularity either within a block of threads (*thread-level coarsening*, Section 3.4) or across multiple blocks of threads (*block-level coarsening*, Section 3.5). We reflect on the importance of preserving the semantics of thread blocks for the correctness of the transformation (Section 3.6). We extend an existing LLVM-based thread coarsening pipeline presented by Magni et al. [49] with the ability to support block-level coarsening.

### *Coarsening factor selection*

We present a method for automated static selection of optimal coarsening factors for a given GPU kernel based on occupancy (Section 4.1). Our approach compiles differently coarsened versions of a kernel in order to gain insight about resource usage by observing the vendor’s compiler output. The resulting information is used to model the predicted occupancy, in order to allow us to choose coarsening factors that will yield an increase in performance. We implement this into an existing coarsening pipeline (Section 4.3). We demonstrate that our approach works for two different modes of coarsening across three different NVidia GPU architectures on code optimised to different degrees, selecting near-optimal coarsening factors that achieve a maximum speedup of 5.08x (Section 4.4) for a series of reduction kernels from the NVidia OpenCL SDK [37]. We show that an average of 1.97x can be achieved across all architectures for these benchmarks, without taking a performance penalty measured to be 0.26x in the worst case. We demonstrate that the optimal coarsening factor selection also depends on the problem size, and show how our approach is able to dynamically select one of the differently coarsened pre-compiled programs at run-time that is suitable to the specified launch configuration (Section 4.5).

### *Cache line re-use analysis*

We explore what types of kernels are likely to suffer a performance backlash as a result of thread coarsening (Section 5.1). To this end, we develop a static heuristic cache line re-use analysis that seeks to automatically identify such kernels (Section 5.4). As the analysis is in nature similar to a re-use distance analysis, we show how without the *distance* measure the analysis can be implemented in a light-weight fashion based on partial symbolic execution, requiring only a small number of simulated threads, and without requiring a cache model. We extend the LLVM coarsening pipeline by incorporating this analysis as a filtering mechanism for kernels we wish to exclude from coarsening (Section 5.7). We show that this heuristic approach offers an improvement of several orders of magnitude in comparison to existing re-use distance analysis, enabling it to be executed at run-time (Section 5.6). We demonstrate that the analysis enables us to apply coarsening to a diverse set of benchmarks from the Rodinia benchmark suite [14], and measure an average speedup of 1.30x out of a maximum of 1.35x for kernels deemed safe to coarsen, and an average speedup of 1.12x out of a maximum of 1.25x across all kernels (Section 5.8). These experiments also enable a detailed comparison of the two coarsening modes to be made, as well as the coarsening factor selection to be re-evaluated on a wider choice of benchmarks.

### *Coarsening and artificial occupancy reduction*

We investigate the relationship between thread coarsening and artificial occupancy reduction (Chapter 6), both of which affect the degree of parallelism of a program execution, as well as occupancy and cache pressure. We conduct a preliminary case study to

explore how these features interact (Section 6.3.1). We show how artificial occupancy can be implemented and integrated with the existing coarsening pipeline (Section 6.2). We show that the cache line re-use analysis is able to identify kernels which may benefit from artificial occupancy reduction (Section 6.3.2), although in practice it would benefit from being complemented by further analysis. We investigate the benefits of coarsening and artificial occupancy reduction when applied in conjunction (Section 6.3.3), and show that a speedup of 8% can be achieved for kernels of the Rodinia benchmark suite. We provide insight into how coarsening and artificial occupancy reduction are able to cancel out each other's negative effects yet without cancelling out each other's positive effects (Section 6.3.1.1), and reflect on practical gain of their combined use.

## **1.1 Thesis structure and originality**

Chapter 2 gives an overview of architectural concepts of NVidia GPUs that are of relevance to our research, and surveys publications related to our work. Chapter 3 presents two previously known modes of thread coarsening, for which we seek throughout to offer new insight and present, for the first time, a detailed comparison between the two. Chapters 4 and 5 are based on the paper 'Predictable Thread Coarsening' [84]. Chapter 6 extends this prior research and is my own work. Much of this has been proof-read several times by my supervisor Dr. Tony Field, whose detailed suggestions have been incorporated throughout. Furthermore, I and Dr. Field co-authored the paper 'Predictable Thread Coarsening' [84], on which parts of Chapters 2-5 are based. Except where otherwise referenced, all remaining contributions are my own.

## **1.2 Copyright declaration**

The copyright of this thesis rests with the author. Unless otherwise indicated, its contents are licensed under a Creative Commons Attribution-Non Commercial 4.0 International Licence (CC BY-NC). Under this licence, you may copy and redistribute the material in any medium or format. You may also create and distribute modified versions of the work. This is on the condition that: you credit the author and do not use it, or any derivative works, for a commercial purpose. When reusing or sharing this work, ensure you make the licence terms clear to others by naming the licence and linking to the licence text. Where a work has been adapted, you should indicate that the work has been changed and describe those changes. Please seek permission from the copyright holder for uses of this work that are not included in this licence or permitted under UK Copyright Law.

# GPU technologies 2

This chapter outlines technologies relevant to this thesis. We describe NVidia GPU micro-processing architectures (Section 2.5), as well as the GPU programming model of CUDA (Section 2.2) and OpenCL (Section 2.3). We outline compilation through LLVM (Section 2.4). The chapter concludes by surveying publications relevant to this thesis (Section 2.6).

## 2.1 GPU Programming History

*... Again I wish to reiterate the point that all the arguments for parallel operation are only valid provided one applies them to the steps which the built in or wired in programming of the machine operates. Any steps which are programmed by the operator, who sets up the machine, should be set up only in a serial fashion. It has been shown over and over again that any departure from this procedure results in a system which is much too complicated to use.* (J. P. Eckert Jr. — 26 August 1946)

On the 14th of February 1946 the ENIAC, the electronic numerical integrator and computer, went operational [39]. Developed by John Presper Eckert and John Mauchly at the Moore School of Electrical Engineering, and weighing in at a respectable 27 tons, two characteristics make it stand out in particular. The ENIAC was the first stored-program computer, using counters “not only for arithmetic purposes, but also as part of the programming circuits to determine when and how a given unit shall perform” [31]. The introduction of such *program counters* was accompanied by a logical separation of arithmetic and programming circuits. The second notable characteristic was that it was a parallel machine. This led to the above quote by its architect J. Presper Eckert, spoken at the first ever conference on computers and computer architecture that became known as the Moore School lectures [11]. Parallel computing is therefore arguably at least as old as stored-program computing. To the modern reader, a statement that parallel computing has been tried ‘over and over again’ may appear as if before its time, in 1946. However, what these early experiences appeared to foreshadow, retrospectively turned out to be a fundamental problem of parallel computing that was to shape much of the history of computer architectures: Namely, that it is simpler to develop parallel hardware than to program it, and that programming in a parallel context can quickly become ‘complicated’.

It is in part due to this characteristic that single-core CPUs dominated the market for end-user CPUs for decades. Moore’s observation that transistor counts on integrated circuits would

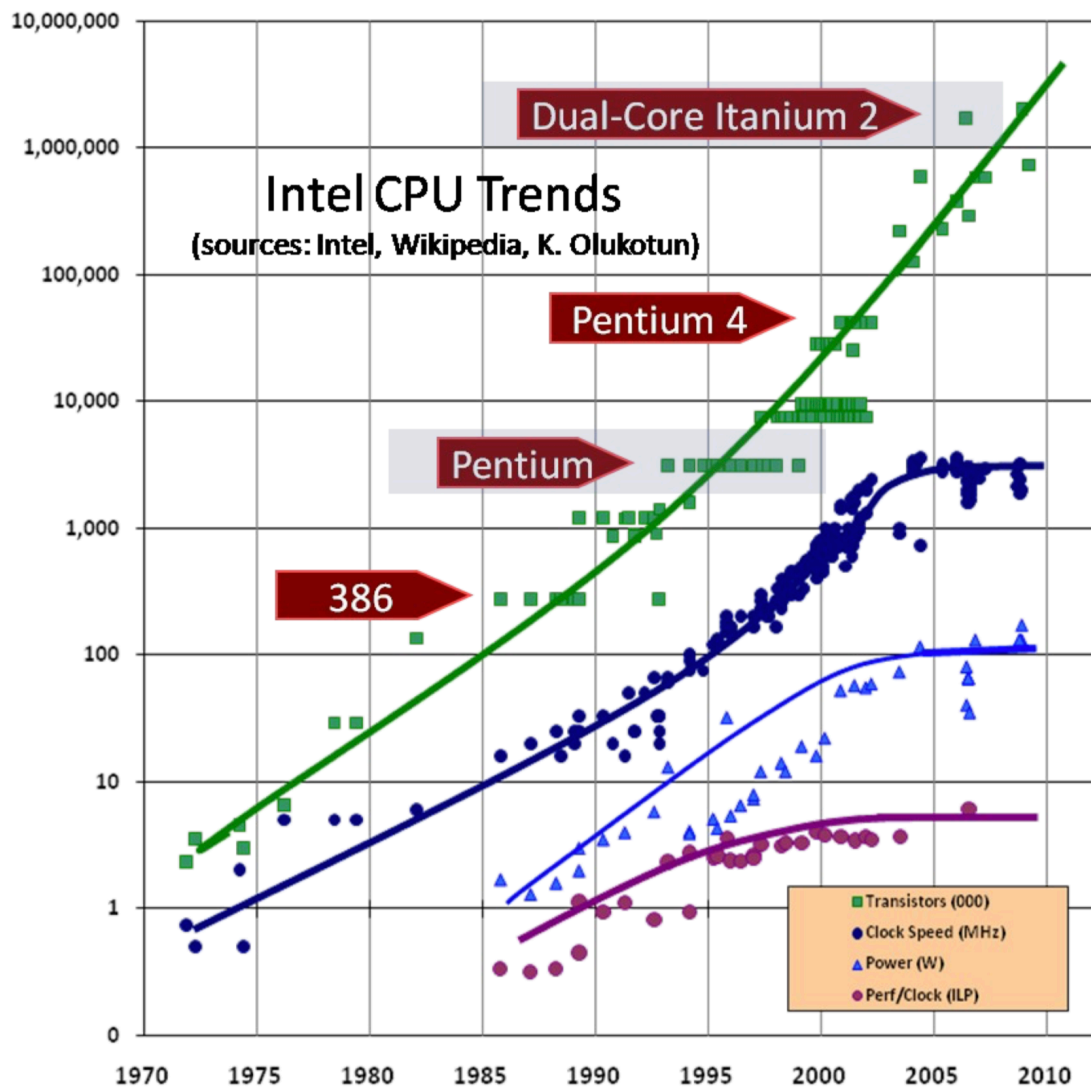


Figure 2.1: Intel microprocessor trends [88]

double approximately every two years held true. As transistors became increasingly smaller, transistor counts and clock cycle frequencies saw a continual increase, giving rise to the hope that parallel computing may be possible to avoid entirely at least in desktop contexts. This trend continued until clock cycle frequencies began to stall around the year 2003, as Figure 2.1 highlights. Processors, at that point, began to reach high temperatures that became increasingly more difficult to cool as required [88]. As Asanovic et al. [3] point out, the International Technology Roadmap for Semiconductors [27] envisioned in 2005 that CPU clock cycle frequencies were to cross the 20 GHz mark around the year 2012. The revised and more conservative roadmap of 2007 placed clock cycle frequencies for the same year 2012 at around 7 GHz. In reality, clock cycle frequencies stayed far behind these predictions and well below the 5 GHz mark to this date, not having seen a significant change since the year 2003.

The following years saw a paradigm shift towards parallel computing. Chip makers turned

to multicore processors [29]. The continued predominance of *fat-core* microprocessor architectures suddenly appeared uncertain [53]. In support of *fat-core* architectures, Seymour Cray famously stated:

*If you were plowing a field, which would you rather use: Two strong oxen or 1024 chickens?*  
(Seymour Cray)

Cray, architect of many world-leading supercomputers in the past and considered by many the “father of supercomputing” [7], believed that a small number of powerful *fat-core* processors would have a performance advantage over a typically larger number of *thin-core* architectures. Fat-core architectures refers to microprocessors that typically employ sophisticated processor technologies in order to maximise single-core performance. Thin-core architectures, on the other hand, typically emphasise light-weight and simple processor designs running at much lower clock frequencies. In this design approach, system performance is based on a large number of simple processors, rather than the processing strength of a few, much like what Cray sought to express in the above analogy.

The paradigm shift towards parallel computing led to an explosion in different varieties of processor architectures on the mainstream market, and also opened up a re-discovery of thin-core architectures for general purpose computing in a variety of contexts. ARM’s *big.LITTLE* architecture [34] combines traditional ‘big’ cores that provide processing power with individual ‘LITTLE’ cores which have low energy consumption. On which cores processing should take place can be determined on the fly by activating or deactivating cores depending on the needs of the application, giving the system the versatility to be efficient both in terms of processing power and in low energy contexts. This design and its successors have been the basis of many world-leading smartphones, including devices from several generations of the Samsung Galaxy series, Sony Xperia, Huawei, HTC, OnePlus, and more.

Graphics processing units (GPUs) offer large-scale parallelism based on thin-core architectures. For instance, one of the GPUs used to evaluate our experiments throughout this thesis comprises as many as 2880 cores, running at a mere 980 MHz<sup>1</sup>. In contrast, comparable Intel fat-core CPUs that are likewise high-end desktop models and released around the same time are clocked at 3.6 GHz, comprising 6-8 cores (12-16 with hyper threading)<sup>2</sup>.

The fact that NVidia GPUs follow a thin-core architectural approach has recently come to public attention in the light of the Spectre attack [44], which exploits a conceptual flaw in speculative execution. Speculative execution utilises a branch predictor to speculate about which instructions may need to be executed in the near future, and begins to execute them without committing the instructions. This technology is typical for deep, complex instruction pipelines of fat-core architectures as found in virtually all server, desktop, and mobile processors. Thin core architectures as employed by GPUs typically do not feature such technologies. This is also the case for NVidia GPUs, making them immune to Spectre attacks.

---

<sup>1</sup>NVidia GK110-430-B1 chipset, Kepler architecture, released in February 2014.

<sup>2</sup>For instance, the Core i7 Extreme 4960X chipset based on the Ivy Bridge, released September 2013.

Yet J. P. Eckert's saying remained true, that parallel architectures are easier to devise than they are to program. To this end, the rise of large-scale parallelism inspired new programming models intended to be better equipped at giving programmers ease of access to the available parallelism.

Among the earliest and most influential platforms for general purpose GPU (GPGPU) programming are Brook [10] and Accelerator [89]. Brook, in particular, laid the ground work for many of the concepts that define the CUDA and OpenCL programming model, and is worth investigating more closely. Brook implements a *stream programming* model, comprising streams, kernels, and reduction operations. Buck et al. define a *stream* as a collection of data records requiring similar computation while *kernels* are functions applied to each element of a stream. Stream programming was previously used in supercomputing contexts, and the authors note that this type of data parallelism appears particularly suitable to applications with high arithmetic intensity (c.f. [20]).

A Brook kernel is executed as a shader program on a GPU, which is so called because these data-parallel programs were traditionally used for shading, and are executed in the rendering pipeline of a GPU. A kernel is assigned input and output registers on which to operate. An example presented by the authors is the following saxpy (scalar alpha X plus Y) Brook kernel, comprising of host and device code:

```

1  kernel void saxpy (float a, float4 x<>, float4 y<>, out float4 result<>) {
2      result = a*x + y;
3  }
4
5  void main (void) {
6      float a;
7      float4 X[100], Y[100], Result[100];
8      float4 x<100>, y<100>, result<100>;
9      ... initialize a, X, Y ...
10     streamRead(x, X);                                // copy data from mem to stream
11     streamRead(y, Y);
12     saxpy(a, x, y, result);                            // execute kernel on all elements
13     streamWrite(result, Result);                        // copy data from stream to mem
14 }

```

The saxpy kernel operates on a constant  $a$ , two input streams  $x$  and  $y$ , as well as an output stream prefixed with  $out$ . Prior to executing this kernel, the input streams must be filled with data and placed in the GPU's memory, as is shown in the main method which is executed on the host, i.e., by the CPU. The kernel is then invoked and executed on the GPU device once for each element of the streams. Each kernel execution, processed by the GPU's vertex shaders, is assigned a set of input/output registers. The input and output streams are therefore accessed with an index that is implied rather than specified explicitly. Not shown in this example are *gather streams*, which are read-only arrays that support random access, such that data can be 'gathered' from them. When kernel execution is finished, the output stream will have been populated with the calculated results, and can subsequently be copied back from device memory to the host's main memory. The Brook system also supports reduction operations on streams along with

dedicated syntax. Further language features include *iterator streams* containing pre-initialised sequential values, and an `indexof` operator that returns the index of the current element in the stream, i.e., the predecessor of OpenCL's `get_global_id()`. Finally, Brook implements two parallel indirect read-modify-write operators called `ScatterOp` and `GatherOp`, but cites GPU hardware limitations such that these have to be executed on the CPU.

Although Brook builds on previous work showing how GPUs may be abstracted as a SIMD [70] or vector [90] processor, it proved to be greatly influential for later mainstream GPU programming models including CUDA and OpenCL. For instance, both languages follow the stream programming model in principle, retaining both the concept and the terminology of compute kernels that describe the processing of one work item. Input/output streams have been discontinued in favour of fully accessible gather/scatter streams (i.e., arrays), with the `indexof` operation being extended and commonly implemented on hardware-level. Finally, Brook's concept of host-device interaction has also been retained in principle and extended further.

## 2.2 The CUDA language and API

In June 2007 NVidia published the initial release of CUDA as a means to support general purpose computing on NVidia GPUs. CUDA is a development platform for GPU computing which includes the CUDA programming language, an extension initially to C and later to C++ supporting the implementation of compute kernels. The CUDA run-time API manages host-device communication including data transfers, synchronisation, and the invocation of kernels from the host.

The original name, Compute Unified Device Architecture, has since been discontinued in favour of the acronym CUDA. Nowadays, the term CUDA is commonly used synonymously for the CUDA programming language and API. The original name, however, suggests a more wholistic view in which CUDA was initially thought of primarily primarily an architecture, with a programming language and API built on top of it. This is reflected in much of the terminology chosen, including, for instance, the naming of stream processing cores, which on NVidia GPUs are known as *CUDA cores* (see Section 2.5.2.1), each of which holds an integer ALU as well as a single-precision floating point unit. As such, CUDA terminology consists predominantly of hardware-based vocabulary. Notably, the concepts of threads and thread blocks in the CUDA programming model are also entirely hardware-managed, and are therefore hardware concepts.

Threads in CUDA are responsible for the processing of stream elements. A CUDA kernel describes the processing of one thread. Brook's `indexof` function was superseded by a `threadIdx` field with 3-dimensional components. CUDA threads reside in the GPU's *Streaming Multiprocessors* (see Section 2.5.2) and are mapped onto CUDA cores for execution. As an extension over Brook, threads in CUDA are organised into thread blocks. The hardware will schedule thread blocks rather than individual threads for execution. This allows for inter-thread communication including synchronisation and data sharing. Data sharing can take place via the

specially available *shared memory*, a software-managed cache shared between threads of the same thread block. Inter-block communication is not part of CUDA's programming model.

The previous section shows a saxpy kernel implemented in Brook. In CUDA, the same program may look as follows:

```

1  __global__ void saxpy(float a, float *x, float *y, float *result) {
2      int tid = blockIdx.x * blockDim.x + threadIdx.x;
3      result[tid] = a * x[tid] + y[tid];
4  }
5
6  int main(void)
7  {
8      int N = 1024;
9      float a, *x, *y, *result, *d_x, *d_y, *d_result;
10
11     ... initialise a, x, and y ...
12
13     cudaMalloc(&d_x, N*sizeof(float));
14     cudaMalloc(&d_y, N*sizeof(float));
15     cudaMalloc(&d_result, N*sizeof(float));
16
17     // copy data from host to device
18     cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
19     cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
20
21     // invoke SAXPY for 8 thread blocks of 128 threads
22     saxpy<<<(N+127)/128, 128>>>(a, d_x, d_y, d_result);
23
24     // copy data back to host
25     cudaMemcpy(result, d_result, N*sizeof(float), cudaMemcpyDeviceToHost);
26
27     cudaFree(d_x);
28     cudaFree(d_y);
29     cudaFree(d_result);
30 }

```

The host code, as specified in the main function, bears strong resemblance to Brook. Memory is allocated both in the host's main memory as well as in the device's main memory (pointers to device memory are commonly prefixed with the letter *d* by convention). Once the data is initialised, it is transferred to the device before the kernel is launched, and the results are transferred in the opposite direction once the execution of the kernel has been completed. The invocation states the parameters passed to the kernel, as well as the number of kernel executions requested. The triple angle bracket notation specifies the number of blocks followed by the number of threads in each block, which taken together make up the *compute grid*. The size of the compute grid is equivalent to the stream size Brook. In this example, the kernel call is placed statically inside the host code. However, CUDA also provides API functionality through which kernels may be specified and invoked dynamically.

The saxpy kernel shown above is in nature similar to its Brook implementation. However, streams (referred to simply as arrays) passed to the kernel must be indexed explicitly. To this end CUDA provides a number of built-in variables which also support the programming model's

| CUDA function | Type  | Description                                         |
|---------------|-------|-----------------------------------------------------|
| gridDim       | dim3  | Number of thread blocks contained in a compute grid |
| blockIdx      | uint3 | Index of the current thread block                   |
| blockDim      | dim3  | Number of threads per block                         |
| threadIdx     | uint3 | Index of the current thread                         |
| warpSize      | int   | Number of threads per warp                          |

Table 2.1: CUDA built-in variables.

approach to organising threads into thread blocks. The thread ID computed in line 2 above is therefore semantically equivalent to Brook’s `indexof` function. CUDA provides a number of built-in variables available to threads for acquiring information about the processing context and their own position in it. These built-in variables are outlined in Table 2.1. All but the last of these variables are three-dimensional values and can be accessed by their x, y, or z components.

CUDA kernels are compiled to PTX (parallel thread execution) ISA [69], an abstract instruction set architecture (ISA). PTX is a virtual ISA that can be compiled to native SASS (source and assembly) code of different GPUs. The PTX representation of the above kernel is as follows:

```

1  visible .entry _Z5saxpyfPfS_(
2      .param .f32 _Z5saxpyfPfS_.__param_0 ,
3      .param .u64 _Z5saxpyfPfS_.__param_1 ,
4      .param .u64 _Z5saxpyfPfS_.__param_2 ,
5      .param .u64 _Z5saxpyfPfS_.__param_3
6  )
7  {
8      .reg .f32      %f<5>;
9      .reg .s32      %r<5>;
10     .reg .s64      %rd<11>;
11
12
13     ld.param.f32    %f1, [_Z5saxpyfPfS_.__param_0];
14     ld.param.u64    %rd1, [_Z5saxpyfPfS_.__param_1];
15     ld.param.u64    %rd2, [_Z5saxpyfPfS_.__param_2];
16     ld.param.u64    %rd3, [_Z5saxpyfPfS_.__param_3];
17     cvta.to.global.u64 %rd4, %rd3;
18     cvta.to.global.u64 %rd5, %rd2;
19     cvta.to.global.u64 %rd6, %rd1;
20     mov.u32         %r1, %ctaid.x;
21     mov.u32         %r2, %ntid.x;
22     mov.u32         %r3, %tid.x;
23     mad.lo.s32      %r4, %r2, %r1, %r3;
24     mul.wide.s32    %rd7, %r4, 4;
25     add.s64         %rd8, %rd6, %rd7;
26     ld.global.f32   %f2, [%rd8];
27     add.s64         %rd9, %rd5, %rd7;
28     ld.global.f32   %f3, [%rd9];
29     fma.rn.f32      %f4, %f2, %f1, %f3;
30     add.s64         %rd10, %rd4, %rd7;
31     st.global.f32   [%rd10], %f4;
32     ret;
33 }

```

The PTX code uses virtual registers to implement the kernel, in this case employing two loads and one store operation as well as arithmetic operations including multiply-add and fused multiply-add instructions (see also Section 2.5.2.1).

From the virtual PTX ISA, the kernel is then compiled into native SASS for a specific, targeted CUDA-capable GPU architecture. Disassembling a CUDA binary file reveals the SASS representation for the above saxpy kernel:

```

1  //----- .text._Z5saxpyfPfS_S_ -----
2          .section      .text._Z5saxpyfPfS_S_,"ax",@progbits
3          .sectioninfo  @"SHI_REGISTERS=10"
4          .align 4
5          .global       _Z5saxpyfPfS_S_
6          .type         _Z5saxpyfPfS_S_,@function
7          .size         _Z5saxpyfPfS_S_,(.L_17 - _Z5saxpyfPfS_S_)
8          .other        _Z5saxpyfPfS_S_,@"STO_CUDA_ENTRY_STV_DEFAULT"
9  _Z5saxpyfPfS_S_:
10 .text._Z5saxpyfPfS_S_:
11      /*0000*/      MOV R1, c[0x1][0x100];
12      /*0008*/      S2R R0, SR_CTAID.X;
13      /*0010*/      S2R R2, SR_TID.X;
14      /*0018*/      MOV32I R5, 0x4;
15      /*0020*/      IMAD R3, R0, c[0x0][0x8], R2;
16      /*0028*/      IMAD R6.CC, R3, R5, c[0x0][0x28];
17      /*0030*/      IMAD.HI.X R7, R3, R5, c[0x0][0x2c];
18      /*0038*/      IMAD R8.CC, R3, R5, c[0x0][0x30];
19      /*0040*/      LD.E R2, [R6];
20      /*0048*/      IMAD.HI.X R9, R3, R5, c[0x0][0x34];
21      /*0050*/      IMAD R4.CC, R3, R5, c[0x0][0x38];
22      /*0058*/      LD.E R0, [R8];
23      /*0060*/      IMAD.HI.X R5, R3, R5, c[0x0][0x3c];
24      /*0068*/      FFMA R0, R2, c[0x0][0x20], R0;
25      /*0070*/      ST.E [R4], R0;
26      /*0078*/      EXIT;
27  .L_17:

```

The S2R instruction (special register to register) is used for making built-in variables available for computational use. For instance, lines 12 and 13 show the `blockIdx.x` and `threadIdx.x` variables being moved, respectively. In PTX and SASS, the block index is referred to as the CTA ID, where CTA stands for *collaborative thread array*, i.e., thread block.

## 2.3 The OpenCL language and API

OpenCL, the Open Compute Language, is an API for executing compute tasks on hardware accelerators in the context of heterogeneous computing platforms. It was initially developed by Apple in collaboration with Intel, AMD, IBM, Qualcomm, and (notably) NVidia, and was proposed to the Khronos Group, whose members are a wider group of technology companies. The apparent purpose behind this was to involve a large number of vendors of different processing devices in the specification of OpenCL early on. The Khronos group holds working groups for several APIs prefixed with the word ‘open’, including the well-known Open Graph-

| OpenCL function | Description                 |                                                                                                                                                                                                                                                                                                                         |
|-----------------|-----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| get_global_id   | Global work item ID value   | <pre> for groupIdZ = 0 .. get_num_groups(2)   for groupIdY = 0 .. get_num_groups(1)     for groupIdX = 0 .. get_num_groups(0)       for localIdZ = 0 .. get_local_size(2)         for localIdY = 0 .. get_local_size(1)           for localIdX = 0 .. get_local_size(0)             __kernel void foo() { ... }; </pre> |
| get_global_size | Number of global work items |                                                                                                                                                                                                                                                                                                                         |
| get_local_id    | Local work item ID          |                                                                                                                                                                                                                                                                                                                         |
| get_local_size  | Number of local work items  |                                                                                                                                                                                                                                                                                                                         |
| get_group_id    | Work group ID               |                                                                                                                                                                                                                                                                                                                         |
| get_num_groups  | Number of work groups       |                                                                                                                                                                                                                                                                                                                         |
| get_work_dim    | Number of dimensions in use |                                                                                                                                                                                                                                                                                                                         |
|                 | (a)                         | (b)                                                                                                                                                                                                                                                                                                                     |

Figure 2.2: (a) OpenCL work-item built-in functions. (b) Implied loop nest within which kernels are executed.

ics Library, OpenGL [102], a framework to support open access to GPUs for the purposes of executing graphical computational tasks. In 2008, the Khronos Group then officially released the first version of the OpenCL specification [36].

The OpenCL API was designed to support a variety of different microprocessing architectures. It is therefore kept purposefully abstract, while leaving both implementation and tool support up to the individual vendors. In case of NVidia GPUs, for instance, the compiler provided by the vendor translates OpenCL kernels into PTX code (see above), such that subsequently they can be handled similarly to kernels translated from CUDA.

OpenCL adheres to using software-based terminology where possible, which is unlike CUDA's hardware-based terminology. This appears to serve as an abstraction from the software API to specific underlying platforms. For instance, OpenCL speaks of *work-items*, a software concept, where CUDA speaks of *threads*, which in CUDA is a hardware concept. Table 2.2 may serve as a guide to translate between OpenCL's and CUDA's key terminology (see Section 2.5.4 for an overview of the memory hierarchy on NVidia GPUs).

| OpenCL terminology | CUDA terminology         |
|--------------------|--------------------------|
| work-item          | thread                   |
| work-group         | thread block             |
| compute unit       | streaming multiprocessor |
| global memory      | global memory            |
| local memory       | shared memory            |
| private memory     | local memory             |
| constant memory    | constant memory          |

Table 2.2: Differences in terminology between OpenCL and CUDA.

The OpenCL API specifies a number of work-item related built-in functions, which are outlined in Figure 2.2(a). Functions in this table can be indexed with a number from zero to the result of 'get\_work\_dim() - 1' (naturally, this excludes the last function listed in the table). The parameter within this range specifies the dimension, starting from 0 as the x-dimension.

The programming model employed by OpenCL is fundamentally similar to that of CUDA, although the languages have since evolved with different extensions. Kernels specify the pro-

cessing of one work-item, and are executed for a number of work-items which must be specified before the kernel is scheduled for execution. OpenCL also supports inter-work-item communication, including synchronisation and data sharing via a programmable cache local to a work-group. As in CUDA, communication between different work-groups is not part of the programming model. OpenCL kernels may be thought of as executing in an implied loop nest, which Figure 2.2(b) seeks to outline. However, the API specifies no order in which threads must be executed, such that the loops shown may effectively be chosen to be re-ordered by the hardware.

In OpenCL, the above saxpy kernel may be implemented as follows:

```

1  __kernel void saxpy(float a,
2                      __global float *x,
3                      __global float *y,
4                      __global float *result) {
5      int tid = get_global_id(0);
6      result[tid] = a * x[tid] + y[tid];
7  }
```

As far as this example is concerned, the differences between the previously shown CUDA implementation to this version of the kernel in OpenCL are entirely syntactical. One difference to the earlier CUDA implementation is that OpenCL specifies a built-in function for obtaining a global thread identifier, whereas in CUDA this has to be computed manually (see above). However, when the kernel is compiled to PTX code, this difference will disappear. As PTX does not have a native way of referring to the global identifier, the NVidia compiler reverts to compute the global identifier as done in CUDA.

OpenCL is a run-time API, through which the kernel can be specified, compiled, and invoked. The following example shows how the saxpy kernel can be invoked. For brevity, this example does not include error handling, although it is highly recommended for OpenCL API calls.

```

1  int main(void)
2  {
3      const char *kernel_source = ...; // kernel source, generated from code or
4      ↪ read from file
5      int N = 1024;
6      float a, *x, *y, *result;
7      // ... initialise a, x, and y ...
8
9      cl_platform_id platform_id;
10     clGetPlatformIDs(1, &platform_id, NULL);
11
12     cl_device_id device_id;
13     clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
14
15     cl_context context = clCreateContext(0, 1, &device_id, NULL, NULL, NULL);
16
17     cl_command_queue queue = clCreateCommandQueue(context, device_id, 0, NULL);
18     cl_program program = clCreateProgramWithSource(context, 1, (const char **)
19     ↪ &kernel_source, NULL, NULL);
20     clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
```

```

20     cl_kernel kernel = clCreateKernel(program, "saxpy", NULL);
21
22     cl_mem d_x = clCreateBuffer(context, CL_MEM_READ_ONLY, N * sizeof(float),
23                               ↪ NULL, NULL);
24     cl_mem d_y = clCreateBuffer(context, CL_MEM_READ_ONLY, N * sizeof(float),
25                               ↪ NULL, NULL);
26     cl_mem d_result = clCreateBuffer(context, CL_MEM_WRITE_ONLY, N *
27                               ↪ sizeof(float), NULL, NULL);
28
29     clEnqueueWriteBuffer(queue, d_x, CL_TRUE, 0, N * sizeof(float), x, 0, NULL,
30                               ↪ NULL);
31     clEnqueueWriteBuffer(queue, d_y, CL_TRUE, 0, N * sizeof(float), y, 0, NULL,
32                               ↪ NULL);
33
34     clSetKernelArg(kernel, 0, sizeof(float), &a);
35     clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_x);
36     clSetKernelArg(kernel, 2, sizeof(cl_mem), &d_y);
37     clSetKernelArg(kernel, 3, sizeof(cl_mem), &d_result);
38
39     size_t global_size = N;
40     size_t local_size = 128;
41     clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global_size, &local_size,
42                               ↪ 0, NULL, NULL);
43     clFinish(queue);
44     clEnqueueReadBuffer(queue, d_result, CL_TRUE, 0, N * sizeof(float), result,
45                               ↪ 0, NULL, NULL );
46
47     clReleaseMemObject(d_x);
48     clReleaseMemObject(d_y);
49     clReleaseMemObject(d_result);
50     clReleaseProgram(program);
51     clReleaseKernel(kernel);
52     clReleaseCommandQueue(queue);
53     clReleaseContext(context);
54 }

```

The OpenCL run-time API offers detailed control to its user at the cost of verbosity. After initialising a platform and identifying the device on which the computation is to take place, a command queue is created, the equivalent of a processing stream in CUDA (see Section 2.5.1). The kernel source is supplied and compiled at run-time, as shown in lines 18-20. Care must be taken when using ‘compile-time’ and ‘run-time’ terminology, which due to the dynamic nature of the API may become ambiguous.

## 2.4 LLVM, SPIR, and AXTOR

LLVM, the Low-Level Virtual Machine [45], is a well-established open-source compiler, supporting a variety of source languages and target architectures. A language frontend, such as the C language family frontend Clang, parses the input source and translates it into the compiler’s *intermediate representation* (hereafter referred to as LLVM IR). The IR forms the basis of a variety of analysis and code transformations implemented in LLVM’s optimiser component. After any number of optimisation passes, the resulting LLVM IR is then compiled to one of the

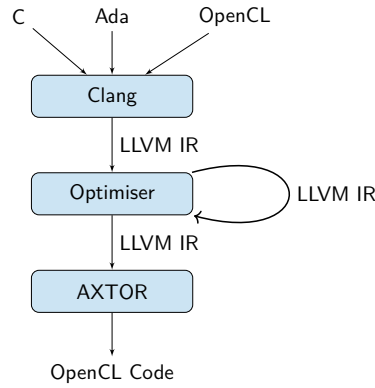


Figure 2.3: LLVM compilation pipeline

supported targeted ISAs. Figure 2.3 presents an outline of this compilation pipeline.

Characteristic of the compiler’s IR is to represent code in static single assignment (SSA) form [19] in order to simplify code analysis. In SSA form, each variable is assigned to exactly once. Code is organised into basic blocks, which are blocks of code connected by jump instructions, such that each block can only be entered at its head. To handle multiple possible assignments to variables such as may be caused by conditional jump instructions, SSA includes a special  $\varphi$  instruction, typically placed at the beginning of basic blocks. It yields a different value depending on the basic block last visited before the most recent jump took place.

In order to translate OpenCL kernels into LLVM IR, the OpenCL Standard Portable Intermediate Representation (SPIR) provided by the Khronos group acts as a mapping from the OpenCL C programming language to LLVM IR [43]. The saxpy kernel implemented in OpenCL (see above) translates to the following LLVM IR; this example is based on LLVM 3.5, which is used throughout this thesis:

```

1 ; Function Attrs: nounwind
2 define void @saxpy(float %a, float addrspace(1)* %x, float addrspace(1)* %y,
   ↪ float addrspace(1)* %result) #0 {
3   entry:
4     %call = call i32 @get_global_id(i32 0) #3
5     %arrayidx = getelementptr inbounds float addrspace(1)* %x, i32 %call
6     %tmp = load float addrspace(1)* %arrayidx, align 4
7     %arrayidx1 = getelementptr inbounds float addrspace(1)* %y, i32 %call
8     %tmp1 = load float addrspace(1)* %arrayidx1, align 4
9     %tmp2 = call float @llvm.fmuladd.f32(float %a, float %tmp, float %tmp1)
10    %arrayidx2 = getelementptr inbounds float addrspace(1)* %result, i32 %call
11    store float %tmp2, float addrspace(1)* %arrayidx2, align 4
12    ret void
13  }
14
15 !openc1.kernels = !{!0, !1}
16 !openc1.global_address_space = !{!3}
17 !openc1.local_address_space = !{!4}
18 !openc1.constant_address_space = !{!5}
19 !llvm.ident = !{!6}

```

```

20
21 !0 = metadata !{void (float, float addrspace(1)*, float addrspace(1)*, float
    ↪ addrspace(1)*)* @saxpy}
22 !1 = metadata !{void (float, float addrspace(1)*, float addrspace(1)*, float
    ↪ addrspace(1)*)* @saxpy, metadata !2}
23 !2 = metadata !{i32 16776966, i32 16776960, i32 16776960, i32 16776960}
24 !3 = metadata !{i32 16776960}
25 !4 = metadata !{i32 16776961}
26 !5 = metadata !{i32 16776962}
27 !6 = metadata !{metadata !"clang_version_3.5.0"}

```

AXTOR, an Abstract syntax tree eXtractor [56], is a publicly available tool that decompiles LLVM IR to OpenCL. The resulting code is more verbose than the original OpenCL code, but is intended to be functionally equivalent. The verbosity stems partly from the use of SSA form, and partly from the fact that loops – which are resolved to jump instructions in LLVM IR – are not reconstructed in their original `for` or `while` forms, but are instead represented as `while(true)` loops with a `break` condition. For the purposes of this thesis, we have applied multiple patches to the tool, for instance, to enable support for statically allocated shared memory regions.<sup>3</sup>

## 2.5 NVidia GPU Architectures

This section surveys key architectural concepts of NVidia GPUs relevant to this thesis. Many of the concepts described in this chapter apply to several generations of NVidia GPUs. Our primary focus are the Kepler, Maxwell, and Pascal architectures, for which we seek to develop architecture-specific approaches to code optimisation in subsequent chapters of this thesis.

### 2.5.1 Thread Block Scheduling

The block scheduler assigns blocks of threads to SMs to be executed. The left-hand side of Figure 2.4 highlights the original design and purpose of the thread block scheduler, while the right-hand side shows a number of extensions introduced with the Kepler architecture.

The work distributor unit takes compute grids that are placed on a stream queue as a result of invoking an API call to launch a given kernel. Compute grids consist of a specified number of thread blocks split over one to three dimensions. Scheduling these thread blocks is the responsibility of the work distributor unit. The incoming grids are processed in order, which means that thread blocks of a given grid will not be scheduled until the previous grid has finished processing entirely<sup>4</sup>. The NVidia CUDA Programming Guide [60] states:

Thread blocks are required to execute independently: It must be possible to execute them in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled in any order across any number of cores [...]<sup>5</sup>

<sup>3</sup>This previously caused AXTOR to loop infinitely, caused by a confusion of LLVM’s use and user functionality.

<sup>4</sup>See [60], Section 3.2.5.5. Streams

<sup>5</sup>See [60], Section 2.2. Thread Hierarchy.

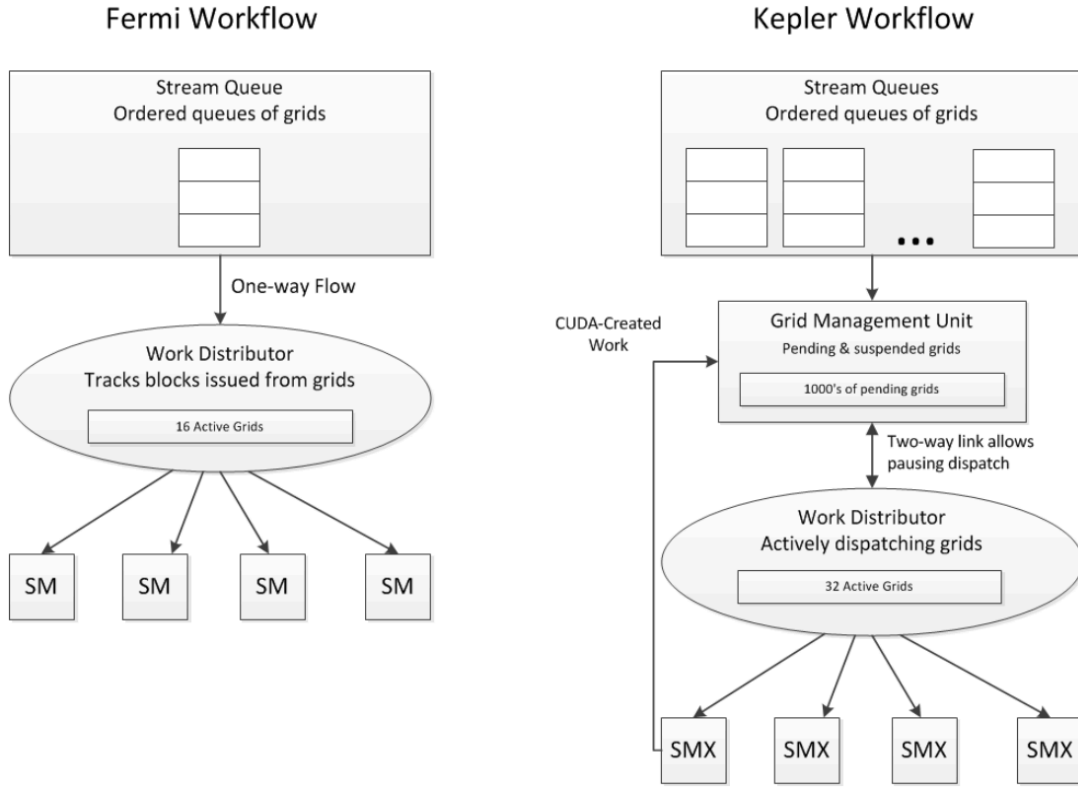


Figure 2.4: Thread block schedulers of the Fermi and Kepler architectures [63].

This means that blocks may be scheduled in any order, the implication being that no assumptions should be made inside a kernel that would require a specific order in the scheduling of blocks. In practice, however, simple experiments reveal that the logic to dispatch thread blocks appears rather simple, and that thread blocks are dispatched starting from the block with an index of zero in incremental order, in the case of one-dimensional kernel launches.

The thread block scheduler does not perform context switches of any form, unlike, for instance, a process scheduler in an operating system. By this we mean that once a thread block has been scheduled for execution, the execution is not interrupted to swap the block out of a processor in order to allocate processing time to a different thread block. Instead, thread blocks are executed to completion once they are scheduled. The reason for this is again that there are no dependencies between thread blocks, and that thread blocks must be able to execute independently. As such, there is no gain in performing context switches, which are typically time-consuming.

The work distributor seeks to maximise the number of thread blocks concurrently executing. This means that it must know about the resource requirements of scheduled kernels, as well as the availability of resources on each SM. This is possible as resource requirements of kernels are statically known at compile time, and are used in the process of thread block scheduling. As such, the responsibility of the thread block scheduler is to maximise the *occupancy* across all SMs. We define theoretical occupancy to be the fraction of active threads occupied on each SM.

For instance, if an SM can support the running of 2048 active threads, and 3 thread blocks of 512 threads each are scheduled to run on that SM, the theoretical occupancy will be 75%, as 1536 of the 2048 maximum active threads are actively used, i.e., occupied (see Section 2.5.5).

The extended block scheduler introduced with the Kepler architecture is shown on the right-hand side of Figure 2.4. It introduced two new features. The first was the addition of further stream queues such as to allow multiple processing streams. The Fermi architecture allowed multiple streams only in software, which were then combined into one queue on the hardware. Multiple hardware stream queues allow grids from multiple processing streams to be dispatched simultaneously. As before, grids from the same processing stream execute in order, as described above. Users who wish to use a single processing stream may use the *default stream*<sup>6</sup>, or its legacy variant, the NULL stream, which adds further implicit synchronisation if desired.

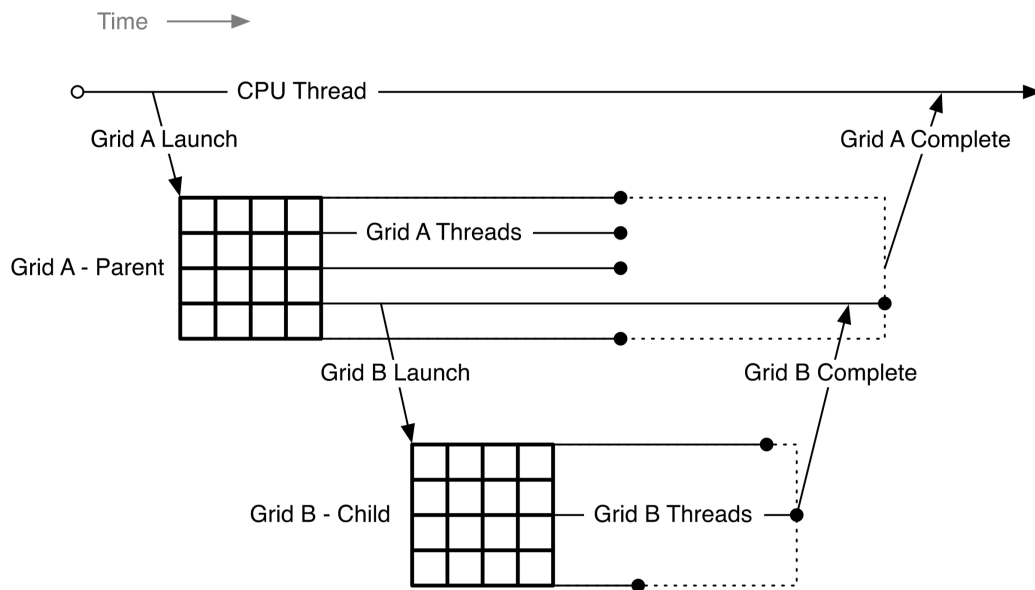


Figure 2.5: Dynamic parallelism in CUDA, see [60], Section D.2.1.1. Parent and Child Grids

The second feature added with the Kepler architecture was the introduction of *dynamic parallelism*. Dynamic parallelism allows grids to launch other grids dynamically. We distinguish between parent and child grids, which are shown in Figure 2.5. Parent grids may launch child grids either at the end of their execution (similar to the concept of tail recursion), or at any other point during their own execution. The launched grids can in turn launch further grids, up to a configurable nesting depth. Importantly, parent thread blocks must wait for their entire child grid hierarchy to finish processing before resuming their own execution. This required a series of changes to the existing thread block scheduling mechanism. First, context switches in the above sense are now required for any thread block that launches child grids and wishes for them to finish executing. As such, thread blocks must be taken out of the SMs and stashed in global memory until their child grids finish executing. These are referred to as *suspended grids* in Figure 2.4. A grid management unit was therefore added in addition to the existing thread

<sup>6</sup>See [60], Section 3.2.5.5.2. Default Stream

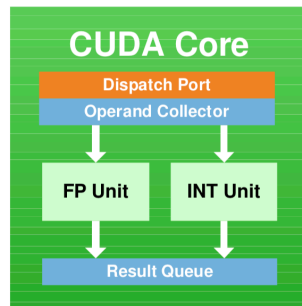


Figure 2.6: Diagram of a CUDA core [61].

block scheduler in order to handle these additional tasks. The grid management unit will hold parent blocks inactive until dependent child grids are finished executing. Child grids are prioritised for execution by the grid management unit. In order to dynamically launch child grids, SM units need to be able to communicate with the grid management unit, as indicated in the same Figure. Dynamic parallelism was introduced with the Kepler architecture in 2012 as part of CUDA 5.0, but is not currently supported by OpenCL.

## 2.5.2 Streaming Multiprocessors

An NVidia GPU holds several *Streaming Multiprocessors* (SMs), each of which forms one independent OpenCL *compute unit*. Streaming Multiprocessors were previously known as SMXs on Kepler, SMMs on Maxwell, but have been referred to as SMs since the Pascal architecture.

A streaming multiprocessor is responsible for executing thread blocks. Several thread blocks may be executed on one SM simultaneously, but a block cannot be split over several SMs. Since the introduction of the Kepler architecture, each SM can hold up to 2048 threads simultaneously. The number of thread blocks concurrently executable on one SM is limited. The limit depends on the CUDA / OpenCL Compute Capability of the device.

A GPU holds an array of streaming multiprocessors, which encapsulate a GPU's processing hardware. As such, SMs hold all hardware required for the processing of thread blocks, including single and double precision processing cores, instruction buffers, warp schedulers, a register file, on-chip memory, etc.

The following subsections provide an overview of the key architectural concepts of streaming multiprocessors, highlighting differences in the design of the Kepler, Maxwell, and Pascal architectures.

### 2.5.2.1 CUDA Cores

CUDA cores are an essential building block of streaming multiprocessors. Figure 2.6 shows a diagram of a CUDA core of the Fermi architecture. It encapsulates an integer unit as well as a single-precision floating point unit. The single precision units implement the IEEE 754-2008 floating point standard, including fused multiply-add (FMA) operations. FMA operations have



Figure 2.7: Streaming Multiprocessor (SMX) of the GK110 Kepler architecture [63]

the advantage over multiply-add (MAD) operations, that rounding only takes place once, at the end of the operation. In contrast, MAD operations round between the multiplication and the addition operation, as well as at the end, thereby increasing the margin of error due to loss of precision.

### 2.5.2.2 Kepler's SMX

Figure 2.7 shows the setup of a streaming multiprocessor of the Kepler architecture. The figure shows a total of 192 CUDA cores able to support 6 full warps of 32 threads simultaneously. Besides this, there are 64 double-precision units (DP), 32 special function units (SFUs), and 32

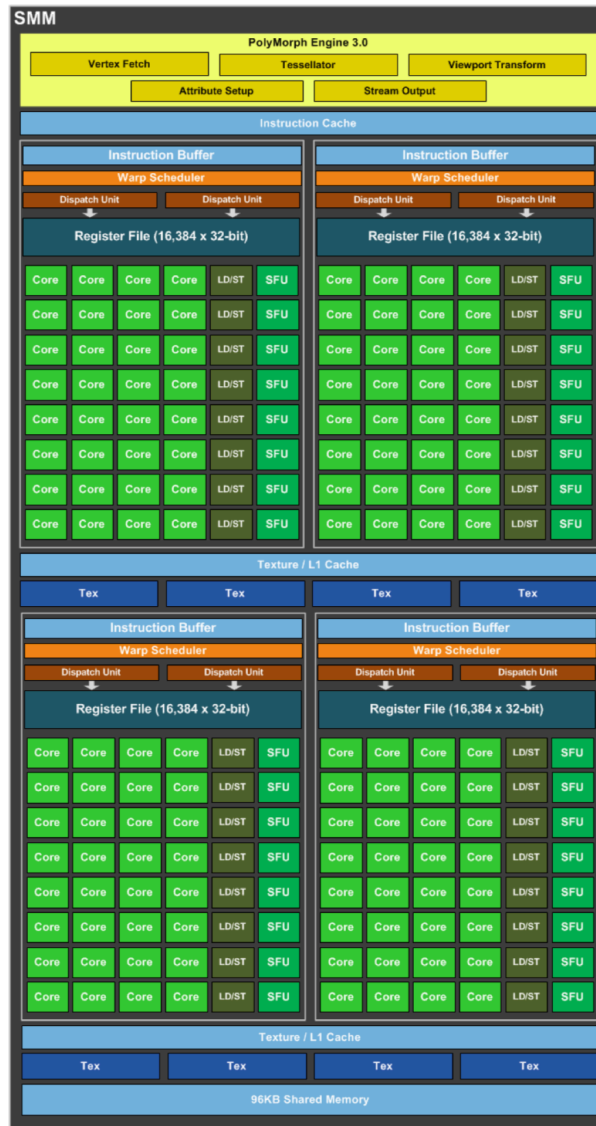


Figure 2.8: Streaming Multiprocessor (SMM) of the GM204 Maxwell architecture [64].

load-store units (LD/ST). The Kepler SMX features a total of 4 warp schedulers with 2 dispatch units each, fed from a shared instruction cache. The Kepler architecture provides a 64K 32-bit register file, which are statically allocated to thread blocks. There is also a 48K read-only data cache, acting as the *constant memory*. Finally, Kepler provides a 64K block of memory which is shared by the L1 cache and *shared memory*, a programmable cache. The split can be configured to be 16K/48K, 32K/32K, or 48K/16K. This shared implicit/explicit cache was present in the earlier Fermi architecture, and was given further configuration options in Kepler [62].

### 2.5.2.3 Maxwell's SMM

Figure 2.8 shows the setup of a streaming multiprocessor of the Maxwell architecture, which was introduced as the successor of the Kepler architecture. The number of CUDA cores has been

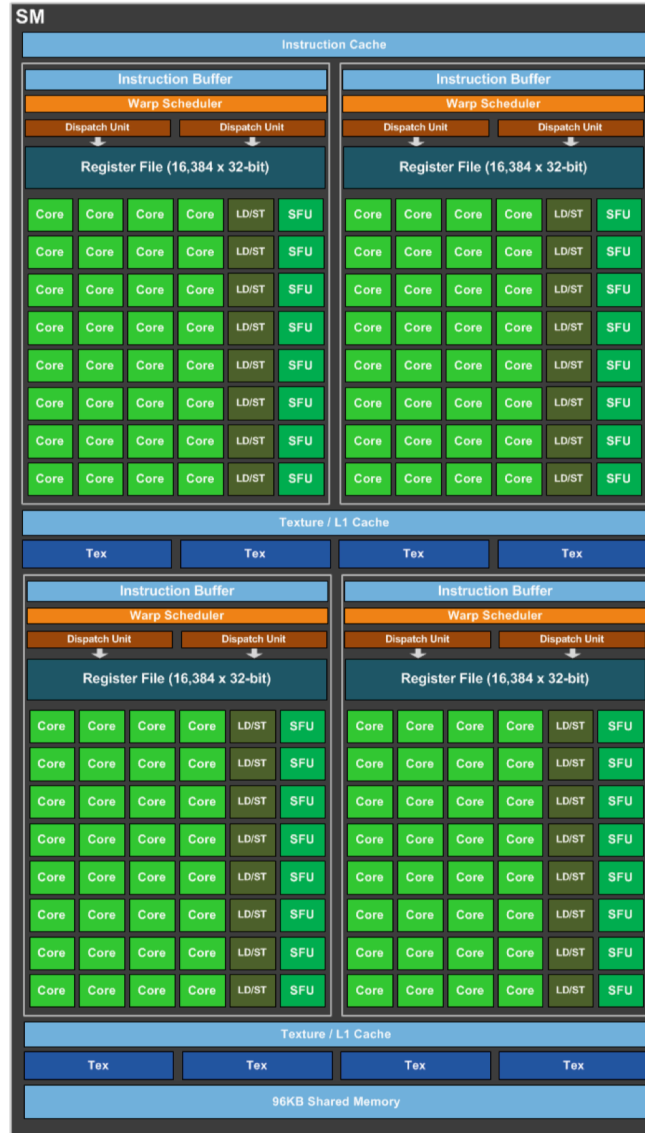


Figure 2.9: Streaming Multiprocessor (SM) of the GP104 Pascal architecture [66].

reduced to 128, supporting a total of 4 full warps of 32 threads simultaneously. The Maxwell whitepaper [64] states that by configuring the number of CUDA cores to be a power of two, the complexity of the datapath organisation could be reduced, improving area and power efficiency. This may be speculated to make more efficient use of the 4 warp schedulers, whose number was not changed. The scheduling logic has been modified, in order to reduce redundant re-compilation [64]. The number of double-precision units has been reduced to 4 per SMM (not shown in this diagram). We speculate that this significant design decision may be due to the high demand of low-precision floating point operations found in machine learning and deep learning applications.

An important difference to the Kepler architecture is that the L1 cache and the shared memory have been separated. The size of the shared memory has been doubled to 96K, although the

maximum amount of addressable shared memory for a single thread block remains at 48K as per Kepler. The L1 cache and texture cache have been combined. Finally, the SMM provides a register file of 64K 32-bit registers.

#### 2.5.2.4 Pascal's SM

Figure 2.9 shows the setup of a streaming multiprocessor of the Pascal architecture, which succeeds the Maxwell architecture. In the GP104 chipset the number of CUDA cores is the same as compared to the Maxwell architecture, comprising of 128 CUDA cores per SM, equivalent to 4 full warps of 32 threads each. Note, that this is not accurately reflected in the diagram. The number of double-precision units is not stated. The single-precision logic of CUDA cores has been adapted to support two 16-bit floating point operations being performed by a single instruction as part of a 'paired operation' [65], a feature added to provide lower-precision data types to improve the performance of deep learning applications.

#### 2.5.2.5 Warp Scheduling

The warp scheduler is responsible for scheduling threads in SIMD style, in chunks of 32 (called a *warp*). Figure 2.10 shows the warp scheduler of the Fermi architecture. The warp schedulers of later generations including Kepler, Maxwell, and Pascal, are similar in concept. Instructions are scheduled for warps rather than for individual threads, with one instruction pointer per warp. This is commonly referred to as *lock-step execution*. One of the implications of lock-step execution is that synchronisation within warps is implicit.

A second implication is that the device must handle *branch divergence*. If threads execute different branches, they are said to *diverge*. For instance, consider the following example:

```
1  int tid = get_global_id(0);
2  if (tid % 2 == 0) {
3      foo();
4  } else {
5      bar();
6  }
```

In this example, threads with even-numbered thread IDs execute the `foo()` function in line 3, while threads with odd-numbered thread IDs execute the `bar()` function in line 5 instead. This means that any given warp holds divergent threads, requiring it to execute both functions. This is implemented using *predicated execution* (see [17]). In predicated execution, the instruction pointer steps through both branches in turn, enabling and disabling threads depending on a predicate stating whether they need to execute the given branch. In our example, even-numbered threads are enabled during execution of the then-branch while odd-numbered threads are disabled, and vice-versa for the else branch. Branch divergence is not an issue if it only occurs on warp-level, such that all threads within the same warp take the same branch, while other warps may take different branches. For instance, this is the case if the condition in the above example was changed to `tid % 32 == 0`. In this case, since individual instruction

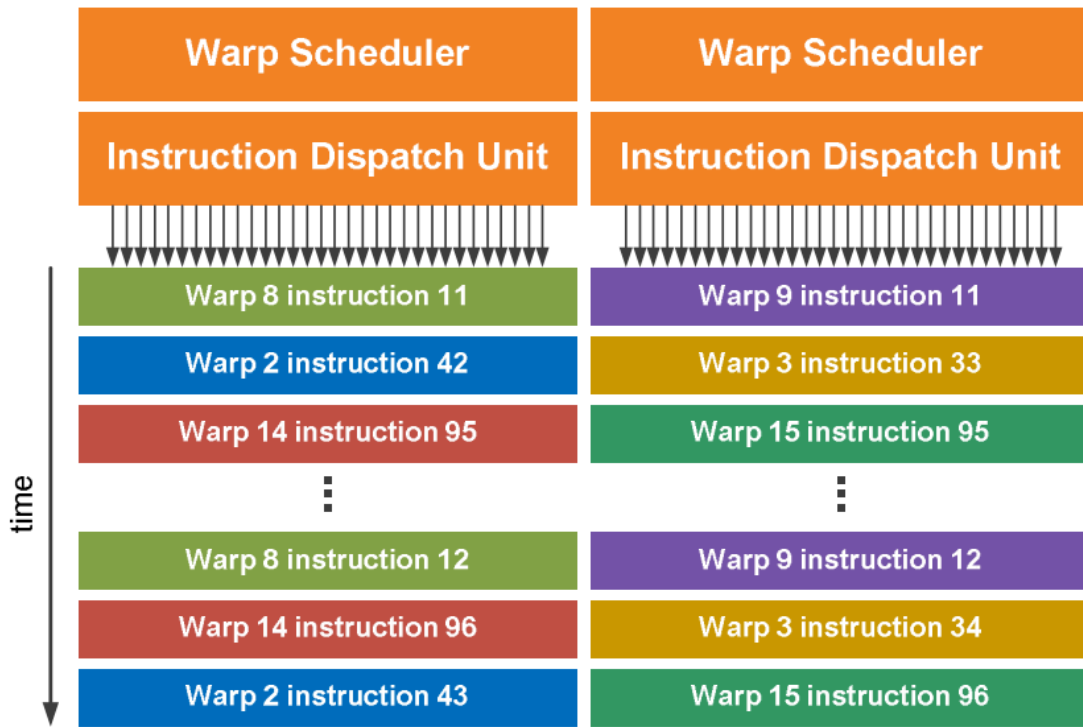


Figure 2.10: Warp scheduler of the Fermi architecture [61]

pointers exist for all warps, branch divergence is not an issue and predicated execution is not relevant.

The warp scheduler is closely linked to the instruction dispatch unit. Figure 2.11 shows an instruction dispatch in the Fermi architecture. This highlights that different types of instructions may be in flight at any one time, occupying different units in the streaming multiprocessors. Dispatching different types of instruction is likely to lower resource contention of any one type of unit, although it depends on the executed code whether and to what degree this is possible. One responsibility of the warp scheduler is to select the most suitable warp for execution at any time. However, implementation details of the warp scheduler are not disclosed, to the best of our knowledge.

As threads belonging to the same warp are executed in lock-step, the hardware seeks to combine accesses to global memory of threads belonging to the same warp to adjacent memory addresses. This process is called *memory coalescing*. For instance, if 32 adjacent threads belonging to the same warp access 32 adjacent 4-byte data items, this may be combined into a single 128-byte memory access. Memory accesses are not required to be sequential in order to be coalesced, such that threads may access data items out of order. In addition, memory accesses are also coalesced if there are gaps in the items being accessed. For instance, if only every second or third item is accessed, the memory access may still be coalesced. However, it is clear that this incurs a performance overhead. In contrast, if all bytes of a cache line are referred to, the access is said to be *fully coalesced*. Fully coalesced accesses offer a significant performance

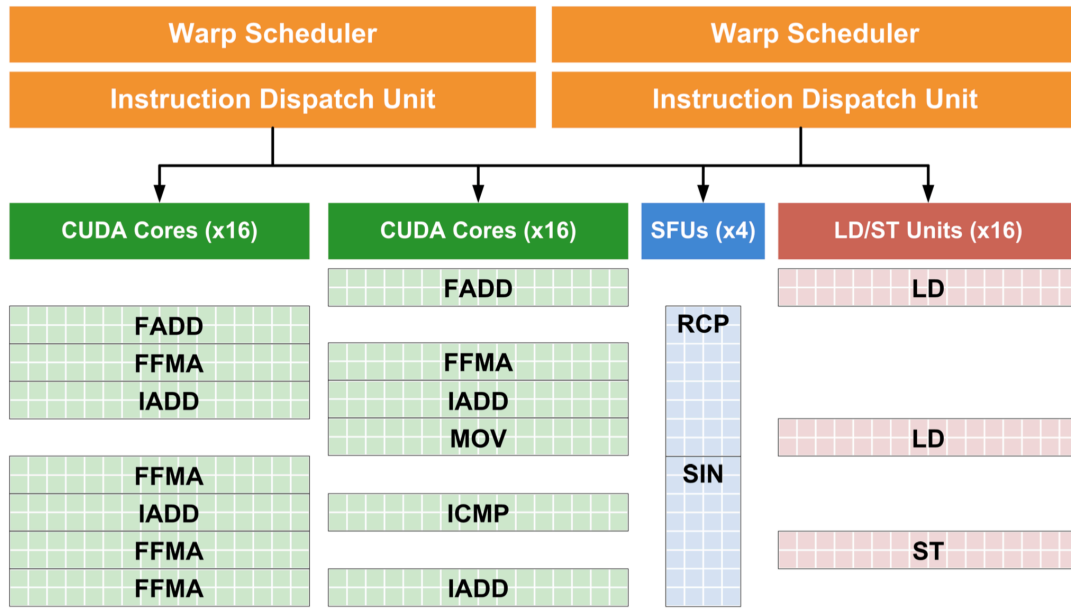


Figure 2.11: Instruction dispatch in the Fermi architecture [30].

advantage over individual memory accesses for each thread, and are a widely used concept in GPU programming.

#### 2.5.2.6 Register Allocation

Each SM has a register file of a fixed number of registers. The number of registers that can be used per thread and per block are limited, specified in the compute capability of the architecture. Registers are assigned statically to blocks in chunks equal to the warp size. As this assignment is static, register spilling is decided at compile time. By default, register spilling is disabled on NVidia GPUs, although it can be enabled if desired by setting the `maxrregcount` compiler option. This compiler option controls the maximum number of registers each thread may use, such that for any further register usage, register spilling will take place. For instance, if `maxrregcount` is set to 13 but a kernel requires to use 14 registers simultaneously, one register will be spilled to the L1 cache on Kepler and Pascal [62, 68], or global memory on Maxwell.

The ratio of the register file size to the number of active threads is typically large. For instance, a 64K 32-bit register file servicing 2048 active threads per SM yields an average of 32 registers per thread. The ratio of registers per thread is higher than the ratio of L2 cache per thread (see below). The intention behind providing a large register file is clearly to avoid register pressure. In practice, the size of the register file is sufficiently large such that many kernels will not experience register spilling. In this case, the implication is that parts of the register file will simply be unused.

The default method to handle register pressure is to schedule fewer thread blocks per SM. This is possible since the number of registers per thread block is known at compile time. The thread block scheduler can therefore map the register requirements of kernels to the availability

of registers on SMs, and choose to schedule fewer thread blocks when resources of the register file are exhausted. This avoids register spilling. The implication is that resource intensive kernels are more likely to run at reduced occupancy. Note, that throughout this thesis we use the default configuration.

### 2.5.3 GPU Chipsets

Figure 2.12 shows a full chip diagram of the Maxwell GM204 chipset. The diagram shows that 8 SMMs are grouped into a Graphics Processing Cluster (GPC). Four GPCs provide and manage a total of 32 SMMs. The number GPCs and SMs differ between architectures and also between different chipsets. For instance, our Maxwell test system holds a total of 24 SMMs (3 GPCs), while our Pascal test system holds 20 SMs managed by 2 GPCs.



Figure 2.12: Full-chip block diagram of GM204 chipset, Maxwell architecture [64].

As the figure highlights, the L2 cache is mounted on board of the GPU. It is therefore not an on-chip cache as in many CPU architectures. The L2 cache acts as a gateway to a large DRAM

base, which acts as the *global memory* of the device, constituting its main memory. Note, that the global memory is not shown in the above diagram. The global memory is typically several gigabytes large.

The L2 cache services all requests that target the global memory, acting as a point of data unification between SMs. In the GM204 chipset shown in Figure 2.12, the L2 cache has a size of 2048K. Consider that in this case the L2 cache is shared by 32 SMMs, capable of running 2048 active threads each. This accounts for a ratio of 32 byte per thread. This means that given the large number of threads that this cache services, this cache can fill up quickly in some applications.

### 2.5.4 Memory Hierarchy

Figure 2.13 shows the memory subsystem of the Kepler architecture. As discussed above, registers (here ‘threads’) as well as shared memory, L1 cache, and read-only data cache are on-chip memory, included in each SM. The L2 memory is shared by all SMs and services transactions to the global memory, a large DRAM base typically consisting of several gigabytes.

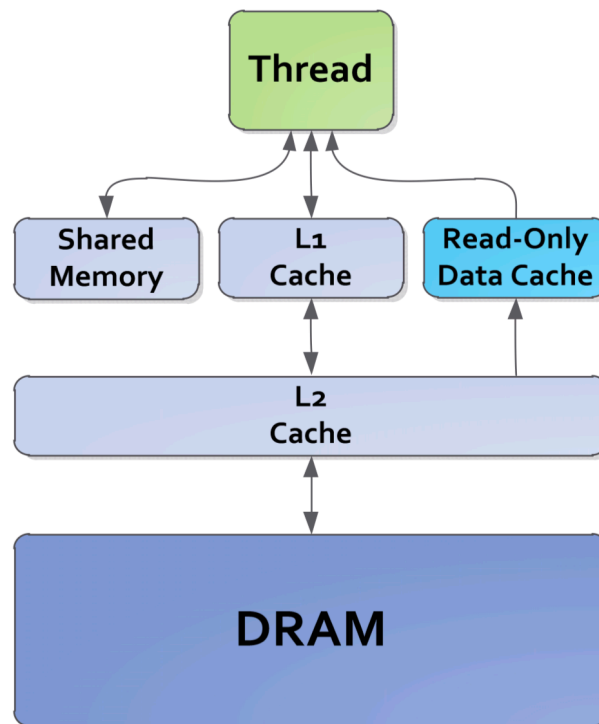


Figure 2.13: Memory hierarchy of the Kepler architecture [63]

The behaviour of caches can be modified using the `--def-load-cache` and `--def-store-cache` compiler options for default cache modifiers [100, 67]. The default behaviour for loads is to be cached in L2 only, and not in the L1 cache [68], while stores are handled by the L1 cache by default. This is the case for Kepler, Maxwell, and some Pascal architectures. Although caching behaviour may be controlled via a compiler option, it is nevertheless not a global configuration

option. Rather, the caching behaviour is specified in each individual memory transaction in PTX code.<sup>7</sup> The L2 cache cannot fully be disabled or bypassed, although an option for write-through exists.

The Pascal Tuning Manual [68] states that:

Kepler serviced loads at a granularity of 128B when L1 caching of global loads was enabled and 32B otherwise.

This behaviour has caused speculation that the L1 cache line size is 128 bytes, while the L2 cache line size is 32 bytes on the Kepler architecture.

Each SM also holds *shared memory*, essentially a programmable cache. Shared memory is organised into several equally-sized modules, referred to as banks. Shared memory requests from threads of the same warp can therefore be served in parallel, an architectural design effectively increasing shared memory bandwidth. This requires, however, that accessed data is stored in different banks. If more than one thread from the same warp accesses the same bank within a single memory transaction, the transaction will have to be serialised. This is referred to as a bank conflict. In this scenario, the device seeks to serialise the memory access into as few transactions as possible.<sup>8</sup>

### 2.5.5 Occupancy

*Occupancy* is an alternative measure to hardware utilisation of a specific kernel, concerned only with the utilisation of active threads across the device. As such, occupancy is defined by the number of active threads as a proportion of the maximum number of active threads that could be executed on one SM. For example, if 1536 active threads are scheduled onto an SM which can handle a maximum of 2048 active threads, the occupancy is 0.75 or 75%. A distinction exists between *theoretical* and *achieved* occupancy. The *theoretical occupancy* captures the upper bound of occupancy that a compiled kernel can achieve on an SM. This can be statically determined by matching a kernel's resource requirements against hardware specifications, in particular register usage, shared memory usage, and requested thread block size. The *achieved occupancy* of a kernel depends on additional run-time factors which may cause parts of the hardware to idle, such that the occupancy observed is lower than its theoretical upper bound. This would occur in the presence of unbalanced workloads, e.g. where a block or a warp within a thread block finishes earlier than another (referred to as a *tail effect*), or towards the end of a computation where some SMs will have no work to do. The achieved occupancy thus depends on the grid size, e.g. for small grid sizes there may be insufficient work to enable the machine to reach 100% occupancy.

Unless otherwise specified we use the term occupancy to refer exclusively to theoretical occupancy. However, where indicated we also take the grid size into consideration, in which case we refer to the achievable (rather than achieved) occupancy.

---

<sup>7</sup>See [69], Section 9.7.8.1. Cache Operators

<sup>8</sup>See [60], Section 5.3.2. Device Memory Accesses

## 2.6 Related Work

In this section we survey publications related to this thesis. We discuss programming techniques and algorithms able to exploit the performance offered by GPUs (Section 2.6.1). Section 2.6.2 reflects on work helpful to aid understanding of GPU performance. We consider general-purpose (Section 2.6.3) and domain-specific languages (Section 2.6.4) for generating GPU code, and survey auto-tuning (Section 2.6.6), as well as other approaches to code optimisation for GPUs (Section 2.6.7). Heterogeneous computing systems (Section 2.6.5) and work related to the GPU programming model (Section 2.6.10) are also described. Section 2.6.11 outlines different GPU benchmarks. We describe different approaches to program analysis (Section 2.6.9).

Publications related to thread coarsening are described in Section 2.6.8. The thread coarsening optimisation is discussed in the following chapter.

### 2.6.1 GPU programming and algorithms

The emergence of large-scale parallelism offered by GPUs opened up opportunities to explore implementations of known algorithms and primitives in a way that is most efficient on GPUs. Harris [37] explores optimising CUDA code, including seven different implementations of a reduction kernel. He notes that algorithmic optimisations account for nearly 12x of the achieved speedup as compared to a naive parallel implementation, whereas code optimisations account for a 2.5x performance increase.

Parallel prefix sums (i.e. scan primitives) were also studied [38, 78, 79], addressing issues around algorithmic choice, parallelisation, segmented and unsegmented scans, as well as the division of work into blocks and warps.

Cederman and Tsigas [12] present a quicksort algorithm tailored for GPUs. Satish et al. [77] present merge sort and radix sort implementations for GPUs. Radix sort has the advantage of possessing linear complexity with respect to the width of the fixed-sized numerical keys to be sorted, and can perform faster than quicksort on a GPU, whereas on a CPU quicksort performs better. Ye et al. [103] present GPU-Warpsort based on bitonic sorting networks followed by a merge sort. Merrill and Grimshaw [54] present a version of radix sort with an improved ‘scan runtime’, employing techniques such as kernel fusion and thread block serialisation to reduce the complexity overhead of a fully-recursive Brent-Kung scan.

### 2.6.2 GPU performance

Early GPGPU research held the widespread conviction that GPUs offered a significant performance advantage over traditional CPU architectures, with speedups of 10x to over 1000x. For instance, Silberstein et al. [82] present a GPU-based solver of the sum-product problem, reporting a 2700x speedup over a randomised data set, and 270x over real-world data. Govindaraju et al. [32] report an 8-40x performance advantage on GPUs for Fast Fourier Transforms. Intel’s

Lee et al. [46] analyse these and further publications of throughput computing. The analysed works are representative of a widespread trend in GPGPU-related research at that time which held the assertion that GPU implementations outperform CPU implementations in a variety of different contexts by one or often several orders of magnitude. Lee et al. show that after further tuning of both the CPU and GPU implementations presented, the performance advantage of GPUs is reduced to an average factor of 2.5x. The techniques applied included the re-ordering of memory accesses, cache blocking, as well as fine-tuned use of multithreading capabilities.

The Roofline performance model [98] was developed in the wake of multicore and large-scale parallel architectures. The model is built around the concept of operational intensity of kernels measured in FLOPS per byte, expressing the ratio of operations to memory transactions. The model observes that the theoretical peak floating point performance of any given architecture can only be achieved if the operational intensity of a kernel is above a certain threshold, which varies depending on the architecture employed. Kernels of lower operational intensity can only achieve a fraction of the maximum peak performance, which is smaller for lower operational intensities. Such kernels are classified as being *memory-bound*, as their maximum performance is limited by the peak memory bandwidth, while kernels with operational intensities above the architectural threshold are *compute-bound* in terms of performance, and are theoretically able to achieve peak floating point performance.

### 2.6.3 High-level general purpose languages

High-level languages such as Haskell [41] were explored for encapsulating general purpose CUDA or OpenCL code. The approach promised an easy solution to the problem of tuning, porting, and re-tuning GPU code, which turned out to fall short of expectations. Haskell is a functional language, and as such abstraction of the von Neumann model of computing (see [4]). This importantly includes an abstraction of actual memory accesses. As memory accesses are not specified in the original program, the idea was that they can be specified in a code-generating compiler. For example, Accelerate [13] explored the use of incorporating CUDA code templates into Haskell. Nikola [52] is an embedded programming language, supporting both compile-time and run-time code generation, data management and transfers to/from GPU, and automated loop parallelisation. The end result was evaluated on two benchmarks, and did not yield convincing performance results. This body of research highlighted that deferring implementation choices until code generation can lead to extensive optimisation spaces, where it is not clear what choices will yield good or even optimal performance.

### 2.6.4 Domain-specific languages

Domain-specific languages (DSLs) have the advantage of being confined to a small set of typically very similar problems, for which an optimisation strategy can be explored more easily due to their similarity. Since, therefore, not only the problem domain but also the type of computational problems are known, many optimisation decisions can be made up-front, leaving a

significantly narrower optimisation space to be explored by the compiler. Firedrake [74] is a DSL for the final element method of solving partial differential equations. It consists of several layers which cover mathematical modelling of linear and non-linear solves, numerical analysis over meshes, matrices, and vectors, a PyOP2 layer [73] responsible for modelling parallel loops and data structures employed, and a parallel programming layer responsible for parallel scheduling and run-time code generation. The resulting system is highly scalable while offering convincing performance statistics.

*Delite* is a Scala-based framework for generating DSL compilers, aimed to simplify the task of implementing DSLs and to explore optimisation strategies [87]. It features generic optimisations such as array-of-structs to struct-of-arrays conversion, domain-specific optimisations which can be controlled via rewrite rules, and lightweight modular staging, a type of staged metaprogramming [75]. *Delite* claims to be the basis of several successfully implemented DSLs. In our own experiments with *Delite*, it emerged that the targeted architecture is only known in the code-generators, yet not in the optimisation layer, suggesting that algorithmic choice and optimisations are committed to at an early stage, independent of the architecture.

### 2.6.5 Heterogeneous computing

Heterogeneous computing refers to using different types of compute resources in unison. GPU computing may in some sense be considered as a type of heterogeneous computing, since CPU and GPU are different types of processors. However, in systems where the CPU is merely used as the controller of one or several GPUs, which perform all actual compute work, such a classification is practically meaningless. Challenges in heterogeneous computing include finding an optimal work split between compute resources, allocating processing tasks to processing resources with respect to type of task and often with respect to costs incurred by data transfers between compute units, etc. In contrast to this body of work, we focus exclusively on single-GPU single-kernel performance.

Ansel et al. present *Petabricks* [1], which offers the option to specify multiple implementations of the same functionality to provide algorithmic choice depending on the utilised compute resource. Phothilimthana et al. have implemented a work-stealing mechanism in this framework [71] to show how it can be used to divide a given work-load across different types of compute resources. Their conclusion is that a work-split in combination with algorithmic choice does have performance advantages over a single static configuration.

### 2.6.6 Auto-tuning

Auto-tuning [22] is a technique of experimentally exploring an unknown optimisation space, in which it is not clear which optimisations should be chosen to be applied and what their parameters should be. Auto-tuning involves generating, compiling, and executing different versions of the same program so as to exhaustively explore the optimisation space. This can be an effective ‘black-box’ approach provided the optimisation space is small, but if a higher number

of parameters and combinations thereof are considered, the approach quickly gets intensely time-consuming.

Auto-tuning has been used to attempt to target GPUs from high-level languages [33], to generate optimised DSL kernels [80], to determine algorithmic choice on heterogeneous architectures [71], and more. To optimise the process of auto-tuning especially in the context of large optimisation spaces, Ryoo et al. [76] proposed optimisation space pruning, while Magni et al. [50] investigate the optimisation of individual experiments to achieve results scalable to larger data sets (interestingly, in the context of thread coarsening). Williams et al. [99] propose hierarchical and distributed auto-tuning.

### **2.6.7 Code optimisation tools and frameworks**

Tools and frameworks to explore individual code optimisations or more complex optimisation strategies have become widely used across research communities.

The early CUDA Lite [91] implemented annotation-based memory management optimisations. Ocelot [23] is a dynamic optimisation and emulation framework for NVidia GPUs.

Heuristics and cost model based optimisation techniques have been employed in contexts ranging from compile-time source-to-source transformations [9] to run-time code-generation [48], in the latter case specifically to avoid the run-time overhead incurred by auto-tuning. Wen et al. [97] use a combination of run-time data and code analysis to schedule concurrent tasks on heterogeneous CPU/GPU platforms in order to maximise system throughput. Huang and Li [40] use performance models to predict the performance impact of data based in different types of memory on a GPU. Interestingly, Chen et al. [15] present a framework for data placement on GPUs which utilises a reuse distance based model.

Fursin et al. [28] apply machine learning techniques to code optimisation, which [51] [51] do in the context of thread-coarsening.

### **2.6.8 Thread Coarsening**

Applying thread coarsening to GPU kernels was first discussed as a manual optimisation technique by Volkov and Demmel [95]. Stratton et al. [85] investigate it as part of a comparison between and investigation into different GPU code optimisations. Unkule et al. [92] provide a more focused study and the first evaluation of thread coarsening, and present an implementation of the thread coarsening code transformation based on annotated CUDA code. Magni et al. [49] present a more refined implementation of a thread coarsening algorithm with improved handling of control flow as an OpenCL source-to-source transformation, and include a cross-platform evaluation of the optimisation.

#### **2.6.8.1 Selecting Coarsening Factors**

Unkule et al. [92] implement thread-coarsening as a semi-automatic optimisation. They mention an experimental heuristic to automatically select coarsening factors, yet implementation

details or an evaluation are not provided. They note that in one of their experiments the maximum speedup is achieved at the lowest occupancy, and hence emphasize the importance of factors other than occupancy for selecting good coarsening factors. Magni et al. [49] comment that they view the authors' evaluation on a set of five kernels, to which coarsening generally applies, as limited.

Magni et al. [49] consider 17 kernels across five architectures, leading the authors to say that developing an analytical model is 'unfeasible' due to the large number of architectures and runtime libraries involved. They present an analysis based on regression trees to characterise the significance of different hardware counters on each platform for achieving good performance when applying coarsening.

The authors follow up their work recommending an auto-tuning-based approach for estimating optimal coarsening factors [50], for which they optimise the size of the required test data set. Finally, they also present a study of applying machine learning techniques to selecting good coarsening factors (see [51]).

Cummins et al. [18] present a deep learning neural network trained over raw code. They evaluate using the same test set as Magni et al. [51] and show an improvement in the selection of coarsening factors.

### 2.6.9 Program analysis

Polyhedral compilation [6] takes the approach of defining the iteration space of a loop nest in a sequential program as a polyhedron. The iteration space needs to fulfil certain criteria, including for instance that it is affine and that its exact dimensions are statically known. A polyhedral representation of a program can be used equally for the purposes of program analysis as well as for automated loop restructuring during code generation. Verdoolaege et al. present PPCG [93], the Polyhedral Parallel Code Generator, a polyhedral source-to-source compiler that parallelises the loop nest of a sequential program, transforming it into a CUDA program.

Polly [35] is an implementation of the polyhedral model in LLVM. In the context of cache line re-use analysis, Polyhedral compilation techniques [6] offer an exact rather than heuristic approach to analysing memory access patterns in loop nests (see Chapter 5). However, at the time of writing we encountered that too many of our test cases were still unsupported by Polly, as the tool was still under active development. We leave the integration of polyhedral compilation techniques for future work.

Namjoshi and Singhanian [58] claim that fully automated loop transformations do not always work and present Loopy, a semi-automated tool providing formally verified loop transformations implemented in Polly. Moll et al. [57] present input-space splitting for OpenCL based on polyhedral analysis. Their goal is to distinguish between divergent and non-divergent code regions in static analysis and generate optimised code for each scenario. This means vectorising compilers, such as for the AVX instruction set targeted by the authors, are able to avoid generating less performant code needed to support kernels whose divergence behaviour is otherwise

not known until run time.

Re-use distance analysis [101] was first studied on GPUs by Shojania and Li [81] in an analysis based on GPGPU-Sim [5]. Nugteren et al. [59] present an implementation of re-use distance analysis with a cache miss analysis model for GPUs (also see Section 5.2).

### **2.6.10 GPU verification and programmability**

Much work has focused on improving the programmability and further developing the programming model of GPUs. Betts et al. [8] present GPUVerify, a tool for detecting race-conditions and to verify divergence-freedom. The motivation behind this was to address the struggle often faced by GPU programmers in identifying these classes of bugs on co-processing hardware with their inevitable complexities. Chong et al. [16] extend this tool by adding barrier invariants, allowing them to formally reason about program correctness in data-dependent program executions. Sorensen et al. [83] implement inter-workgroup synchronisation barriers, provided the degree of software parallelism is sufficiently small to allow a full inter-group synchronisation.

### **2.6.11 Benchmarks**

The Rodinia Benchmark Suite [14] covers various problem domains, inspired by the Berkeley Dwarfs of Parallel Computing [2]. It consists of over 20 benchmarks, each comprising one or several kernels. The complexity of kernels ranges from small arithmetic kernels to complex, algorithmic, and data-dependent kernels. Benchmark sizes are not prescribed, but instead users are encouraged to choose suitable test sizes and generate corresponding test data sets by using the accompanying data set generators included in the distribution.

The PolyBench suite of polyhedral benchmarks [72] comprises a large number of predominantly smaller, arithmetic kernels. In contrast to that, the Parboil benchmark suite [86] contains typically large, complex kernels.

The Scalable Heterogeneous Computing (SHOC) benchmark suite [21] is designed to measure the performance of heterogeneous computing systems, including measuring inter-node and intra-node communications.

A popular approach among GPU researchers is assemble individual test suites comprised of kernels taken from SDKs of different vendors, including for instance the SDKs made available by AMD, NVidia, or Intel.



# Thread Coarsening 3

The optimisation we will focus on throughout this thesis is *thread coarsening*. It was originally proposed by Volkov and Demmel [95], and later studied and formalised by Unkule et al. [92] and Magni et al. [49]. In this chapter we will recapitulate and extend the theory behind this optimisation, and contrast two different modes in which it can be applied.

## 3.1 Thread coarsening to reduce parallelism

Thread coarsening is an optimisation that reduces the degree of software parallelism employed by a GPU kernel, by increasing the workload per thread. This approach differs from conventional CPU code optimisations. For example, when implementing a program for a conventional CPU architecture, a programmer may choose to implement a serial version of the code initially, before subsequently proceeding to parallelise the program by adding multi-threading capabilities. The goal is to improve performance by optimally exploiting available multi-core processing resources. In contrast, a GPU program is parallelised from the outset, with each kernel typically describing the processing of a single parallel work-item, which will be executed by one hardware thread. This often results in parallelism that is more fine-grained than required to achieve optimal performance, and may thus incur a performance overhead. One may therefore wish to increase the number of work-items processed per thread, a process we refer to as *thread coarsening*.

We therefore define thread coarsening as increasing the number of work-items processed per thread. By merging several threads into one, the resulting program thus has fewer, more ‘coarsely grained’ threads. Thread coarsening has also been termed *granularity coarsening* for this reason (c.f. Stratton et al. [85]). Alternatively, thread coarsening can also be described as *kernel unrolling*, as it affects an unrolling of the implied loop nest within which kernels are executed (see Figure 2.2(b)). We define the coarsening factor  $C$  to be the number of threads merged into one.

The reduction of parallelism effected by thread coarsening can have both a beneficial and a detrimental effect on run-time requirements. To give the intuition, consider the simplified example shown in Figure 3.1a (we will discuss a more extensive example in Figure 3.4 below). While the actual program logic is abstracted away into methods not shown, it highlights a simple pattern underlying many kernels, i.e., the reading and processing of data, which may happen either in registers or in shared memory, a synchronisation barrier (line 8), and the subsequent

|                                                                                                                                                                                                                               |                                                                                                                                                                                                                                                                                                                                                                                               |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> 1 kernel foo(global float* in, out) { 2   uint idx = get_global_id(0); 3 4   float data = read_data(in, idx); 5 6   float result = do_some_work(data); 7 8   barrier(); 9   write_data(out, idx, result); 10 11 }</pre> | <pre> 1 kernel foo(global float* in, out) { 2   uint idx0 = 2 * get_global_id(0); 3   uint idx1 = 2 * get_global_id(0) + 1; 4   float data0 = read_data(in, idx0); 5   float data1 = read_data(in, idx1); 6   float result0 = do_some_work(data0); 7   float result1 = do_some_work(data1); 8   barrier(); 9   write_data(out, idx0, result0); 10  write_data(out, idx1, result1); 11 }</pre> |
| (a) Original code                                                                                                                                                                                                             | (b) Coarsened code                                                                                                                                                                                                                                                                                                                                                                            |

Figure 3.1: Effects of applying thread coarsening by a factor of 2 to a simple kernel.

writing back computed results to global memory. When applying thread coarsening to the kernel, all instructions that are dependent on the thread ID (`idx`) are duplicated, whilst all other instructions are shared between the two coarsened instances. One such *shared instruction* is a synchronisation barrier (line 8). Note, that for simplicity of presentation we apply coarsening with respect to `get_global_id()`, whereas below we discuss coarsening with respect to the underlying `get_local_id()` and `get_group_id()` functions.

The transformed code in Figure 3.1b has the potential to outperform the original code for two reasons:

1. As the coarsened version of the code requires only half the number of threads to be launched, the barrier is executed half as many times, when taken across the whole program execution.
2. The increased number of instructions that will be generated from the coarsened code increases the scope for exploiting hardware instruction-level parallelism.

These effects scale to some extent with increasing coarsening factor, although there will come a point where the reduction in the number of threads limits the ability of the application to exploit the available parallelism efficiently. Increasing the coarsening factor also raises a kernel's resource consumption, e.g., of registers, eventually resulting in reduced occupancy. An increased workload per thread can also increase the pressure on the cache in some kernels, which we will discuss in further detail in Chapter 5.

Sections 3.4 and 3.5 of this chapter describe two modes of coarsening – the first applies coarsening on a *thread-level*, merging threads into fewer, more coarsely grained threads per block, keeping the total number of thread blocks unchanged. The second approach applies coarsening on a *block-level*, merging thread blocks to obtain fewer, more coarsely grained blocks, leaving the number of threads per block unchanged. Figure 3.2 demonstrates how the launch configuration is affected in either case when applying a coarsening factor of  $C = 2$ . Note, that the total number of executing threads is reduced to  $\frac{B*N}{C}$ , independent of the coarsening mode.

Consider again the shared barrier statement in Figure 3.1b. Applying thread-level coarsen-

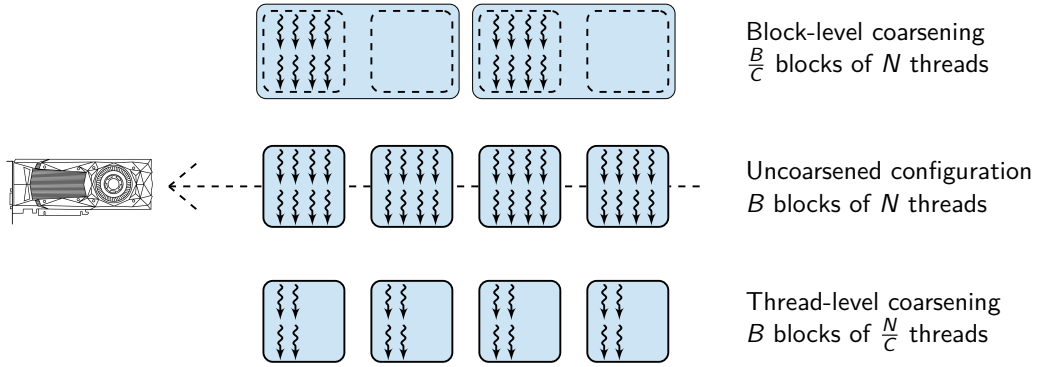


Figure 3.2: Effects on launch configuration of applying a coarsening factor of 2

ing will cause the barrier to be executed for the same number of blocks, yet with fewer threads per block waiting to synchronise at the barrier. In contrast, block-level coarsening will result in fewer blocks hitting the barrier, with an unchanged number of threads per block waiting to synchronise at each barrier invocation. We have informally conducted experiments to investigate a potentially different impact on performance, and found that the achieved speedup in case of a barrier is approximately the same for both modes of coarsening.

### 3.2 The code transformation

On a source code level the coarsening transformation effects a code duplication, which is in nature similar to an unrolled loop body. The code duplication is based on the *coarsening variable*, the choice of which determines the coarsening mode. As such, if coarsening is to be applied on a thread-level the coarsening variable is `get_local_id()`, while block-level coarsening is based on `get_group_id()`. Both coarsening modes are discussed in detail below. In either case, the transformation duplicates instances of the coarsening variable by assigning different values, i.e. multiple thread IDs ('local IDs') or multiple block IDs ('group IDs'), respectively. Subsequently, all instructions that use these duplicated definitions directly or indirectly are also duplicated. Code regions whose entry condition depend on a duplicated instruction are duplicated in their entirety. This algorithm, besides an LLVM-based implementation of thread-level coarsening (see Section 3.3), was proposed by Magni et al. [49].<sup>1</sup>

The algorithm avoids duplicating code regions where possible. This has the advantage of maximising the number of shared instructions. Consider the following example:

<sup>1</sup>The original description of the algorithm referred to duplicated/shared instructions as divergent/non-divergent. This choice of terminology may be seen as confusing due to its similarity to thread divergence, which is not meant by this 'coarsening divergence'. This has led the authors to state that 'non-divergent regions' can be identified by the presence of `barrier` statements, which according to the OpenCL specification should only occur in non-divergent code regions. However, the OpenCL specification is referring to thread divergence, not coarsening divergence.

```

1  int gid = get_global_id(0);
2  int tid = get_local_id(0);
3  __local float sdata[SIZE];
4  if (p) {
5      sdata[tid] = data_in[gid];
6      barrier();
7      sdata[tid] = f(sdata, tid);
8      out[gid] = sdata[tid];
9  }

```

A naive implementation could default to duplicating all code regions, e.g.:

```

1  int gid0 = ...;
2  int gid1 = ...;
3  int tid0 = ...;
4  int tid1 = ...;
5  __local float sdata[SIZE];
6  if (p) {
7      sdata[tid0] = data_in[gid0];
8      barrier();
9      sdata[tid0] = f(sdata, tid0);
10     out[gid0] = sdata[tid0];
11 }
12 if (p) {
13     sdata[tid1] = data_in[gid1];
14     barrier();
15     sdata[tid1] = f(sdata, tid1);
16     out[gid1] = sdata[tid1];
17 }

```

However, if predicate *p* is a shared instruction, the algorithm will not duplicate the branch, such that:

```

1  int gid0 = ...;
2  int gid1 = ...;
3  int tid0 = ...;
4  int tid1 = ...;
5  __local float sdata[SIZE];
6  if (p) {
7      sdata[tid0] = data_in[gid0];
8      sdata[tid1] = data_in[gid1];
9      barrier();
10     sdata[tid0] = f(sdata, tid0);
11     sdata[tid1] = f(sdata, tid1);
12     out[gid0] = sdata[tid0];
13     out[gid1] = sdata[tid1];
14 }

```

The resulting code is likely to have a performance advantage, as it has less control flow logic and also more shared instructions (i.e. the barrier instruction) in comparison to the previous version.

The choice of coarsening mode can also influence which code regions are duplicated. Recall that the two coarsening modes are based on different coarsening variables. It is thus possible to conceive that predicate *p* may have a dependence on `get_group_id()`, but not on

`get_local_id()` (or vice versa). This would mean that `p` requires duplication in block-level coarsening, but is considered a shared instruction in thread-level coarsening. In this example, the branch would require duplication for block-level coarsening (similar to the above naively coarsened code), but would not require duplication for thread-level coarsening, as shown in the second coarsened version. The choice of coarsening mode can thus significantly affect the structure of the coarsened code.

To highlight an important limitation of the algorithm presented by Magni et al. [49], consider the following bounds-check:

```

1  __kernel foo(__global float * data, int N) {
2      int gid = get_global_id();
3      if (gid < N)
4          // kernel body
5      }
6  }
```

In this example, the kernel body (not shown here) is wrapped in a bounds-check, to ensure that the launch configuration does not exceed the size of the data passed in. In practice, this is a wide-spread pattern employed by developers, although it could probably be omitted in many cases. In this case, thread coarsening will be forced to duplicate the branch, and thus effectively duplicate the entire kernel body, such that the resulting code is:

```

1  __kernel foo(__global float * data, int N) {
2      int gid0 = 2 * get_global_id();
3      int gid1 = 2 * get_global_id() + 1;
4      if (gid0 < N)
5          // kernel body
6      }
7      if (gid1 < N)
8          // kernel body
9      }
10 }
```

In this case, thread coarsening is not expected to have any performance advantage over the uncoarsened code due to a lack of shared instructions. Alternatively, the above example could be rewritten as:

```

1  __kernel foo(__global float * data, int N) {
2      int gid = get_global_id();
3      if (gid >= N)
4          return;
5      }
6      // kernel body
7  }
```

The coarsened code thus becomes:

```
1  __kernel foo(__global float * data, int N) {
2      int gid0 = 2 * get_global_id();
3      int gid1 = 2 * get_global_id() + 1;
4      if (gid0 >= N)
5          return;
6      }
7      if (gid1 >= N)
8          return;
9      }
10     // coarsened kernel body
11 }
```

Since the bounds-checking now happens before the kernel body, the body itself does not have to be duplicated in its entirety. However, this implementation is likely to break the coarsening logic entirely: If a single thread fails the bounds-check, both coarsened instances executed by this thread will terminate, leading to incorrect behaviour at run-time. This poses a limitation of the current implementation of the coarsening algorithm.

### 3.3 Implementation

An LLVM-based source-to-source compiler for OpenCL kernels is provided by Magni et al. [49]. In order to implement coarsening as a source-to-source transformation, calls to the OpenCL run-time API are intercepted and the coarsening pipeline described in Figure 3.3 is invoked. The coarsening pipeline parses an OpenCL kernel via Clang into LLVM IR, to which the coarsening transformation, implemented as an LLVM optimisation pass, is subsequently applied. The transformed LLVM IR is decompiled to OpenCL using the AXTOR tool [56], the ‘abstract syntax tree extractor’. The compilation chain is then continued by passing the transformed OpenCL code to the vendor’s compiler.

The coarsening pipeline is realised by intercepting calls to the OpenCL run-time API which

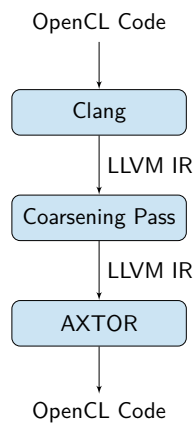


Figure 3.3: An LLVM Coarsening Pipeline.

handles the compiling and launching of a kernel. In particular:

- Intercepting calls to `clBuildProgram()`, which handles the compilation of a kernel. The intercepted kernel is parsed into LLVM IR via Clang, before running the coarsening pass which is implemented as an LLVM optimisation pass, and translating the result back to OpenCL using AXTOR. The intercepted method call then proceeds to invoke the implementation of `clBuildProgram()` provided by the vendor's driver, passing the modified coarsened kernel.
- Intercepting calls to `enqueueNDRangeKernel()`, which handles the launching of a kernel. The method arguments specify the requested launch configuration. The number of thread blocks and the size of each thread block are adjusted depending on the specified coarsening factor, as discussed below.

The system is designed to be controlled via environment variables covering the parameters of the optimisation, including the coarsening factor, coarsening dimension, stride, the name of the coarsened kernel, as well as further parameters such as compilation options for the underlying vendor's compiler. This allows for detailed manual configuration of the system. The implementation supports the thread-level coarsening mode, which is outlined below.

### 3.4 Thread-level coarsening

In thread-level coarsening each thread block performs the same amount of work but with fewer threads. The total number of thread blocks therefore remains unchanged, while the number of threads per block and the number of threads in total are reduced. This does not necessarily result in an immediate reduction in occupancy, as more smaller thread blocks can be scheduled simultaneously if sufficient resources are present. However, as each SM has limitations in terms of registers, shared memory, and concurrently runnable thread blocks, a reduction in occupancy is bound to occur eventually.

In order to combine the workload of several threads into one, it is necessary to specify which threads are to be merged. This is controlled via the *stride* parameter, acting as an offset between the IDs of threads that are to be combined.

To ensure that duplicated thread IDs calculated by using the stride value are within the boundaries of the thread block, the stride  $S$  must satisfy

$$S \leq \frac{\text{get\_local\_size}(d)}{C} \quad (3.1)$$

where  $d$  is the dimension in which coarsening is applied, and  $C$  is the coarsening factor. When choosing the stride parameter it is important to avoid breaking any existing memory coalescing [49]. On NVidia architectures,  $S$  should be at least as large as the warp size, in order to ensure that any memory coalescing (Section 2.2) exploited by an uncoarsened kernel is retained in its

```

1 #define BLOCK_SIZE 512
2 #define T double
3
4 __kernel void reduce2(__global T *g_idata, __global T *g_odata, uint n) {
5     uint tid = get_local_id(0);
6     uint i = get_global_id(0);
7     __local T sdata[BLOCK_SIZE];
8     sdata[tid] = (i < n) ? g_idata[i] : 0;
9     barrier(CLK_LOCAL_MEM_FENCE);
10
11     // do reduction in shared mem
12     for(uint s=get_local_size(0)/2; s>0; s>>=1) {
13         if (tid < s) {
14             sdata[tid] += sdata[tid + s];
15         }
16     }
17     barrier(CLK_LOCAL_MEM_FENCE);
18
19     // write result for this block to global mem
20     if (tid == 0) g_odata[get_group_id(0)] = sdata[0];
21 }

```

```

1 __kernel void thread_coarsened(__global T *g_idata, __global T *g_odata, uint n) {
2     uint tid0 = (get_local_id(0) / 32) * 32 * 2 + get_local_id(0) % 32;
3     uint tid1 = tid0 + 32;
4     uint i0 = get_group_id(0) * 2 * get_local_size(0) + tid0;
5     uint i1 = get_group_id(0) * 2 * get_local_size(0) + tid1;
6     __local T sdata[BLOCK_SIZE];
7     sdata[tid0] = i0 < n ? g_idata[i0] : 0;
8     sdata[tid1] = i1 < n ? g_idata[i1] : 0;
9     barrier(CLK_LOCAL_MEM_FENCE);
10    for (uint s = 2*get_local_size(0)/2; s>0; s>>=1) {
11        if (tid0 < s) {
12            sdata[tid0] = sdata[tid0] + sdata[tid0+s];
13        }
14        if (tid1 < s) {
15            sdata[tid1] = sdata[tid1] + sdata[tid1+s];
16        }
17        barrier(CLK_LOCAL_MEM_FENCE);
18    }
19    // only one write - the condition of the second will never be true
20    if (tid0==0) g_odata[get_group_id(0)] = sdata[0];
21    if (tid1==0) g_odata[get_group_id(0)] = sdata[0];
22 }

```

```

1 __kernel void block_coarsened(__global T *g_idata, __global T *g_odata, uint n) {
2     uint tid = get_local_id(0);
3     uint i0 = 2 * get_group_id(0) * get_local_size(0) + get_local_id(0);
4     uint i1 = (2 * get_group_id(0) + 1) * get_local_size(0) + get_local_id(0);
5     __local T sdata0[BLOCK_SIZE];
6     __local T sdata1[BLOCK_SIZE];
7     sdata0[tid] = (i0 < n) ? g_idata[i0] : 0;
8     sdata1[tid] = (i1 < n) ? g_idata[i1] : 0;
9     barrier(CLK_LOCAL_MEM_FENCE);
10    for (uint s=get_local_size(0)/2; s>0; s>>=1) {
11        if (tid < s) {
12            sdata0[tid] += sdata0[tid + s];
13            sdata1[tid] += sdata1[tid + s];
14        }
15        barrier(CLK_LOCAL_MEM_FENCE);
16    }
17    if (tid == 0) {
18        g_odata[2 * get_group_id(0)] = sdata0[0];
19        g_odata[2 * get_group_id(0) + 1] = sdata1[0];
20    }
21 }

```

Figure 3.4: A reduction kernel from the NVidia OpenCL SDK shown uncoarsened, with thread-level, and block-level coarsening, for stride  $S = 32$  (thread-level coarsening only) and coarsening factor  $C = 2$ .

coarsened equivalent. In a similar manner, the stride parameter can affect accesses to shared memory by introducing bank conflicts for kernels that make use of shared memory resources. Bank conflicts occur when several threads from the same warp access data stored in the same bank of shared memory, causing the device to serialise the access.

In the kernel body, code duplication is based on calls to *get\_local\_id()*, and includes all dependent instructions. Rewrite rules for relevant OpenCL functions are outlined in Table 3.1.

Dealing with allocated shared memory resources under thread-level coarsening is straightforward, as shared memory is allocated on a per-block basis in the OpenCL programming model. Although the actual number of threads in a block will change as a result of coarsening, each block still needs to perform the work of all threads specified in the original launch configuration. Consequently, shared memory allocations inside a kernel are left unaltered, as the same amount of shared memory per block is required. However, as discussed, shared memory requirements per SM may increase due to an increase in the number of concurrently executable blocks per SM.

The first two code blocks of Figure 3.4 show the original version of a reduction kernel (*reduce2*) from the NVidia OpenCL SDK together with the coarsened version (*thread\_coarsened*) built using the thread-level coarsening rules of Table 3.1. The kernel is one-dimensional, as indicated by the zero-argument with which OpenCL built-in functions are invoked. It can be seen that all instructions depending on *get\_local\_id(0)* have been duplicated, including code regions whose entry condition depends on a duplicated value, such as the conditional inside the *for*-loop. The chosen value for the stride in this case is 32. Note, that for presentational purposes only the rewrite of the *i* variable has been simplified, to avoid repeating the expression of the previous line. Since the kernel will be executed by half the number of threads per block, the value of *get\_local\_size(0)* has been scaled by a factor of two, to yield the original value. No changes are required to the use of the shared memory resources.

### 3.5 Block-level coarsening

With block-level coarsening the number of threads per block remains unchanged so that the number of executed thread blocks is reduced by the coarsening factor. Because each thread block has to handle an increased workload, resource requirements per block, in terms of register and shared memory usage, will typically increase, as with thread-level coarsening.

The stride parameter now determines which blocks will be merged. Unlike thread-level coarsening, the stride has no influence on memory coalescing or bank conflicts, as the original memory access patterns of the uncoarsened code within each block are preserved. Choosing a different stride gives a degree of control over the order in which blocks are scheduled. This is in line with NVidia’s CUDA and OpenCL programming guidelines, which state that the thread block scheduler does not commit to schedule blocks in any particular order (see [60]). In practice we have not encountered any significant correlation between stride parameter and performance and therefore choose a static value of 1 for the stride throughout our evaluation.

| OpenCL function                | After thread-level coarsening                                                                                  | After block-level coarsening          |
|--------------------------------|----------------------------------------------------------------------------------------------------------------|---------------------------------------|
| <code>get_local_id()</code>    | $\lfloor \frac{\text{get\_local\_id}()} {S} \rfloor * S * C$<br>+ <code>get_local_id() * mod(S) + i * S</code> | <code>get_local_id()</code>           |
| <code>get_group_id()</code>    | <code>get_group_id()</code>                                                                                    | $C * \text{get\_group\_id}() + i * S$ |
| <code>get_global_size()</code> | $C * \text{get\_global\_size}()$                                                                               | $C * \text{get\_global\_size}()$      |
| <code>get_local_size()</code>  | $C * \text{get\_local\_size}()$                                                                                | <code>get_local_size()</code>         |
| <code>get_num_groups()</code>  | <code>get_num_groups()</code>                                                                                  | $C * \text{get\_num\_groups}()$       |
| <code>get_global_id()</code>   | (rewrite using def: $\text{get\_group\_id}() * \text{get\_local\_size}() + \text{get\_local\_id}()$ )          |                                       |

Table 3.1: Implementation details for thread and block-level coarsening, with *stride*  $S$ , coarsening factor  $C$ , and  $i$  as the index of the coarsened thread or thread block such that  $0 \leq i < C$ .

In the kernel body, code duplication is based on calls to `get_group_id()`, and includes all dependent instructions (c.f. Table 3.1).

Shared memory is allocated on a per-block basis. As the work of several thread blocks is combined into one, shared memory regions need to be duplicated along with other instructions of the kernel body. However, a shared memory region needs to be duplicated if and only if it handles a value which is also duplicated in the process of thread coarsening. For dynamically allocated shared memory regions, passed as function parameters to the kernel, such duplication is not possible. Throughout our experiments, we have thus chosen to restructure dynamically allocated shared memory into static allocations as a manual step, as can be seen in Figure 3.4, where shared memory pointers were originally passed in as function arguments.

The third code block of Figure 3.4 shows the coarsened version (`block_coarsened`) of the original code (first code block in the figure), this time built using the block-level coarsening rules of Table 3.1. The example shows that after rewriting the call to `get_global_id(0)` using its definition, all instances of `get_group_id(0)` and dependent instructions are duplicated. Note, that this includes the allocated shared memory region, giving each of the two merged blocks their own memory region. Unlike thread-level coarsening, no code region requires duplication (this characteristic is more important for longer-running code regions). As such, if the duplicated code regions were expensive to execute, the performance achieved by the two coarsening modes could be expected to differ significantly from each other. The kernel will be executed by half the number of thread blocks, with each block performing twice the number of accesses to global memory as the original.

### 3.6 Preserving block semantics

It is important to ensure that coarsening is a sound transformation in that the semantics of each thread block are preserved. By this we simply mean that the output of a block is the same before and after coarsening, in order to preserve the way a kernel communicates with other kernels or host code.

To illustrate this, consider the reduction kernel shown in Figure 3.4, where each block computes one value of a list of partial sums. The transformation is sound if the produced list is

unaltered, i.e., if each block produces the same output value to the same list index as before. In the particular case of a reduction operation this rule could potentially be relaxed if the intention is to perform a full reduction, but this is not generally possible in other cases.

Block semantics are preserved by modifying the launch configuration in a way that corresponds to the applied code transformation rules stated above. Consider again the examples in Figure 3.4. Both versions should produce the same unaltered list of partial sums as the uncoarsened code. Intuitively, thread-level coarsened code should produce one value per block, while block-level coarsened code should produce two values per block in this example (i.e., one value per ‘merged’ block for a coarsening factor of 2). Each value should then be written to its intended index, to retain the same output as without coarsening. However, this is the case if and only if for thread-level coarsening (assuming coarsening factor 2) the number of threads per block is halved and the total number of blocks is retained, whereas for block-level coarsening the thread block size is retained while the total number of blocks is halved. This is also reflected in the above code transformation rules. Any additional modification to the configured number of threads per block will result in a differently sized list of partial sums produced by the kernel, which may break program semantics. This means that thread block sizes cannot be chosen freely during coarsening, as suggested by Magni et al. [49]. The above transformation rules therefore preserve block semantics and thereby program semantics, by ensuring that each block or ‘merged’ block has the same input/output behaviour before and after coarsening.

A notable exception where block sizes can be modified during coarsening, are kernels that do not refer to any block semantics in their implementation, such as many linear algebra kernels, or the examples shown below in Figure 5.1 and Figure 5.2, which allow free manipulation of thread block sizes, in line with Magni et al. [49]. The underlying issue is that the CUDA and OpenCL programming models allow writing a kernel body both from the perspective of an individual thread (c.f. Figure 5.1) and from the perspective of a thread block (c.f. Figure 3.4) (or a mixture thereof). The latter, however, is the more general and should be followed in discussions involving code transformations.



# Predicting Optimal Coarsening Factors 4

Thread coarsening has been extensively applied as a manual, semi-automatic, auto-tuning-based, and machine-learning-based optimisation (c.f. [95], [92], [49], [51], respectively).

Throughout this chapter we will consider kernels to which thread coarsening applies as an optimisation technique, which is to say that a performance gain can be expected. For these kernels we will focus on coarsening factor selection, showing how code analysis in combination with architectural insight can be utilised to select optimal coarsening factors (Section 4.1). We compile differently coarsened versions of a kernel to gain insight about resource usage by observing the compiler output, without requiring to execute the kernel. The resulting information is used to model the achievable occupancy, in order to select the maximum coarsening factor that does not reduce occupancy (Section 4.2). We extend the existing LLVM coarsening pipeline to support fully automated selection of coarsening factors (Section 4.3). We evaluate the performance of our coarsening factor predictions when applied to code pre-optimised to different degrees, and across three different architectures (Section 4.4). We also explore the effects of varying problem sizes on optimal coarsening factor selection (Section 4.5).

In the following chapter we will consider kernels that are not safe to coarsen due to risks associated with cache pressure, and will propose a way to identify and handle these kernels. At that point a much broader evaluation of fully automated thread coarsening becomes possible, such that we plan to revisit some of the methods presented here by the end of the following chapter.

## 4.1 Coarsening Factor Selection

Assuming that a given kernel would benefit from coarsening, the challenge is to determine the coarsening factor with which the optimisation should be applied, so as to maximise the resulting performance gain.

One primary effect of thread coarsening is to reduce the workload stored on a GPU’s processing queue and increase the workload actively residing in its processors. If sufficient resources are present, we may expect the ideal coarsening factor to be such that no work resides on any processing queue, while all SMs are fully and equally occupied, to the effect that all threads are simultaneously processed in one batch. Intuitively, we may expect that every increase in coarsening factor up to this exact point is likely to yield a continuous increase in performance.

As such, we wish to select a coarsening factor  $C$  that reduces the degree of software parallelism employed in such a way that the degree of parallelism encoded in the software  $P_S$  is equal to the degree of parallelism exhibited by the hardware  $P_H$ . In other words, the ideal coarsening factor  $C_I$  is such that

$$C_I = \frac{P_S}{P_H} = \frac{\text{numThreadsLaunched}}{\text{numSMs} * \text{threadsPerSM}}, \text{ assuming } P_S \geq P_H \quad (4.1)$$

This assumes  $P_S \geq P_H$  - otherwise,  $C_I$  defaults to 1.

In practice, the optimal coarsening factor  $C_{opt}$  may be significantly lower than  $C_I$ , and finding  $C_{opt}$  is as such a *maximisation problem*. For instance, consider that a large computational grid implies a large  $C_I$ , resulting in kernels with high register and shared memory requirements when coarsened with this factor. In practice, however, we expect to encounter hardware limitations that may cause the observed optimal coarsening factor to be lower than this, such that  $C_{opt} \leq C_I$ . We expect  $C_{opt}$  to be the maximum coarsening factor not affected by resource limitations. As a consequence, we also expect that every coarsening factor  $C$  such that  $1 \leq C \leq C_{opt}$  will yield a continuous performance increase as  $C$  is increased.

It is universally agreed that no single coarsening factor is optimal either for all kernels on a given platform, or across different platforms for a given kernel (e.g., see [49]). This includes platforms which are architecturally similar, such as different generations of GPUs from the same vendor. Pre-determining the optimal coarsening factor therefore encapsulates two challenges:

- To understand a kernel's resource requirements and how these requirements are affected by different coarsening factors.
- To understand the underlying architecture sufficiently well to identify the point at which resource limitations are likely to become a performance issue.

Combining these two factors forms the basis for selecting optimal coarsening factors in an automated fashion. The first point concerning a kernel's resource requirements splits into two separate parts, covering explicitly and implicitly managed resources. Explicitly managed resources on NVidia GPUs are registers and shared memory, and will be the focus of this chapter. Implicitly managed resources are primarily concerned with data caches, and will be considered in more detail in the following chapter. Explicitly managed resources can likely be modelled or predicted in some fashion by a compiler analysis. Such results are heuristic by nature. However, details concerning register and shared memory usage are retrievable from the verbose output of the underlying compiler, such that invoking the compiler yields reliable and precise information regarding resource usage.

The remaining part of the problem thus is to identify the point at which resource limitations will start affecting the performance gained by thread coarsening. One important consideration here is the occupancy lowering mechanism employed by NVidia GPUs. As described in Section 2.5.5, the GPU's thread block scheduler may choose to schedule fewer thread blocks per SM than the maximum, thereby artificially lowering  $P_H$ . This is done for kernels with high

register usage or high shared memory requirements, of which only a limited number of thread blocks can be simultaneously handled by any of the SMs. Thread coarsening naturally increases a kernel’s resource consumption, an effect which is amplified for higher coarsening factors. An artificially lowered  $P_H$ , however, is unlikely to yield a performance increase. Although a performance increase remains possible, it is currently unclear how to reason about performance at lower occupancy (also see Section 5.8.5). Applying coarsening factors which result in lowered occupancy is thus in principle unsafe in terms of performance.

The optimal coarsening factor is therefore estimated to be the largest factor that preserves the occupancy of the original kernel. Beyond this the occupancy will start to drop, as more resources are required to support increasingly larger kernels, and our current assumption is that this will lead to reduced performance. This is one sense in which our model is approximate, as it is well known that some kernels can exhibit better performance at lower occupancy [94], an effect that we observe in some of our benchmarks. Our objective here is to evaluate how well the simple occupancy-based metric works in practice.

Finally, for kernel launches with small computational grids, the maximum coarsening factor we select is such that  $P_S = P_H$  after coarsening. This means that, for instance, if  $P_S < P_H$  for  $C = 1$ , the computational grid specified by the kernel’s launch configuration does not exhibit a sufficient degree of software parallelism for further coarsening. Every increase in  $C$  will cause a reduction of *achieved* occupancy, having the same effect as the reduction in theoretical occupancy discussed above. As before, at this point reasoning about performance becomes significantly harder.

## 4.2 An occupancy-based model for coarsening factor selection

We seek to define a model for the selection of coarsening factors, with respect to the interaction between a set of NVidia GPU architectures, a given kernel implementation, the effects of the thread coarsening optimisation when applied to a kernel, and the requested grid size.

Our model characterises kernels by their resource usage, referring to register and shared memory usage. For a set of candidate coarsening factors, the resource usage of each coarsened version of the code is determined. In practice, this is done by compiling the kernel multiple times and parsing the compiler’s output which states register and shared memory usage. While this has low overhead, we nevertheless consider that other, possibly heuristic approaches to determine resource usage may be used as an alternative.

GPU architectures are characterised by their compute and memory resources, in particular those that determine a given kernel’s occupancy. These are, the number of SMs, active threads per SM, the limit of runnable thread blocks per SM, total shared memory per SM, shared memory addressable per block, and registers per SM.

The interaction between coarsened code and the GPU device is modelled by *achievable occupancy* (see Section 2.5.5), effectively a whole-device theoretical occupancy with respect to the requested launch configuration. Our approach is to determine the achievable occupancy of

each coarsened version of the code on a given architecture and for a given launch configuration. We select the maximum coarsening factor that does not reduce the achievable occupancy below the level of the uncoarsened kernel.

One implication that we wish to reiterate, is that this approach is to a large degree architecture specific, relying both on architectural concepts as well as – in its current form – on the vendor’s compiler. In the following chapter, we will extend this model by with respect to a kernel’s caching behaviour.

### 4.3 Implementation

We have extended the existing thread coarsening pipeline outlined in Section 3.3 by adding support for block-level coarsening alongside the existing thread-level coarsening functionality. The AXTOR (abstract syntax tree extractor) component has also been extended with the ability to handle statically allocated shared memory regions. Finally, we have implemented the re-compilation of a given kernel for a selection of coarsening factors, parsing of the compilers output to gain information on the kernel’s resource requirements, a store for these data and the compiled kernels, a mechanism to predict each kernel’s achievable occupancy once the launch configuration is known, and a model to select a suitable coarsening factor based on these factors as described above. Storing differently coarsened versions of the program means that coarsening factors can be selected dynamically at run-time based on the problem size, which is not known until a kernel launch is requested; the version corresponding to the selected coarsening factor is then launched for execution.

The implementation of block-level coarsening follows from the specification given in Section 3.5 above. This required changes to be made both in the coarsening pass, as well as in the `enqueueNDRangeKernel()` function, where the launch configuration had to be adjusted accordingly. In line with the existing design, we added an environment variable to control which mode of coarsening is to be selected. Adding support for block-level coarsening required multiple changes to be made to AXTOR, which did not support statically allocated shared memory. However, block-level coarsening is incompatible with dynamically allocated shared memory, as the allocated memory region may require duplication in the process of applying coarsening. Throughout this thesis, we have, therefore, manually re-written dynamic allocations of shared memory resources as static allocations.

For the implementation of the coarsening factor selection, intercepted calls to `clBuildProgram()` are extended by statically compiling the kernel multiple times with different coarsening factors. The additional compilation cost depends on the number of coarsening factors considered (in our experiments we use only powers of two but this is an arbitrary decision). Importantly, since the optimal coarsening factor depends on the specified problem size, the choice of coarsening factor can only be made when the launch configuration is known, i.e. when `enqueueNDRangeKernel()` is called. Compiling differently coarsened versions of the same kernel with the compiler’s verbosity flag switched on allows us to extract the compiler output by

querying `clGetProgramBuildInfo()`. For NVidia compilers, the build log contains parsable information for each instance of the code including register usage per thread and statically allocated shared memory usage per thread block. The stated register requirements per thread can be directly used for the purposes of calculating the kernel's theoretical occupancy. However, with statically allocated shared memory, this is not possible at this stage. Since the value is stated on a per-thread-block basis, it cannot, however, be meaningfully interpreted at kernel compile-time. Instead, it needs to be interpreted in the context of the thread block size, which is not known until `enqueueNDRangeKernel()` is called when the kernel is about to be launched. The resource requirements extracted from the build log are therefore entered into a cache alongside the compiled kernels.

When `enqueueNDRangeKernel()` is called to execute a kernel with a given launch configuration, the system has a number of pre-compiled, differently coarsened versions of the kernel to choose from. Register requirements for each version are known and can be retrieved from the store. Shared memory requirements can at this stage be interpreted in the context of the requested thread block size specified in the launch configuration passed as a parameter to the `enqueueNDRangeKernel()` API call. For instance, 12K of shared memory used by a block of 128 threads will affect the occupancy calculation differently as compared to 12K of shared memory used by 1024 threads (assume, for instance, a maximum of 48K shared memory and 2048 maximum active threads). The occupancy achievable by each of the coarsened instances of a given kernel is calculated in terms of the number of requested thread blocks, number of threads per block, registers per thread, and the amount shared memory per thread block. This is compared against architectural parameters, such as the available shared memory per SM, maximum executable thread blocks per SM, registers per SM, maximum active threads per SM, etc. These are currently provided as environment variables. The occupancy calculation is described in the NVidia SDK documentation.

The calculated occupancies for each version of the kernel, in combination with the problem size, are used to select the desired coarsening factor. As explained previously, we expect the optimal coarsening factor to be the maximum coarsening factor which does not reduce the occupancy below that of the uncoarsened version of the kernel. The system will then execute the kernel with the automatically selected coarsening factor, although a coarsening factor may also be specified manually if desired.

## 4.4 Evaluation

In our evaluation we wish to investigate whether occupancy is indeed useful as a guiding principle for coarsening factor selection. Furthermore, we seek to establish how well our method of selecting coarsening factors applies to code running at different fractions of theoretical peak performance, i.e., whether our model works for both unoptimised and optimised code. We also aim to establish whether the model can be adapted to different generations of NVidia GPUs. Finally, we wish to investigate our claim that coarsening factor selection needs to be done with

respect to the problem size as specified in the launch configuration. As mentioned before, we will revisit further questions about thread coarsening policy at the end of the following chapter.

To evaluate our findings, we use a set of differently optimised versions of the same program provided by the NVidia OpenCL SDK. The NVidia OpenCL SDK includes seven reduction kernels that are optimised to different degrees as described in [37]. Table 4.2 outlines the optimisations applied to each version of the code. As the kernels are provided by vendor, the optimisations applied are highly hardware-specific. As an example, consider the `reduce2` kernel discussed in detail in Figure 3.4. The same figure also highlights the effects of thread-level and block-level coarsening on the kernel, as discussed above. By choosing these kernels for the purposes of our evaluation, we are able to investigate the efficiency of our model under different degrees of utilised peak performance. Our experimental setup is described in Table 4.1.

| Architecture                          | NVidia Kepler           | NVidia Maxwell      | NVidia Pascal         |
|---------------------------------------|-------------------------|---------------------|-----------------------|
| <b>Device Name</b>                    | GeForce GTX TITAN Black | GeForce GTX TITAN X | GeForce GTX 1080      |
| <b>CUDA Compute Capability</b>        | 3.5                     | 5.2                 | 6.1                   |
| <b>OpenCL Version</b>                 | 1.2                     | 1.2                 | 1.2                   |
| <b>CPU</b>                            | Intel Core i7-4770K     | Intel Core i7-4770K | Intel Xeon E5-1630 v3 |
| <b>OS</b>                             | Ubuntu 14.04            | Ubuntu 14.04        | Ubuntu 16.04          |
| <b>Streaming Multiprocessors (SM)</b> | 15                      | 24                  | 20                    |
| <b>Max. active threads per SM</b>     | 2048                    | 2048                | 2048                  |
| <b>Max. thread blocks per SM</b>      | 16                      | 32                  | 32                    |
| <b>Registers per SM</b>               | 64k                     | 64k                 | 64k                   |
| <b>Shared Memory per SM</b>           | 48kB                    | 96kB                | 96kB                  |
| <b>Max. Shared Memory per block</b>   | 48kB                    | 48kB                | 48kB                  |

Table 4.1: Properties of the test systems.

In our experiments we chose to execute versions of the code for all coarsening factors, rather than only those coarsening factors selected by our model. This is to allow for a more detailed comparison of the performance of different coarsening factors, leading to a clear conclusion regarding whether the selected coarsening factors are optimal for the tested benchmarks. Each benchmark has been tested across three different architectures (see Table 4.1), for both modes of coarsening, and a selection of coarsening factors which are powers of 2 ranging from 1 to 32, both ends inclusive. Each experiment was repeated ten times, and we report the average. The standard deviation was measured to be 0.19%, with a worst case of 5%. We measured the throughput of each kernel for each experiment using a test data set of 450MB of single-precision floating-point values. Kernels were executed with a size of 512 threads per block. For thread-level coarsening, we chose a constant stride of 32. We choose this stride value across all our experiments, such as to not break any potential memory coalescing, which the reduction kernels use to read data from global memory.

Analysing the code of the kernels reveals that cache pressure will not be an issue in any of the experiments, as data is accessed in a streaming manner, and not subsequently accessed again. Recall, that cases where cache pressure may become an issue will have to be treated differently, as discussed in the following chapter. Performance graphs for each architecture and coarsening

mode are shown in Figure 4.1, each graph plotting the applied coarsening factor (vertical axis) against the achieved throughput (horizontal axis) for each of the seven reduction kernels. The results shown are averages from ten independent executions of each kernel for each coarsening factor.

The coarsening factors selected are highlighted with a cross. The model predictions for `reduce0`-`reduce5` turn out to be identical. The reason for this is that all versions of the code allocate the same amount of shared memory, which in this case acts as the limiting factor to occupancy. Therefore, the estimated maximum coarsening factor is similar for all but the last kernel. For all other kernels, the optimal coarsening factor is 4 on the Kepler architecture, and 8 on the Maxwell and Pascal architectures, which our model is able to correctly identify in each case.

The graphs show a performance increase that corresponds to a speedup of up to 2.84x for thread-level coarsening (`reduce3` on Kepler), and up to 5.08x for block-level coarsening (`reduce0` on Pascal). The average performance increase (geometric mean) achieved by thread-level and block-level coarsening when taken across all benchmarks and architectures is 1.73x and 1.97x, respectively.

The graphs show that kernels coarsened according to our model match or exceed the performance of uncoarsened kernels that are optimised to roughly two degrees more – all of which can be achieved in a fully automated fashion.

The `reduce6` kernel is launched with a constant number of thread blocks, small enough that all can be actively processed at the same time. This renders coarsening obsolete as applying the optimisation would in practice mean a reduction in occupancy. As our model considers the requested launch configuration in its occupancy calculation, this is correctly identified, and the kernel is not coarsened. Without this restriction the kernel would run at  $0.26x$  to  $0.57x$  of its original performance.

Tables 4.3 - 4.8 list the performance gained in each benchmark using our method for coarsening factor selection. The original throughput is stated (i.e., of the uncoarsened code) alongside

| Kernel               | Optimisation                                                                                                               |
|----------------------|----------------------------------------------------------------------------------------------------------------------------|
| <code>reduce0</code> | Interleaved addressing (using modulo arithmetic) with divergent branching.                                                 |
| <code>reduce1</code> | Interleaved addressing (using contiguous threads) with bank conflicts.                                                     |
| <code>reduce2</code> | Sequential addressing, no divergence or bank conflicts.                                                                    |
| <code>reduce3</code> | Uses $n/2$ threads, performs first level during global load.                                                               |
| <code>reduce4</code> | Unrolled loop for last warp, intra-warp synchronisation barriers removed.                                                  |
| <code>reduce5</code> | Completely unrolled, using template parameter to assert whether the number of threads is a power of two.                   |
| <code>reduce6</code> | Multiple elements per thread, small constant number of thread blocks launched. Requires very few synchronisation barriers. |

Table 4.2: Optimisation levels of reduction kernels from the NVidia OpenCL SDK.

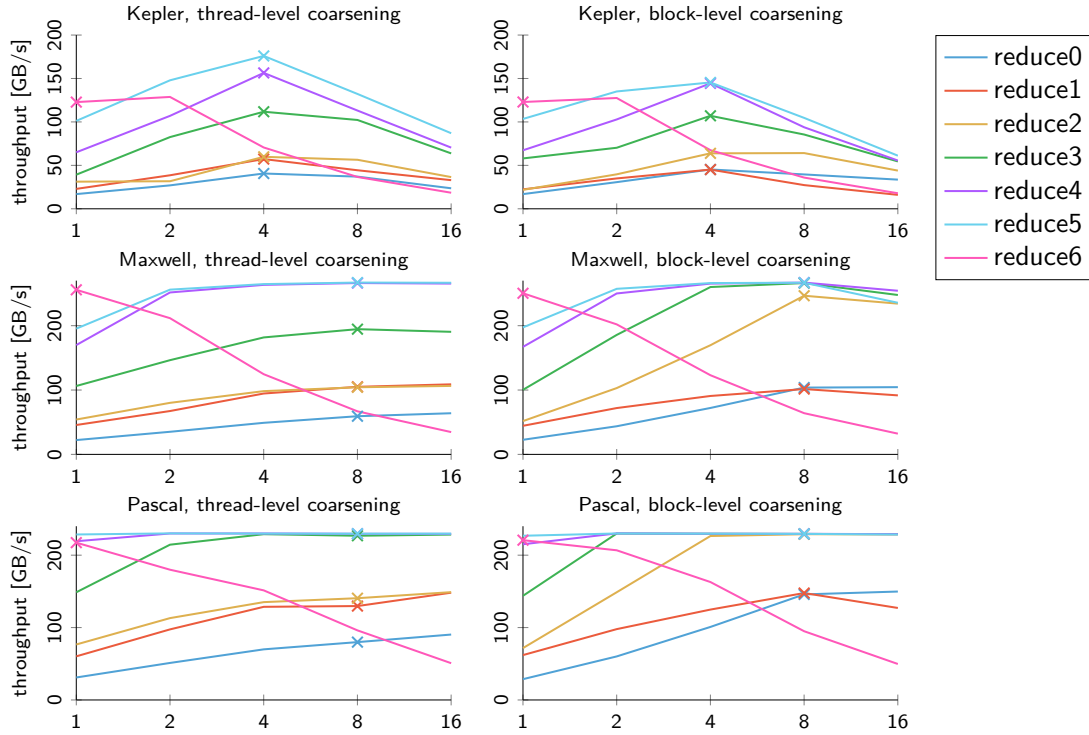


Figure 4.1: Reduction benchmarks, plotting achieved throughput over coarsening factor. Marked are the points selected by our model.

the throughput achieved when following our model, as well as the optimal observed throughput. The speedup achieved by our coarsening factor predictions is also stated. The final column of each table states how close to the optimum our selected coarsening factors performed, where for instance 100% means that the performance of the selected coarsening factor was optimal.

Our experiments comprised of seven benchmarks executed across three architectures and for two different modes of coarsening, resulting in a total of 42 coarsening factor predictions. Of these, 25 yielded the optimal performance, while only 4 selected coarsening factors yielded less than 95% of the optimal performance, of which the lowest was 87.54% (see Table 4.5). This means that on average our predictions were within 98.7% of the optimal performance.

The results appear to suggest that in several of the experiments the performance gain stabilises at a certain point, hitting a performance roofline (c.f. [98]). This effect is particularly prevalent in experiments on the Pascal architecture, but also occurs to a lesser degree on the Maxwell architecture. We do not clearly observe the same effect on the Kepler architecture. For some experiments, several coarsening factors were able to hit the performance roofline on the Maxwell and Pascal architectures. The effect is that some of our predictions are within 1% of the optimal performance. The difference that remains may be explained by small errors and deviations in measurements.

| Kernel  | original [GB/s] | model [GB/s] | best [GB/s] | model speedup | within % of optimal |
|---------|-----------------|--------------|-------------|---------------|---------------------|
| reduce0 | 16.92           | 40.62        | 40.62       | 2.40x         | 100.00%             |
| reduce1 | 22.93           | 57.25        | 57.25       | 2.50x         | 100.00%             |
| reduce2 | 31.23           | 59.82        | 59.82       | 1.92x         | 100.00%             |
| reduce3 | 39.28           | 111.73       | 111.73      | 2.84x         | 100.00%             |
| reduce4 | 65.03           | 156.41       | 156.41      | 2.41x         | 100.00%             |
| reduce5 | 101.17          | 175.95       | 175.95      | 1.74x         | 100.00%             |
| reduce6 | 122.82          | 122.82       | 128.75      | 1.00x         | 95.40%              |

Table 4.3: Kepler, thread-level coarsening.

| Kernel  | $T_{init}$ [GB/s] | $T_{model}$ [GB/s] | $T_{best}$ [GB/s] | achieved speedup | within % of optimal |
|---------|-------------------|--------------------|-------------------|------------------|---------------------|
| reduce0 | 22.23             | 59.30              | 63.94             | 2.67x            | 92.76%              |
| reduce1 | 45.66             | 105.01             | 108.85            | 2.30x            | 96.47%              |
| reduce2 | 54.16             | 104.30             | 106.40            | 1.93x            | 98.03%              |
| reduce3 | 106.29            | 194.59             | 194.59            | 1.83x            | 100.00%             |
| reduce4 | 169.86            | 266.25             | 266.25            | 1.57x            | 100.00%             |
| reduce5 | 195.41            | 267.00             | 267.00            | 1.37x            | 100.00%             |
| reduce6 | 255.76            | 255.76             | 255.76            | 1.00x            | 100.00%             |

Table 4.4: Maxwell, thread-level coarsening.

| Kernel  | $T_{init}$ [GB/s] | $T_{model}$ [GB/s] | $T_{best}$ [GB/s] | achieved speedup | within % of optimal |
|---------|-------------------|--------------------|-------------------|------------------|---------------------|
| reduce0 | 31.06             | 79.90              | 90.36             | 2.57x            | 88.43%              |
| reduce1 | 60.26             | 129.72             | 148.18            | 2.15x            | 87.54%              |
| reduce2 | 76.63             | 140.56             | 148.93            | 1.83x            | 94.38%              |
| reduce3 | 148.75            | 226.85             | 228.98            | 1.53x            | 99.07%              |
| reduce4 | 219.26            | 229.53             | 229.99            | 1.05x            | 99.80%              |
| reduce5 | 228.59            | 229.71             | 229.84            | 1.00x            | 99.95%              |
| reduce6 | 217.31            | 217.31             | 217.31            | 1.00x            | 100.00%             |

Table 4.5: Pascal, thread-level coarsening.

The experiments highlight that thread-coarsening applies well as an optimisation technique to a variety of reduction kernels. One of the reasons may be that all but the last of the reduction kernels may be considered barrier-intensive. Liu et al. [47] have recently highlighted the importance of optimising stalls at synchronisation barriers. This characteristic seems to account for a significant proportion of the achieved speedups, although a difference exists between the two modes of coarsening. Specifically, synchronisation barriers are either hit by fewer thread blocks (block-level coarsening), or by fewer threads per block (thread-level coarsening); both appear to have a similar effect on performance.

| Kernel  | $T_{init}$ [GB/s] | $T_{model}$ [GB/s] | $T_{best}$ [GB/s] | achieved speedup | within % of optimal |
|---------|-------------------|--------------------|-------------------|------------------|---------------------|
| reduce0 | 16.93             | 45.27              | 45.27             | 2.67x            | 100.00%             |
| reduce1 | 22.39             | 45.13              | 45.13             | 2.02x            | 100.00%             |
| reduce2 | 21.82             | 63.85              | 64.13             | 2.93x            | 99.55%              |
| reduce3 | 57.98             | 107.05             | 107.05            | 1.85x            | 100.00%             |
| reduce4 | 67.36             | 144.40             | 144.40            | 2.14x            | 100.00%             |
| reduce5 | 103.44            | 145.42             | 145.42            | 1.41x            | 100.00%             |
| reduce6 | 122.93            | 122.93             | 127.53            | 1.00x            | 96.39%              |

Table 4.6: Kepler, block-level coarsening.

| Kernel  | $T_{init}$ [GB/s] | $T_{model}$ [GB/s] | $T_{best}$ [GB/s] | achieved speedup | within % of optimal |
|---------|-------------------|--------------------|-------------------|------------------|---------------------|
| reduce0 | 22.68             | 103.80             | 104.46            | 4.58x            | 99.37%              |
| reduce1 | 44.41             | 101.52             | 101.52            | 2.29x            | 100.00%             |
| reduce2 | 51.75             | 246.55             | 246.55            | 4.76x            | 100.00%             |
| reduce3 | 100.38            | 266.22             | 266.22            | 2.65x            | 100.00%             |
| reduce4 | 167.09            | 266.95             | 266.95            | 1.60x            | 100.00%             |
| reduce5 | 197.36            | 266.78             | 266.78            | 1.35x            | 100.00%             |
| reduce6 | 250.37            | 250.37             | 250.37            | 1.00x            | 100.00%             |

Table 4.7: Maxwell, block-level coarsening.

| Kernel  | $T_{init}$ [GB/s] | $T_{model}$ [GB/s] | $T_{best}$ [GB/s] | achieved speedup | within % of optimal |
|---------|-------------------|--------------------|-------------------|------------------|---------------------|
| reduce0 | 28.73             | 145.83             | 149.77            | 5.08x            | 97.37%              |
| reduce1 | 61.99             | 147.68             | 147.68            | 2.38x            | 100.00%             |
| reduce2 | 71.67             | 228.95             | 228.95            | 3.19x            | 100.00%             |
| reduce3 | 143.87            | 229.33             | 229.89            | 1.59x            | 99.76%              |
| reduce4 | 215.09            | 229.49             | 229.63            | 1.07x            | 99.94%              |
| reduce5 | 226.82            | 229.40             | 229.98            | 1.01x            | 99.75%              |
| reduce6 | 220.72            | 220.72             | 220.72            | 1.00x            | 100.00%             |

Table 4.8: Pascal, block-level coarsening.

## 4.5 Performance for varying problem sizes

The coarsening factor prediction model is designed to be problem-size aware. This means that coarsening will only be applied if a program exhibits sufficient software parallelism, i.e. launches a sufficient number of threads. Recall from Equation 4.1, that the current implementation restricts the maximum selected coarsening factor depending on the number of threads and thread blocks launched. For the purposes of this thesis, we define the *problem size* to be the total number of threads launched for execution. This number is commonly, although not in all cases, linked to the amount of data being processed, which we do not consider as relevant for the purposes of applying thread coarsening.

In Section 4.1 we reflected on the link between problem sizes and coarsening factor: small problem sizes may consist fewer threads than can be simultaneously processed, while applying coarsening can have the same effect of resulting in fewer threads than are concurrently executable. In each case, the calculated theoretical occupancy may be as high as 100%, although in practice entire SMs may be idle due to an insufficient number of threads to execute. In either case, applying coarsening irrespective of the problem size may clearly be detrimental.

One limitation of the ‘occupancy’ measure is, therefore, that it applies only within one SM, yet not across the device so as to measure a combined occupancy of all SMs. For a similar concept to such ‘device occupancy’, we use the notion of *waves of computation*. One wave of computation is defined as the exact number of threads that can be processed simultaneously across all SMs of a given device and for a given kernel. We use this notion in our considerations of problem sizes, as it allows us to reason about the specified problem size in relation to the degree of parallelism available across the device.

To illustrate this, consider the Pascal architecture used throughout our experiments: Each of the 20 SMs can hold up to 2048 active threads concurrently, totalling 40k active threads across the device. A kernel launch consisting of 40k threads at full occupancy therefore yields one full wave of computation. Similarly, a kernel launching 20k threads at full occupancy yields only half a wave of computation, while a kernel launching 120k threads at 75% occupancy yields four full waves of computation. The number of waves of computations yielded by a given kernel launched under the pre-determined occupancy acts as a constraint on the coarsening factor: for a kernel launching  $n$  waves of computation, the maximum coarsening factor  $C$  selected by the model is  $n$ .

In order to investigate the relationship between problem size and selected coarsening factor, we have conducted experiments using the reduction benchmarks with different problem sizes. To reduce any potential interference, problem sizes are expressed in terms of waves of computation. The smallest problem size consists of half a wave of computation, meaning 20k floating point numbers. The problem size is doubled in each series of experiments; we do not consider the handling of oddly-shaped problem sizes at this stage, but instead leave this for future work. Experiments were repeated for different coarsening factors across nine different problem sizes. Each experiment was executed ten times, and we report the averaged speedup. Having previously explored the cross-platform capabilities of our system, we choose to conduct experiments on the Pascal architecture for block-level coarsening.

Figure 4.2 shows the speedup achieved by the `reduce1` kernel for different coarsening factors and problem sizes. Model predictions are marked with an ‘x’ on each performance curve. The coarsening factor selected by our model for a half wave and one wave is  $C = 1$  in both cases. For larger problem sizes, the selected coarsening factor increases as the number of waves of computation is increased, up to a maximum of 8. From this point onwards, the coarsening factor does not increase further, being constrained by shared memory resources, as explored in the previous section.

The graph shows that with fewer than eight waves of computation, the optimal coarsening

factor is not 8. The curves also show the effect that a static selection of  $C = 8$  would have had if the mechanism were not to take the problem size into account. For two and four waves, selecting  $C = 8$  would have incurred a speedup, but not the optimal one chosen by the model. However, selecting  $C = 8$  in the case of a half wave or one wave would have incurred a 10% performance penalty. These two curves also highlight that in this benchmark our model under-predicted the selected coarsening factor, having chosen  $C = 1$  in both cases, while the optimal  $C$  is 2 and 4, respectively.

We have conducted further experiments for all of the remaining reduction benchmarks, and have observed the same patterns as exhibited by the `reduce1` benchmark. A number of characteristics stand out:

- For small problem sizes, the optimal coarsening factor is typically lower than the one to which it stabilises for larger problem sizes (we refer to the stabilised coarsening factor as  $C_S$ )
- The coarsening factor may stabilise at different points. In the experiment shown in Figure 4.2, the best performing coarsening factor stabilises to  $C_S = 8$  at the exact point when the number of waves of computation  $n$  is equal to 8, i.e.,  $C$  stabilises at  $n = C_S$ . However, we also observed that the coarsening factor may stabilise for both smaller and larger problem sizes, depending on the experiment.
- When selecting a coarsening factor less than  $C_S$ , our model tends to under-predict. For instance, this is the case at  $n = 1$  in Figure 4.2, where the selected coarsening factor is 1, while the best performing coarsening factor is 4. It is currently not clear whether this is characteristic of our model or of the reduction benchmarks chosen for our experiments.

The experiments shown consider problem sizes that are divisible by the product of the coarsening factor and the number of active threads per device. In this thesis, we consider coarsening factors that are powers of two for matching problem sizes. As a result, applying coarsening in our experiments is guaranteed to yield a whole number of waves of computation. To illustrate this, compare coarsening  $n = 2$  and  $n = 3$  waves of computation with a factor of  $C = 2$ . In this case, applying coarsening will yield  $n = 1$  and  $n = 1.5$  coarsened waves, respectively. The latencies of these are equivalent to  $n = 1$  and  $n = 2$  waves, respectively, as executing a half wave takes an approximately similar amount of time as executing a full wave. The coarsening factor  $C = 2$  affects a doubling of the workload of threads, which may therefore run twice as long. This causes the run-time to be equivalent to executing  $n = 2$  and  $n = 4$  waves of computation of the uncoarsened code, without taking into account any performance improvement gained by applying thread coarsening. In this example, coarsening must yield a speedup of at least 1.33x in order *not* to have a detrimental effect on the performance of coarsening  $n = 3$  waves of computation. However, this problem does not occur where  $C$  evenly divides  $n$ .

For ‘odd’ problem sizes, we therefore expect that choosing coarsening factors other than powers of two will be preferential, for instance, prime number coarsening factors. We investigated

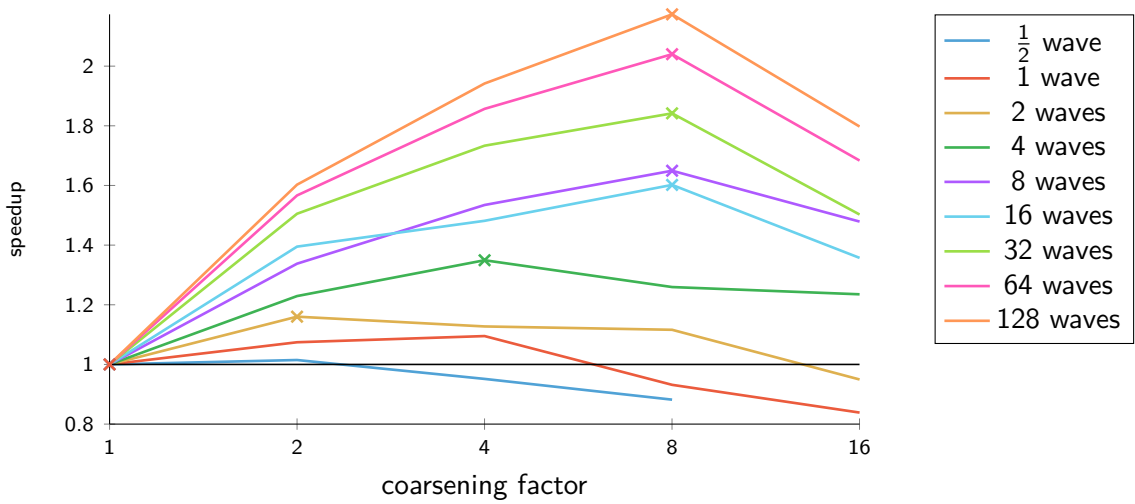


Figure 4.2: The reduction1 benchmark with different input sizes.

applying coarsening factors that are powers of two to non-matching problem sizes in informal experiments, in which we observed the described detrimental effects on performance. Importantly, these performance penalties are more significant for small problem sizes, but nearly disappear for large problem sizes. This is explained by the fact that waves of computation do not finish perfectly simultaneously on GPUs; this effect is increased for larger number of waves, such that ‘half waves’ introduced by unmatching coarsening factors have a diminished negative effect on performance for large problem sizes.

## 4.6 Discussion

The method presented here shows how occupancy can be used as a guiding principle for automated coarsening factor selection on NVidia GPUs, if a kernel’s resource requirements after coarsening are known such that they can be compared against the amount of resources available on the targeted platform. The experiments presented here seem to indicate that the coarsening factor selection model can be parameterised to support different generations of NVidia GPUs. Our evaluation has shown that our method applies both to unoptimised and optimised code alike, achieving speedups averaging 1.73x and 1.97x for thread-level and block-level coarsening, respectively. Finally, the experiments highlighted that the problem size as specified in the launch configuration constitutes an important aspect that needs to be considered in the process of selecting coarsening factors in an automated fashion. Compiling differently coarsened versions of the program is, therefore, not only useful for determining the occupancy of each version, but also provides a range of pre-compiled programs from which the most suitable one can be dynamically selected for execution once the problem size is known at run-time.



# Static analysis for identifying cache pressure 5

In the previous chapter we presented a method to select coarsening factors based on calculating the occupancy from explicitly managed resources. In this chapter, we consider cases that experience cache pressure, which is increased by thread coarsening (see Section 5.1). We present a light-weight cache line re-use analysis based on partial symbolic execution in order to identify such kernels and exclude them from coarsening. We integrate this analysis into the existing coarsening pipeline, and evaluate our system on a diverse set of benchmarks from the Rodinia Benchmark Suite [14].

## 5.1 Effects of coarsening on cache pressure

For some kernels coarsening can significantly increase pressure on the cache, as illustrated by the example in Figure 5.1. The implementation copies a matrix, using one thread to copy an entire row and each thread loops over adjacent elements belonging to the same cache line, before moving on to the next. This is non-optimal for a GPU, but it serves to illustrate the effect. The number of cache lines being accessed at any time is the same as the number of simultaneously active threads. If we coarsen the kernel then, in the worst case, the same total number of threads will be executing simultaneously as in the original kernel, but with each thread doing more work. In this worst case the number of cache lines being accessed at any time will thus scale with the coarsening factor and this will increase the pressure on the cache. In practice, since higher coarsening factors can yield lower occupancy and result in fewer active threads, the effect is not always linear. However, in the presence of cache line re-use the cache pressure frequently increases with the coarsening factor and that this typically outweighs the benefits that coarsening might otherwise bring. Interestingly, the issue here is cache line re-use, which is generally seen as a good thing for performance on conventional architectures, but which is a potential problem when thread coarsening for a GPU. Cache pressure is, however, not an issue when data is accessed in a streaming manner, where there is no data re-use or cache line re-use.

The problem is not just restricted to GPU kernels containing loops, but can also happen in simultaneously executing instances of a single program statement. As an example, Figure 5.2 illustrates a matrix transpose kernel based on the NVidia OpenCL SDK. Here each thread reads a single element from a source array at a given position  $(x, y)$  and writes it to a target array at its transposed index  $(y, x)$  (c.f. [55]). Cache lines are read in a coalesced fashion by threads of

```

1  kernel void memcpy(global float* input,
2                      global float* output,
3                      int size) {
4
5      size_t globalId = get_global_id(0);
6
7      for (unsigned int index = 0; index < size; ++index) {
8          output[globalId * size + index] = input[globalId * size + index];
9      }
10 }
11
12 kernel void memcpy_coarsened(global float* input,
13                              global float* output,
14                              int size) {
15
16     size_t globalId0 = 2 * get_group_id(0) * get_local_size() + get_local_id();
17     size_t globalId1 = (2 * get_group_id(0) + 1) * get_local_size() +
18         ↪ get_local_id();
19
20     for (unsigned int index = 0; index < size; ++index) {
21         output[globalId0 * size + index] = input[globalId0 * size + index];
22         output[globalId1 * size + index] = input[globalId1 * size + index];
23     }
24 }

```

Figure 5.1: The memcpy kernel before and after coarsening.

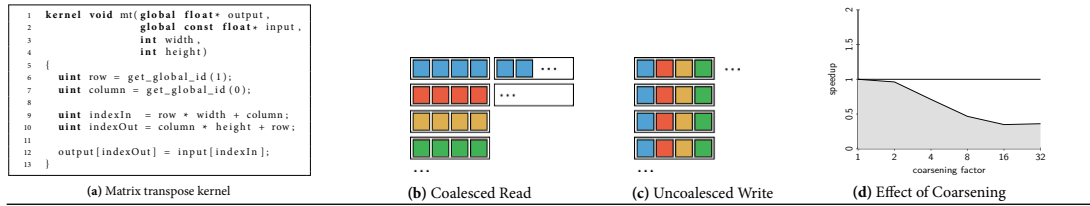


Figure 5.2: Performance effects of coarsening on the matrix transpose kernel.

single warps (indicated by colour), but the data is subsequently written to differing cache lines in an uncoalesced way. Data that is cached as a result of a read will not be accessed again for the remainder of the program. However, writes to a cache line originate from multiple warps, so the line should ideally not be evicted until it has been written to by each of those warps.

How, then, does coarsening affect the behaviour of this kernel with regard to cache pressure? The answer depends on how individual threads are scheduled on the SM and how outstanding writes are handled by the memory system. If, for example, we coarsen by a factor of two then in the worst case twice as many read and write instructions may be in flight at any time, as there will be two instances of the code on line 12 of Figure 5.2. The reasoning is similar to the above case. With more write instructions in flight the average time taken to complete the writes to a given cache line will increase<sup>1</sup> and this will increase pressure on the cache.

In practice, however, many kernels do not re-use cache lines in this way. For instance, if the above example were rewritten to exploit coalesced memory accesses, data from an entire cache line will be read or written in a single memory access and the data held in the cache line evicted.

<sup>1</sup>For a machine in the steady state this would be a consequence of Little's Law

As for the matrix transpose kernel shown in Figure 5.2, however, it is clear that any coarsening factor chosen either manually or by our model presented in the previous chapter would necessarily lead to a slow-down in performance. In this particular case, no coarsening factor exists that would yield an improvement in performance. It is therefore necessary that the kernel would be excluded from coarsening altogether. In general, kernels which re-use cache lines, thereby requiring these cache lines to remain in cache for a period of time without an eviction taking place, are typically unsafe to coarsen. Our goal in this chapter is to establish a method to identify these kernels analytically, and adapt our previous model (defined in Section 4.2) such as to exclude them from coarsening in an automated way.

## 5.2 Re-use distance analysis

The problem of determining whether a kernel performs cache line re-use in the above sense can in theory be solved by applying re-use distance analysis, which is a well-known technique for analysing cache locality [101, 24]. Knowing the exact distance of a re-use makes it possible to predict whether a given memory access is likely to be served from cache, or is likely to trigger a cache miss. The former scenario poses a potential performance risk when applying thread coarsening, as coarsening effectively increases re-use distances, which in turn accounts for increased cache miss rates.

Re-use distance analysis requires an ordered memory access trace, making it not directly suitable for GPUs due to the unknown warp schedule. In practice, this can be circumvented by making assumptions on the order in which warps are scheduled [59]. The approach requires that a substantial number of warps be analysed in order to get an accurate prediction of the distance. One problem here is that the applied thread block dimensions are not actually known until run-time. The more significant problem, however, is that the approach is typically expensive in both memory footprint and run-time. For instance, Nugteren et al. [59] report a memory footprint of typically around 2 GB with an execution time of around 10 seconds run-time (plus 4 minutes additional time for emulation in Ocelot [23]) for the `cutcp` benchmark.

We require an analysis which is able to execute at program run-time (i.e. kernel compile-time). This is in order to integrate with our coarsening pipeline (see Section 5.7), which is implemented as part of the OpenCL run-time API, and can thus, for instance, analyse kernels which were generated at run-time.

## 5.3 Polyhedral analysis

Polyhedral analysis [6] appears to be a suitable alternative to re-use distance analysis, offering precise analytics of affine loop nests (also see Section 2.6.9). However, the requirement for loop nests to be affine would exclude a large number of programs with non-affine loop structures from coarsening, for instance, the reduction kernels considered in the previous chapter (see Figure 3.4, and Section 4.4). This ruled out the use of polyhedral code analysis at an early stage.

In addition, tools such as Polly [35] were still undergoing active development and presented us with engineering challenges concerning, for instance, supported data types.

## 5.4 Approximate cache line re-use analysis

Because we are only interested in *whether* a kernel re-uses cache lines, we do not require a full re-use distance analysis. Instead, we analyse solely for re-use, regardless of the distance.

To do this, we implement an LLVM pass that performs partial symbolic execution of a given kernel, with a view to computing the access locations associated with pointer and array accesses to global memory. These are the only accesses handled by the cache hierarchy. Kernel implementations typically feature separate sets of instructions dealing with memory locations on the one hand, and data stored at these locations on the other. Purely data-related instructions are disregarded by our analysis, as they do not affect the outcome. Location-related instructions are based on integer arithmetic, and are therefore frequently computationally cheaper than data-related instructions, which often involve more expensive floating point arithmetic. The first stage of the analysis performs a pass over the LLVM IR to identify the set of instructions associated with calculating access locations of global memory accesses. A separate pass is then performed over these instructions which implements the rules described below.

Across our target platforms thread blocks may assume maximum dimensions of up to 1024 threads in a space of 1024×1024×64 threads. However, our symbolic analysis assumes a single thread block of 32, 32×2, or 32×2×2 threads for 1/2/3-dimensional kernels, respectively, as the exact launch configuration is not known when the analysis is run. As a result of that, the memory footprint and run-time of the analysis is thus effectively kept low. (Note, that the low number of simulated threads suffices to analyse for a simple yes/no answer, whereas more threads would be needed to obtain an accurate distance measure.)

The idea is to analyse accesses to global memory by constructing Memory Access Descriptors (MADs) that determine at which offset each thread would access a given pointer or array variable. MADs are implemented as matrices of up to three dimensions, with each entry in the matrix corresponding to one particular thread of a single simulated thread block. MADs assume the dimensions encountered in the corresponding arithmetic expressions. As an example, the matrix transpose kernel in Figure 5.2 will have two-dimensional MADs, while one dimension is sufficient to capture accesses in the memcpy kernel of Figure 5.1.

MADs are constructed by performing a pass over the previously identified location-related instructions described above. When processing the LLVM IR the following cases apply (the processing rules stated here are illustrated by examples given in Section 5.5):

- **OpenCL functions:** For calls to OpenCL built-in functions (such as `get_global_id()` or `get_local_size()`, etc.), we construct a new MAD containing sample IDs. To do this, we assign consecutive global IDs starting with 0. Corresponding values for all other OpenCL functions can be derived and set accordingly, also with respect to thread block

dimensions described above. Recall, that intention for the analysis of a single thread block is to yield insight into the memory access patterns of any thread block, whichever IDs we choose.

- **Variables, Function Parameters:** Variables whose values are unknown are substituted by arbitrarily-chosen sample values, making sure that different values are used for different variables. This also applies to function parameters. Naturally, for any variable with an existing definition the predefined value will be used.
- **Arithmetic operations:** The arithmetic operations considered correspond to array index calculations, e.g.  $A[i * n]$ . In this case we simulate the corresponding operation, retrieving the existing MADs for the operands, e.g.  $i$  and  $n$  in the example. The resulting MAD has the combined dimensions of the operand MADs, to correctly represent the captured expression (see above).
- **Conditionals:** For conditionals, both branches are evaluated separately from each other. Thus, if two branches construct different versions of the same MAD, all versions are retained and used separately for analysing memory accesses. MADs that are jointly constructed by different branches in the case of thread divergence are not considered in the current version of the analysis. The benefit is that all branches are analysed regardless of their entry condition. The disadvantage is that cache line re-use which occurs *only* in the presence of thread divergence and does not occur otherwise, is currently not detected. However, this appears to constitute a relatively uncommon scenario – it is not encountered in any of our benchmarks, for example.
- **Loops:** For loops, the goal is to detect cache line re-use both within the loop body and between iterations. We found in our experiments that this can be captured surprisingly accurately for a large number of loops by analysing only two adjacent iterations, i.e., the first two – a number which is naturally compounded for nested loops. If, for instance, the loop nest contains a conditional expression causing it to apply cache line re-use only in later iterations, this will be detected by the fact that the analysis considers all branches. Our analysis will not detect cache line re-use if the re-use occurs in non-adjacent iterations of a loop (in practice, such a scenario of cache line re-use increases the re-use distance and hence the likelihood that a cache eviction will have occurred, rendering the correctness of the result of the analysis less important). We do not require loops to have an affine iteration space (c.f. [6]), but support a variety of loop structures, so long as there is an induction variable.
- **Memory Accesses:** When a memory access is encountered, the existing MADs are used to test for potential cache line re-use. If multiple versions of the MAD exist (e.g. after a conditional), these are tested independently of each other. Each MAD is processed in chunks equal to the warp size (i.e. 32 threads). For each warp, the address offsets are

translated into cache line offsets, taking cache line width and alignment of the memory access into consideration. Duplicates within a warp are treated as a single coalesced access. A memory access is considered to *not* be performing cache line re-use, if:

- The cache line offsets accessed by any warp have not been accessed by a previously simulated warp within the same memory access instruction.
- The cache line offsets accessed by any warp have not been accessed in a previous memory access instruction to the same symbol. (The assumption is that pointer arguments to a given kernel are not aliases of one another<sup>2</sup> ; this is typically the case in OpenCL kernels, although the property is rarely declared explicitly.)
- All warps access the same cache line within a single instruction (this case is ignored due to its small impact).
- The memory access is fully coalesced and is a store operation (this case would not trigger a cache miss).

Note, that the analysis can simply ‘opt out’ if it encounters any unsupported feature or loop structure. In this case, coarsening will not be applied to the kernel.

Data-dependent memory accesses (typically of the form  $A[B[i]]$ ) constitute a special case that cannot be handled by this analysis, as they require either full symbolic execution or a dynamic approach based on actual program execution.

The analysis will terminate as soon as any cache line re-use or data dependency is detected. If the analysis completes without detecting either of these scenarios, the analysed kernel is considered to be safe to coarsen.

The analysis is *symbolic* in nature as it considers sets of MADs rather than specific MADs. In doing so, despite selecting sample values for OpenCL built-in functions, the objective is to validate the analysis for all possible execution paths, rather than simulate one specific execution path. Recall, that all branches of conditionals are evaluated. This effectively renders the chosen thread block ID irrelevant for the purposes of the analysis. That said, we consider it possible for our heuristic approach to detect false negatives in the case of potentially complex arithmetic introducing differences in cache line re-use behaviour between different thread blocks – but such a scenario seems largely theoretical, and we have also not encountered this in practice. Consequently, in practice the analysis can be expected to yield the same result independent of the chosen block ID for the vast majority of cases. Therefore, we consider it safe to use sample values in place of unresolved symbols for improved performance, yet to the same effect.

---

<sup>2</sup>It may be tempting to annotate function parameters with the `NoAlias` attribute within LLVM, equivalent to the `__restrict__` keyword in C99 and CUDA. However, this turns out to be a fallacy, as it causes the compiler to combine memory accesses, now being able to reason that it is safe to do so. This would lead to discrepancies with the code generated by the OpenCL compiler, resulting in the detection of false negatives. It is therefore important that the no-alias assumption is not propagated within LLVM, not even in the branch to the cache line re-use analysis.

## 5.5 Memory Access Descriptors for cache line re-use analysis

To explore in more detail how Memory Access Descriptors (MADs) are used to analyse for cache line re-use within a kernel body, consider the following examples. The first example illustrates the analysis of a simple, one-dimensional mapping function (Section 5.5.1). Further examples give insight into two-dimensional MADs (Section 5.5.2), as well as the handling of conditionals (Section 5.5.3) and loops (Section 5.5.4).

### 5.5.1 Example 1: A simple mapping function

To illustrate, consider the following example, which maps a  $2n + 1$  function over an array of values as an in-place operation:

```
1 __kernel void ex0(__global float* data) {
2     int index = get_group_id(0) * get_local_size(0) + get_local_id(0);
3     data[index] = 2 * data[index] + 1;
4 }
```

The kernel body compiles to the following LLVM IR:

```
1 Basic block [entry]:
2 >> %call1 = call i32 @get_group_id(i32 0) #3
3 >> %call11 = call i32 @get_local_size(i32 0) #3
4 >> %mul = mul i32 %call1, %call11
5 >> %call12 = call i32 @get_local_id(i32 0) #3
6 >> %add = add i32 %mul, %call12
7 >> %arrayidx = getelementptr inbounds float @data, i32 %add
8     %tmp = load float @data, i32 %arrayidx, align 4
9     %tmp1 = call float @llvm.fmuladd.f32(float 2.000000e+00, float %tmp,
10         ↪ float 1.000000e+00)
11     store float %tmp1, float @data, i32 %arrayidx, align 4
12     ret void
```

The first step of the analysis is to identify accesses to cached memory, which are the load and store instructions in lines 8 and 10, respectively. Starting from these, the analysis recursively identifies all definitions used by these instructions to determine access locations. This is done by recursively traversing the definition-use chains of the operators. The identified instructions are marked as location-relevant, as highlighted in the IR code above. These marked instructions will be used for calculating Memory Access Descriptors (MADs) in the next step. Note, that all other instructions, such as the floating point multiply-add (line 8), are not necessary for the purposes of this analysis, and will not be considered in the simulation stage. One important benefit of basing the analysis on *partial* symbolic execution is that set of instructions responsible for calculating access locations are integer based, while the set of instructions related to data processing, which is ignored by the analysis, is in practice frequently more computationally expensive than the former, as it may comprise more complex floating point instructions.

At the end of the pre-processing stage, the analysis has identified a subset of the instructions responsible for determining access locations for all accesses to cached memory regions (i.e. global memory). Accesses to un-cached memory (i.e., shared memory) can be ignored

for the purposes of this analysis. The load and store operations themselves are not marked, as they are not used for calculating access locations – if they were, this would imply that the kernel is *data-dependent*; in this case, the analysis would terminate here, as data-dependent kernels cannot be analysed for cache line re-use by static analysis.

In the next step, the analysis steps through the IR code line by line to construct Memory Access Descriptors (MADs), which are subsequently used to analyse memory accesses as they are encountered in the program flow. MADs are scalars, vectors, or matrices that capture expressions describing memory access locations of threads belonging to one thread block. The MADs for the LLVM IR shown above are:

$$\begin{aligned}
 \%call &= (0) \\
 \%call1 &= (1) \\
 \%mul &= \%call * \%call1 = (0) \\
 \%call2 &= (0 \quad 1 \quad \dots \quad 31) \\
 \%add &= \%mul + \%call2 = (0 \quad 1 \quad \dots \quad 31) \\
 \%arrayidx &= \%add = (0 \quad 1 \quad \dots \quad 31)
 \end{aligned}$$

The MADs are explained as follows:

- The `%call` instruction (line 2 of the LLVM IR) invokes the OpenCL built-in function `get_group_id(0)`. The analysis uses pre-defined values to model built-in functions. In this case, the invoked `get_group_id()` function returns the ID of the work group to be simulated, for which the analysis returns a value of zero. Because the returned value is identical for all simulated threads, the MAD class will internally represent this as a scalar.
- Likewise, the MAD for `%call1` captures the invocation of the `get_local_size()` function, and the size of the simulated work-group is set to 32 threads; the representation is as before.
- The `%mul` instruction multiplies these two scalars, which yields a new MAD containing 0.
- The `%call2` instruction holds the IDs of all simulated threads, namely values from 0 to 31. Since each thread has a different ID, the values are represented in a vector.
- The `%add` instruction describes the addition of a scalar-based MAD of value 0 to an array-based MAD. Note, that while an MAD may be represented as a scalar, it nevertheless represents one value per thread. However, as these values do not diverge within the simulated thread block, but instead are all zero, this can be represented in a scalar. The implementation of the addition operation trivially follows.

- Finally, the `%arrayidx` MAD represents the `getelementptr` instruction (line 7), sets up the pointer which will be dereferenced in the load and store instructions, outlining the accessed pointer variable as well as the MAD used in the access, as computed in `%add`. The values are therefore equivalent.

The goal of the analysis to detect intra-block cache line re-use, which should be done independently of the block ID. To this end, the analysis simulates a single thread block, which always assumes ID 0. The intuition is that by analysing one block, it is possible to detect cache line re-use that could occur in any block. This intuition can – in theory – be refuted by conditional branching that checks the block ID, and we will re-visit this specific case when discussing conditionals (see Section 5.5.3).

For one-dimensional kernels, we statically select a block size of the 32 for the symbolically executed thread block and assign thread IDs starting from zero. This corresponds to one full warp of threads. The launch configuration is not known by the time the analysis is run, which is at compile-time of the kernel. However, the analysis does not require to know the number of threads specified at kernel launch, as instead a small number of symbolically executed threads suffices (also see Sections 5.2 and 5.4). This avoids repeated execution of the analysis at run-time. This configuration is able to accurately analyse all one-dimensional kernels considered throughout this thesis (see Section 5.8). The number of simulated threads has not undergone rigorous optimisation, and could in practice be set much lower in many cases, yet it is also possible to construct cases where more warps are required for an accurate analysis. Note, that the number of symbolically executed threads differs for multi-dimensional kernels, as discussed in Section 5.5.2.

After the MADs shown above have been constructed in order of the program flow (lines 1-7 of the LLVM IR), the analysis encounters a load operation in line 8. The *align* field of the `getelementptr` instruction specifies that fields in the accessed pointer structure have a width of 4 byte. It also specifies the offsets at which the pointer will be accessed, which in this case is based on the MAD of `%add`, i.e., offsets 0 to 31 inclusive. The analysis uses the specified index offsets and alignment to calculate the offsets of the accessed underlying bytes, as well as the offsets of cache lines holding the accessed bytes. For example, a (single-threaded) 32-bit access to a pointer variable at index 1 (the ‘offset’) would require loading bytes [4, 5, 6, 7] (the ‘byte offsets’), which are held by cache line 0 (the ‘cache line offset’). Note, that on GPU architectures each thread may access memory at its own offset. For illustrative purposes, we assume a cache line width of 32 bytes – although in practice using an accurate cache line width is rarely important for gaining accurate results. The accessed offsets for the `%arrayidx` MAD are as follows:

$$\begin{aligned}
\text{offsets} &= (0 \quad 1 \quad 2 \quad \dots \quad 31) \\
\text{byteOffsets} &= (0 \quad 4 \quad 8 \quad \dots \quad 124) \\
\text{cacheLineOffsets} &= (0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 32 \quad 32 \quad \dots \quad 96 \quad 96) \\
\text{uniqueCacheLineOffsets} &= (0 \quad 32 \quad 64 \quad 96)
\end{aligned}$$

Since accesses within a warp are coalesced, the access will cover a total of four unique cache lines at the offsets shown above. The unique cache line offsets are used to detect cache line re-use. The analysis checks whether the %data pointer (line 7 of the LLVM IR) has previously been accessed at these offsets, but as this is the first access to this variable, no conflict was found and the analysis resumes.

The next instruction encountered in order of the program flow is the multiply-add instruction of line 9. However, this instruction is discarded by the analysis, as it is not required for the analysis of access locations.

Finally, the store instruction (line 10) is analysed. Since the same MAD is used for accessing the pointer variable as in the previous load instruction, the offsets, byte offsets, cache line offsets, and unique cache line offsets are the same as those used in the load instruction shown above. Stores on NVidia GPUs have a unique characteristic which is of crucial for determining cache line re-use. For a store operation in a conventional CPU architecture, the underlying cache line is loaded into cache, the value to be written is replaced in the cache line, and the cache line is written back to main memory. However, in a fully coalesced store all values of a cache line are over-written. There is, therefore, no technical reason for the hardware to trigger a cache miss and load the line to be modified into cache, since no value in the cache line will be retained. Instead, the entire cache line can be written at once. A store operation, therefore, does not in all situations re-use a previously stored cache line, and cache line re-use depends on whether the store is a fully coalesced access.

This means that if the store is not fully coalesced, the analysis at this point will detect cache line re-use and terminate. However, from analysing offset vector it is established that the store is fully coalesced, and that therefore the memory access in line 10 does not constitute cache line re-use. The accessed cache lines are nevertheless stored in order to detect future cache line re-use. However, since there are no further instructions to be analysed, the analysis terminates, stating that no cache line re-use has been detected.

### 5.5.2 Example 2: Two-Dimensional Matrix Transpose

Consider the following example of a multi-dimensional kernel for a matrix transpose, which was also discussed above in Figure 5.2:

```

1  __kernel void ex2(__global float* output,
2                      __global const float* input,
3                      int width,
4                      int height) {
5      unsigned int row = get_global_id(1);
6      unsigned int column = get_global_id(0);
7      unsigned int indexIn = row * width + column;
8      unsigned int indexOut = column * height + row;
9      output[indexOut] = input[indexIn];
10 }

```

This is compiled into LLVM IR:

```

1  Basic block [entry]:
2  >> %call = call i32 @get_global_id(i32 1) #2
3  >> %call1 = call i32 @get_global_id(i32 0) #2
4  >> %mul = mul i32 %call, %width
5  >> %add = add i32 %mul, %call1
6  >> %mul2 = mul i32 %call1, %height
7  >> %add3 = add i32 %mul2, %call
8  >> %arrayidx = getelementptr inbounds float @rspace(1)* %input, i32 %add
9      %tmp = load float @rspace(1)* %arrayidx, align 4
10 >> %arrayidx4 = getelementptr inbounds float @rspace(1)* %output, i32 %add3
11     store float %tmp, float @rspace(1)* %arrayidx4, align 4
12     ret void

```

As before, the analysis identifies two memory accesses in lines 9 and 11. By traversing the use-define chain the set of instructions responsible for calculating the memory access locations are identified, as highlighted above. For this set of instructions, the above code is then stepped through line by line, during which the the following MADs are constructed:

$$\begin{aligned}
\%call &= \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\
\%call1 &= \begin{pmatrix} 0 & 1 & \dots & 31 \end{pmatrix} \\
\%mul &= \begin{pmatrix} 0 \\ 100 \end{pmatrix} \\
\%add &= \begin{pmatrix} 0 & 1 & 2 & \dots & 31 \\ 100 & 101 & 102 & \dots & 131 \end{pmatrix} \\
\%mul2 &= \begin{pmatrix} 0 & 200 & 400 & \dots & 6200 \end{pmatrix} \\
\%add3 &= \begin{pmatrix} 0 & 200 & 400 & \dots & 6200 \\ 1 & 201 & 401 & \dots & 6201 \end{pmatrix}
\end{aligned}$$

The MAD implementation internally supports parametric unary and binary operations, as well as automatically expanding dimensions as required, such that various MADs can be constructed with relative ease when processing the IR code. In this example, the invocation of `get_global_id(1)` (line 2) yields a vector in the second dimension, as indicated by the argument. Note, that in practice each GPU thread holds its own value, although the implementation

of MADs abbreviates this if values do not diverge. This is taken into consideration in the `%add` and `%add3` instructions, each of which combine two partially represented MADs of different dimensions. The analysis symbolically executes a total of  $32 \times 2$  threads comprising two full warps for two-dimensional kernels. This is indicated by the MADs comprising the return values from the two calls to `get_global_id()` in lines 2 and 3.

The example shown has two function parameters, `width` and `height`, which are not known at the time the analysis is run. The approach here is to generate sample values that differ for each encountered variable. The intuition is that will be a useful approximation in the majority of cases, such as in this. However, it may in theory be possible to conceive of kernels in which the choice of generated values could influence the outcome of the analysis. However, in practice we have not encountered such kernels. Identifying such scenarios will be left for future work. In this example, the values generated are `%width = 100` and `%height = 200`.

The MAD of `%add` is used for the analysis of the `load` instruction in line 9. Note, that it is equal to `%arrayidx`, which we have left this out for brevity. The memory access is analysed for 64 threads, equivalent to two warps of 32 threads each. Accesses within a warp may be coalesced, but across warps coalescing is not possible, as warps are scheduled and executed independently of each other. Each row of the `%add` and the `%add3` MAD corresponds to a full warp. Both MADs are scaled by the specified alignment of 4, and the method described above (Section 5.5.1) is applied to determine the uniquely accessed cache line offsets:

$$\begin{aligned} \text{uniqueCacheLineOffsets}(\%add) &= \begin{pmatrix} 0 & 32 & 64 & 96 \\ 100 & 132 & 164 & 196 \end{pmatrix} \\ \text{uniqueCacheLineOffsets}(\%add3) &= \begin{pmatrix} 0 & 800 & 1600 & 2400 & \dots & 18800 \\ 0 & 800 & 1600 & 2400 & \dots & 18800 \end{pmatrix} \end{aligned}$$

The offsets shown here for the `%add` MAD are used to analyse the `load` instruction of line 9. Each row contains the unique cache line offsets for a single warp. The first warp will access the four cache lines shown in the first row, while the second warp will access four different cache lines, shown in the second row. In this case, no cache line re-use is detected.

The analysis of the `store` instruction (line 11) uses the unique cache line offsets determined for the MAD of `%add3`. Analysing these offsets reveals that the first warp accesses 32 unique cache lines, while the second warp accesses the same unique cache lines (also see Figure 5.2(c)). This example, therefore, highlights how cache line re-use on a GPU can happen within a single memory access instruction due to the warp-based schedule. At this point, the analysis has detected cache line re-use and will abort.

### 5.5.3 Example 3: Conditionals

Conditionals present a number of challenges to MAD-based analysis that require fundamental design decisions to be made. For instance, the cache line re-use behaviour of a kernel can differ between blocks executing the same kernel because of a conditional, as discussed earlier

(Section 5.5.1). Likewise, different iterations of a loop can behave differently in terms of cache line re-use based on a conditional inside a loop, as will be discussed later (Section 5.5.4). In both cases it is desirable to choose an approximation in order to retain the cost of partial symbolic execution at a minimum.

The analysis, therefore, evaluates both branches separately, regardless of the condition. In some scenarios an accurate evaluation of the branching condition is not possible at the time the analysis is run, for instance if the condition depends on function arguments, or in the case of data-dependent control flow.

To illustrate the handling of conditionals by a simple example, the previous kernel is extended with a flag to indicate whether a given matrix should be copied or transposed.

```

1  __kernel void ex3(__global float* output ,
2                      __global const float* input ,
3                      int width ,
4                      int height ,
5                      int transpose) {
6      unsigned int row = get_global_id(1);
7      unsigned int column = get_global_id(0);
8      unsigned int indexIn = row * width + column;
9      unsigned int indexOut;
10     if (transpose) {
11         indexOut = column * height + row;
12     } else {
13         indexOut = indexIn;
14     }
15     output[indexOut] = input[indexIn];
16 }

```

The corresponding LLVM IR consists of three basic blocks, due to the added control flow:

```

1  Basic block [entry]:
2  >> %call = call i32 @get_global_id(i32 1) #2
3  >> %call1 = call i32 @get_global_id(i32 0) #2
4  >> %mul = mul i32 %call, %width
5  >> %add = add i32 %mul, %call1
6      %tobool = icmp ne i32 %transpose, 0
7      %tmp = xor i1 %tobool, true
8      % = select i1 %tmp, i1 false, i1 true
9      br i1 %, label %if.then, label %if.end
10 Basic block [if.then]:
11 >> %mul2 = mul i32 %call1, %height
12 >> %add3 = add i32 %mul2, %call
13     br label %if.end
14 Basic block [if.end]:
15 >> %indexOut.0 = phi i32 [ %add, %entry ], [ %add3, %if.then ]
16 >> %arrayidx = getelementptr inbounds float @rspace(1)* %input, i32 %add
17 %tmp3 = load float @rspace(1)* %arrayidx, align 4
18 >> %arrayidx4 = getelementptr inbounds float @rspace(1)* %output, i32
    ↳ %indexOut.0
19     store float %tmp3, float @rspace(1)* %arrayidx4, align 4
20     ret void

```

As before, the analysis identifies a set of instructions relevant to calculating access locations,

as highlighted above. The branching condition is not considered, as discussed above. The branch is immediately followed by a  $\phi$  instruction (line 15), a special form of SSA semantic, which allows a variable to assume one of several values. The return value of the  $\phi$  instruction depends on where the control flow originated from, i.e., which basic block was last visited and, therefore, which branch was executed. The  $\phi$  instruction can assume one of two values, %add or %add3, which are calculated as described above in Sections 5.5.1 and 5.5.2. By the time the partial symbolic execution reaches line 15, both values that can be assumed by the  $\phi$  instruction are present, such that:

$$\%index.Out0 = \left\{ \begin{pmatrix} 0 & 1 & 2 & \dots & 31 \\ 100 & 101 & 102 & \dots & 131 \end{pmatrix}, \begin{pmatrix} 0 & 200 & 400 & \dots & 6200 \\ 1 & 201 & 401 & \dots & 6201 \end{pmatrix} \right\}$$

The analysis retains both possible values to be used for calculating downstream MADs, as well as for analysing memory accesses. The analysis of the load instruction in line 17 follows from Section 5.5.2.

When the store instruction in line 19 is reached, both partial MADs are used independently of each other to analyse the memory access. The first partial MAD is analysed warp by warp, the unique accessed cache lines are identified, and the analysis determines that there is no conflict between the warps. In addition, there is also no conflict with previous accesses, since no previous accesses have occurred at this stage.

After this, the memory access is analysed using the second partial MAD. Conflicts between partial MADs are not detected. Instead, the analysis seeks to prove the existence of a conflict either between warps of the same (partial) MAD, or of a conflict with a previous access. Since a conflict between warps can be proven for this partial MAD, the analysis will terminate having detected cache line re-use.

The approach taken to handle conditionals is subject to limitations. Evaluating branches separately of each other assumes that all threads execute the same branch at all times, regardless of thread divergence. This is clearly not the case in a large number of GPU programs, with thread divergence being a commonly encountered feature. However, it is important to note that this does not imply that the presented analysis is not a good approximation. The reason is that thread divergence, though commonly encountered, does not commonly influence whether or not a kernel performs cache line re-use. A kernel whose cache line re-use depends on thread divergence in a conditional can in theory be constructed, although we have not encountered any such kernels in practice. Therefore, the proposed handling of conditionals in the analysis appears a useful approximation for the vast majority of cases.

One practical limitation of our approach is that MADs can grow exponentially in case of a series of conditionals, as MADs can double in size each time an if-statement is traversed. A similar effect happens in the presence of nested loops, described in the next example. These

constitute the worst-case scenarios for this algorithm. Nevertheless, it is not idiomatic of realistic kernels to expose a number of conditionals sufficiently large to cause a noticeable impact on performance.

#### 5.5.4 Example 4: Loops

To illustrate the analysis of loop structures, consider the following example:

```

1  __kernel void ex4(__global float* input, __global float* output, int size) {
2      // memcpy - CLR
3      size_t globalId = get_global_id(0);
4      for (unsigned int index = 0; index < size; ++index) {
5          output[globalId * size + index] = input[globalId * size + index];
6      }
7  }
```

The kernel performs a two-dimensional memcpy, where each thread loops over one row of a two-dimensional array to create a copy of that row. Note, however, that the kernel itself is one-dimensional. The kernel is compiled to the following LLVM IR in SSA representation:

```

1  Basic block [entry]:
2  >> %call = call i32 @get_global_id(i32 0) #2
3      br label %for.cond
4  Basic block [for.cond]:
5  >> %index.0 = phi i32 [ 0, %entry ], [ %tmp1, %Flow ]
6      %cmp = icmp ult i32 %index.0, %size
7      br i1 %cmp, label %for.body, label %Flow
8  Basic block [for.body]:
9  >> %mul = mul i32 %call, %size
10 >> %add = add i32 %mul, %index.0
11 >> %arrayidx = getelementptr inbounds float @array, i32 %add
12      %tmp = load float @array, i32 %arrayidx, align 4
13 >> %arrayidx3 = getelementptr inbounds float @array, i32 %add
14      store float %tmp, float @array, i32 %arrayidx3, align 4
15 >> %inc = add i32 %index.0, 1
16      br label %Flow
17 Basic block [Flow]:
18 >> %tmp1 = phi i32 [ %inc, %for.body ], [ undef, %for.cond ]
19      %tmp2 = phi i1 [ false, %for.body ], [ true, %for.cond ]
20      br i1 %tmp2, label %for.end, label %for.cond
21 Basic block [for.end]:
22      ret void
```

As before, the pre-processing stage of the analysis identifies a set of instructions responsible for calculating access locations. As with conditionals, the loop condition is not evaluated, allowing for the analysis of loops with data-dependent entry conditions, as shown above.

In an SSA representation, loops carry  $\phi$  instructions. In case of this for-loop, the  $\phi$  instruction in line 5 acts as the induction variable, and is placed at the beginning of the loop. One unique property here is that besides the initialisation value of 0, the  $\phi$  instruction also has a forward definition %tmp1, whose value is unknown by the time the analysis first reaches line 5. The value of the forward definition is only known when the loop body has executed once.

With the known initialisation value the loop body is executed for a single iteration, yielding the following MADs:

$$\begin{aligned}
\%call &= (0 \quad 1 \quad \dots \quad 31) \\
\%index.0 &= (0) \\
\%mul &= \%call * \%size = (0 \quad 100 \quad 200 \quad \dots \quad 3100) \\
\%add &= \%mul + \%index.0 = (0 \quad 100 \quad 200 \quad \dots \quad 3100) \\
\%arrayidx &= \%add = (0 \quad 100 \quad 200 \quad \dots \quad 3100) \\
\%arrayidx3 &= \%add = (0 \quad 100 \quad 200 \quad \dots \quad 3100) \\
\%inc &= \%index.0 + 1 = (1) \\
\%tmp1 &= \%inc = (1)
\end{aligned}$$

By the time the analysis reaches the load instruction in line 12, all of the above MADs up to `%arrayidx3` have been computed. The memory access is analysed as before:

$$\begin{aligned}
offsets &= (0 \quad 100 \quad 200 \quad \dots \quad 3100) \\
byteOffsets &= (0 \quad 400 \quad 800 \quad \dots \quad 12400) \\
cacheLineOffsets &= (0 \quad 384 \quad 800 \quad \dots \quad 12384) \\
uniqueCacheLineOffsets &= (0 \quad 384 \quad 800 \quad \dots \quad 12384)
\end{aligned}$$

As this is the first access to the input pointer (line 11), no cache line re-use is detected. The unique cache line offsets are stored for analysing future accesses. Likewise, the output pointer (line 13) is accessed at the same unique cache line offsets. Again, no conflict is detected, and the accessed offsets are stored for this pointer as well. The accessed unique cache line offsets are stored separately for each pointer. Recall, that the assumption here is that pointers passed as function arguments do not alias each other (Section 5.4). Note, that for the same reason it also suffices to analyse offsets, rather than memory addresses.

The analysis symbolically executes one iteration of the loop, starting from the definition of the loop variable in line 5 up to the point of the forward definition, namely the  $\phi$  instruction in line 18. Afterwards, all assignments made during the processing of the loop are cleared, such that only `%call` remains as a known MAD, with other MADs having to be re-computed. The accessed memory offsets are, however, retained for subsequent analysis.

Internally, collections of MADs are stored in a stack-like structure. Before the first iteration of the loop, an empty layer is created on top of the stack in which all MADs encountered in the loop body including the loop variable will be stored. This layer is removed at the end, clearing

all definitions made in the loop body. For the second iteration of the loop, a new layer is created. The layer is initialised with a new definition of the loop variable `%index.0` which is initialised with the value of the `%tmp1` instruction of the first iteration. Note, that the `undef` part of `%tmp1` (line 18) is ignored by the analysis. At the end of the second iteration, the layer will again be removed, before the iteration continues. The second iteration of the loop is then analysed and the MADs are re-constructed (the definition of `%call` is repeated):

$$\begin{aligned}
\%call &= \begin{pmatrix} 0 & 1 & \dots & 31 \end{pmatrix} \\
\%index.0 &= \begin{pmatrix} 1 \end{pmatrix} \\
\%mul &= \%call * \%size = \begin{pmatrix} 0 & 100 & 200 & \dots & 3100 \end{pmatrix} \\
\%add &= \%mul + \%index.0 = \begin{pmatrix} 1 & 101 & 201 & \dots & 3101 \end{pmatrix} \\
\%arrayidx &= \%add = \begin{pmatrix} 1 & 101 & 201 & \dots & 3101 \end{pmatrix}
\end{aligned}$$

When in the second iteration of the loop the load instruction in line 12 is reached, the memory access is analysed as follows:

$$\begin{aligned}
offsets &= \begin{pmatrix} 1 & 101 & 201 & \dots & 3101 \end{pmatrix} \\
byteOffsets &= \begin{pmatrix} 4 & 404 & 804 & \dots & 12404 \end{pmatrix} \\
cacheLineOffsets &= \begin{pmatrix} 0 & 384 & 800 & \dots & 12384 \end{pmatrix} \\
uniqueCacheLineOffsets &= \begin{pmatrix} 0 & 384 & 800 & \dots & 12384 \end{pmatrix}
\end{aligned}$$

At this point, the analysis establishes that the unique cache line offsets of this load instruction have been accessed before, namely in the previous iteration of the loop. The analysis thus terminates, having detected cache line re-use.

In this case, the analysis was able to detect cache line re-use analysing only two iterations of a loop. The intuition underlying this heuristic approach to loop analysis is that analysing only two iterations of a loop is sufficient to detect cache line re-use in a large number of cases. This is furthermore aided by previous design decisions, in particular to not evaluate branching conditions. As a result, a loop cannot introduce cache line re-use in later iterations purely based on a conditional. It may, however, be possible to construct a loop which performs cache line re-use at a distance of more than one iteration. In practice, a re-use distance of more than one full loop iterations is reasonably likely to be sufficiently large such that a cache eviction will have occurred, making a non-detected cache line re-use the preferable outcome.

Because of the inductive approach to analysing loops, the analysis in its current form requires loops to have an induction variable, as well as a single back edge. This could potentially be re-

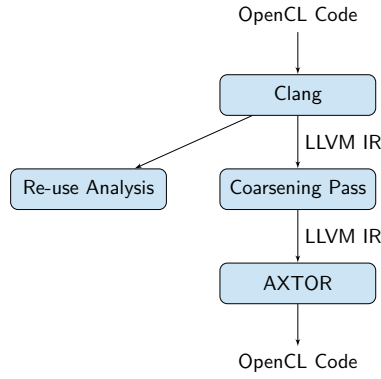


Figure 5.3: The coarsening pipeline extended by cache line re-use analysis.

laxed in future work. The specific application of the analysis in the context of thread coarsening legitimises opting out if unsupported features are encountered, such as unsupported loop structures, unimplemented instructions, or calls to functions that cannot be inlined. If the analysis decides opt out, the result is that coarsening will not be applied. In this fail-safe approach no negative consequences are incurred.

Nested loop structures are also supported by the mechanism shown here. In a loop nest, two iterations of each individual loop are analysed as before, thereby covering both the loop initialisation and the iterative step of each loop. In case of a single nested loop (i.e. an  $xy$ -loop), the inner loop is thus analysed a total of four times, namely twice in the first iteration of the outer loop, and twice in the second iteration. This effect compounds for deeply nested loops. The stack-like structure of MADs helps the memory footprint to remain low, while also simplifying recursive analysis of loops.

## 5.6 Comparison with re-use distance analysis

Compared with re-use distance analysis, the order of accesses is now unimportant making it unnecessary to consider specific warp schedules. The analysis can be executed with a small number of warps, as no ‘filler’ warps are required to accurately determine the distance between two accesses. Finally, no caching model, including knowledge of the hierarchy or replacement policy, is required. Instead, it suffices to remember which cache lines are requested.

However, the approach surrenders some of the accuracy captured in the re-use distance metric. For instance, if the re-use distance is sufficiently large, it can be reasoned with the help of a cache model, that the accessed data will have been evicted and is guaranteed to cause a miss. A model that is unaware of distance between accesses is not able to identify this. However, Wang and Xiao [96] claim that many kernels perform memory accesses either in a streamed manner (accessing data only once, rendering caching redundant), or typically exhibit short re-use distances. This could in practice be beneficial to our approach, as it lacks accuracy in particular for long re-use distances, where it is not able to predict that an eviction has taken place.

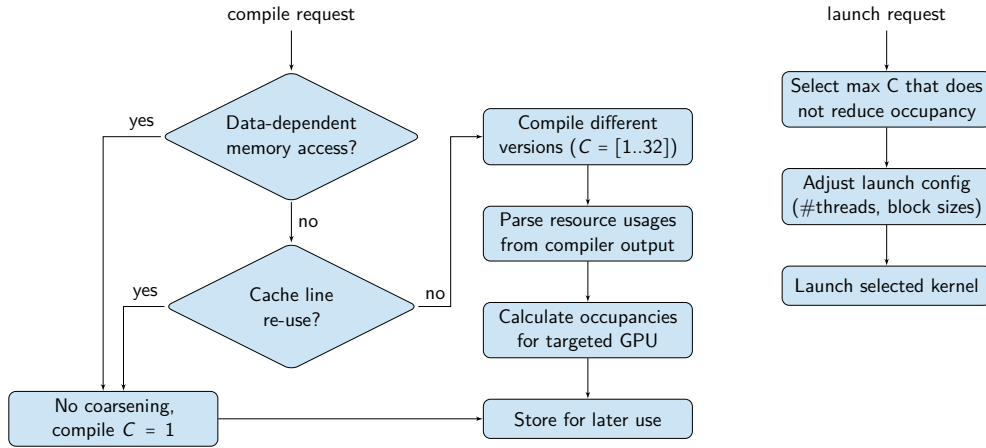


Figure 5.4: Coarsening flowchart

Evaluating our approach on 19 kernels from the Rodinia Benchmark Suite (Section 5.8), the maximum memory requirement was measured to be around 4KB. This is in addition to any memory requirements of the LLVM opt tool, which typically requires 50-100MB. The total execution times (incl. opt) ranged from 0.01 to 0.1 seconds, an improvement of several orders of magnitude both in terms of run-time and memory footprint compared to full re-use distance analysis [59] (c.f., 2 GB memory footprint with an execution time of around 10 seconds run-time plus 4 minutes additional time for emulation in Ocelot).

## 5.7 Implementation

We have extended our implementation presented in Section 4.3 by the cache line re-use analysis as shown in Figure 5.3. The analysis is built as an LLVM pass, implementing the algorithm described in Sections 5.4 and 5.5. The analysis acts as a filtering mechanism to exclude kernels from being coarsened. It is invoked to analyse the LLVM IR of the uncoarsened version of a given kernel. Applying coarsening to a kernel should not affect the results of the analysis, such that one invocation per kernel is sufficient to analyse for cache line re-use. If the analysis detects no cache line re-use, the pipeline behaves as described in the previous chapter, and will proceed to select a coarsening factor for execution based on the kernel’s predicted occupancy. If the analysis detects cache line re-use, however, the coarsening factor selected for execution will be 1, i.e., the uncoarsened code will be executed. At this point, compiling differently coarsened versions of the code is rendered unnecessary. This coarsening workflow is depicted in Figure 5.4.

## 5.8 Evaluation

With a method in place to exclude kernels from being coarsened which likely would experience a performance slow-down, we are now able to conduct a broader evaluation of automated thread coarsening as presented throughout this thesis up to this point. The objective of our evaluation is to answer the following questions about thread coarsening policy:

- How useful is occupancy as a guiding principle for selecting good coarsening factors?
- Does our framework ever reduce the performance of an uncoarsened kernel, i.e. is any kernel being coarsened when it should not?
- How does the performance of block-level coarsening compare with that of thread-level coarsening?

However, we are not primarily concerned with re-evaluating potential performance benefits of thread coarsening, which others have done before (see Section 2.6).

The Rodinia Benchmark Suite covers various problem domains, inspired by the Berkeley dwarfs of parallel computing [2]. We choose the Rodinia suite for our evaluation as it allows us to test both our coarsening factor selection model as well as our cache line re-use analysis on a wider variety of kernels from different problem domains. We chose these benchmarks over those assembled by Magni et al. [51], as our initial experiments seemed to indicate that the Rodinia suite would provide a better testing ground for the cache line re-use analysis, giving a balanced selection of test cases for either outcome of the analysis.

It should be noted, that the Rodinia OpenCL kernels do not implement any vendor-specific optimisation, as they are targeted at a variety of platforms, and should hence be viewed as unoptimised code. This is unlike the reduction kernels considered in the previous chapter, which are provided in hardware vendor’s SDK and implement different levels of optimisation as outlined above. Our experimental setup is as before, described in Table 4.1.

The benchmarks offer the functionality to freely specify the desired problem size and to generate corresponding test data sets. We select kernels from benchmarks that are easy to scale up in size and which were compilable using the existing coarsening/AXTOR pipeline, after various bug fixes and extensions. This left us with 19 kernels, which are summarised in Table 5.1.

Figure 5.5 plots the performance of these kernels for both thread-level and block-level coarsening on each architecture (see Table 4.1), averaged over ten executions, measuring a standard deviation of 0.57%. In each graph, the left axis (solid line) indicates the achieved speedup of the kernel and the right axis (dashed line) the theoretical occupancy of the kernel for each coarsening factor. Occupancies are expressed as a percentage of the peak (100%). The circles mark the coarsening factor that would be chosen in the absence of cache line re-use analysis and the crosses mark the one actually selected, i.e., taking that analysis into account. Note, that our model predictions are limited by both theoretical occupancy and problem size, the latter of which is not highlighted in the graphs.

For readability, we have grouped the benchmarks by the outcome of the cache line re-use analysis, as indicated to the right of the diagram: 8 kernels analysed as safe to coarsen, 5 kernels for which cache line re-use has been detected, and 6 kernels which were analysed to be data-dependent. The decisions of the analysis were straightforward to verify by manually inspecting the code of the benchmarks.

The choice of a constant stride value of  $S = 32$  in thread-level coarsening reveals a problem which did not occur with the reduction benchmarks. Rearranging inequality (3.1) in Section refsec:tlc we have the constraint that  $C \leq \text{get\_local\_size}(d)/S$ . However, many of the Rodinia benchmarks have relatively small thread block sizes (see Table 5.1) which means that many benchmarks can only be coarsened by small factors when applying thread-level coarsening; this is reflected in the graphs of Figure 8. We could pick a smaller stride, e.g.  $S = 8$ . However, this risks breaking memory coalescing, as discussed in Section 3.4. An advantage of block level coarsening is that this problem does not arise.

### 5.8.1 Kernels Safe to Coarsen

The first group of eight benchmarks consists of kernels which our analysis assessed as safe to coarsen, meaning that coarsening should either have a beneficial or at least a neutral effect on the kernels' performance. The `bpnn_forward` kernel is largely algebraic and of medium complexity, featuring some control flow elements as well as synchronisation barriers. The kernel benefits well from coarsening, gaining between 2.2x and 3.3x speedup in block-level coarsening mode, being able to benefit from shared instructions in the coarsened code as well as having a control flow structure that synthesises well with the coarsening optimisation.

The `nn` kernel is an example of kernel on which coarsening has a neutral effect, which results from a feature in its implementation: All code of the kernel body is wrapped inside an `if(get_global_id(0) < N)` statement, which essentially performs bounds-checking to ensure that the kernel body is not executed by more threads than intended. This is a common feature in GPU codes, which, however, does not synthesise well with the coarsening algorithm. Recall, that the coarsening algorithm as described by Magni et al. [49] (see Section 3.2) duplicates code regions where the entry condition depends on a variable that itself requires duplication. This means, if coarsening is applied with a factor of two, there will be two derivatives of `get_global_id`, and one `if`-statement for each of those, which are processed separately. The coarsened version is thus not able to profit from any of the benefits of thread coarsening (e.g. shared instructions), resulting in a largely neutral performance effect. The `kmeans` kernel follows the same pattern of performing a check on the global ID, although it is not clear why this resulted in a performance decrease in the Pascal experiments. The `pf_sum` kernels exhibits the same pattern, while the `pf_normalize` kernel has several successive code regions each having such conditionals. In both cases the performance effect is neutral as expected.

The `sc_memset` kernel is a particularly trivial kernel, setting each field of an array to the same value. The performance benefit of thread-level coarsening does not match that of block-level

| Name          | Benchmark            | Kernel              | Dwarfs               | Domain              | Threads per block |
|---------------|----------------------|---------------------|----------------------|---------------------|-------------------|
| findK         | B+ Tree              | findK               | Graph Traversal      | Search              | 256               |
| findRangeK    | B+ Tree              | findRangeK          | Graph Traversal      | Search              | 256               |
| bpnn_adjust   | Back Propagation     | bpnn_adjust_weights | Unstructured Grid    | Pattern Recognition | 16x16             |
| bpnn_forward  | Back Propagation     | bpnn_layerforward   | Unstructured Grid    | Pattern Recognition | 16x16             |
| bfs1          | Breadth-First Search | BFS_1               | Graph Traversal      | Graph Algorithms    | 256x1             |
| bfs2          | Breadth-First Search | BFS_2               | Graph Traversal      | Graph Algorithms    | 256x1             |
| hotspot       | HotSpot              | hotspot             | Structured Grid      | Physics Simulation  | 16x16             |
| hotspot3D     | HotSpot 3D           | hotspotOpt1         | Structured Grid      | Physics Simulation  | 64x4              |
| kmeans        | Kmeans               | kmeans_kernel_c     | Dense Linear Algebra | Data Mining         | 256               |
| kmeans_swap   | Kmeans               | kmeans_swap         | Dense Linear Algebra | Data Mining         | 256               |
| lavamd        | LavaMD               | kernel_gpu_opengl   | N-Body               | Molecular Dynamics  | 128               |
| nn            | k-Nearest Neighbors  | NearestNeighbor     | Dense Linear Algebra | Data Mining         | 256               |
| dynproc       | PathFinder           | dynproc_kernel      | Dynamic Programming  | Grid Traversal      | 256               |
| pf_findIdx    | Particle Filter      | find_index          | Structured Grid      | Medical Imaging     | 512               |
| pf_normalize  | Particle Filter      | normalize_weights   | Structured Grid      | Medical Imaging     | 512               |
| pf_sum        | Particle Filter      | sum                 | Structured Grid      | Medical Imaging     | 512               |
| pf_likelihood | Particle Filter      | likelihood          | Structured Grid      | Medical Imaging     | 512               |
| sc_memset     | Streamcluster        | memset_kernel       | Dense Linear Algebra | Data Mining         | 256x1             |
| sc_pgain      | Streamcluster        | pgain_kernel        | Dense Linear Algebra | Data Mining         | 256x1             |

Table 5.1: Overview of Rodinia kernels used in our experiments.

coarsening, which is potentially due to a higher overhead that thread-level coarsening incurs. However, the kernel does perform best at a very low occupancy, an effect that may be explained by [94].

### 5.8.2 Kernels with Cache Line Re-use

The `hotspot3D`, `kmeans_swap`, `pf_likelihood`, and `sc_pgain` kernels were analysed to be performing cache line re-use. Across these benchmarks, coarsening (at lower occupancy) yielded a significant speedup for one experiment. While the `hotspot3D` kernel would not be coarsened on the basis of our occupancy analysis, the remaining three benchmarks yielded 0.88x, 0.91x, and 0.83x of their original performance in individual experiments, respectively (assuming model predictions are followed).

The observed performance impacts are not as large as we observed in Figure 5.2, and in some cases the performance curve remains neutral throughout. What this highlights is that although these kernels are classified as unsafe to coarsen, applying the coarsening optimisation is not guaranteed to be detrimental. Rather, in practice a kernel might have a complex interplay of benefits and risks of coarsening, such that both a decrease and an increase in performance are possibilities, causing the exact performance impact to be somewhat unpredictable.

A clear example for this is the `bpnn_adjust` kernel. This kernel is a small, algebraic kernel, featuring a synchronisation barrier as well as one conditional, but no loops. It benefits from coarsening, as well as from running under lower occupancy due to its cache line re-use. In this particular case, the reason for this is that a reduction in occupancy eases pressure on the cache

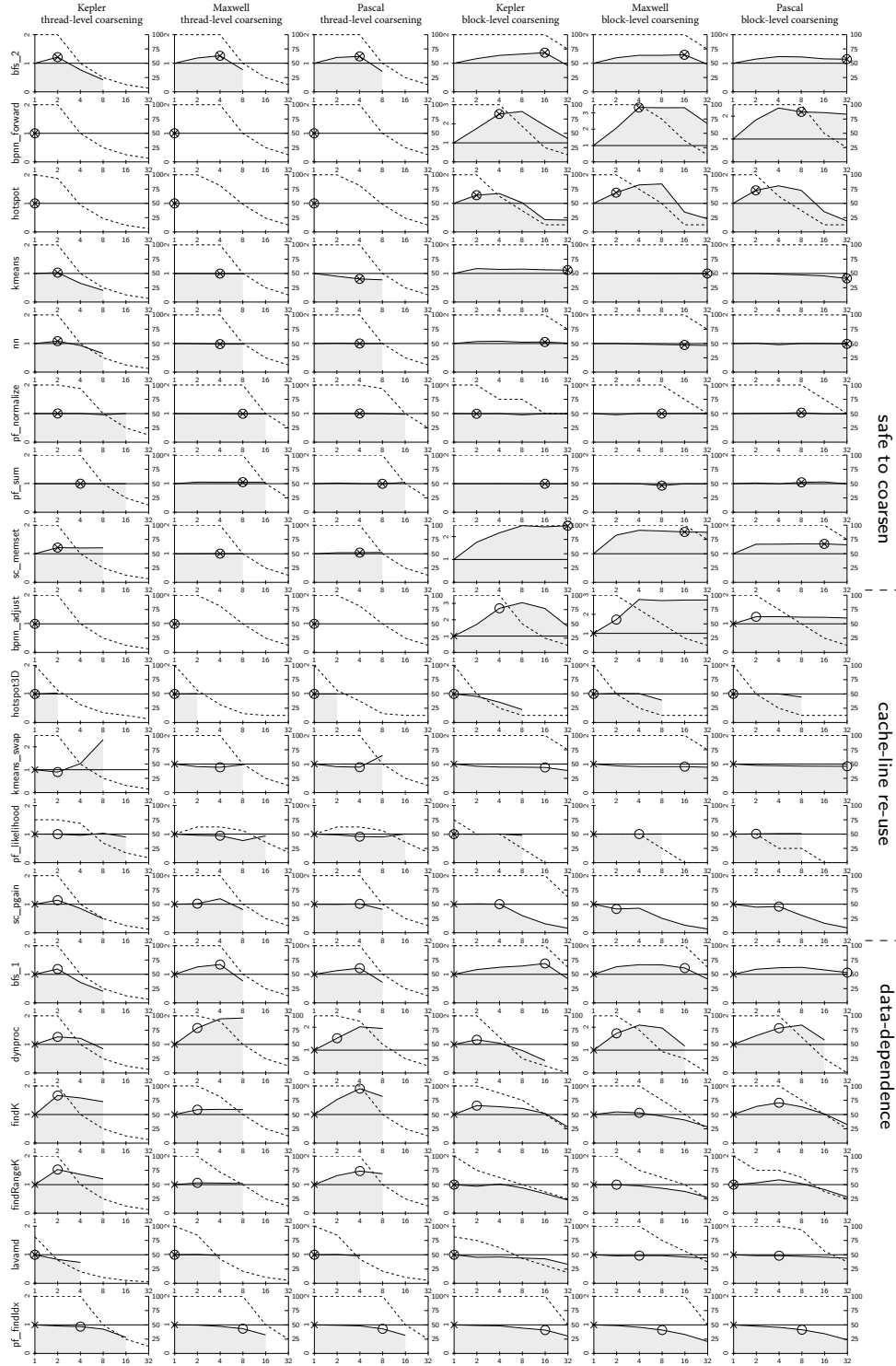


Figure 5.5: Rodinia – plotting speedup over coarsening factor (left axis, straight line), against occupancy (right axis, dashed line), with unadjusted model predictions (X mark) and adjusted by CLR analysis (O mark).

| Category          | Max Speedups (Avg) |             | Unchecked Model (Avg) |             | Model Predictions (Avg) |             |
|-------------------|--------------------|-------------|-----------------------|-------------|-------------------------|-------------|
|                   | thread-level       | block-level | thread-level          | block-level | thread-level            | block-level |
| Safe to coarsen   | 1.05x              | 1.35x       | 1.03x                 | 1.30x       | 1.03x                   | 1.30x       |
| Cache line re-use | 1.10x              | 1.17x       | 0.98x                 | 1.08x       | 1.00x                   | 1.00x       |
| Data-dependence   | 1.28x              | 1.19x       | 1.22x                 | 1.10x       | 1.00x                   | 1.00x       |
| Total             | 1.13x              | 1.25x       | 1.07x                 | 1.17x       | 1.01x                   | 1.12x       |

Table 5.2: Average speedups for the different categories of kernels from the Rodinia benchmark suite. Listed are average of maximum speedups, speedups for unchecked and actual model predictions (corresponding to circles and crosses in Figure 5.4, respectively), for thread-level and block-level coarsening.

in addition to the benefits this kernel experiences from being coarsened. This is also why the unadjusted model predictions fall short of the optimal speedup, e.g. in the case of block-level coarsening on Maxwell 1.72x out of 2.78x.

### 5.8.3 Data-dependent Kernels

Kernels with data-dependent memory accesses (i.e. typically with memory access patterns of the form  $A[B[x]]$ ) are by default excluded from coarsening, as their memory access pattern cannot be reconstructed by the cache line re-use analysis. Instead, profiled execution or full simulated execution would be required to decide whether coarsening should be applied – and this step may have to be repeated if any change in the data that controls memory accesses has taken place between successive kernel invocations, effectively making this class of kernels hard to deal with. The experiments show that, for instance, the `findK` and `dynproc` kernels are able to achieve consistently good speedups ranging from 1.05x to 1.9x, averaging a speedup of 1.4x if the model had been followed. In contrast to that, the benchmarks for the `pf_findIdx` kernel show that coarsening does not apply to this kernel, such that naively following the model predictions would yield between 0.82x and 0.93x of the kernel’s original performance. While it may appear that following the model predictions regardless of the cache line re-use analysis would result in a performance increase on average, it must be considered that this effect is likely due to chance, also with respect to the relatively small sample size, such that this class of kernels remains essentially unpredictable in terms of the performance impact of coarsening.

### 5.8.4 Speedups

Table 5.2 lists averaged results for maximum speedups and speedups achieved by our model for the Rodinia experiments shown in Figure 5.5. As before, we distinguish between ‘unchecked’ model predictions, which have not been adjusted by the results of the cache line re-use analysis, and the model predictions which take the results of the analysis into account.

For kernels that are safe to coarsen, the model can achieve a 1.30x average speedup of a

maximum possible 1.35x speedup for block-level coarsening, while for thread-level coarsening the realised average speedup is 1.03x out of 1.05x. This reflects the practical difficulties we encountered based on the stride parameter.

Kernels classified as performing cache line re-use averaged a performance of 0.97x and 1.08x for the two coarsening modes, respectively, and were excluded from being coarsened. The latter number is positive only because of the `bpnn_adjust` kernel, without which the speedup for block-level coarsening amounts to 0.95x. On the other hand, kernels deemed safe to coarsen yielded a clear average performance speedup, with the exception of corner cases (e.g. `kmeans`), as discussed above.

### 5.8.5 Coarsening at lower occupancy

Throughout our experiments we observed multiple examples where coarsened kernels performed better at reduced occupancy. Unkule et al. [92] also observe this effect, and we are interested to understand why this is happening. Specifically, we distinguish between two different types of performance improvements at lower occupancy: The first category are performance spikes at lowest occupancy. This can be observed for instance in the `k_means_swap` kernel (thread-level coarsening), and is likely due to reasons of cache pressure. This kernel was also diagnosed as performing cache line re-use.

The second category appears to function differently, however. As can be seen for instance in the `hotspot` (safe to coarsen), `bpnn_adjust` (cache line re-use), and `dynproc` (data-dependence) kernels for block-level coarsening, the performance does not spike but instead drops at lowest occupancy. Repeatedly, our model seemed to narrowly under-predict the optimal coarsening factor by a single factor. In these experiments, the peak of the performance curve lies not at lowest occupancy, but just after the occupancy starts to decrease – and as occupancy decreases further, the performance likewise declines.

The reason for this appears to be found in the occupancy measure, which is defined in terms of utilised active threads per SM. However, this is not precisely how the occupancy measure is used in our model. Instead, our model is based on hardware and software degrees of parallelism. The assumption is that occupancy models the degree of parallelism exhibited by the hardware, with the intention to not reduce it. Occupancy, however, is only an approximation of the true degree of parallelism offered by the hardware. While an SM can hold typically 2048 active threads, not all of these are actually processed simultaneously. The number of CUDA cores processing integer and single-precision floating point operations may for instance be 192 on one of our systems, while the number of load-store units and double-precision floating point units is even lower. Naturally, it depends on the program how great the demand for each of these hardware resources is, making it difficult to determine the exact number of threads that are processed concurrently at any moment in time – and naturally, the number may dynamically change throughout program execution.

Not all active threads are therefore truly ‘active’, and some may idle while others await mem-

ory transactions and others are actively using compute function units. The true degree of parallelism, therefore, is lower than captured by the occupancy. This means that it is possible to reduce the occupancy, without reducing the degree of parallelism and without incurring negative performance effects. The degree to which this is possible is largely program dependent, and a negative impact on performance will happen eventually. This explains why many but not all benchmarks perform best under occupancy that is just slightly reduced. We consider therefore that in this respect our model is accurate, yet only as accurate as occupancy approximates the actual degree of hardware parallelism exploited by the program.

### 5.8.6 Discussion

Our experiments show that thread occupancy is a useful metric to use when determining coarsening factor, but also that it is essential to model the effect that coarsening is likely to have on caching. The occupancy metric on its is not sufficient. The analysis we perform is conservative, in the sense that we will only coarsen a kernel if we can establish that the pressure on the cache will not increase by doing so. Of course, it may be that any increase in cache pressure may not have significant detrimental effects on performance and/or that any such effects are, in practice, outweighed by other factors. This can be seen to a small extent in benchmarks like `sc_pgain` and `kmeans_swap` for some configurations. It may also be the case that performance increases, despite a reduction in occupancy, and we can see this in some of the benchmarks, e.g. `hotspot`.

In almost all cases performance either increases or remains flat when we elect to coarsen a kernel, so we conclude that the combination of occupancy metric and cache line re-use analysis is doing a good job of ensuring that our optimisations are *safe* in that they avoid degrading performance. The only exception in our benchmark suite was `kmeans`, as discussed above.

Table 5.2 shows that block-level coarsening results in better overall performance than thread-level coarsening for the Rodinia benchmarks. However, this was not as clearly the case for the reduction benchmarks (see Section 4.4). This highlights the significant practical difficulty of choosing a suitable stride parameter for small thread block sizes. The problem is amplified in particular for multi-dimensional kernels, where coarsening with a higher stride is frequently impossible, depending on the exact thread layout of the block. However, as noted earlier, reducing the stride parameter, is not safe in all cases as it may break memory coalescing.

Independent of the stride, applying thread-level coarsening can lead more quickly to occupancy reduction than for block-level coarsening. The reason is that while the number of threads per block is reduced, the number of blocks running on each SM is increased to retain the same occupancy. However, the number of blocks running per SM is limited. In practice very small thread blocks cannot be coarsened at all using thread-level coarsening without effecting a reduction in occupancy.

The practical issues encountered by block-level coarsening appear to be less significant in comparison. Two points in particular are worth mentioning. First, the coarsening factor must evenly divide the number of thread blocks launched. We anticipate that in practice this can be

circumvented by pre-compiling different or additional coarsening factors, whereas our study is limited to coarsening factors that are powers of two. Secondly, block-level coarsening produces fewer, longer-running thread blocks and this can lead to longer ‘tail effects’ where the last few blocks are running on a small number of SMs. Depending on the total workload this can reduce the overall device utilisation.

The decision as to which coarsening mode should be applied to a given kernel can also depend on which code regions require duplication, and as such, the exact structure of the resulting coarsened kernel – which in many cases differ for the two coarsening modes. Because of this, we can imagine that in some cases it may be desirable to choose among the two options based on static analysis of the kernel being compiled.

Magni et al. [51] train a neural network using 17 candidate static code features. Their approach is complex in nature, requiring extensive training across of the network. They evaluate on four different architectures, comprising GPUs from NVidia’s Fermi and Kepler architectures, as well as AMD’s Cypress and Tahiti. They report speedups of 1.16x, 1.11x, 1.33x, 1.30x, respectively. Cummins et al. [18] replicate their experimental setup, but report speedups of 1.21x, 1.01x, 0.86x, and 0.94x for the Magni et al. model. For their own DeepTune they report speedups of 1.10x, 1.05x, 1.10x, and 0.99x, respectively. They also add, that for their set of benchmarks a perfect heuristic would achieve a speedup of 1.36x. In comparison, our block-level coarsening achieves an average of 1.12x out of a maximum of 1.25x speedup. While this appears to outperform machine learning based approaches for NVidia GPUs, we nevertheless acknowledge the limitations of such a comparison due to different benchmark suites involved. Importantly, however, machine learning based approaches as those discussed are highly complex and time-consuming to train, requiring many repeated program executions, and results may be difficult to replicate (as reported by Cummins et al. [18]). The advantage of our analytical approach, however, lies in its relative simplicity, requiring only static analysis and no experimental program executions for selecting near-optimal coarsening factors.



# Artificial occupancy reduction 6

Up to this point we have considered thread coarsening as a means to reduce software parallelism. In this chapter, we consider a closely related optimisation technique for reducing hardware parallelism, which has been shown to lead to a performance increase in some cases [94]. The effect can be achieved by artificially lowering a kernel's occupancy, implemented by reserving superfluous shared memory resources. We consider such an *artificial* reduction in occupancy as an optimisation technique, to be distinct in nature from occupancy reduction through natural causes, such as increased resource usage which may be caused, for instance, by thread coarsening. The goal of this chapter is to experimentally explore the relationship between thread coarsening and artificial occupancy reduction.

## 6.1 Motivation

There are obvious similarities between thread coarsening and artificial occupancy reduction. Thread coarsening reduces software parallelism, while artificial occupancy reduction limits the hardware parallelism available on the targeted platform. As such, the former reduces the number of threads launched, while the latter reduces the number of threads that can be processed concurrently by the device. Thread coarsening, therefore, increases the concurrent workload of SMs as the workload per thread is increased, while artificial occupancy reduction lessens it as the number of threads residing in the processors is reduced. The former can therefore increase cache pressure, while the latter relieves it.

As the similarities are apparent, our goal is to answer the following questions on the relationship between the two optimisation techniques:

- The cache line re-use analysis identifies kernels for which cache pressure may be an issue, in order to exclude them from coarsening. Since artificial occupancy reduction eases pressure on the cache, are the kernels identified by the cache line re-use analysis therefore a set of prime candidates for which an artificial occupancy reduction should be attempted?
- Can thread coarsening and artificial occupancy reduction be used in conjunction to achieve better performance than either optimisation by itself? That is, if artificially reducing a kernel's occupancy yields a performance benefit, does a further reduction in

occupancy, and the consequential easing of cache pressure, allow for the application of (additional) thread coarsening?

- Both optimisations are concerned with tampering with the degree of parallelism of a kernel execution, both modify the pressure on the cache. While this apparently makes artificial occupancy reduction closely related to thread coarsening, there is also a possibility of potential interference. Therefore, should interference between the optimisations be expected, or is it possible to apply them entirely independently of each other?

## 6.2 Implementing a compiler pass

In order to automate the code transformation to artificially reduce the occupancy of a kernel, We have extended our existing LLVM pipeline with a compiler pass to automate the code transformation required for achieving artificial occupancy reduction. This is implemented by reserving redundant shared memory resources at the beginning of a kernel, e.g.:

```
1 __local char dummy[2048];
```

However, in practice this declaration will be removed during compilation, as it introduces an unused array definition. Optimising compilers can trivially identify and remove such unused definitions. In our setup, this could happen at three different stages, namely in LLVM when applying coarsening, in the AXTOR tool when translating back to OpenCL, and by the NVidia OpenCL driver in the process of compiling to PTX. However, by inserting an additional code snippet, the dummy definition can be retained throughout all stages of the compilation process.

The LLVM IR code inserted by our compiler pass is equivalent to the following OpenCL code:

```
1 __local char dummy[2048];
2 if (get_local_size(0) == MAX_INT) {
3     *((volatile __local char*)&(dummy[0])) = (char)(0x00);
4 }
```

The code snippet works as follows: Line 3 shows a memory access to the allocated memory region, whereby the allocated memory ceases to be an unused definition. Declaring the access as volatile prevents the compiler from reasoning about circumstances surrounding the memory access. A non-volatile access, on the other hand, may again be identified and removed by an optimising compiler.

The memory access is placed inside a conditional branch whose entry condition will never be true. The trick is again to select a condition which an optimising compiler is not able to reason about. Conditional branches whose entry condition statically evaluates to false can be identified as ‘dead’ (i.e., unreachable) code regions, and subsequently be removed. Thread blocks of size MAX\_INT are not supported by current devices, which is unknown to the compiler.

The above solution seeks to minimise the overhead incurred at run-time by the optimisation. This is achieved by two properties: First, the memory access is hidden in an unreachable code region and is therefore not performed. Second, the branching condition is quick to check, as the

return values of work-item related built-in functions are stored in special registers on NVidia GPUs.

This code snippet is inserted at the beginning of a kernel body. On the level of LLVM byte-code, this means that two basic blocks are inserted before the first basic block of the kernel, typically named the entry block. The following example shows artificial occupancy reduction applied to a reduction kernel:

```

1  @reduce0..aor = internal addrspace(3) global [2048 x i8] zeroinitializer,
    ↪ align 1
2  @reduce0.sdata = internal addrspace(3) global [512 x i8] zeroinitializer,
    ↪ align 1
3
4  define void @reduce0(float addrspace(1)* %g_idata, float addrspace(1)*
    ↪ %g_odata, i32 %n) #0 {
5  aor..entry:
6    %0 = call i32 @get_local_size(i32 0)
7    %1 = icmp eq i32 %0, 2147483647
8    br i1 %1, label %aor..memaccess, label %entry
9
10 aor..memaccess:                                ; preds = %aor..entry
11   store volatile i8 0, i8 addrspace(3)* getelementptr inbounds ([2048 x i8]
    ↪ addrspace(3)* @reduce0..aor, i8 0, i8 0), align 1
12   br label %entry
13
14 entry:                                          ; preds = %aor..memaccess, %aor..entry
15   %call = call i32 @get_local_id(i32 0) #3
16   %2 = call i32 @get_group_id(i32 0) #3
17   [...]
18   [...]
19   }

```

Line 1 shows the additional shared memory allocation used to artificially reduce the occupancy of the kernel. This is alongside the existing shared memory used by the kernel, as shown in line 2. The basic block labelled `aor..entry` shown in line 5 performs the conditional described above, branching to the entry basic block in line 14. This block contains the original code of the kernel, and is preserved without modifications. The `aor..memaccess` basic block in line 10 performs a stub memory access to the dummy array, but this part of the code is never executed.

When invoking the compiler pass, two arguments needs to be specified. First, the name of the kernel which should be modified, and second, the amount of shared memory which should be reserved besides any existing allocations of shared memory resources. Besides existing shared memory usage, the initial occupancy of the kernel also has to be taken into account, which could be limited by factors other than shared memory usage. It can therefore be misleading to pursue a simplistic approach to calculating the amount of additional shared memory required to limit execution to a desired number of thread blocks.

We seek to determine the amount of shared memory  $S$  that needs to be allocated to limit concurrent execution to a maximum of  $n$  thread blocks per SM. Let  $S_{existing}$  be the amount of shared memory of any existing definitions in the kernel, and  $S_{total}$  be the total amount of

shared memory available on each SM:

$$S(N) = \frac{S_{total} - (N + 1) * S_{existing}}{N + 1} + 1 \quad (6.1)$$

Intuitively, to obtain the amount of shared memory required to execute  $N$  thread blocks, we calculate the amount of memory required to execute  $N + 1$  thread blocks, and add one byte to it. This has the effect of limiting the number of concurrently executed thread blocks to  $N$ .

In the above equation,  $N$  is constrained by two terms, such that  $N_{min} \leq N \leq N_{init}$ , where  $N_{init}$  is the initial number of concurrently executed thread blocks per SM with respect to the launch configuration and the initial occupancy of the kernel, and  $N_{min}$  acts as a lower bound on the number of thread blocks that can be achieved by this technique. These upper and lower bounds are calculated as follows:

$$N_{init} = O * \frac{T_{maxActive}}{T_{block}}$$

$$N_{min} = \frac{S_{total}}{S_{maxPerBlock}}$$

Here,  $O$  is the initial occupancy of the kernel,  $T_{maxActive}$  is the number of maximum active threads per SM,  $S_{total}$  is the total amount of shared memory per SM, and  $S_{maxPerBlock}$  is the maximum amount of shared memory addressable by one block. For instance, our Pascal architecture is equipped with 96K of shared memory on each SM, while a thread block can only address a maximum of 48K of shared memory. Therefore,  $N_{min} = 2$  on the Pascal architecture, which is the same of the Maxwell architecture. However, this differs for older architectures, including Kepler, where  $S_{total} = S_{maxPerBlock}$ . If  $N_{min}$  is erroneously set lower than permitted by the architecture, the result is that  $S(N) > S_{maxPerBlock}$ , causing a compiler error of the kernel.

We have implemented an execution engine to execute a given kernel at all possible occupancy levels, and to determine how much additional shared memory needs to be allocated to achieve a given occupancy according to the formula stated in Equation 6.1 above. The task of the execution engine is to determine the boundaries  $N_{init}$  and  $N_{min}$ , as well as the number of iterations required to cover all possible occupancies, which can vary depending on the thread block size. For instance, a kernel launch with  $T_{block} = 512$  and  $O = 1$  (that is, 100% initial occupancy) and  $N_{min} = 2$  will be executed for  $[2048, 1536, 1024]$  threads per SM with a step size of 512, which corresponds to three distinct occupancies, namely  $[1, 0.75, 0.5]$ , respectively. However, for  $T_{block} = 128$  and  $O = 0.875$  will be executed for  $[1792, 1664, \dots 512, 384, 256]$  with a step-size of 128, resulting in 13 experiments at occupancies  $[0.875, 0.8125, \dots 0.25, 0.1875, 0.125]$ , respectively. The number of experiments, and therefore unique occupancies, as well as  $N_{init}$  are only known dynamically at run-time because of a dependency on  $T_{block}$ , the number of

threads per block, which is only available at run-time.

The coarsening pipeline has therefore been extended to integrate with the execution engine. The existing coarsening pipeline has knowledge of all information required by the execution engine, which are  $T_{block}$ ,  $S_{existing}$ , and  $O$ , which are gathered or calculated for the purposes of automatic coarsening factor selection. This information is available in the override to the `enqueueNDRangeKernel()` method, at which point kernel execution has been requested but has not taken place yet. The interaction between the existing coarsening pipeline and the execution engine works as follows:

- The execution engine invokes the original program without artificial occupancy reduction.
- The existing coarsening pipeline, which handles the program execution, has knowledge of all information required by the execution engine, which are  $T_{block}$ ,  $S_{existing}$ , and  $O$ . These are gathered or calculated for the purposes of automatic coarsening factor selection. These information are available in the override to the `enqueueNDRangeKernel()` method, at which point kernel execution has been requested but has not taken place yet. This information is made available to the execution engine.
- The execution of the original program is continued without applying artificial occupancy reduction. The reason for this is that we are interested in executing the program at all possible occupancies, including the initial occupancy. However, we can imagine a scenario in which artificial occupancy reduction should be applied before executing a program. This is possible to do, since all necessary information (see above) is available at this point. However, this is not currently implemented.
- After the initial run at full occupancy completes, the execution engine uses the information obtained from the coarsening pipeline to calculate the required  $S(N)$  for all  $N_{min} \leq N < N_{init}$ , and to establish the total number of executions it will perform. The execution engine finally executes the program for all possible occupancies.

## 6.3 Experiments

Throughout this section we will re-investigate benchmarks from previous chapters, focusing in particular on benchmarks of the Rodinia suite (see Chapter 5). However, further benchmarks are also considered, specifically to perform a preliminary case study on a simple kernel. As before, any time measurements reported are averaged over a total of ten executions. Unlike in previous chapters, cross-platform portability is not part of our main objective here. The experiments throughout this chapter are therefore focused on the Pascal architecture (see Table 4.1).

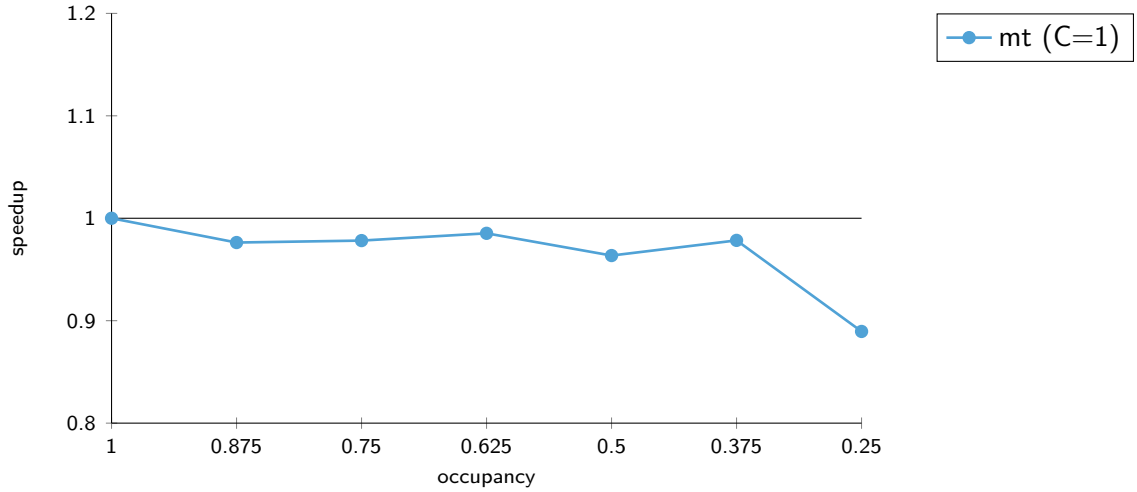


Figure 6.1: Matrix-transpose kernel with artificially reduced occupancy.

### 6.3.1 Case study: Matrix transpose

As an initial experiment, recall the matrix-transpose kernel discussed in Figure 5.2. The kernel reads an element at position  $(x, y)$  of a matrix and stores it to position  $(y, x)$ . For optimal performance, it is essential that the number of cache misses be kept low, as the kernel performs several subsequent store operations to the same cache line, which should not be evicted from cache and written back to global memory until all writes to it have completed.

Figure 6.1 shows the effects of artificial occupancy reduction on this kernel, plotting the achieved speedup over occupancy. Here, occupancy is expressed as a factor ranging from 0 to 1, with the factor 1 on the left hand side corresponding to 100% or full occupancy, and with occupancy levels decreasing from left to right.

In this example thread coarsening has the effect of increasing the pressure on the cache, thereby causing a decline in performance as the coarsening factor is increased (see Figure 5.2). Artificial occupancy reduction might be expected to have the opposite effect by easing pressure on the cache. However, Figure 6.1 shows that this does not translate into a performance improvement in this case. As the graph shows, even a minor reduction of occupancy will immediately result in a performance penalty of 2.4% (measured at 87.5% occupancy), while the performance loss at minimal occupancy (in this case, 25% occupancy) exceeds 11%. The results of Figure 6.1 are in some sense not surprising, showing that reducing the parallelism of the program has caused performance to decline.

The memory footprint of the kernel is important to consider. Each thread performs a 4-byte load of adjacent memory addresses which the device is able to fully coalesce. This results in  $4 * 32 * C$  byte for each warp consisting of 32 threads. Subsequently, each thread performs an uncoalesced store to a unique cache line, resulting in  $32 * C$  unique accessed cache lines per warp. This causes pressure when performing stores, which is significantly increased as coarsening is applied to the kernel.

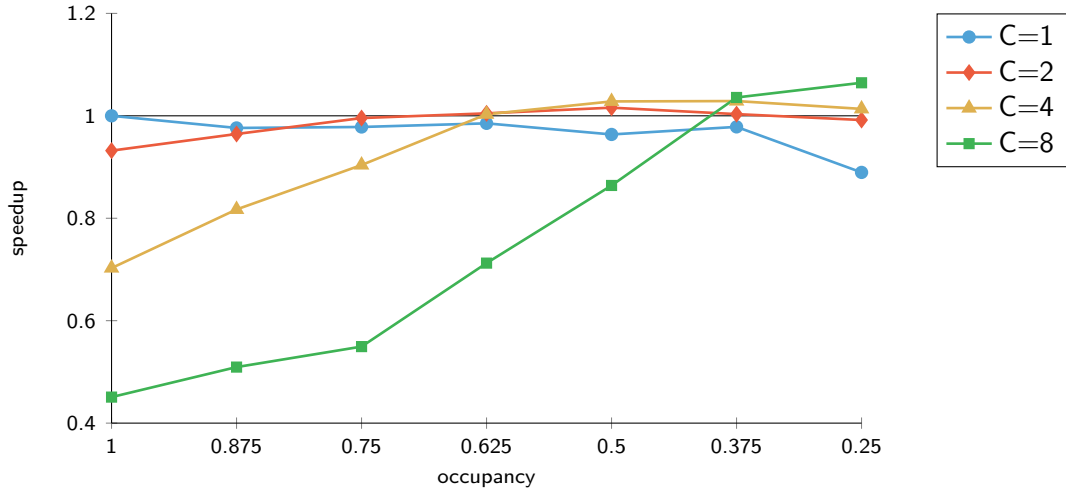


Figure 6.2: Coarsened matrix-transpose kernels with artificially reduced occupancy.

But the question remains, whether an artificial occupancy reduction could have yielded a performance improvement given higher initial cache pressure. To explore this, we apply coarsening to the kernel before subsequently applying artificial occupancy reduction. The simplistic nature of the kernel means that coarsening is expected to have diminishingly small effects on the kernel other than to increase its cache pressure – any opportunities to exploit shared instructions or ILP appear non-obvious.

Figure 6.2 plots the results of these experiments, for which we measured a standard deviation of 0.22% in the run times. The graph shows one curve for each coarsened version of the kernel, each labelled according to its coarsening factor  $C$ . The performance baseline of all experiments is the run time of the uncoarsened version at full occupancy. The original version of the code is the curve labelled  $C = 1$  and corresponds to the curve shown in Figure 6.1.

The performance curve for the coarsening factor  $C = 2$  shows that artificial occupancy reduction can have a positive impact on kernels for which cache pressure is an issue. At full occupancy, this coarsened version of the code performs at 93.2% of the original kernel, but this increases to 101.6% as the occupancy is reduced to 50%. This improvement means that artificial occupancy reduction can improve the performance of the  $C = 2$  program, which suffers from cache pressure. Cache line re-use has also been detected for the program by our analysis.

It may therefore appear that the cache line re-use analysis is able to identify programs that are likely to benefit from artificial occupancy reduction, as is the case for the matrix transpose when applying a coarsening factor  $C \geq 2$ . However, the results for the uncoarsened program shown in the  $C = 1$  curve appear to suggest that the cache line re-use analysis by itself may not be a sufficiently strong indicator to attempt occupancy reduction of a kernel.

The second insight that the  $C = 2$  performance curve offers is that coarsening and occupancy reduction in conjunction exceed the performance of either optimisation on its own. Recall, that neither optimisation by itself yielded performance improvement for any parameterisation (see Figures 5.2 and 6.1), whereas both optimisations used in conjunction results in a narrow

performance improvement of 1.6%.

The potential performance gain of artificial occupancy reduction increases for versions of the kernel with higher initial cache pressure. The  $C = 4$  program starts at 70.3% of the original performance and improves to 102.9%, a 1.46x speedup compared to itself. The  $C = 8$  program improves from 45.08% to 106.45% performance improvement at 25% occupancy compared to the original version, corresponding to 2.36x speedup of the  $C = 8$  program compared to its own original performance.

Figure 6.2 shows that at some point all coarsened programs with  $C \geq 2$  exceed the performance of the original program  $C = 1$ . This suggests that coarsening must have some positive performance benefit caused by shared instructions or increased ILP. However, this benefit only becomes apparent when the negative effects of coarsening – an increase in cache pressure – are cancelled out by artificial occupancy reduction. In this particular example, the kernel body does not feature any expensive shared instructions such as synchronisation barriers or control flow. It can therefore be supposed that for kernels with more costly shared instructions the combined benefit of coarsening and occupancy reduction will be higher than in this scenario.

#### 6.3.1.1 Discussion

These preliminary results appear to answer some of our initial questions. First, that the cache line re-use analysis might highlight good candidates for occupancy reduction. However, it is likely not a strong indicator by itself. Second, it appears to be the case that the two optimisations can work together in conjunction, achieving a performance improvement of 6.5% when applied together, although when applied individually both optimisations caused a performance degradation.

The two optimisations also have the ability to cancel out each other's negative effects. We observed that applying artificial occupancy reduction to a matrix transpose kernel coarsened with  $C = 8$  improves the performance from 45.08% to 106.45%, a 2.36x speedup. Conversely, applying artificial occupancy reduction can, in the worst case, reduce the performance to 88.95%. Subsequently applying coarsening improves this performance by a factor of 1.20x. In some sense, the worst coarsening factor ( $C = 8$ , 0.45x speedup) and the worst artificial occupancy reduction ( $O = 0.25$ , 0.88x speedup) yielded the best measured speedup across all experiments when applied in combination with each other (1.065x speedup).

In the following, we seek to investigate these questions experimentally on a wider set of benchmarks.

#### 6.3.2 Cache line re-use analysis as an indicator for artificial occupancy reduction

The objective of these experiments is to explore whether cache line re-use analysis can be used to find candidates for artificial occupancy reduction. The idea is that these kernels may experience cache pressure, which artificial occupancy reduction is able to reduce so as to yield an increase in performance.

| Kernel name   | Analysis          | Best speedup | At occupancy | Speedup at min occupancy | Min occupancy |
|---------------|-------------------|--------------|--------------|--------------------------|---------------|
| bfs2          | –                 | 1.00x        | 100.00%      | 0.29x                    | 25.00%        |
| bpnn_adjust   | –                 | 1.00x        | 75.00%       | 0.99x                    | 25.00%        |
| bpnn_forward  | –                 | 1.02x        | 87.50%       | 0.44x                    | 25.00%        |
| hotspot       | –                 | 1.00x        | 100.00%      | 0.48x                    | 25.00%        |
| kmeans        | –                 | 1.00x        | 100.00%      | 0.77x                    | 25.00%        |
| nn            | –                 | 1.00x        | 100.00%      | 0.77x                    | 25.00%        |
| pf_normalize  | –                 | 1.00x        | 100.00%      | 1.00x                    | 50.00%        |
| pf_sum        | –                 | 1.01x        | 50.00%       | 1.01x                    | 50.00%        |
| sc_memset     | –                 | 1.00x        | 100.00%      | 0.31x                    | 25.00%        |
| hotspot3D     | cache line re-use | 1.00x        | 75.00%       | 0.73x                    | 25.00%        |
| kmeans_swap   | cache line re-use | 3.11x        | 25.00%       | 3.11x                    | 25.00%        |
| pf_likelihood | cache line re-use | 1.00x        | 75.00%       | 1.00x                    | 50.00%        |
| sc_pgain      | cache line re-use | 1.00x        | 100.00%      | 0.42x                    | 25.00%        |
| bfs1          | data-dependent    | 1.00x        | 100.00%      | 0.30x                    | 25.00%        |
| dynproc       | data-dependent    | 1.00x        | 100.00%      | 0.55x                    | 25.00%        |
| findK         | data-dependent    | 1.00x        | 100.00%      | 0.34x                    | 25.00%        |
| findRangeK    | data-dependent    | 1.00x        | 100.00%      | 0.38x                    | 25.00%        |
| lavamd        | data-dependent    | 1.00x        | 100.00%      | 0.98x                    | 12.50%        |
| pf_findIdx    | data-dependent    | 1.00x        | 100.00%      | 0.99x                    | 50.00%        |

Table 6.1: Performance of Rodinia benchmarks at artificially reduced occupancy.

Table 6.1 shows the results of artificial occupancy reduction on kernels of the Rodinia benchmark suite. The table lists the result of the cache line re-use analysis for each kernel, as well as:

1. The optimal speedup achieved by artificial occupancy reduction, and the occupancy at which it was measured.
2. The speedup measured at minimal occupancy, including the minimal occupancy itself. Here, we use ‘minimal occupancy’ to refer to the minimal value achievable by reserving superfluous shared memory resources. Note, that on the Pascal architecture this corresponds to two thread blocks, regardless of their size.

The results presented in this table reveal three distinct outcomes to applying artificial occupancy reduction to a given kernel. First, a kernel may experience a performance gain. This was the case for the `kmeans_swap` kernel, which yielded a 3.11x performance improvement when executed at 25% of its original occupancy. Secondly, artificial occupancy reduction may have a neutral effect on a kernel’s performance. The six kernels exhibiting this characteristic are `bpnn_adjust`, `pf_normalize`, and `pf_sum` (all classified as not performing cache line re-use), `pf_likelihood` (cache line re-use), `lavamd`, and `pf_findIdx` (both data-dependent). Finally, the remaining twelve kernels experience a performance slowdown, in some cases yielding 0.3x of their original performance, for instance in the case `bfs1` and `bfs2`.

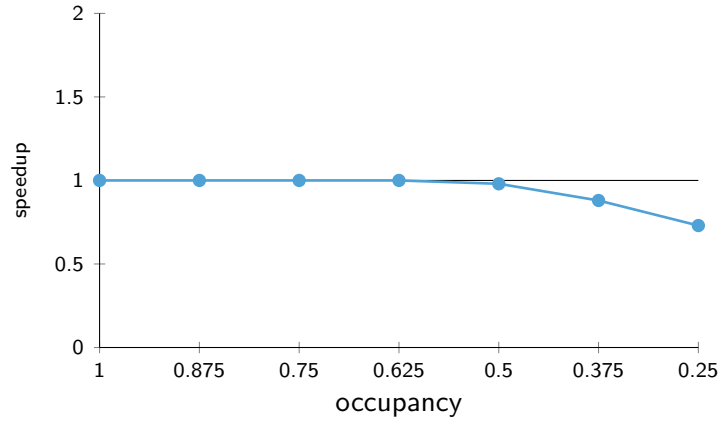


Figure 6.3: Performance impact of artificial occupancy reduction on the hotSpot3D kernel.

Six experiments of Table 6.1 exhibited a neutral performance on a flat line across all levels of occupancy between the initial and minimal occupancy levels, without exhibiting significant measurable changes. In contrast to that, across all these experiments we observed that if artificial occupancy reduction impacts the performance of a kernel, then even a slight alteration in occupancy results in a measurable change in performance, in the manner of the experiment shown in Figure 6.1. The only exception to this is the hotSpot3D kernel, as shown in Figure 6.3. For this particular case, artificial occupancy reduction initially has a neutral effect. However, performance starts to decline as the occupancy of the kernel is reduced to 50% or less. This holds a clue to understanding the set of kernels whose performance appears unaffected by artificial occupancy reduction. In similar manner to the hotSpot3D kernel, it is important to highlight that any kernels may be unaffected by artificial occupancy reduction only up to a certain point, yet not universally, as some of our experiments appear to suggest. The neutral performance impact must be understood only within the boundaries and possibilities of our test setup. As in the hotSpot3D kernel, there will come a point when performance starts to decline, even though this is outside the measurable range. However, for parallel programs executed on parallel hardware an eventual decline in performance is both natural and expected for a sufficiently large reduction in the degree of parallelism employed.

The neutral performance impact may be explained similarly to the effect described in Section 5.8.5, where we discussed the limitations of the occupancy measure in capturing the degree of parallelism. Instead, occupancy captures the number of utilised active threads per SM. The actual degree of parallelism exhibited by the hardware, however, is approximately equal to the number of active threads required to maximise utilisation of the available processing resources such as CUDA cores, load-store units, double-precision units, etc. The actual degree of parallelism exploitable by a program is thus typically lower than what is captured by the number of maximum active threads and therefore the occupancy measure. The degree of exploitable parallelism may also vary depending on program structure and use of computational resources. This implies that during program execution some active threads may be idle, such that in practice it

may make no difference whether they are queuing for processing within or without an SM. We believe this explains why in some cases artificial occupancy reduction makes no difference in practice, thus causing the observed neutral effect on performance. We have also investigated a link between neutral performance effects and the operational intensities of kernels, although this appears to have proven inconclusive.

Table 6.1 furthermore shows that only one kernel experienced a significant performance improvement, namely the `kmeans_swap` kernel, yielding a 3.11x improvement when executed at 25% occupancy. The kernel was identified as performing cache line re-use by the analysis. However, none of the other kernels belonging to the same group showed a performance improvement when their occupancy was reduced.

The results obtained in these experiments show that the cache line re-use analysis is not a reliable indicator on its own in regards to whether a kernel will benefit from artificial occupancy reduction. The analysis merely finds whether cache pressure has the potential to be an issue for a given kernel, yet does not have the ability to predict whether in practice cache pressure will be an issue, and to what degree this would be so. The results appear to indicate that in practice not many of the cache line re-using kernels appear to suffer from issues caused by cache pressure. For a better predictor it would therefore be necessary to have more extensive knowledge of the memory footprint and caching behaviour of a kernel. This could be done by a simple capacity estimation, or by a more sophisticated cache model, e.g. in the manner of Nugteren et al. [59].

### 6.3.3 Coarsening and artificial occupancy reduction

Previous experiments involving thread coarsening have shown that some kernels perform best when the selected coarsening factors affect a reduction in occupancy (see Section 5.8.5). This effect is known in literature and has been observed by Unkule et al. [92]. The same effect could also be observed in our experiments and was discussed in Section 5.8.5. The kernels from the Rodinia benchmark suite for whom this behaviour could be observed are `bpnn_adjust`, `hotspot`, and `dynproc`, as well as, to a lesser degree, the `bpnn_forward` kernel.

In order to investigate the relationship between coarsening and artificial occupancy reduction further, we have executed benchmarks from the Rodinia suite, applying both optimisations in conjunction. Each benchmark is executed for a total of 6 different coarsening factors and for all levels of occupancy, which dynamically varies from 4 to 16 different levels.<sup>1</sup> As before, each experiment was repeated ten times, resulting a total of 5760 individual executions. The reported numbers are as before based on the averaged run times. To aid readability, speedups are reported as factors while occupancy levels are reported as percentages. The results are shown in Table 6.2. The table lists the kernel name and the outcome of the cache line re-use analysis in the first two columns, as well as:

- The best speedup achieved by applying both optimisations in conjunction

---

<sup>1</sup>Note, a minimum of two blocks are executed on each SM, so the actual number is one level less than stated.

| Kernel name   | Analysis          | Best combined speedup |   | Coarsening Factor | Occupancy          | $\Delta$ Occupancy | Occupancy reduction only |       | Coarsening only | Gain |
|---------------|-------------------|-----------------------|---|-------------------|--------------------|--------------------|--------------------------|-------|-----------------|------|
| bfs2          | –                 | 1.27x                 | 4 | 100.00%           |                    |                    |                          |       |                 |      |
| bpnn_adjust   | –                 | 1.35x                 | 4 | 25.00%            | -50.00% (4 blocks) |                    | 0.99x                    | 1.31x | 1.03x           |      |
| bpnn_forward  | –                 | 2.28x                 | 4 | 100.00%           |                    |                    |                          |       |                 |      |
| hotspot       | –                 | 1.61x                 | 4 | 75.00%            |                    |                    |                          |       |                 |      |
| kmeans        | –                 | 1.00x                 | 1 | 100.00%           |                    |                    |                          |       |                 |      |
| nn            | –                 | 1.01x                 | 2 | 100.00%           |                    |                    |                          |       |                 |      |
| pf_normalize  | –                 | 1.00x                 | 1 | 100.00%           |                    |                    |                          |       |                 |      |
| pf_sum        | –                 | 1.00x                 | 1 | 100.00%           |                    |                    |                          |       |                 |      |
| sc_memset     | –                 | 1.40x                 | 2 | 87.50%            | -12.50% (1 block)  |                    | 0.94x                    | 1.29x | 1.08x           |      |
| hotspot3D     | cache line re-use | 1.00x                 | 2 | 50.00%            |                    |                    |                          |       |                 |      |
| kmeans_swap   | cache line re-use | 3.11x                 | 1 | 25.00%            | -75.00% (6 blocks) |                    | 3.11x                    | 1.00x | 1.00x           |      |
| pf_likelihood | cache line re-use | 1.00x                 | 1 | 100.00%           |                    |                    |                          |       |                 |      |
| sc_pgain      | cache line re-use | 1.00x                 | 1 | 100.00%           |                    |                    |                          |       |                 |      |
| bfs1          | data-dependent    | 1.25x                 | 4 | 100.00%           |                    |                    |                          |       |                 |      |
| dynproc       | data-dependent    | 1.69x                 | 4 | 62.50%            | -12.50% (1 block)  |                    | 0.90x                    | 1.67x | 1.01x           |      |
| findK         | data-dependent    | 1.47x                 | 4 | 100.00%           |                    |                    |                          |       |                 |      |
| findRangeK    | data-dependent    | 1.19x                 | 4 | 75.00%            |                    |                    |                          |       |                 |      |
| lavamd        | data-dependent    | 1.00x                 | 1 | 100.00%           |                    |                    |                          |       |                 |      |
| pf_findIdx    | data-dependent    | 1.00x                 | 1 | 100.00%           |                    |                    |                          |       |                 |      |

Table 6.2: Performance of coarsened Rodinia benchmarks at artificially reduced occupancy.

- The coarsening factor and occupancy at which this speedup was measured
- The reduction in occupancy as compared to the kernels original occupancy
- The speedup achieved by applying occupancy reduction on its own at the level stated in the ‘occupancy’ column, i.e., with a coarsening factor of  $C = 1$
- The speedup achieved by applying thread coarsening on its own with the coarsening factor specified in the fourth column, without artificially reducing the occupancy
- The gain of applying both optimisations in conjunction as compared to either one on its own. For instance, the dynproc kernel achieved a 1.69x speedup from applying both optimisations in conjunction, but could have yielded a 1.67x speedup if only coarsening was applied. The improvement of combining the optimisations is therefore approximately 1%, or 1.01x expressed as a factor. In comparison, the kmeans\_swap kernel draws its entire speedup from artificial occupancy reduction, while the addition of coarsening does not yield any further performance improvement.

The final four columns of the table are only shown for kernels that experience a performance increase at artificially reduced occupancy. For kernels which benefit from thread coarsening at either un-reduced or naturally reduced occupancy the table is kept sparse to improve readability, as these were discussed in more detail in the previous chapters.

The bpnn\_adjust kernel achieves its maximum speedup of 1.35x when executed at minimal occupancy, with a coarsening factor of 4. This may appear to indicate that cache pressure is an issue. However, this seems unlikely given that the addition of artificial occupancy reduction only affects an improvement of 3% as compared to executing the same coarsened kernel at

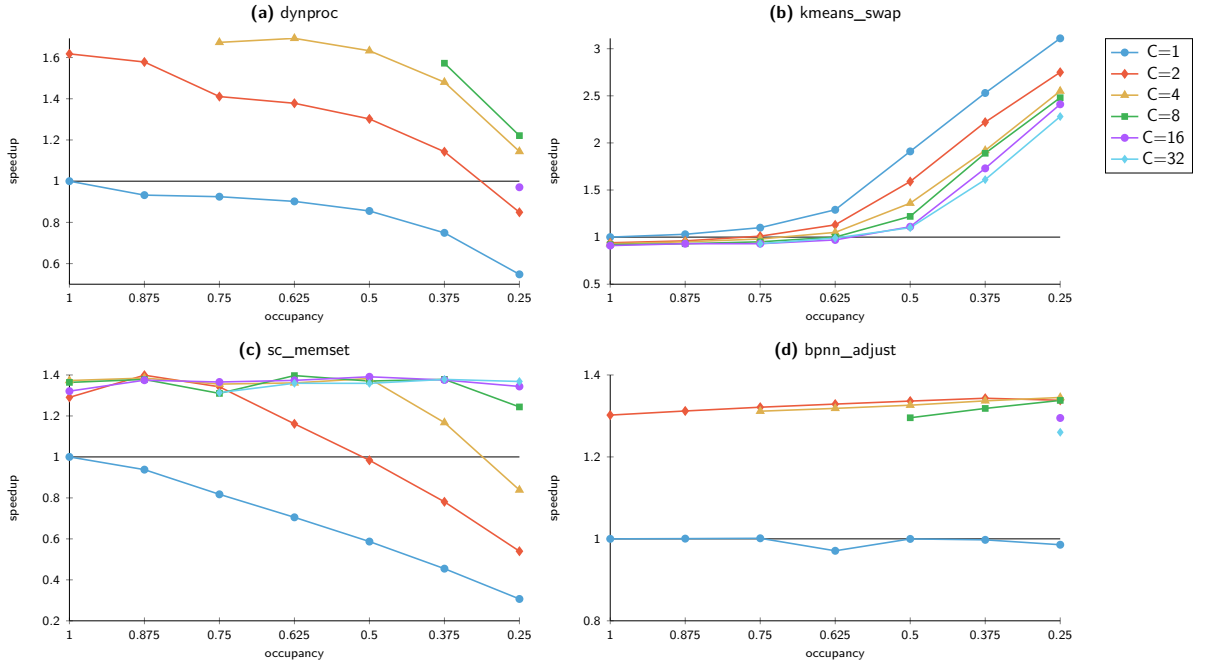


Figure 6.4: Effects of coarsening and artificial occupancy reduction.

full occupancy. Recall, that from the previous experiments it emerged that artificial occupancy reduction had a largely neutral effect on this kernel (see Table 6.1). It appears that this is also the case when coarsening is applied, which seems to account for small magnitude of the achieved additional performance improvement, as compared to the improvement that can be attained by applying thread coarsening on its own.

The `sc_memset` kernel experiences an 8% performance gain when both optimisations are applied in conjunction. The kernel’s task is to set all fields of an array to a specified value, performing one coalesced memory access per thread. This causes the kernel to be particularly simplistic, as well as short-running. There appears to be a parallel between this kernel and the matrix transpose kernel discussed in the preliminary case study, as both are simplistic in nature and both experience a combined performance gain of 6 – 8% when both optimisations are applied, the highest measured throughout our experiments. This may give rise to the question whether the approach of combining the optimisations works best for simple, memory-bound kernels but does not readily translate to more complex kernels.

Figure 6.4b shows the speedup over decreasing occupancy for differently coarsened versions of the `kmeans_swap` kernel. As before, a factor of 1 corresponds to 100% occupancy, and so on. The graph shows that, independent of the applied coarsening factor, an artificial reduction in occupancy has a very similar impact on the performance of each of the coarsened programs, such that the curves follow an approximately similar pattern. In a similar manner, applying thread coarsening at any given point on the occupancy axis has a similar impact throughout. At any given occupancy, the application of a higher coarsening factor causes a loss of performance,

with performance decreasing further with increasing coarsening factor.

Coarsening and artificial occupancy reduction therefore appear to work entirely independently for the `kmeans_swap` kernel based on these findings. The same observations can also be made for other kernels shown in Figure 6.4. The `dynproc` kernel is shown in Figure 6.4a. While coarsening has a clear performance impact on the kernel, it again appears that artificial occupancy reduction has the same (negative) effect on each coarsened version of the program. Note, that for this kernel coarsening factors of  $C \geq 4$  affect a natural reduction in occupancy, as shown in the graph.

The `sc_memset` kernel shown in Figure 6.4c was shown to have a ‘flat’ performance curve in previous experiments for the Pascal architecture in block-level coarsening mode (see Figure 5.5). The figure showed that for coarsening factors  $C \geq 2$  the speedup settles to be nearly constant at around 1.35x. This is also the case for the results shown in Figure 6.4c at full occupancy: As expected, the programs for  $C \geq 2$  all exhibit a performance improvement exceeding 1.3x at this point. As in previous examples, artificial occupancy reduction has approximately the same effect on all versions of the program, causing a reduction in performance in each case. However, what is particularly noticeable about the results shown in this graph is that the effect occurs later the higher the coarsening factor is. As such, the  $C = 1$  program suffers an immediate decline in performance of 6.2% when occupancy is reduced from 100% to 87.5%. In contrast, the  $C = 4$  program, for instance, only experiences a performance degradation when occupancy is reduced from 50% to 37.5%, at which point the performance declines by 14.5%. levels sink well below 50%. This appears to resemble the neutral effect that artificial occupancy reduction can have on the performance of uncoarsened code (see Section 6.3.3). However, the likely explanation at this point appears to be that coarsening cancels out the negative effects that artificial occupancy reduction can have on performance, an effect increased for higher coarsening factors, although coarsening factors of  $C > 2$  do not appear to result in any additional improvements in performance.

We have previously observed that artificial occupancy reduction has a neutral impact on the `bpnn_adjust` kernel. As shown in Figure 6.4d, this property holds for all coarsened versions of the program. The two optimisations therefore operate entirely independently for this kernel.

## 6.4 Discussion

The experiments conducted in this chapter have been designed to explore the trade-off between the performance effects of cache pressure, which can increase in some kernels with increasing coarsening factors, and artificial occupancy reduction.

Our experiments to establish whether the cache line re-use analysis can be used to identify kernels that are likely to benefit from artificial occupancy reduction have been inconclusive. In combination with the preliminary experiments we performed, it appears that the cache line re-use analysis can indeed identify kernels which may benefit from artificial occupancy reduction. However, the analysis by itself is not able to predict whether the optimisation will yield a ben-

efit. This would instead require a approximation of cache capacity and usage, or else a more sophisticated model of the cache with symbolic partial execution of code.

Our experiments have also shown that from the 19 kernels of Rodinia suite used in our experiments, only one kernel experiences a significant performance improvement, which is the `kmeans_swap` kernel with a speedup of 3.11x.

We have also explored whether it is beneficial to apply thread coarsening and artificial occupancy reduction in conjunction with each other. The idea is that artificial occupancy reduction may be used to ease cache pressure, thereby providing more headroom for the application of higher coarsening factors, which typically has the opposite effect on cache pressure. In our preliminary experiments we found that both optimisations applied individually may each have a negative effect on performance, but when applied in conjunction can increase the performance of a kernel, albeit by a small amount. We have been able to reconstruct these findings for some the Rodinia benchmarks, and in some cases obtained performance improvements of 3-8%.

We have also investigated whether there is any potential for interference between the two optimisations, since both are modifying the degree of parallelism of a program. Experiments with four kernels from the Rodinia suite suggest that thread coarsening and occupancy reduction operate largely independently or else complement each other, so as to cancel out each other's negative effects. In our case study on the matrix transpose kernel, it appeared that applying the worst performing coarsening factor in combination with the worst performing artificially reduced occupancy yielded a speedup of 6%. In contrast, we did not observe any case where – to borrow a common phrase – "two positives make a negative", in the sense that the two optimisations would be able to cancel out each other's positive benefits. From our experiments we presume that such a case does not in fact appear to exist. This means that negative interference between the two optimisation is not to be expected, despite their close similarity, as both effect a modification of the degree of parallelism of a program execution, and both can alter the pressure on the cache experienced by a program. In order to implement thread coarsening as a fully automated compiler pass, we considered the question of interference between these optimisation techniques as worth exploring, with artificial occupancy reduction being a likely candidate for potential interference due to its similarities to thread coarsening. In practice, fully automated thread coarsening would still be expected to follow the occupancy-based model and be guarded by the cache line re-use analysis, which does not appear to recommend kernels for coarsening that are also likely candidates to benefit from artificial occupancy reduction.



# Conclusions 7

Software performance on GPUs arguably depends making optimal use of the large-scale parallel processing power offered by them, while efficiently accessing data through a complex memory subsystem. In this thesis we have explored how software parallelism (i.e., the total number of threads launched) and hardware parallelism (i.e., the total number of threads simultaneously executed) can be re-balanced with respect to cache pressure to execute code with improved performance on the available architecture. We have shown how reducing software parallelism by applying coarsening, can increase performance when hardware parallelism, as measured by occupancy, is not reduced. We have described how this can be implemented as a fully automatic static compiler optimisation by re-compiling differently coarsened code and predicting their achievable occupancy through analysing the resource usage of each version. Moreover, we have demonstrated the usefulness of this approach for two different modes of coarsening and across three different, yet related, GPU architectures. Experiments have yielded a maximum speedup of 5.08x across a series of differently optimised reduction kernels, and an average speedup of 1.30x across a subset of the Rodinia benchmark suite.

In addition, we have explored how a light-weight static analysis, based on partial symbolic execution using memory access descriptors, can detect potential cache line re-use. This approximate method is significantly faster than a more substantial re-use distance analysis based on detailed cache models, which makes run-time execution (i.e., during run-time of the host program) entirely feasible. Moreover, we have demonstrated how this can be integrated into an existing coarsening pipeline to act as a filtering mechanism, identifying and preventing those kernels from being coarsened that might otherwise perform worse due to increased cache pressure. In practice, designers of code-generating DSLs may statically know the outcome of the analysis for generated kernels, without requiring to execute the analysis. To this end, we consider that our implementation has proved a useful platform for exploring what questions should be asked of a kernel implementation to determine its suitability for coarsening.

Moreover, we have explored how reducing hardware parallelism can counteract the effects of increasing cache pressure, and that reducing hardware and software parallelism in conjunction may cancel out negative effects of the other, resulting in improved overall performance. To this end, a case study on a matrix transpose kernel has revealed that the worst-performing coarsening factor and the worst-performing level of artificial occupancy reduction, which cause the program to run at 0.45x and 0.88x of its original performance, respectively, can achieve a 6.5% speedup when applied in conjunction. Our attempts to establish whether the cache line

re-use analysis can be used to identify kernels which may benefit from a reduction in hardware parallelism have proven inconclusive. While it appeared that this is theoretically possible, it seems that in practice the analysis would require the ability to reason about the cache pressure a kernel experiences before and after its occupancy is reduced.

## 7.1 Applications and future work

The work presented here is, by design, specific to NVidia GPU architectures and exploits information extracted from the NVidia compiler. The idea of making coarsening decisions on the basis of resource usage could, in principle, be adapted to other GPU platforms, such as those of AMD and ARM. Although there may not be a direct corresponding notion of thread occupancy on other platforms, other resource utilisation measures may prove to be applicable and it will be interesting to explore this further.

A remaining challenge is to automatically reason about kernel behaviour at lower occupancy. It is well known that peak performance of some kernels occurs at occupancies well below 100% (see [94]). The problem becomes one of detecting the characteristics of such kernels and then determining a suitable reduced occupancy metric.

The correctness of our transformations is also an interesting issue. Given a formal semantics for the OpenCL (or similar) language it seems entirely feasible to prove that our transformations preserve the meaning of the program by formulating, and proving, a suitable correspondence theorem. The somewhat subtle issue of block semantics (Section 3.6) would be interesting to capture, although this lies well beyond the objectives of this thesis.

There are clearly improvements that can be made to the compilation pipeline. For example, having compiled a kernel for a particular configuration and coarsening factor, it would be straightforward to cache the result in order to avoid re-compiling the same configuration at a later time. With that said, the low cost of our analysis means that dynamic compilation of kernels, which is the standard model in the OpenCL API, is completely tractable.

The current implementation could be extended by the ability to coarsen in multiple dimensions; the selection of the stride parameter or the dimension in which coarsening is applied could also be automated, as well as the selection of coarsening mode. The latter could be achieved by identifying which code regions of a given kernel will be duplicated in either coarsening mode, and comparing the resulting shared instructions, before applying the transformation.

In the context of cache line re-use analysis, exploring the adaptation of polyhedral analysis may likely prove beneficial for understanding loop nests. In its current form, the analysis does not support thread divergence. The existing analysis could be combined with a benefit analysis, e.g. based on a simple cost-model, to improve accuracy of what kernels should be excluded from coarsening. The analysis could also be extended to reason about how the number of re-used cache lines will scale when coarsening is applied in a given dimension. In combination with a benefit analysis this could allow kernels to be coarsened which are currently excluded from coarsening, including data-dependent kernels. Projecting the number of accessed as well

as re-used cache lines per thread, per thread block, and across the device might approximate the cache pressure experienced by a program, although a more substantial and detailed re-use distance analysis may prove necessary. Approximating the cache pressure experienced by a program may provide insight at compile-time as to whether artificial occupancy reduction of a kernel should be attempted, possibly in connection with thread coarsening.

The application of artificial occupancy reduction could be decided based on auto-tuning or run-time profiling, and this information could be cached for future executions. Alternatively, we can imagine that using a more detailed caching analysis (see above) make it possible to reason about applying artificial occupancy reduction in conjunction with thread coarsening at compile-time.

## **7.2 Closing remarks**

Optimising software performance is vital to realise the available processing. Automating performance optimisations is arguably more important still for software written by programmers who may not have expert knowledge of the intricacies of specific GPU architectures and their complex memory subsystems. Fully automatic optimisations reduce the amount of non-functional performance-related code explicitly programmed into kernel bodies, thereby increasing the performance portability of software to and from specific platforms. We believe that an integrated approach of studying micro-architectures, program behaviour, and specific performance optimisations has yielded valuable insight into how to make thread coarsening a fully automated compile-time optimisation. Our hope is that our techniques will be of use to GPU programmers and DSL designers, and that it will enable thread coarsening to be implemented in commercially available compilers.



# Bibliography

- [1] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. *PetaBricks: a language and compiler for algorithmic choice*, volume 44. ACM, 2009.
- [2] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The landscape of parallel computing research: A view from Berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.
- [3] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubitowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, et al. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009.
- [4] John Backus. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Communications*, 1978.
- [5] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174. IEEE, 2009.
- [6] Cedric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 7–16. IEEE Computer Society, 2004.
- [7] F Brett Berlin. Seymour Cray, 1925-1996 [In Memoriam]. *IEEE Computational Science and Engineering*, 3(4):90–92, 1996.
- [8] Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. GPU-Verify: A Verifier for GPU Kernels. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '12*, pages 113–132, 2012. ISBN 978-1-4503-1561-6.

- [9] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, 2008.
- [10] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *ACM transactions on graphics (TOG)*, volume 23, pages 777–786. ACM, 2004.
- [11] Martin Campbell-Kelly and Michael Roy Williams. *The Moore School lectures: theory and techniques for design of electronic digital computers*, volume 9. Mit Press, 1985.
- [12] Daniel Cederman and Philippas Tsigas. A practical quicksort algorithm for graphics processors. In *European Symposium on Algorithms*, pages 246–258. Springer, 2008.
- [13] Manuel MT Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pages 3–14. ACM, 2011.
- [14] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. Ieee, 2009.
- [15] Guoyang Chen, Bo Wu, Dong Li, and Xipeng Shen. PORPLE: An extensible optimizer for portable data placement on GPU. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 88–100. IEEE Computer Society, 2014.
- [16] Nathan Chong, Alastair F. Donaldson, Paul H.J. Kelly, Jeroen Ketema, and Shaz Qadeer. Barrier Invariants: A Shared State Abstraction for the Analysis of Data-dependent GPU Kernels. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 605–622, 2013. ISBN 978-1-4503-2374-1.
- [17] Peter Collingbourne, Alastair F Donaldson, Jeroen Ketema, and Shaz Qadeer. Interleaving and lock-step semantics for analysis and verification of GPU kernels. In *European Symposium on Programming*, pages 270–289. Springer, 2013.
- [18] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. End-to-end deep learning of optimization heuristics. In *Parallel Architectures and Compilation Techniques (PACT), 2017 26th International Conference on*, pages 219–232. IEEE, 2017.
- [19] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph.

- ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.
- [20] William J Dally, Francois Labonte, Abhishek Das, Patrick Hanrahan, Jung-Ho Ahn, Jayanth Gummaraju, Mattan Erez, Nuwan Jayasena, Ian Buck, Timothy J Knight, et al. Merrimac: Supercomputing with streams. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 35. ACM, 2003.
  - [21] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74. ACM, 2010.
  - [22] Jim Demmel, Jack Dongarra, Victor Eijkhout, Erika Fuentes, Antoine Petit, Rich Vuduc, R Clint Whaley, and Katherine Yelick. Self-adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93(2):293–312, 2005.
  - [23] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 353–364. ACM, 2010.
  - [24] Chen Ding and Yutao Zhong. Predicting Whole-program Locality Through Reuse Distance Analysis. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, PLDI '03*, pages 245–257, 2003.
  - [25] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*, 38(8):391–407, 2012.
  - [26] Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri, and Shihjong Kuo. Intel AVX: New frontiers in performance improvements and energy efficiency. *Intel white paper*, 19: 20, 2008.
  - [27] The International Technology Roadmap for Semiconductors. Executive summary. Technical report, 2005 and 2007. <http://www.itrs2.net>.
  - [28] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, et al. Milepost gcc: Machine learning enabled self-tuning compiler. *International journal of parallel programming*, 39(3):296–327, 2011.
  - [29] David Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.
  - [30] Peter N Glaskowsky. NVIDIA's Fermi: the first complete GPU computing architecture. *White paper*, 18, 2009.

- [31] Herman H Goldstine and Adele Goldstine. The electronic numerical integrator and computer (eniac). *Mathematical Tables and Other Aids to Computation*, 2(15):97–110, 1946.
- [32] Naga K Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High performance discrete fourier transforms on graphics processors. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–12. Ieee, 2008.
- [33] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cava-zos. Auto-tuning a high-level language targeted to GPU codes. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10. IEEE, 2012.
- [34] Peter Greenhalgh. Big. little processing with arm cortex-a15 & cortex-a7. *ARM White paper*, 17, 2011.
- [35] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly-Polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, volume 2011, 2011.
- [36] Khronos OpenCL Working Group et al. The OpenCL specification. *version*, 1(29):8, 2008.
- [37] Mark Harris. Optimizing CUDA. *SC07: High Performance Computing With CUDA*, 2007.
- [38] Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel prefix sum (scan) with CUDA. *GPU gems*, 3(39):851–876, 2007.
- [39] Douglas Rayner Hartree. The ENIAC, an electronic computing machine. *Nature*, 158 (4015):500, 1946.
- [40] Yingchao Huang and Dong Li. Performance Modeling for Optimal Data Placement on GPU with Heterogeneous Memory Systems. In *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*, pages 166–177. IEEE, 2017.
- [41] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.
- [42] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Ramin-der Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12. ACM, 2017.
- [43] *SPIR 1.2 Specification for OpenCL*. Khronos Group, 2013.

- [44] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. *arXiv preprint arXiv:1801.01203*, 2018.
- [45] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.
- [46] Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, et al. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *ACM SIGARCH computer architecture news*, 38(3):451–460, 2010.
- [47] Yuxi Liu, Zhibin Yu, Lieven Eeckhout, Vijay Janapa Reddi, Yingwei Luo, Xiaolin Wang, Zhenlin Wang, and Chengzhong Xu. Barrier-Aware Warp Scheduling for Throughput Processors. In *Proceedings of the 2016 International Conference on Supercomputing*, page 42. ACM, 2016.
- [48] Fabio Luporini, Ana Lucia Varbanescu, Florian Rathgeber, Gheorghe-Teodor Bercea, J Ramanujam, David A Ham, and Paul HJ Kelly. COFFEE: an Optimizing Compiler for Finite Element Local Assembly. *arXiv preprint arXiv:1407.0904*, 2014.
- [49] Alberto Magni, Christophe Dubach, and Michael FP O’Boyle. A large-scale cross-architecture evaluation of thread-coarsening. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, page 11. ACM, 2013.
- [50] Alberto Magni, Christophe Dubach, and Michael O’Boyle. Exploiting GPU Hardware Saturation for Fast Compiler Optimization. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, page 99. ACM, 2014.
- [51] Alberto Magni, Christophe Dubach, and Michael O’Boyle. Automatic optimization of thread-coarsening for graphics processors. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 455–466. ACM, 2014.
- [52] Geoffrey Mainland and Greg Morrisett. Nikola: Embedding compiled GPU functions in Haskell. In *Proceedings of the Third ACM Symposium on Haskell (Haskell ’10)*, pages 67–78, September 2010.
- [53] Ami Marowka. Back to thin-core massively parallel processors. *Computer*, 44(12):49–54, 2011.

- [54] Duane Merrill and Andrew Grimshaw. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters*, 21(02):245–272, 2011.
- [55] Paulius Micikevicius. GPU performance analysis and optimization. In *GPU technology conference*, volume 84, 2012.
- [56] Simon Moll. Decompilation of LLVM IR. *Master’s thesis*, 2011.
- [57] Simon Moll, Johannes Doerfert, and Sebastian Hack. Input Space Splitting for OpenCL. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 251–260. ACM, 2016.
- [58] Kedar S Namjoshi and Nimit Singhanian. Loopy: Programmable and formally verified loop transformations. In *International Static Analysis Symposium*, pages 383–402. Springer, 2016.
- [59] Cedric Nugteren, Gert-Jan van den Braak, Henk Corporaal, and Henri Bal. A detailed GPU cache model based on reuse distance theory. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 37–48. IEEE, 2014.
- [60] CUDA NVidia. Programming guide, 2010.
- [61] *Whitepaper - NVIDIA’s Next Generation CUDA Compute Architecture: Fermi*. NVIDIA Corporation, 2009.
- [62] *TUNING CUDA APPLICATIONS FOR KEPLER*. NVIDIA Corporation, 2012.
- [63] *Whitepaper - NVIDIA’s Next Generation CUDA Compute Architecture: Kepler GK110*. NVIDIA Corporation, 2012.
- [64] *Whitepaper - NVIDIA GeForce GTX 980*. NVIDIA Corporation, 2014.
- [65] *Whitepaper - NVIDIA Tesla P100*. NVIDIA Corporation, 2016.
- [66] *Whitepaper - NVIDIA GeForce GTX 1080*. NVIDIA Corporation, 2016.
- [67] *CUDA COMPILER DRIVER NVCC*. NVIDIA Corporation, 2018.
- [68] *TUNING CUDA APPLICATIONS FOR PASCAL*. NVIDIA Corporation, 2018.
- [69] *PARALLEL THREAD EXECUTION ISA - Application Guide*. NVIDIA Corporation, 2018.
- [70] Mark S Peercy, Marc Olano, John Airey, and P Jeffrey Ungar. Interactive multi-pass programmable shading. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 425–432. ACM Press/Addison-Wesley Publishing Co., 2000.

- [71] Phitchaya Mangpo Phothilimthana, Jason Ansel, Jonathan Ragan-Kelley, and Saman Amarasinghe. Portable performance on heterogeneous architectures. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 431–444. ACM, 2013.
- [72] Louis-Noël Pouchet. Polybench: The polyhedral benchmark suite. URL: <http://web.cse.ohio-state.edu/pouchet.2/software/polybench/>, 2012.
- [73] Florian Rathgeber, Graham R Markall, Lawrence Mitchell, Nicolas Lorient, David A Ham, Carlo Bertolli, and Paul HJ Kelly. PyOP2: A high-level framework for performance-portable simulations on unstructured meshes. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1116–1123. IEEE, 2012.
- [74] Florian Rathgeber, David A Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew TT McRae, Gheorghe-Teodor Bercea, Graham R Markall, and Paul HJ Kelly. Firedrake: automating the finite element method by composing abstractions. *ACM Transactions on Mathematical Software (TOMS)*, 43(3):24, 2017.
- [75] Tiark Rompf and Martin Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE '10*, pages 127–136, 2010.
- [76] Shane Ryoo, Christopher I Rodrigues, Sam S Stone, Sara S Baghsorkhi, Sain-Zee Ueng, John A Stratton, and Wen-mei W Hwu. Program optimization space pruning for a multithreaded GPU. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 195–204. ACM, 2008.
- [77] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore GPUs. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10. IEEE, 2009.
- [78] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D Owens. Scan primitives for GPU computing. In *Graphics hardware*, volume 2007, pages 97–106, 2007.
- [79] Shubhabrata Sengupta, Mark Harris, and Michael Garland. Efficient parallel scan algorithms for GPUs. *NVIDIA, Santa Clara, CA, Tech. Rep. NVR-2008-003*, 1(1):1–17, 2008.
- [80] Jaewook Shin, Mary W Hall, Jacqueline Chame, Chun Chen, Paul F Fischer, and Paul D Hovland. Speeding up nek5000 with autotuning and specialization. In *Proceedings of the 24th ACM International Conference on Supercomputing*, pages 253–262. ACM, 2010.
- [81] H Shojania and B Li. Cache miss analysis for gpu programs based on stack distance profile. In *Proc. IEEE Int'l Conference on Distributed Computing Systems (ICDCS)*, 2009.

- [82] Mark Silberstein, Assaf Schuster, Dan Geiger, Anjul Patney, and John D Owens. Efficient computation of sum-products on GPUs through software-managed cache. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 309–318. ACM, 2008.
- [83] Tyler Sorensen, Alastair F. Donaldson, Mark Batty, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. Portable Inter-workgroup Barrier Synchronisation for GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016*, pages 39–58, 2016.
- [84] Nicolai Stawinoga and Tony Field. Predictable Thread Coarsening. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(2):23, 2018.
- [85] J.A. Stratton, N. Anssari, C. Rodrigues, I-Jui Sung, N. Obeid, Liwen Chang, G.D. Liu, and W. Hwu. Optimization and architecture effects on GPU computing workload performance. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10, May 2012.
- [86] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127, 2012.
- [87] Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):134, 2014.
- [88] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs's journal*, 30(3):202–210, 2005.
- [89] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: Using Data Parallelism to Program GPUs for General-purpose Uses. *SIGOPS Oper. Syst. Rev.*, 40(5):325–335, October 2006. ISSN 0163-5980.
- [90] Chris J Thompson, Sahngyun Hahn, and Mark Oskin. Using modern graphics architectures for general-purpose computing: a framework and analysis. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 306–317. IEEE Computer Society Press, 2002.
- [91] Sain-Zee Ueng, Melvin Lathara, Sara S Baghsorkhi, and W Hwu Wen-mei. CUDA-Lite: Reducing GPU Programming Complexity. In *LCPC*, volume 8, pages 1–15. Springer, 2008.

- [92] Swapneela Unkule, Christopher Shaltz, and Apan Qasem. Automatic Restructuring of GPU Kernels for Exploiting Inter-thread Data Locality. In *Proceedings of the 21st International Conference on Compiler Construction, CC'12*, pages 21–40, 2012.
- [93] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):54, 2013.
- [94] Vasily Volkov. Better Performance at Lower Occupancy. Presentation, 2010. <http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf>.
- [95] Vasily Volkov and James W Demmel. Benchmarking GPUs to tune dense linear algebra. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–11. IEEE, 2008.
- [96] Dongwei Wang and Weijun Xiao. A reuse distance based performance analysis on GPU L1 data cache. In *Performance Computing and Communications Conference (IPCCC), 2016 IEEE 35th International*, pages 1–8. IEEE, 2016.
- [97] Yuan Wen, Zheng Wang, and Michael FP O’Boyle. Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms. In *High Performance Computing (HiPC), 2014 21st International Conference on*, pages 1–10. IEEE, 2014.
- [98] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.
- [99] Samuel Williams, Leonid Oliker, Jonathan Carter, and John Shalf. Extracting ultra-scale lattice Boltzmann performance via hierarchical and distributed auto-tuning. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12. IEEE, 2011.
- [100] Nicholas Wilt. *The CUDA handbook: A comprehensive guide to GPU programming*. Pearson Education, 2013.
- [101] Michael E. Wolf and Monica S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, PLDI '91*, pages 30–44, 1991.
- [102] Mason Woo, Jackie Neider, Tom Davis, Dave Shreiner, et al. *OpenGL programming guide*, volume 3. Addison-wesley Reading, 1999.
- [103] Xiaochun Ye, Dongrui Fan, Wei Lin, Nan Yuan, and Paolo Ienne. High performance comparison-based sorting algorithm on many-core GPUs. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10. IEEE, 2010.

- [104] Ying Zhang, Lu Peng, Bin Li, Jih-Kwon Peir, and Jianmin Chen. Architecture comparisons between Nvidia and ATI GPUs: Computation parallelism and data communications. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 205–215. IEEE, 2011.