Analysis and Parameter Prediction of Compiler Transformation for Graphics Processors

Alberto Magni

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2015

Abstract

In the last decade graphics processors (GPUs) have been extensively used to solve computationally intensive problems. A variety of GPU architectures by different hardware manufacturers have been shipped in a few years. OpenCL has been introduced as the standard cross-vendor programming framework for GPU computing. Writing and optimising OpenCL applications is a challenging task, the programmer has to take care of several low level details. This is even harder when the goal is to improve performance on a wide range of devices: OpenCL does not guarantee performance portability.

In this thesis we focus on the analysis and the portability of compiler optimisations. We describe the implementation of a portable compiler transformation: thread-coarsening. The transformation increases the amount of work carried out by a single thread running on the GPU. The goal is to reduce the amount of redundant instructions executed by the parallel application.

The first contribution is a technique to analyse performance improvements and degradations given by the compiler transformation, we study the changes of hardware performance counters when applying coarsening. In this way we identify the root causes of execution time variations due to coarsening.

As second contribution, we study the relative performance of coarsening over multiple input sizes. We show that the speedups given by coarsening are stable for problem sizes larger than a threshold that we call saturation point. We exploit the existence of the saturation point to speedup iterative compilation.

The last contribution of the work is the development of a machine learning technique that automatically selects a coarsening configuration that improves performance. The technique is based on an iterative model built using a neural network. The network is trained once for a GPU model and used for several programs. To prove the flexibility of our techniques, all our experiments have been deployed on multiple GPU models by different vendors.

Lay Summary

In the last decade processors originally designed to compute the color of the pixels of a computer screen (GPUs) have also been used to solve computationally intensive problems. A variety of GPU architectures by different hardware manufacturers have been shipped in a few years. The OpenCL programming language has been introduced to provide a standard programming framework for GPUs.

Writing and optimising OpenCL applications is a challenging task, the programmer has to take care of several low level details. This is even harder when the goal is to improve performance on a wide range of devices. OpenCL applications do not perform equally well on devices of different types.

In this thesis we focus on the analysis and the portability of code transformations. We describe the implementation of a portable compiler transformation: thread-coarsening. Its goal is to reduce the amount of redundant instructions executed by the parallel application.

We first develop a technique to automatically analyse performance improvements and degradations given by thread-coarsening. We also developed machine learning technique that automatically selects a coarsening configuration that improves performance on a four different GPU models.

Acknowledgements

I would like to thank first my advisor Michael O'Boyle for his constant support, he taught me how to get things done. I am greatly indebted towards my second supervisor Christophe Dubach. He gave innumerable suggestions about life as a PhD student.

I would like to thank Dominik, Thibaut, Toomas, Erik, Michel, Yuan, Tobias, Alex, Chris, Sid, Kiran and all the members of the CARD group. A special thank to my Greek friends Vasilis, Konstantina, Christos and to Juan, who almost speaks Greek.

I would like to thank Anton Lokhmotov and ARM for hosting me for three months in Cambridge. I thank the colleagues I met there: Murat, Georgia and Alexey. I had a great time discussing with Cedric about GPU caches and thread scheduling.

I am grateful to Vinod Grover, Sean Lee and Mahesh Ravishankar at Nvidia for giving me the opportunity to work with them. I would like also to thank my colleague Nhat and my flatmate in Redmond Alexander.

I thank my office mate Andrea for sharing his brownies with me and for hosting me in Munich. My Italian friends in Edinburgh helped me free my mind from research and fill my belly with food. I would like to thank Tiziana, Manuela, Gianpaolo, Marta, Marina, Fabio, Simona and Cesare. A special thank to Marcello for all our endless discussions about computers and for not letting me down in San Francisco. I thank Ettore for convincing me to come to Edinburgh to do a PhD and for driving me around Silicon Valley. I would like to thank Antonio and Yanick for our trips to Berlin and for hosting me in California.

My parents Annalisa and Danilo always had words of encouragement and support.

Finally, thank you Giada for being close to me.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Some of the material used in this thesis has been published in the following papers:

- Alberto Magni, Christophe Dubach, Michael F.P. O'Boyle.
  "A Large-Scale Cross-Architecture Evaluation of Thread-Coarsening".
  In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC), November 2013.

- Alberto Magni, Christophe Dubach, Michael F.P. O'Boyle.
  "Exploiting GPU Hardware Saturation for Fast Compiler Optimization".
  In Proceedings of the Workshop on General Purpose Processing Using GPUs (GPGPU), March 2014.

- Alberto Magni, Christophe Dubach, Michael F.P. O'Boyle.
  "Automatic Optimization of Thread-Coarsening for Graphics Processors".
  In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), August 2014.

(*Alberto Magni*)

Table of Contents

Chapter 1

Introduction

For decades the performance of computing processors has been improved by increasing clock speeds. The side-effect of this trend was the consequent increment of power consumption and operating temperatures. *"Power Wall"* is the term used to identify the impossibility to scale up processor frequency further due to heat constraints. The limits of frequency scaling became apparent to the broad public in 2004 when Intel stopped the development of two architectures: Tejas (the would-be successor of the Pentium 4) and Jayhawk. [1] These two processors were both designed for high operating frequencies. The computer architecture industry then aimed at lowering single core performance and integrating multiple cores on the same die. Heterogeneous devices have then been introduced with the goal of improving power efficiency. In this scenario, fully programmable graphics processors (GPUs) have become mainstream for solving highly parallel tasks. The application of GPUs for computational workloads has been called General Purpose computation on Graphics Processors (GPGPU). With respect to CPUs, GPUs offer much higher floating point performance per Watt, making them suitable for both embedded and high performance computing.

GPUs are radically different from CPUs. CPU cores are designed to minimise the latency of individual threads. GPUs, instead, are massively parallel machines that aim at minimising the latency of the application by instantiating thousands of parallel threads, thus maximising the throughput, *i.e.*, the number of threads terminating per unit of time. CPUs are mounted with a small number (usually less than 10) of powerful cores that aim at improving single-thread performance with branch predictors, reorder buffers and several levels of data and instruction caches. The primal way to improve performance on the CPU is by exploiting the maximum amount of instruction level parallelism (ILP) available. GPUs have instead a much larger number (thousands) of far simpler cores. Such cores are often aggregated hierarchically into clusters (called streaming multiprocessors by Nvidia). These clusters implement the logic to schedule threads onto their cores. To simplify the design of the architecture all the cores in

[1] http://www.nytimes.com/2004/05/08/business/intel-halts-development-of-2-new-microprocessors.html

a cluster share the same instruction pointer, so that hardware threads running on a processor cluster execute in lockstep. GPU cores have a much larger register file with respect to CPUs, this is to enable the concurrent execution of a large number of threads. GPUs achieve high performance through the synchronized scheduling of multiple threads to hide memory latencies, this is called thread level parallelism (TLP). In summary, CPUs strive to keep their pipeline busy with instructions, while GPUs aim to keep their clusters of cores busy with threads.

The wide success of GPUs in delivering high performance for non-graphics applications has lead to the introduction in the market of many different hardware devices in a few years. As a result the field of GPU architecture has rapidly evolved: many vendors have come into play. Vendor diversity implied also diversity in the architecture and ISA designs.

Unfortunately, heterogeneity hinders code portability across devices. The first GPGPU programming frameworks to be introduced where Close To Metal by ATI/AMD (2006) [56] and shortly after CUDA by Nvidia (2007) [98]. Being developed by two different companies these two programming models were incompatible. A programmer that wanted to deploy code for the two architecture had to write the application twice.

In 2008 the Khronos consortium introduced a new GPGPU API and language called OpenCL [67]. This is the result of the efforts of multiple companies to define a cross-platform API for GPGPU. Being developed by a wide range of hardware vendors, the primal design goal of OpenCL is to ensure code portability, *i.e.*, a single program can successfully and transparently execute on devices of multiple manufacturers. This is a major productivity boost, allowing programmers to extensively reuse code.

The OpenCL API manages the GPU as an external accelerator and it is responsible for scheduling parallel work on the GPU. It can allocate buffers on the graphics memory and transfer data between the CPU and the GPU.

The efforts to define a portable standard fall short of achieving performance portability. This means that different devices require different optimisation techniques. Performance portability affects all the aspects of the computation: memory management, data transfers between CPU and GPU and compiler optimisations to apply to GPU code. For example, improving memory transfers between CPU and GPU must take into account the device-specific memory system. Software pipelining [58] can successfully improve performance for GPUs that do not share main memory with the CPU. The same transformation might have little or no benefit on systems that share main memory between CPU and GPU. The same applies to other code transformations, such as the use of the GPU texture memory [138]. The relative performance of texture memory and texture caches with respect to main GPU memory changes across hardware vendors. Optimisation techniques that aim at adapting to the underlying hardware to extract the maximum performance are various and touch all the aspects of the computation.

In this work we focus on the performance portability of the compiler optimisation settings for GPU code (called the *kernel* function). To achieve high performance in this context the programmer can either optimise code by hand for each device or rely on the compiler to perform aggressive optimisations. The first approach is impractical, since optimising parallel programs is extremely demanding. On the other side, the development of an optimising compiler for multiple architectures is a challenging task for the compiler engineer. The heuristics that control the transformations must be tuned for all the devices of interest. This type of tuning can by applied to high level transformations *e.g.* loop unrolling, tiling, fusion and fission. Back-end optimisations, instead, are inherently non-portable, our work does not deal with those.

To showcase our analysis and prediction techniques we use a compiler transformation specifically designed for a data parallel language: thread-coarsening. Thread-coarsening increases the amount of work performed by a single thread and, to preserve the semantics of the application, reduces the amount of threads concurrently running on the GPU. This is done by logically merging the computation of multiple threads together. By applying this transformation it is possible to reduce the overhead of redundant computation performed by the application across threads. Previous works on coarsening [138, 125] lack implementation in an optimising compiler and rigorous analysis of effects on performance. In this thesis we provide a complete implementation, analysis and a methodology to select parameters for the coarsening transformation that can improve the performance of OpenCL kernels on four GPU architectures.

1.1 Contributions

This thesis makes the following contributions:

1. We provide implementation of the thread-coarsening compiler transformation. The transformation is implemented using the LLVM compiler infrastructure and it works on LLVM intermediate representation. The transformation makes sure to preserve high performance memory accesses by address remapping. We also present a tool that we call `SymEngine` that analyses GPU memory access patterns. The source code of our compiler passes is available as open-source [83, 84]. The design and the implementation of all the compiler passes are documented in chapter 4.

2. We show the performance improvements given by thread-coarsening across four architectures [85]. We show how hardware performance counters change when applying coarsening on four devices and we propose a technique based on regression trees to simplify the task of performance analysis. The trees automatically identify the counters that are affected the most by the transformation and correlate their run-time changes with execution speedup. This method is helpful in the identification of the root causes of performance differences among program versions. Chapter 5 presents this work.

3. We study the impact of the problem input size on the performance given by thread-coarsening [87]. In particular we analyse how the best coarsening configuration changes across input sizes. We show that the best coarsening configuration remains the same for input sizes bigger than a threshold that is benchmark- and platform-specific. We call this threshold *saturation point*. We propose to leverage saturation to speedup iterative compilation. This study is reported in Chapter 6.

4. We developed a machine learning technique to attempt to predict the optimal coarsening factor for a given program and architecture [86]. We also present a heuristic to estimate the best thread-remapping policy to preserve the coalescing of memory accesses. Thus we improve over previous chapter by automatically selecting the coarsening factor and stride without the need of iterative compilation. The policy relies on the results of symbolic execution of the kernel code using our tool `SymEngine`. This work is described in chapter 7.

1.2 Thesis organization

This thesis is organized as follows:

Chapter 2 provides the technical background needed to understand the remainder of the work. It includes: a description of GPGPU computing and OpenCL in specific, an overview of the architectural structure of Nvidia and AMD GPUs, we describe the LLVM compiler infrastructure and the relevant machine learning techniques employed in the thesis.

Chapter 3 presents the related work. It introduces and critically discuss prior work about relevant compiler transformations, performance modelling and optimisation for GPU, and the use of machine learning in compiler tuning.

Chapter 4 describes the design and the implementation of thread-coarsening. It presents the design choices made in the construction of the compiler infrastructure, the implementation of divergence analysis, the coarsening transformation and the symbolic execution engine `SymEngine` for memory access pattern analysis.

Chapter 5 presents the results of performance analysis. It shows how coarsening affects performance of 17 GPGPU benchmarks on four GPU architectures. The chapter shows how regression trees are used to ease the task of cross-platform performance analysis relying on hardware performance counters.

Chapter 6 studies how the coarsening transformation affect run-time performance across multiple problem input sizes. It shows that the best coarsening configuration remains constant across problem sizes bigger than a lower-bound that we call minimum saturation point. We describe how to exploit this property to speed up iterative compilation of thread-coarsening.

Chapter 7 describes the prediction technique used to look for the best coarsening factor and stride factor. We employ a neural network to decide whether to apply coarsening or not. By iteratively querying the network we show that is possible to improve performance on the four architectures taken into account. We also describe our heuristic to remap threads so to preserve coalescing of memory accesses.

Chapter 8 concludes the thesis by summarising the contributions, the limitations and by describing possible future direction of research.

Chapter 2

Background

This chapter presents the relevant technical background needed to understand the remainder of the thesis. Section 2.1 introduces the most important concepts of the OpenCL programming language. Section 2.2 gives an overview of hardware features of Nvidia and AMD GPUs. Section 2.3 describes the LLVM compiler infrastructure. Section 2.4 explains how we collected hardware performance counter on Nvidia and AMD GPUs. Section 2.5 lists the benchmarks that we used for experiments in our work. Section 2.6 presents the machine learning techniques that we used: regression trees, neural networks and principal components analysis. Section 2.7 describes how we evaluate the performance of our machine learning methods. Section 2.8 concludes the chapter.

2.1 OpenCL

OpenCL [67] is the standard cross-platform data parallel programming model for multi-core hardware. The OpenCL specification is extremely flexible, giving vendors the possibility to support many different types of devices:

- Multicore CPUs, *e.g.* Intel i7 or ARM Cortex family.
- Graphics Processors:
  - Discrete GPUs (the CPU and the GPU have different physical memories), *e.g.* Nvidia GTX, Nvidia Tesla, AMD Radeon
  - Integrated GPUs (the CPU and the GPU share the same physical memory), *e.g.* ARM Mali, Intel HD series, AMD Kaveri APUs.
- Accelerators, *e.g.* Xeon Phi.

An OpenCL application compliant with the standard is guaranteed to correctly execute on any of these devices. This improves the productivity developers who can write an application

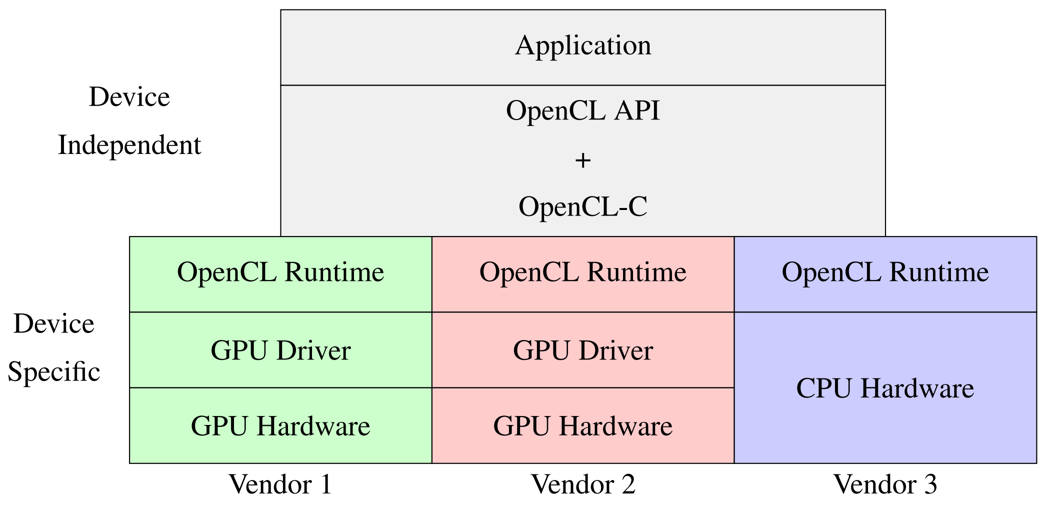


Figure 2.1: OpenCL abstraction layers. A single application can be deployed on multiple platforms.

once and run it to many devices. Figure 2.1 describes the OpenCL abstraction layers, clearly identifying the portable and non-portable layers of the stack. In the figure, the OpenCL runtime is a library that provides a C interface to schedule computation and to manage buffers on the computing device. In the case of GPUs these low-level tasks are performed by the device driver. Notice that the non-portable components are all provided by the hardware vendors, the application developer does not have to deal with device-specific issues to achieve the expected functionalities.

The OpenCL standard describes both an application programming interface and a programming language called OpenCL-C. OpenCL programs are made of two components: a *host* program written in C/C++ which run on the CPU and a *device* program written in OpenCL-C.

Host code The host code is a C program that implements the main functionalities of the application. It uses the OpenCL API functions to schedule the compute-heavy sections of the code to the parallel computing device available. The OpenCL application is initialized with the creation of a Context, a container for the following data objects:

- Devices: set of available computing devices.
- Kernels: functions running on the parallel devices.
- Program Object: instance of the kernel function containing the code compiled for the target device.
- Memory Objects: device memory buffers used to communicate with the host.

```
kernel void transposeMatrix(global float* input,
                            global float* output,
                            size_t width, size_t height) {
  size_t column = get_global_id(0);
  size_t row = get_global_id(1);
  size_t inputIndex = row * width + column;
  size_t outputIndex = height * column + row;
  output[outputIndex] = input[inputIndex];
}
```

Figure 2.2: Example of an OpenCL kernel written in OpenCL-C.

All these components are created by the programmer when setting up the computation. The typical work-flow of an OpenCL application is the following:

1. Select the computing *platform* to use, a platform is a collection of devices from the same hardware vendor.
2. Create a *context* for the chosen platform.
3. Select a computing device from the ones available for the chosen platform.
4. Create the *command queue*. The queue is responsible for sending commands to the device, so to trigger data transfers and kernel invocations.
5. Create the *program* by compiling the kernel source code at run-time.
6. Allocate the buffers for data input/output on the device.
7. Transfer data from the host to the device.
8. Launch kernel execution
9. Transfer data from the device to the host.
10. Release all the OpenCL resources.

Device code The device code defines a *kernel* function, which is executed by the parallel device on each data point of the iteration space. The kernel is written in OpenCL-C, a subset of C99 with extensions to support multiple address spaces, vector data types and textures. Figure 2.2 shows an example of an OpenCL kernel that transposes the matrix `input` into `output`. Each parallel thread reads and writes a single element of the matrix. The kernel is compiled at run-time calling the API function `clBuildProgram` which invokes the vendor-specific compiler.

Iteration space definition Kernels are instantiated within a multi-dimensional index-space, specified at run-time, called `NDRange` space. A thread executing the kernel function in the `NDRange` space is called a *work-item*. Each work-item is uniquely identified by a tuple called the *global-id*. Work-items are associated to the cores in *work-group*s whose size is specified at the kernel launch time. Each work-group is identified by a *group-id*. Within work-groups work-items are identified by *local-id*s. The following list specifies all the names we use to identify work-items, work-groups and their sizes. We will use the same notation throughout the thesis.

- `NDRange` space size: (G_x, G_y)
- Size of each work-group: (S_x, S_y)
- Number of work-groups: (W_x, W_y)
- Work-item global indexes: (g_x, g_y)
- Work-group global indexes: (w_x, w_y)
- Work-item local indexes, inside the work-group: (l_x, l_y)

The following formulas correlate together the work-item identifiers and the sizes of the iteration space:

- $(g_x, g_y) = (w_x \cdot S_x + l_x, w_y \cdot S_y + l_y)$
- $(W_x, W_y) = (G_x/S_x, G_y/S_y)$
- $(w_x, w_y) = ((g_x - s_x)/S_x, (g_y - s_y)/S_y)$

For the reminder of the thesis we consider only one- and two- dimensional kernel. We map direction 0 to the x coordinate, while direction 1 to y.

2.1.1 Device abstraction

In order to support a wide range of devices the OpenCL standard provides a high level abstraction of the hardware. Figure 2.3 visualizes the abstract representation of an OpenCL device. A parallel device is made of multiple *compute units* each containing several *processing elements* (PE). Threads are scheduled in work-groups: all the work-items in work-group are associated to a single compute unit. The individual work-items are then executed by the processing elements. Each PE has a segment of private memory, not visible to the other processing elements. All the PEs in a compute unit share a piece of low-latency memory called *local memory*. Without atomic instructions the only way for work-items to safely communicate among each other is

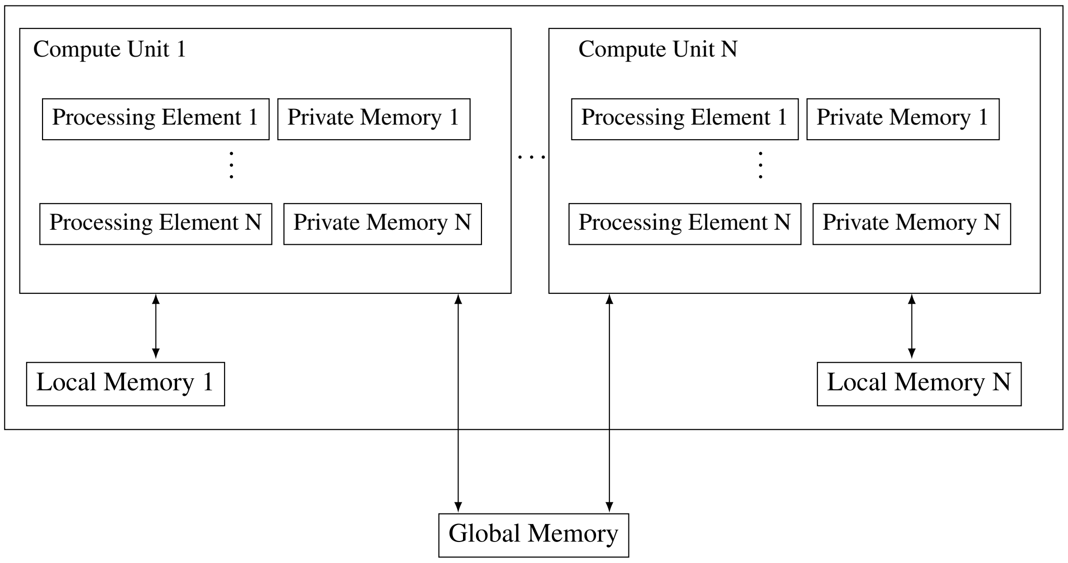


Figure 2.3: Abstract representation of a parallel computing device according to the OpenCL standard.

through local memory. Writes to local memory are guaranteed to be visible by the other work-items in a work-group only after the invocation of a synchronization function called `barrier`. Since work-items can communicate through local memory and local memory is private for a compute unit, work-groups can not be migrated across compute units. Finally all the compute units have access to *global memory*. This is where input/output buffers for host/device communication are allocated. The OpenCL memory model is relaxed: the memory state visible by one work-item is not guaranteed to be the same across work-items. This implies that, for instance, writes to global memory are guaranteed to be visible to all the work-items only at the termination of the kernel.

In summary, for the remainder of the thesis and in particular for the coarsening transformation it is important to remember the following properties of the standard:

- Thread synchronization is possible only within the work-items of a work-group with the `barrier` function.
- Communication between the host and the device is possibly only through buffers in device memory. Pointers to these buffers must be passed to the kernel function as parameters.

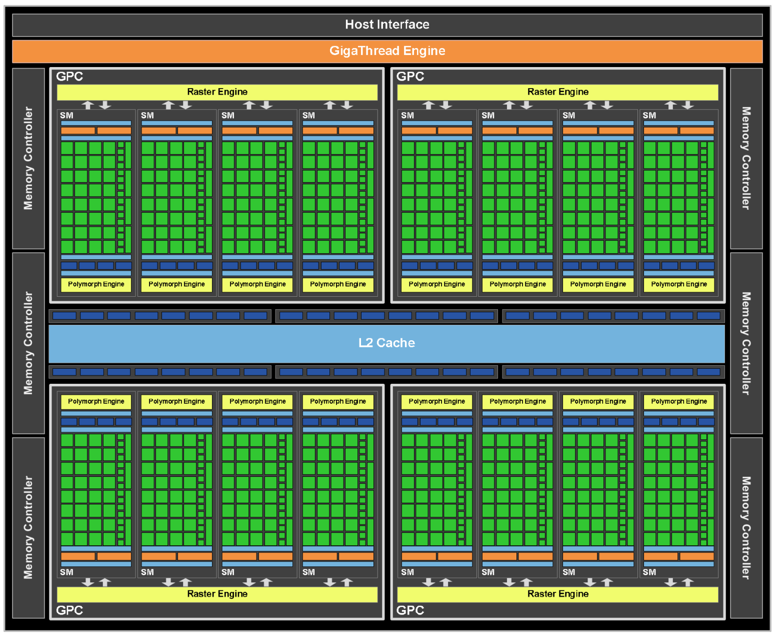


Figure 2.4: Schematic representation of the Fermi GF100 architecture. Taken from [100]

2.2 GPU Architecture

This section presents the hardware characteristics of modern Nvidia and AMD GPU architectures.

2.2.1 Nvidia

In this section we present the structure of the Nvidia Fermi Architecture [100]. In particular we focus on the version named GF100.

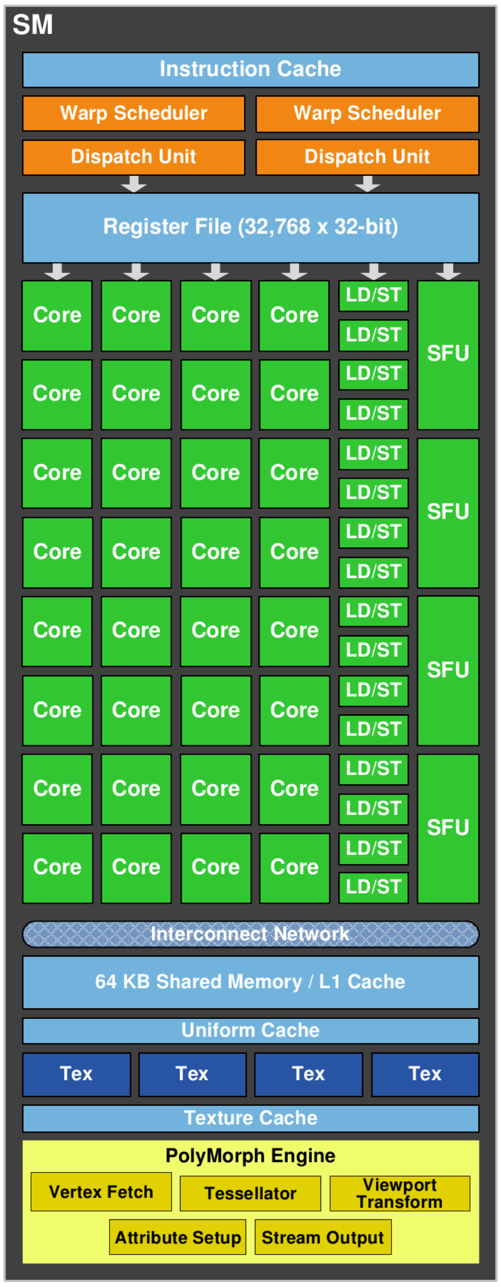
The Nvidia Fermi architecture is the successor of the Tesla architectures, which was the first architecture by Nvidia to support unified shader-cores (see section 3.1 for an overview of GPU hardware). Fermi features scalable and reusable graphics cores. A variable amount of cores can be employed at design time to match the performance needs of the device. Figure 2.4 presents an overview of the architecture. The GPU is organized in Graphics Processing Clusters which contain multiple Streaming Multiprocessors (SMs). Work scheduling happens in a hierarchical way, the *Giga Thread Engine* is responsible for the scheduling of work-groups to the SMs, while the per-SM *Warp Scheduler* schedules batches of threads (see section 2.2.1). To ensure high bandwidth GF100 has six memory controllers, an L2 cache shared among all the cores and private configurable L1 caches.

Streaming Multiprocessor Streaming Multiprocessors group together 32 CUDA cores, see figure 2.5. Threads are grouped into batches of 32 (called warps) which run concurrently on the cores of the SM. Threads in a warp execute in lock-step, *i.e.*, they share the same program counter, meaning that they all execute always the same instruction. This execution model is called SIMT (Single Instruction Multiple Threads): many threads execute a given instruction at the same time. This methodology is effective in reducing the overheads associated with instruction issue and decode. Each SM can support up to 48 warps concurrently. Mapping the Nvidia notation to the OpenCL standard definitions, a CUDA core corresponds to a *processing element*, while a SM corresponds to a *compute unit*.

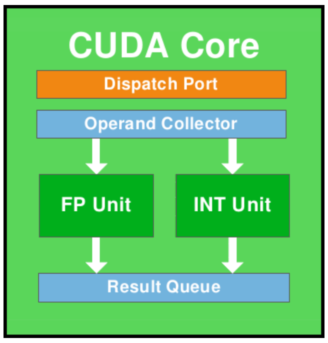
Warp Scheduler Each SM has two warp schedulers. This allows two instructions coming from different warps to be issued and executed concurrently. A schematic representation of the scheduling of warps is given in figure 2.6. Warps in a work-group associated to a SM are placed in a ready queue and sent to execution to the computing cores in round-robin fashion. When threads stall due to a L1 cache miss rate they are placed in a waiting queue, the *Miss Holding Status Register*. When the data comes from memory the warp in the MSHR waiting for that data is reissued for the write-back stage.

Local Memory and Caches Each core is equipped with local memory. Local memory is used as L1D cache and as a programmable scratch pad. See section 2.1.1 for description and uses of local memory. L1 cache and local memory serve opposite roles. Local memory is designed to improve memory accesses for applications with regular and predictable access pattern. L1 cache, instead, improves accesses for irregular algorithms where data addresses are resolved only at run-time. GF100 streaming multiprocessors have 64KB of on-chip memory, 48KB for L1 cache and 16KB for local memory. GF100 also has a 768 KB L2 cache shared among all the cores of the GPU.

Coalescing of Memory Accesses An important concept for the optimisation of GPU applications is the coalescing of memory accesses. On GPUs the best memory pattern is achieved when consecutive work-items in a warp access consecutive memory locations. This is called a *coalesced access*. When a warp makes a memory request, say a `float` (4 Bytes) per thread, data has to be fetched from main memory to the private L1 cache of the core. The size of a transaction from main memory to the cache is equal to a cache line, 128 Bytes. So, if all the addresses requested by the warp fall in the same cache line the request can be serviced in a single transaction. In this case, all the data brought in the cache is used with no waste of bandwidth. This case is depicted in figure 2.7a. If instead, the addresses requested by the threads are scattered in memory (*i.e.*, they are not consecutive for consecutive threads) multiple transactions are needed to service the request. Thus, multiple lines have to be fetched into the



(a) Schematic representation of a Streaming Multiprocessor.



(b) Schematic representation of a CUDA Core.

Figure 2.5: Schemas of an Nvidia Streaming Multiprocessor and a CUDA Core. Both figures are taken from [100].

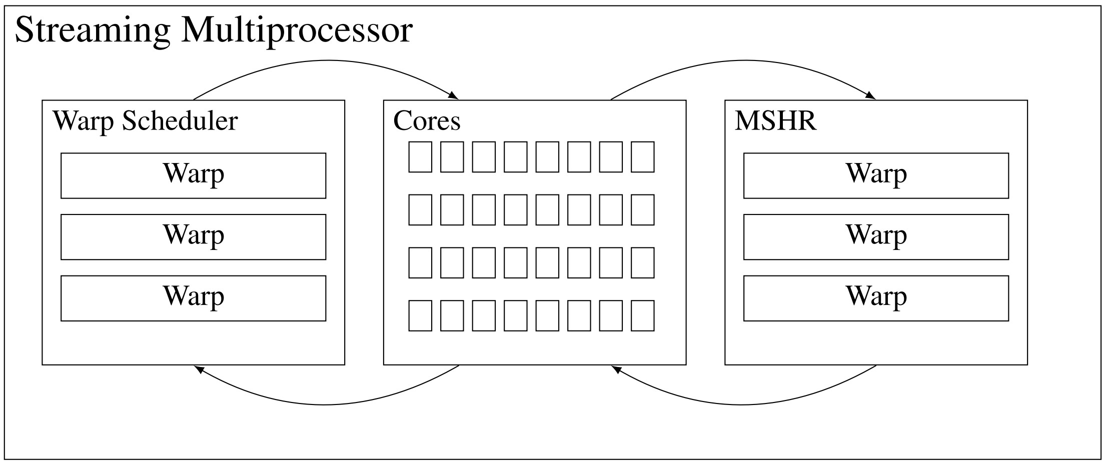


Figure 2.6: Schematic representation of the stages of warp scheduling for an Nvidia streaming multiprocessor. The arrows represent the flow of warps in the functional units.

cache. Only part of the data brought in the cache is going to be actually used by the threads. This type of accesses are called *uncoalesced* and must be avoided as much as possible. This case is depicted in figure 2.7b. Similar concepts apply for AMD graphics hardware.

Compiler Toolchain Along with GPU hardware Nvidia ships to developers a library to compile and execute OpenCL and CUDA programs. This library contains a proprietary compiler that translates OpenCL programs to an intermediate representation called PTX (Parallel Thread Execution) [4]. PTX is then further translated by a backend compiler into SASS: Nvidia assembly format.

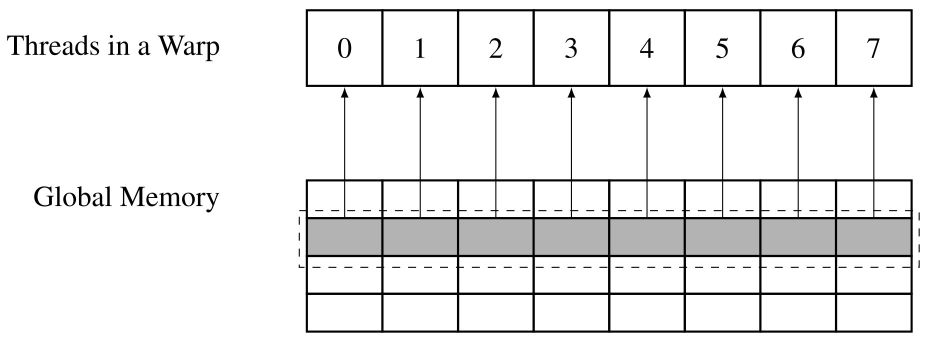
2.2.2 AMD

In this thesis we worked with two AMD architectures, the first one is called *TeraScale* and the second one *Graphics Core Next*. These two are radically different.

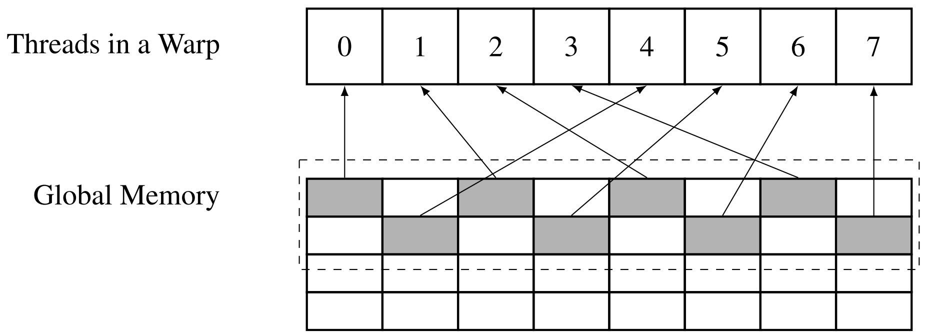
2.2.2.1 TeraScale

The Terascale architecture [94] is the second one from AMD to ship a unified graphics core, the first one was C1/Xenos[1]. TeraScale manages the massive parallelism grouping threads in batches of 64, called *wavefronts*. The most distinctive characteristic of TeraScale are its 5-wide VLIW cores. They offer high performance for graphics computation and for matrix operations. This architecture has two main limitations. The first one is that instructions have to be statically scheduled by the compiler to fill the VLIW slots. This is a hinders the performance of general purpose workloads, where branches and unpredictable outcomes are more likely to happen

[1]http://www.beyond3d.com/content/articles/4/2 (June 2005)



(a) Coalesced access to memory. Notice that the touched locations are aligned and contiguous. This memory request can be serviced in one transaction.



(b) Uncoalesced access to memory. The touched locations are scattered. This memory request is serviced in two distinct transactions.

Figure 2.7: Examples of coalesced and uncoalesced memory accesses. Each row of the memory cells corresponds to a cache line. The grey squares represent memory locations that are touched by a thread in the warp. The dashed rectangle represents all the data fetched from main memory to the private caches. Notice that in the uncoalesced case the amount of data touched by the warp is the same of the coalesced one, but the total amount of data transferred is the double.

than in graphics workloads. The second limitation lies in the limited parallelism available in the register file. It is not possible to schedule two consecutive ALU instructions, wavefronts have to be interleaved to mitigate this problem. These issues were addressed in the design of the next architecture: Graphics Core Next.

2.2.2.2 Graphics Core Next

The most recent AMD architecture is called Graphics Core Next [7, 31, 9] and it is a complete redesign over TeraScale. It implements an entirely different ISA, which is more amenable for computational workloads.

Compute Unit Compute Units (CU) are the main processing components of the GCN architecture. A GPU is mounted with a certain amount of CUs depending on its design require-

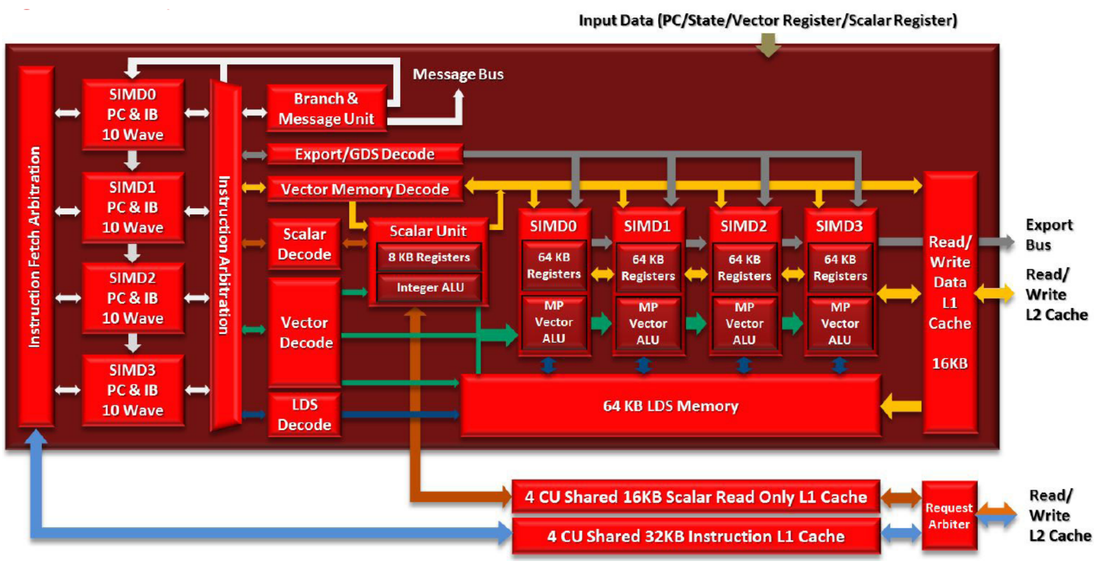


Figure 2.8: Schematic Representation of the GCN architecture. Notice the four SIMD vector units and the scalar unit. Figure taken from [8]

ments and its performance budget, the typical number is 32. Each CU contains 4 SIMD units, called processing elements in the OpenCL parlance. These SIMD units execute the bulk of the computational work. Each SIMD Unit executes simultaneously one instruction from 16 work-items. The GPU scheduler can assign up to 10 wavefronts to each SIMD Unit, therefore they will execute instructions from different wavefront, possibly coming from different applications. The possibility to dynamically schedule work is a significant improvement in flexibility over the TeraScale architecture. The CU schedules work for one SIMD Unit at the time (over the four available) in round-robin fashion. Each CU is equipped with a scalar unit, used to manage conditional branches and to generate the target addresses of jumps. Figure 2.8 shows the architecture of a compute unit.

Memory System Each CU has a private scratch-pad memory called Local Data Share. The data cache hierarchy is organized as follows: L1 is 16KB, 4-way set associative with lines of 64 Bytes using the Least-Recently-Used replacement policy. Note that to achieve memory coalescing the memory accesses for a quarter of a wavefront (16 threads) must be all consecutive. GCN cores introduce another improvement over TeraScale: cache coherency. L1 is coherent for all the threads in a work-group and global coherency is achieved relying on L2. When a wavefront finishes execution or when a barrier statement is executed the private content of L1 cache is written L2 and becomes globally coherent. The last important characteristic of GCN cores is virtual memory support. This has been introduced to make GCN available for integrated GPUs as well, where CPU and GPU share the same physical memory.

2.3 Compiler Infrastructure

This section presents the LLVM compiler we used to implement the coarsening pass.

2.3.1 LLVM

LLVM is an open-source compiler infrastructure initially developed by University of Illinois at Urbana Champaign [73]. The original design goal of LLVM is to provide a framework for aggressive code optimisation so to achieve high code quality keeping compilation times short. LLVM is a robust modern compiler designed in a modular way that provides a wide range of analysis and optimisations. In our experimental framework the OpenCL host code is compiled with GCC (the compiler of choice of the benchmarks) while only the device code *i.e.*, the kernel function, is compiler with LLVM.

LLVM-IR The most significant innovation introduced by LLVM is its intermediate representation, called LLVM-IR. LLVM has the following design choices:

- IR uses operations simple enough to be easily mapped onto real hardware.
- IR is portable, allowing generation of assembly code for a wide range of devices.
- IR is independent of both the operating system and the input language. The type system is generic enough to allow multiple languages to be easily mapped.
- IR supports metadata to enable optimisation transformations and generation of good machine code.

LLVM-IR is in static single assignment form [110, 29], this means that variables are assigned to in only one location. This simplifies the implementation of analysis and transformations such as constant propagation, dead code elimination and register allocation. LLVM programs are organized in *Modules*, which correspond to a translation unit of the original program. Modules contain a set of *Functions*. Functions are made of *Basic Blocks* which then contain IR *Instructions*. This organization reflects well the structure of C and C++ programs.

Pass Manager One of the most important components of the LLVM compiler is the *Pass Manager*. It schedules analysis and transformation passes. LLVM supports passes working on different IR structures: modules, functions and basic blocks. A function pass, for example, works on a function at the time and the pass manager makes sure to run it on all the functions in a module.

Scalar Evolution This thesis makes use of an important LLVM analysis called Scalar Evolution (SCEV). A SCEV of an LLVM value is a mathematical expression that describes its value during the execution of the program. It provides an abstraction over the actual LLVM instructions used to compute the value. The Scalar Evolution analysis is based on the work of [11, 145, 126, 127, 18]. SCEV expressions are constructed recursively using a set of basic types:

- Integer constant.
- Unknown value. For example an input parameter of the current function.
- Cast expression.
- Binary mathematical operation.
- N-ary operation.
- Add recurrence. This is used to represent the evolution of loop variables. This expression is made of a starting value, an operator and a increment value. At each iteration of the associated loop the current value of the expression is update using the operator and the increment.

The following expression is an example of a SCEV expression for the computation of the load address for `transposeMatrix` kernel (figure 2.2): `((4 * ((%width * %call) + %call1))+ %input)`. The values `%width`, `%call` and `%call1` are local function variables, while `%input` is the base pointer of the input buffer. SCEV expression can be rewritten and simplified [126] thus to easily compare the values of different variables. Scalar Evolution works effectively for integer computations, this makes it a good candidate for the analysis and the transformation of loop iteration variables memory addresses. The Scalar Evolution pass fails in analysing computation which depend on conditions evaluated at run-time, for example the trip-cont of a loop iterating a NULL-terminated linked list. More details on Scalar Evolution can be found in [48].

LLVM and OpenCL Thanks to the flexibility of the compiler it is relatively easy to map OpenCL constructs to LLVM-IR. Kernel functions in a module are listed in the metadata section, which can be read by all the passes. Vector data types are natively supported by LLVM-IR. Finally, memory address spaces (*e.g.* the ones identified by the keywords `__local` and `__global`) can be simply represented using the `addrspace` keyword.

LLVM Tools LLVM has been designed in a modular way. The front-end, called `clang`, transforms the input language, in our case OpenCL-C, into LLVM-IR. The `opt` tool invokes the pass manager to schedule optimisation passes on the IR. Finally `llc` lower LLVM-IR to the target. The whole process is summarized by figure 2.9.

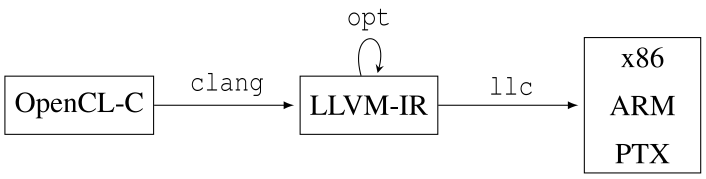


Figure 2.9: LLVM compilation stages. `clang` (the front-end) generates LLVM-IR, which is optimised by `opt`. `llc` lowers the IR to the target assembly language.

2.4 Hardware Performance Counters

Our work makes extensive use of profiler counters. In particular, chapter 5 presents a methodology to quickly relate changes in the value of performance counters with changes in the performance of the application. Here we present how we collected counters on Nvidia and AMD devices. The available counters reflect the architectural configuration of the device and differ across vendors and generations.

Nvidia The Nvidia profiler ships with the OpenCL SDK library, it is enabled by setting the environmental library `COMPUTE_PROFILE` to `1` before the invocation of the executable. Tables A.1, A.2, A.3 and A.4 in Appendix A list the available counters for an Nvidia GTX 480 and for an Nvidia Tesla K20c. In these tables we only report the counters we collect. We ignore texture memory counters, since the benchmarks we use do not use this type of memory. These two devices are used in the experiments of chapter 5. Nvidia counters are highly accurate giving information about all the aspects of the computation. Nvidia hardware limits the execution time of profile runs by collecting counters only on one of the cores of the GPU. Therefore, to obtain a total estimate of the profiler values the output values have to be post-processed, according to this formula:

$$Thread\_ratio = \frac{G_x \cdot G_y}{thread\_launched} \tag{2.1}$$

$$Counter\_value = profiler\_counter \cdot thread\_ratio \tag{2.2}$$

Where $thread\_launched$ is a counter recording the number of hardware threads that are actually profiled by the Nvidia profiler. G_x and G_y identify the x and y dimensions of the `NDRange` thread space.

AMD The AMD profiler is an executable called `sprofile`. It takes in input the name of the OpenCL executable and outputs instruction counts for each kernel in the application. Tables A.5 and A.6 in Appendix A list the available counters for an AMD HD5900 and for an

Program name	Source
`binarySearch`	AMD SDK
`blackscholes`	Nvidia SDK
`convolution`	AMD SDK
`dwtHaar1D`	AMD SDK
`fastWalsh`	AMD SDK
`floydWarshall`	AMD SDK
`mriQ`	Parboil
`mt`	AMD SDK
`mtLocal`	AMD SDK
`mvCoal`	Nvidia SDK
`mvUncoal`	Nvidia SDK
`nbody`	AMD SDK
`reduce`	AMD SDK
`sgemm`	Parboil
`sobel`	AMD SDK
`spmv`	Parboil
`stencil`	Parboil

Table 2.1: OpenCL applications used as benchmarks in this work.

AMD Tahiti 7970. Notice that the counters exposed by the AMD profiler are fewer an much less precise than the one given by Nvidia.

2.5 Benchmarks

This section presents the benchmark programs that we used in this work.

Table 2.1 lists the OpenCL programs that we used in out experiments. We selected these kernels from the AMD SDK, Nvidia SDK and Parboil benchmarks suites. For the *Parboil* benchmarks we used the *opencl_base* version. In the benchmarks selection process we avoided programs with features that our coarsening pass does not support, such as atomic instructions or texture memory. We also selected benchmarks that are easy to parametrize. In particular, we changed the C/C++ host code so to be able to change freely the work-group size. In our experiments we used various problem input sizes for the benchmarks, these are specified in each technical chapter.

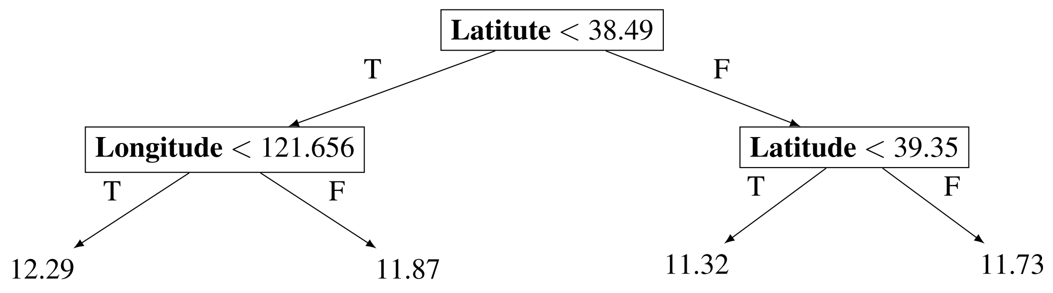


Figure 2.10: Example of a regression tree. The tree maps geographical locations (**Latitude** and **Longitude**) to house prices (the values in the leaf nodes). Example taken from: http://www.stat.cmu.edu/ cshalizi/350/lectures/22/lecture-22.pdf

2.6 Machine Learning

This section presents the relevant statistical techniques used in this thesis. Machine Learning is the field of computer science that develops algorithms that can improve with experience over time. This technique is usually applied to model data and to make predictions about it. In this thesis we apply supervised learning techniques. Supervised learning has the task of determining an unknown function from labelled data. Data used for training the statistical model comes in pairs $< \mathbf{x}, y >$, one element of the pair is the set of features describing the data point ($\mathbf{x}$), while the other element is the value of the unknown function in the given point (y). The algorithm uses training data to learn the shape of unknown function: $\mathbf{x} \rightarrow y$ and be able to predict the value of y for a new data point. In the remainder of the section we describe the characteristics of two techniques we used in the thesis: Regression Trees and Neural Networks.

2.6.1 Regression Trees

Regression problems deal with output responses y having real values. Regression trees [109] attempt to solve these problems by splitting the input domain $\mathbf{x}$ in smaller regions and fitting a simple model for each region. The recursive split of the input space is represented by a decision tree. The tree contains two types of nodes: internal nodes and leaves. Internal nodes represent a split in the input space according to one dimension with a comparison against a predefined value, *e.g.* $x_i < threshold$. Leaf nodes, instead, represent the prediction for the output variable in the given subregion of the space. This is usually performed averaging the response variable (y) in the given subregion. Figure 2.10 gives an example of a regression tree mapping geographical coordinates to house prices. Finding the optimal regression tree for a given space is NP-complete [97], therefore many different heuristics have been proposed in the literature. A typical approach is the following. The construction of the tree starts top-down, with the root containing all the training points. Various split points are evaluated for each

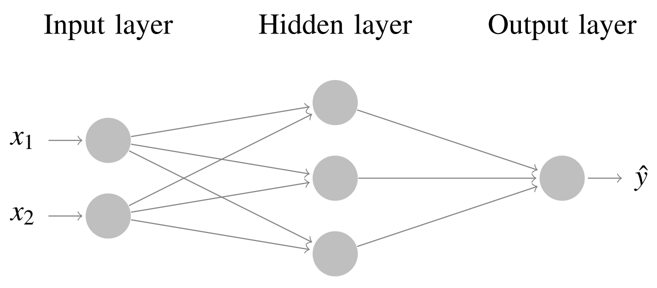


Figure 2.11: Example of a multi layer neural network. This network is fully connected with three layers: input, hidden and output

input dimension (feature). A pair input dimension / threshold point is selected if it maximizes a heuristic value computed on the two resulting sets. One of the heuristics used is often the difference between the variances of the two nodes with respect to the prediction.

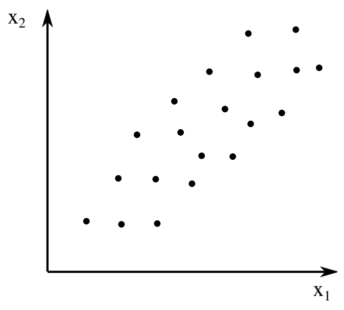
Regression trees have advantages over other machine learning techniques: making predictions from a tree is fast and they are easy for a human to read (in particular the features most relevant for prediction appear close to the root of the tree). The most relevant drawbacks are: trees are not well suited for predicting highly irregular spaces and trees tend to overfit. This means that the resulting tree performs well on the training data but poorly on testing (unseen) data. Techniques such as early pruning are employed to mitigate this issue.

2.6.2 Neural Networks

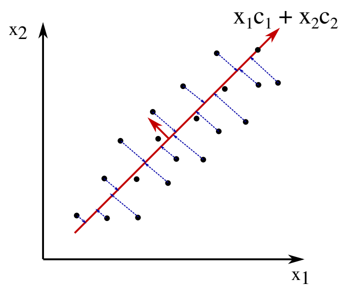
Artificial Neural Networks (NN) [19] are often used when the regression function to be learned is non-linear and highly irregular. NN are composed of multiple neurons organized in layers. Each neuron has a fixed number of input and output links to other neurons.

Figure 2.11 gives a schematic representation of a feed-forward neural network. Each edge in the topology graph is associated with a weight w, while the output of each neuron is computed according to this function: $output(\mathbf{x}) = f(\sum_{i=1}^{n} w_i \cdot x_i)$. x_i are the input values of the neuron. f is called the activation function an it usually has domain over the real numbers and range in $[-1,1]$ or $[0,1]$. Examples of possible activation functions are: the hyperbolic tangent ($f(x) = tanh(x)$) and the logistic function ($f(x) = \frac{1}{1+e^{-x}}$).

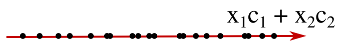
The output function is learned by updating the weights for all the neurons so that the error between the output of the network and output value of the training data is minimised. The current state of the art learning algorithm is called *backpropagation*. The algorithm works iteratively. The weights are first initialized with random values. For each iteration the method computes the output of the network and compares it against the actual output value of the training data point. Then for each weight associated with the edges from the hidden layer to the output layer the algorithm computes Δw according to the formula $\Delta w = \eta \cdot (y - \hat{y}) \cdot x_i^j$ (η



(a) Example of data in a 2-dimensional space having one dominating direction.



(b) Visualization of the two principal components and projection of the data to the one having largest magnitude.



(c) Visualization of the data projected to the most significant component.

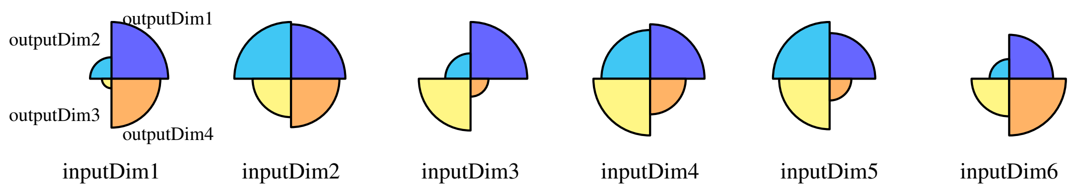
Figure 2.12: Three stages of PCA

is the learning rate and x_i^h is the value from the j-th hidden neuron to the i-th output neuron). It then computes the same delta for the weight between the input and hidden layers. Finally, update all the weights in the network using the deltas just computed. The algorithm stops when the error converges below a pre-defined threshold or a maximum number of iterations have been reached.

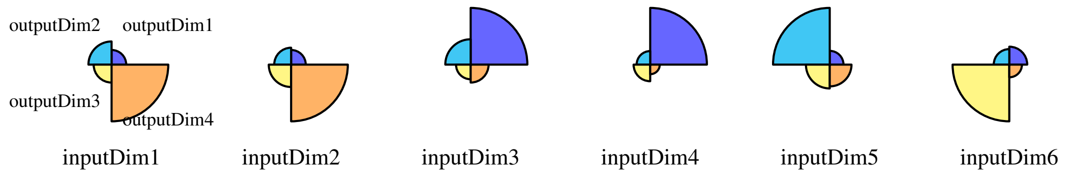
2.6.3 Principal Components Analysis

Principal Components Analysis (PCA) [20] is a statistical technique widely used for dimensionality reduction. PCA is usually applied to feature spaces that are highly dimensional and contain redundant information. Redundancy comes from dimensions that are highly correlated with others, this means that they can be expressed as a linear combination of other dimensions. PCA removes redundancy in a highly dimensional space by projecting the available data onto the most representative directions. This can effectively improve the accuracy of predictive models.

Consider the data in figure 2.12a. It lays out in a two-dimensional space data having a strong distribution along a specific direction. PCA computes first the covariance matrix. This is determined computing the correlation factor between all dimension pairs of the data. The method then computes the eigenvalues and eigenvector for the covariance matrix. Figure 2.12b shows the eigenvectors scaled by the eigenvalue after they have been made orthogonal. The principal components are then the eigenvectors corresponding to the eigenvalues of greatest magnitude. In the example the principal component is given by the symbolic formula $c_1x_1 + c_2x_2$, where c_1 and c_2 are called *loadings*. PCA involves the determination of a threshold used to define what components to discard. The last step of the PCA algorithm Figure 2.12c is to project all the data onto the space of the selected principal components. This way the output data is defined in a different space, with smaller dimensionality and less redundancy than the



(a) A slice in the pie-charts represents the value of a loading of for the given input dimension.



(b) After Varimax rotation one input dimension strongly contributes only to one output dimension.

Figure 2.13: Visual representation of the value of loadings for a hypothetical space before and after Varimax rotation.

original space, and it still coveys similar amount of information.

Varimax Rotation One of the drawbacks of PCA is that the features of the transformed space are a complex linear combination of the original ones. It is often hard to identify the meaning of the new components and how they correlate to the original ones. In this context the Varimax Rotation can be of help [88].

Varimax Rotation rotates the output space in such a way that the resulting principal components have formulas having a large number of loadings set to 0 and a few (ideally only one) loadings set to large magnitude value. This enables the possibility to identify what directions of the input space strongly affect directions in the output space.

Figure 2.13 gives an example of the effects of Varimax rotation on a hypothetical space. Subfigures represents a pie-chart for each dimension of a hypothetical six-dimensional space $D[1-6]$. The application of PCA, in our example, shrinks the original space to a four-dimensional one. Each slice of a pie represent the relative contribution of the given input dimension to the four output dimensions in figure 2.13a. We can see that normally each feature strongly contributes to multiple output dimensions (for example the top right slice is large in area for dimensions $1,2,3,4$). After the application of Varimax an input dimension strongly contributes to only one output dimension, see figure 2.13b. This property is particularly useful since it is now possible to label output dimensions (a new feature) with the name of the original direction (feature) that contributes to its value.

2.7 Performance Evaluation

2.7.1 Performance Metrics

In the context of machine learning the metrics used to quantify the performance of a technique are usually the mean squared error for regression problems and accuracy and precision for classification problems. In the context of compiler optimisations, instead, the most important performance metric is execution time difference between the newly tested technique and a reference baseline for a run of the tested program. For this thesis we always use the execution time speedup to measure this difference. The speedup is defined as

$$speedup = \frac{t_{baseline}}{t_{new}} \tag{2.3}$$

Where $t_{baseline}$ is the execution time of the program run with the configuration considered as baseline, t_{new} is the execution time of the same program run with the compiler configuration under exam. When aggregating speedups of multiple programs we use the geometric mean. In this work we always measure the effectiveness of our compiler techniques against the best attainable in our search space, the so called oracle. We thus compute the percentage of performance that we achieve. Since in our experiments the maximum speedup is always greater than one we compute performance percentage in the following way. If *speedup* is greater than 1:

$$performance\_percentage = 100 \cdot \frac{speedup - 1}{max\_speedup - 1} \tag{2.4}$$

If *speedup* is smaller than 1:

$$performance\_percentage = 100 * (speedup - 1); \tag{2.5}$$

This is a stricter metric over the simple ratio $100 \cdot \frac{speedup}{max\_speedup}$ since we only account for the speedup that we achieve *over the baseline*.

2.7.1.0.1 Correlation In this thesis we make use of the correlation coefficient to measure the similarity of two sets of data, these are usually predicted values and actual experimental data. Correlation measures the interdependence between two set of variables, in particular their linear relationship. Given two set of values A and B, the correlation coefficient is computed as:

$$corr(A,B) = \frac{cov(A,B)}{var(A) \cdot var(B)} \tag{2.6}$$

Where cov is the covariance between the two input sets and var is the variance of the input set. A positive correlation value means that when values in the A set increase values in the B increase as well, and vice-versa. Negative value of correlation instead means that when values in the A set increase values in the B set decrease. Values of 1 or -1 mean that data in set A can be transformed into data in set B with a linear transformation and vice-versa. A value of 0 means that there is no linear relationship between the two sets.

2.7.2 Cross Validation

Machine learning techniques are evaluated by dividing the available labelled data into training and testing set. The model is then fitted on the training data and then evaluated on the testing set. This process is called *cross validation*. Rigorous performance evaluation prescribes not to draw conclusions about the performance of the model when evaluated on the training data. In this work we employ a specific version of Cross Validation called Leave one out. For the neural network model in chapter 7 we train the network on 16 of the 17 benchmark we use and we evaluate its performance of the 17th. We repeat this process 17 times, one for each test case.

2.8 Summary

This chapter presented the technical background necessary to understand the remainder of the work. We introduced the OpenCL language for programming graphics processors and we gave an overview of the main characteristics of Nvidia and AMD processors. We then described the LLVM compiler infrastructure, focusing on the components that we use in the implementation of the work. Finally we gave an overview of regression trees, neural networks and PCA, these are the machine learning techniques that we use to analyse the results of thread-coarsening.

Next chapter will give an overview of the academic publications related to this thesis.

Chapter 3

Related Work

This chapter reviews publications relevant for this thesis. Section 3.1 presents an overview of the evolution of graphics hardware, from fixed hardware functionalities to fully programmable processors. Section 3.2 describes the context of general purpose processing for GPUs, presenting its most important applications. Section 3.3 discusses the topic of performance analysis and modelling for GPUs. Section 3.4 presents prior works on loop optimisations for single- and multi-processors. These techniques have found new applications in the context of GPU programming. Section 3.5 introduces techniques for the optimisation of GPU programs. In Section 3.7 we give an overview the use of machine learning for compiler optimization. Section 3.8 concludes the chapter.

3.1 History of Graphics Processing

This chapter overviews the history of the design of graphics processors.

Graphics processors have been designed with the goal of quickly computing the colour of screen pixels when displaying a computer-generated image. Figure 3.1 gives a high level overview of the stages to display images on the computer screen starting from a representation of a 3D scene. This process is called *rendering*. The 3D scene is made of objects described in memory by a set of points and 2D images called textures. These points represent the vertices

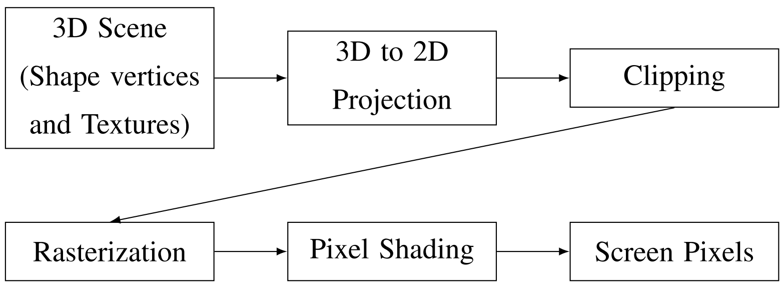


Figure 3.1: Overview of the graphics pipeline.

of triangles (called primitives) positioned on the surface of objects. Textures are images that represent the colour of the surface of objects. The first step of the pipeline is the projection of the 3D scene onto the 2D surface of the screen, transforming the world coordinates into screen coordinates. This projection is described by a 4x4 matrix. The second step is the removal of all the vertices that are invisible given the current orientation of the point of view in the 3D world. Rasterization is the process of determining which triangle in the scene covers a pixel of the screen. Finally pixels are coloured according to the colour of the triangle coving it and the position of the lights, this process is called shading.

Fixed-Function Custom Hardware Dedicated hardware for rendering has been proposed as early as the 1980s [27, 120]. These solutions we not parallel and had refresh times of the order of seconds or minutes. Massively parallel architectures were then introduced to take advantage of data parallelism inherent in the problem (the colour of each pixel can be determined independently of all the others) [44]. These where limited in scalability or in programmability, for example by fixing the number of cores to be able to manage a predefined number of pixels.

The Triangle Processor The work by Deering *et al.* [30] represents a major step forward in the design of graphics processors, introducing a system architecture used as a reference for years to come. The novel idea of this work is to split the rendering pipeline in two stages. The first stage, called *Triangle Processors*, works on one triangle at a time to determine the pixels that cover it. The second stage, called *Normal Vector Shader*, implements the shading according to the Phong model [107]. This organization of the work-flow has been implemented by all modern graphics processors. More in detail, the *Vertex Processor* (VP) works on the vertices of a primitive transforming its coordinates from world-space to screen-space, and also sets up colours and textures to be used by the fragment processor. The *Fragment Processor* (FP) interpolates the data associated with the vertices of the primitive to set each pixel of the appropriate colour. The two types of processors have different types of workloads, the VP works with high precision on mathematical operations, while the FP works on texture filtering (needed to map textures to primitives).

Programmable Vertex Processor The next big step in the definition of modern GPU architecture is the introduction of a programmable Vertex Processor [80]. The first GPU to ship this type of processor was the Nvidia GeForce 256 in 1999. This device addressed the need for a wider range of lighting effects. Graphics core are now programmable using the OpenGL [68] (from the Khronos Group) and Direct3D [93] (from Microsoft) frameworks. It is important to notice that usually graphic workloads are not evenly distributed among VPs and FPs. Scenes with a large number of small triangles are put pressure on VPs, while scenes with a small number of large triangles make the VPs almost idle while keeping the FPs under high utilization.

Any fixed choice of the number of FPs and VPs leads to workload imbalance and possible under-utilization of the device.

Unified architecture The solution to this problem came with the introduction of an unified architecture for the VPs and the FPs, so that vertex and fragment programs can be executed on the same processor. The main drawbacks of the unification comes from the increase in complexity in the processor logic and in power consumption. The first examples of this innovation came from the Nvidia Tesla Architecture [81] and ATI Xenos. The introduction of a single processor core on GPUs means having a fully-programmable highly parallel processors available for computation. Exploiting this architecture for non-graphics workload is the goal of GPGPU.

3.2 General Purpose Computing on GPUs

This section gives an overview of General Purpose computing on graphics processors.

Even though GPUs are designed for graphics workloads the massive parallelism they expose and their power efficiency make them an attractive candidate for general purpose computation. The first programming framework that exploited the capabilities of modern GPUs for General Purpose Computing is Brook [24]. The work introduces a programming model that abstracts the graphics hardware using streams, kernels and reductions. Data stored in memory is organized in *streams*, *i.e.*, a set of data that can be modified in parallel. Streams have a base type and a dimensionality. A *kernel* is a data parallel function which is executed for each element of the data stream. *Reduction*s provide a data-parallel method to compute a single output value from a stream. From the implementations point of view Brook is composed of a compiler and a runtime library. The compiler translates Brook programs into Cg shaders [89] (a shading language by Nvidia) which are then translated to assembly by the vendor compiler. The runtime implementation lowers the programming model down to OpenGL and DirectX constructs.

In 2007 Nvidia introduced its solution for general purpose computing for GPUs: CUDA [98]. CUDA, like Brook, is made of a compiler and a runtime system. The compiler translates `__global` functions into PTX assembly code and the runtime dispatches threads onto the GPUs. The runtime system is also responsible for the management of memory buffers on the GPU and the data transfers between the host CPU and the GPU.

The work by Owens *et al.* [103] presents an overview of the type of applications that can benefit from GPGPU computing. Here follows a description of the most important applications of GPGPU following the guidelines given by the paper.

Basic Algorithms GPUs have proved to be capable of solving fundamental tasks of computer science much more efficiently than multicore CPUs. Merril *et al.* [92] show that radix sort can be improved more than 3x over a CPU implementation using an Nvidia GPU. Ye *et al.* [141] have proposed an efficient implementation of merge sort, while Tanasic *et al.* [122] have extended the work to support multiple GPUs. Hierarchical parallel array reduction is a good fit for GPUs as showcased by an Nvidia tutorial by Mark Harris [52]. Merril *et al.* [91] also provided an efficient implementation of array prefix sum (scan). This implementation is one of the building blocks of a parallel graph Breadth First Search [90] on GPUs.

Linear Algebra Classical linear algebra problems expose high degrees of parallelism that can be efficiently exploited by graphics processors. Dense matrix-matrix multiplication [121] is a perfect candidate for GPUs. State of the art implementations all rely on graphics hardware. Also sparse sparse matrix-vector multiplication have been extensively studied [137].

Domain-Specific Applications GPU are successful in a wide range of application-specific numerical problems. Their superior power-efficiency makes them good candidate for large-scale physical simulations for Nbody problems [47] or fluid simulation [51]. In these contexts a single thread usually simulates a single body or a single node in a grid.
In 2012 the machine learning annual competition for identification of objects in photographs called *Imagenet* was won for the first time by a neural network trained implementing the backward propagation algorithm on a GPU [71]. The run-time improvements given by GPUs when training deep neural networks allow the utilization of much larger data sets, leading to improvements in accuracy. The results are so significant that, after 2012, all winners of the annual competition have used DNN trained on GPUs [1].
In the context of image synthesis with ray-tracing Optix [105] allows programmers to write CUDA-like kernels to describe the properties of materials in 3D scenes. A custom compiler generates GPU code and a runtime system dispatches threads on a GPU to execute the propagation of the rays in the environment.
Lastly we cite the application of GPUs to data base query processing. The tool Red Fox by Wu *et al.* [134] compiles relational queries to GPU code to exploit the massive parallelism for the execution of new independent queries on large data sets.

In this section we saw how GPUs can be used to efficiently solve a variety of problems. These machines are not a perfect fit for every problem. Lee *et al.* [77] and Bordawekar *et al.* [22] show that carefully tuned CPU code can match or outperform GPU programs for highly parallel problems. The authors claim that works advertising massive performance improvements given by GPUs were comparing against suboptimal CPU implementations.

[1]http://on-demand.gputechconf.com/gtc/2015/presentation/S5715-Keynote-Jen-Hsun-Huang.pdf (March 2015)

3.3 GPU Performance Analysis and Modelling

This section describes in techniques to analyse and predict performance of GPU accelerators. We identified three main methodologies for the analysis of GPU performance: manual modelling, simulation-based modelling and machine learning-based modelling. We also present works that evaluate the portability of OpenCL application across different devices.

Manual Modelling The work by Hong *et al.* [53] is, to our knowledge, the first attempt to develop an analytical performance predictor for GPU architectures. The paper approaches the task of performance modelling analysing the available thread level parallelism and the amount of computation performed by the application. The methodology integrates hardware parameters, extracted with benchmarking, in an analytical model to estimate the execution time of CUDA applications. The follow-up paper [54] extends the performance model to integrate a power prediction as well. A more recent work from the same group extends the model to analyse multiple program versions so to guide compiler optimisations [116]. The model takes in input information about the dynamic instruction mix and other profiling information obtained with Ocelot and with the Nvidia profiler [1]. The transformations used for analysis involve prefetching, promotion to shared memory, unrolling and vectorization and were applied manually. They show that the model can successfully predict the performance of multiple code versions.

Baghsorkhi *et al.* [14] have developed a GPU performance model based on the control flow graph. Instructions in basic blocks are coalesced together into groups and marked with a label to identify the most dominant operation: global memory access, local memory access or computation. The dependencies between the nodes and the latencies of the individual nodes are used to determine the overall latency of the threads. The model also takes into account how many warps execute a given node (depending on the control flow) by looking at the program dependency graph [43]. The follow-up work from the same group [15] focuses on the analysis of GPU memory and cache performance. It introduces a tool to instrument memory accesses and to statistically select which accesses to instrument using a Monte Carlo method.

Zhang *et al.* [144] propose a set of micro-benchmarks to measure memory latencies to identify performance bottlenecks. The model relies on code written directly in GPU native instruction set. The authors use their performance model to improve performance of matrix multiplication, tridiagonal solver and sparse matrix multiplication.

All the models presented in this section provide high predictive accuracy at the cost of extensive manual labour. Such models are fit for a specific GPU model and cannot be easily extended to others.

Simulation-Based Modelling GPGPUSim [45, 16] is the state-of-the-art academic simulator for Nvidia architectures. The tool simulates an Nvidia GPU by interpreting PTX and SASS instructions generated from CUDA functions. GPGPUSim includes timing models for the SIMT cores, caches, DRAM and interconnect. This simulator can provide valuable information in particular about thread divergence within warps and the utilization of the cache hierarchy. It is consistently used by researches for design-space exploration of GPU architectures such as [59] and [99]. Multi2Sim [124], a simulator based on AMD TeraScale architecture that aims at studying heterogeneous systems. The tool integrates a GPU and a x86 CPU simulator. The GPU-side of the tool is used to study the utilization of the VLIW cores, the branching divergence and the performance scalability of OpenCL kernels. Since it models an architecture which has been replaced by newer ones this simulator has lost interest over time.

Machine Learning for Performance Modelling The works by Lee *et al.* [75], Dubach *et al.* [38] and Ipek *et al.* [57] are the most significant in the context of performance modelling of CPU cores with machine learning. Their goal is to predict the performance of various configurations of the processor design space given a small sample to learn from. The remainder of this section deals with the performance modelling of GPUs, these are challenging to model due to their massive parallelism.

Jia *et al.* [59] tackle the task of hardware design space exploration for graphics processors. The goal is to quickly identify the best performing GPU hardware configuration for a given program. The authors explore a large space of parameters for GPGPU-Sim [16]. Using linear regression to interpolate a small fraction of the search space they show it is possible to achieve high prediction accuracy over the entire space. The paper describes also the process of selecting the most relevant parameter for model accuracy using the R^2 metric.

The same researchers have approached the problem of rapidly selecting algorithmic parameters [60]. They build a search space for GPGPU applications considering algorithmic optimisation factors specific for each application. The performance of testing configurations is evaluated using a regression tree which predicts both execution time and power consumption. The methodology has been evaluated both on Nvidia and AMD GPUs. This paper is similar to the work we present in chapter 5. Three are the main differences: (1) We employ a compiler transformation to generate the search space, while they rely on human intervention (see chapter 4). (2) Our regression tree methodology is device-specific, while their is kernel-specific. (3) Our technique partitions the space of profiler counters, while theirs works on the program parameter space. In summary, Jia *et al.* aim at providing insightful information to the application programmer, while our work is meant to be used by compiler engineers.

Kerr *et al.* [65] developed a methodology capable of generating performance models for both multicore CPUs and GPUs. The authors employ the Ocelot emulator [33] to collect statistics about the dynamic characteristics of CUDA applications. They gather information such as instruction count, branch divergence, memory footprint, and more. The large number of programs features extracted are then compacted into a small number using Principal Component Analysis. Each output component is then labelled with a significant name depending on the features that contribute to it the most, this is possible after a space rotation called Varimax [88]. The principal components are then used to perform linear regression on the program execution time. The methodology to reduce and analyse the feature space is similar to the one that we employ in chapter 7.

The follow-up paper [64] by the same group aims at automatically develop an infrastructure for performance prediction formalising the development of models and reporting of results to the users.

Memory Modelling In section 4.6 we propose a methodology to analyse memory access patterns of GPU programs. Previous research in this field has been directed towards two main goals: to analyse the GPU cache and to estimate the coalescing of memory transactions. Tang *et al.* [123] propose a model for the L1 data cache of Nvidia GPUs. The paper extends the concept of memory access reuse distance [17] to adapt it to graphics processors by taking into account the concurrent execution of multiple threads, warps and work-groups. The authors statically analyse the memory accesses of CUDA kernels and use them to feed an analytical model of the memory system of a GPU. An improvement over this model is provided by Nugteren *et al.* [99]. The authors enrich the reuse distance model to take into account the cache latency and the presence of the miss status hold register. Both this aspects are ignored by the previous work. Nugteren *et al.* use low-level micro-benchmarking to reverse-engineer properties of the GPU cache to be used in the cache model. The accuracy of the model is validated comparing against hardware profiler counters extracted from the Nvidia profiler [1]. This model is proved to have accuracy comparable to the one of GPGPUSim: a fully-featured GPU simulator.

The other branch of research in this field studies the coalescing properties GPU programs. Boyer *et al.* [23] propose to instrument CUDA source code so to record all the accesses made by a CUDA application. With this information is then possible to compute the number of coalesced accesses and the presence of race conditions among CUDA threads. Fuzia *et al.* [41] propose to use dynamic instrumentation of NVIDIA PTX programs to guide the remapping of memory accesses. Their method first identifies uncoalesced memory accesses and then applies polyhedral code generation to remap accesses in local memory so to ensure coalescing. They show performance improvements between 2x and 35x by removing uncoalesced patterns. Karrenberg *et al.* [63] propose a technique to check whether consecutive OpenCL work-items access consecutive memory locations. Their goal is to verify the safety of the application of whole function vectorization [61] on CPU architectures which do not support gather and scatter operations. For each memory access the authors construct a logic formula using Presburger Arithmetic. A SAT solver is then used to verify or disprove that consecutive work-item access consecutive elements in memory. Their technique is extremely formal and capable of handling complex scenarios. Since most memory accesses of OpenCL programs are regular or indirect such formalization is rarely useful in practice. This is confirmed by the limited number of benchmarks that the authors use in the paper.

OpenCL Performance Portability This section presents work focused on the evaluation of performance portability of OpenCL. These papers compare the performance of OpenCL programs on multiple devices.

Rul *et al.* [111] consider three programs (`cp`, `mri-fhd` and `mriQ`) from the *Parboil* benchmark suite [2] and measure their execution time on Intel i7, Tesla, ATI FirePro and IBM Cell. The authors apply manual transformations by unrolling and vectorising the code. Exploring this search space they evaluate the performance of a benchmark optimised for a device and run on another. The penalties for not optimising code for the target processors are as high as 100x. Similarly Komatsu *et al.* [70] compare various manual optimisation strategies and evaluate performance difference between different runtime libraries: OpenCL and CUDA on Nvidia GPUs. The authors also make absolute comparisons in terms of raw execution times among the devices. This paper again confirms the idea that applications must be individually tuned for the target hardware.

These works on performance portability often just present performance numbers without providing a unique performance model capable of explaining performance in a portable way.

Pennycook *et al.* [106] focus on the performance evaluation of parallel LU decomposition. The authors provide an interesting comparison between OpenCL and Fortran code, showing how the Fortran PGI compiler can deliver better performance thanks to its broader set of optimisations. Further, the paper presents an MPI/OpenCL implementation on a 2-Nodes clusters also comparing with the possibility to perform device fission [67] *i.e.*, creating multiple virtual GPUs aggregating the physical cores of a single one.

Dolbeau *et al.* [34] studies the performance implications of using a single OpenACC program version [3] for a set of accelerators. The paper shows that it is possible to write a single program version to perform on all the devices within 12% of the optimal. This is a rather surprising result, in light of the previous body of work on the subject. A possible explanation for this is the limited size of code transformations, since the authors only look at work-group size.

Grewe *et al.* [46] present a tool to automatically transforms OpenMP code into OpenCL code to run on the GPU. The original CPU-friendly code is automatically transformed so to ensure coalesced accesses on the GPU. The new runtime system is completed with a classification tree model to decide on which device to execute a given piece of code depending on static code features and the problem input size.

3.4 Loop optimisations

This section gives an overview of loop optimisations for single- and multi-processors.

Most of the works cited here were developed for sequential or wide vector machines in the 1980s and 1990s. This body of work is still relevant today for highly parallel GPUs. Ideas such as reduction of redundancy, exploitation of spatial and temporal locality are important for modern hardware. Techniques such as tiling and unrolling can be successfully applied to GPGPU programs, the main difference is that the outer-most loop will be implicit and the data-parallelism is given by the programming model. In particular, the thread-coarsening transformation described in the thesis is remarkably similar to loop unrolling applied to the parallel loop of OpenCL programs.

Unrolling Unrolling replicates the body of the loop multiple times and reduces the number of iterations of the loop accordingly. This technique has been extensively evaluated for Fortran and C programs [35]. Bacon *et al.* [12] explain that the main advantages of unrolling come from reducing the overhead of the computation of loop iteration variables, increasing instruction level parallelism and improving data locality of registers and caches. Increasing the number of instruction in the loop body can significantly improve the quality of code generation. In particular, register allocation and instruction scheduling have more scope for optimisations. This property makes unrolling an effective transformation for statically scheduled architectures, such as VLIW machines [39]. The main drawback coming from the increase in the number of instructions in the loop body is the consequent increase in instruction cache pressure. Unrolling is relatively easy to implement for single-nested loops. Unrolling outer-most loops, instead, implies the replication of inner loops. The overhead that incurs in the replication of the inner loop can be mitigated by fusing the resulting loops. This technique is called *unroll and jam* [26]. Unrolling can also be applied at the assembly level as explained by Abrash [5].

Transformations for data locality Given the relative performance differences between computing cores and memory latencies efficient utilization of the cache hierarchy is central to achieve high performance. The work by Lam *et al.* [72, 133] aims at increasing the locality of loop nests with a mathematical model of the tiling transformation. The idea behind this is to reuse data in the cache as much as possible before it is evicted. This is achieved with a transformation of the iteration space called tiling. Tiling transforms the iteration space that spans the whole dimensions of the original matrices to work on sub-matrices. The tiling transformation is composed of two steps, the first one transforms the code inserting appropriate parameters that define the size of the blocks. The second stage involves the determination of such sizes. The optimal sizes depend on the memory layout of the system, in particular on the associativity and size of the caches. The authors provide a mathematical description of the iteration space based on polyhedral shapes. Linear algebra transformations are employed to define in mathematical form the tiling optimisation. These works rely on data flow analysis for loop nests developed by Feautrier [42].

3.5 GPU Optimisations

This section presents works in the context of code transformations for GPU hardware.

Ryoo *et al.* [112] present the first piece of research to propose optimisation for general purpose computation on Nvidia graphics hardware with CUDA. Coalescing of global memory accesses and tiling to local memory and loop unrolling are the primal optimisation techniques presented in the paper. The follow up paper from the same group [113] addresses the problem of pruning the optimisation space, so to speedup the search for the best performing configuration. The authors introduce two static metrics to estimate performance: Utilization and Efficiency. They then focus the search for the optimal configuration to the configuration laying on the Pareto frontier of the Utilization/Efficiency plot.

Mitigate branch divergence The high cost of executing branches on GPUs makes them targets for aggressive optimisations. The work by Han *et al.* [49] focuses on this aspect proposing two optimisation techniques. The first one is called *iteration delaying*. It targets divergent branches inside loops and it works by changing the execution order of iterations by grouping together iterations that all take the same path of a branch. The second one, *branch distribution*, moves out of a divergent region instructions that are shared between the two paths of a divergent branch. The two techniques have been applied manually to both micro-benchmarks and full application showing speedups up to 80% and 16% respectively. A follow-up work from the same group [50] proposes a way to reduce divergence of inner loops that have a trip count that varies across work-items. This kind of irregularity leads to poor utilization of the computing cores of the GPU. The authors propose to merge inner loops with outer ones, thus creating a single loop that has a constant number of iterations for all the threads in a warp. This technique improves performance up to 4.3x on Nvidia hardware. The main drawback is the limited applicability of the technique only to kernels with nested loops. Zhang *et al.* [142] address the problem of divergence by proposing a novel runtime system that attempts to make all the threads in a warp go through the same execution path. This is accomplished by remapping memory accesses so that all the threads in a warp loads values that make them not diverge. The authors propose two methods, one changes the layout of the data in memory, while the second redirects accesses using indirection. Taking into account the overhead of remapping the system gives within 1.3x and 2x over naive execution. Similar techniques have been applied by the to mitigate the problem of uncoalesced memory accesses [143].

Input-Aware optimisations Liu *et al.* [82] propose an auto-tuner to adapt optimisation options to the size of the input buffers. The authors introduce a source to source compiler based on Cetus called G-ADAPT that automatically searches the optimisation space using a hill-climbing approach spanning multiple input sizes. The optimisations used in the paper are limited to loop unrolling and thread block size. Once that the exploration terminates the tool builds a regression tree that maps input sizes to compiler options. This tree is meant to be used inside a JIT compiler to select the best options given the execution configuration. Samadi *et al.* [114] tackle the problem of tuning StreamIt programs for multiple input sizes. All the optimisations presented in the paper are specific for the streaming programming model but they correspond to common patterns: promotion to local memory, coarsening and coalescing. By coupling the optimiser with a performance model they are able to outperform CUBLAS on highly irregular input sizes.

3.5.1 Thread-coarsening

This section presents previous work on the thread-coarsening transformation for GPUs.

The first work to introduce thread-coarsening for general purpose computing for graphics processors is from Volkov *et al.* [129, 128]. They tackled the problem of optimising linear algebra codes on Nvidia GPUs. The benchmarks of interests were SGEMM, LU, QR and Cholesky factorizations. The authors performed detailed benchmarking of memory latencies and arithmetic performance for four GPU models. They showed that is possible to hide memory latencies by increasing the computational intensity of individual threads. By applying this and other manual optimisation they achieved state-of-the-art performance approaching full utilization of the computing resources. These impressive results were obtained with painstaking low-level performance benchmarking and analysis. The lack of compiler support for such optimisation make this approach non-portable across different architectures.

Yang *et al.* [139, 138] proposed a fully automatic optimising compiler for OpenCL. Their goal is to enable programmers to write easy unoptimised code and leave the burden of code transformations to the compiler. The optimiser takes care of generating coalesced accesses to global memory and to appropriately share data among threads in local memory. Thread-coarsening is among the set of transformations proposed.

Their compiler works source-to-source relying on Cetus [13] thus modifying the AST. This limits the applicability of their methodology only to simple linear algebra programs. Lastly they do not provide an explanation for the different effects that the same code transformation has on different devices.

Coarsening for Nvidia GPUs has been further proposed by Unkule *et al.* [125]. Their study of the transformation is limited to only five programs optimised by hand. The authors do not provide any performance evaluation of the results obtained.

Coarsening is similar to the Whole Function Vectorization algorithm proposed by Karrenberg *et al.* [61]. In the paper the authors describe in the detail the steps to merge together logical threads. Their most important contribution is an algorithm that use masking to linearise the control flow graph removing branches and managing loops appropriately. This is need to ensure uniformity of the code for all vector lanes. Performance results on multicore CPUs show both large improvements (up to 5x) and performance degradation for programs with scattered accesses to memory. The authors do not address the problem of determining when vectorization is beneficial.

3.5.1.1 Divergence Analysis

Thread-coarsening greatly benefits from an analysis pass known as *Divergence Analysis*. This is because divergence analysis can identify uniform instructions. The coarsening transformation can safely avoid the replication of uniform instructions since their output does not change across threads. Divergence analysis has been studied with the overall goal of reducing the overhead of redundant instructions in parallel programs. Coutinho *et al.* [28] introduced a domain specific language to express the relationship of assembly instructions with the register that identified the current thread. The DSL enables reasoning about the divergence properties of branch instructions. They use divergence analysis to apply a branch fusion transformation: reduce the number of divergent instructions by hoisting out of a divergent region common instructions between the two branches. Karrenberg *et al.* [62] use divergence analysis to improve their Whole Functions Vectorization algorithm [61]. By identifying branches that depend on the variable that identifies the current thread the authors are able to limit the linearisation of the control flow graph to only the branches whose outcome could change across threads. This improvements leads to speedups of up to 2x over naive vectorization.

Divergence analysis can be used to improve the run-time performance of code instrumentation, as showed by [40]. The authors use a combination of symbolic execution and instrumentation for analysing memory traces of GPU applications. Symbolic execution (much faster than instrumentation) can be used on non-divergent code sections. By applying divergence analysis the authors were able to find the largest regions of code that can be analysed with symbolic execution, thus reducing the penalty of instrumentation to a minimum. These techniques have been implemented within the Ocelot project [33].

Lee *et al.* [79] address the problem of reducing the overhead of issuing and executing uniform instructions among the threads in a GPU warp. They propose to replace conventional Nvidia assembly instructions with scalar instructions whenever the kernel properties allow so.

A similar study has been performed by Xiang *et al.* [136]. They propose to extend the register file specifically for uniform instructions. They analyse the profitability of uniform instructions optimisation from the point of view of energy efficiency as well run time performance. Finally Lee *et al.* [78] extend their previous work by exploring the design space of options concerning divergent branch management. They leverage divergence analysis to reduce the overhead of a complete software-based solution for the reconvergence of control-flow after a branch executes a divergent branch. Their improvements show that a software solution can match the performance of a hardware solution, thus simplifying VLSI design of the GPU architecture.

3.6 Iterative Compilation

Iterative compilation is the technique that aims at finding the fastest compiler configuration in a set of alternatives. Research in this field has been directed towards the definition of methods to reduce the time taken to explore the space of compiler configurations. Bodin *et al.* [21] address the problem of selecting a fast configuration for tiled and unrolled loops for embedded platforms. They reduce the search space by selecting only configurations at predefined values. Vuduc *et al.* [130] considered the problem of tuning tiling factors for a matrix multiplication program for super-computing applications. They employ a statistical method to stop the search based on the expected probability of find a configuration faster than the best found so far. Agakov *et al.* [6] rely on machine learning modelling to focus the search for the optimal configuration. The technique uses code features to compare the program of interest against a set of benchmarks used as training set. The search is then focused on the configurations that gave high performance for a programs similar to testing one. This methodology is particularly effective since it can be applied to any program/device search space. The authors demonstrate this by using both an embedded device and a high-end CPU for experiments. Pan *et al.* [104] approach the challenging problem of orchestrating compiler optimizations *i.e.*, decide which optimizations to apply to a piece of code and in what order. They focus the search by removing optimizations that have showed to have negative effect on performance during the search. In chapter 6 we propose a technique to improve the performance of iterative compilation for thread-coarsening by reducing the problem input size. We show that by carefully selecting an input size on which to perform the search we can find a coarsening configuration that improves performance on a much larger program size.

3.7 Compiler Optimisation Tuning with Machine Learning

This section gives an overview of the most relevant publications for compiler optimisation techniques that employ machine learning.

One of the first examples of statistical learning applied to compiler optimisation is the work by Calder *et al.* [25]. The authors introduce the concept of *evidence-based static prediction* which goal is to estimate branch probabilities using only static code features. They show that using neural networks and regression trees is possible to improve the accuracy of branch prediction over the state of the art. Stephenson *et al.* [118] introduce the concept of *Meta-optimisations*. They develop new heuristics using genetic programming to control the instruction scheduling and register allocation. Two pieces of work by Mosifrot *et al.* [96] and Stephenson *et al.* [117] have the goal of improving the heuristics that control loop unrolling. The first paper classifies loops based on the performance when unrolled using three categories: unchanged, improved, degraded. The authors employ static code features to train classification trees to characterize a loop. The system has been trained both on simple kernels and on loops coming from SPEC. Stephenson *et al.* [117] attempt to solve the same problem using a more sophisticated approach. The authors rely on lightweight profiler counters to correctly classify loops according to the best performing unrolling factor. Both a Nearest Neighbour approach and a Support Vector Machine are used for classification.

Chapter 7 presents a machine learning model for the prediction of the best coarsening parameter for GPU architectures. This problem presents similarities with the problem of tuning loop unrolling addressed by the papers presented in this section. The technique that we propose in this thesis takes inspiration from these works. The process of feature selection and model tuning that we employed is similar to the one described in [117]. The novelty of this thesis lies in adapting these techniques to the context of graphics processors for general purpose computation. This process involves mainly the development of new set of features capable of describing the behaviour of massively parallel applications. We describe the prediction technique in section 7.6.

Dubach *et al.* [37] address the problem of selecting the right set of optimisations for a given micro-architecture. The model selects compiler optimisations relying on a description of the architecture, source code features and profiler counters extracted with a single run of the unoptimised program. The model achieves 67% of the maximum performance achievable. The same group has proposed a *reaction-based* approach to compiler optimisation [36]. Given a target program the code is randomly transformed and executed on the target processor. The results of this exploration process are fed into a neural network that is trained to predict the relative performance of a new compiler configuration. Leather *et al.* [74] tackle one of the most challenging problems related to machine learning: the selection of features to characterize program behaviour. They propose generate program features using a language grammar specifically designed. Features are then selected based on their contribution to the performance of prediction models.

Stock *et al.* [119] developed a model to tune automatic loop vectorization. The authors developed a source to source compiler that apply high level loop optimisations to reshape loops with the goal making them more amenable to vectorization. The model then ranks loop versions based on their expected performance. This works is different from previous publications because the model is trained on features collected from assembly programs rather than source source code features. Zheng *et al.* [131] tackle the problem of mapping streaming applications written in StreaMIT to multi-core CPUs. Random streaming partitions are automatically generated so to have a large number of training data.

Sanchez *et al.* [115] apply Support Vector Machines to guide code specialization for the Java Virtual Machine. The technique aims at deciding which optimisations to enable for a given snippet of code. The machine learning model is trained with dynamic code features automatically extracted from the code during execution inside the JVM.

Lee *et al.* [76] address the problem of performance prediction in the context of high performance applications for supercomputing. The goal is to estimate the performance of the whole optimisation space given only a subset. The authors employ both a simple linear regression and a neural network. The paper also describes techniques to filter the features based on their correlation with the output variable.

3.8 Summary

This chapter has presented an overview of the publications related to the topic of this thesis. The survey stated with a history of GPU computing and the most important applications of general purpose programming on GPUs. We presented the most important techniques for performance modelling and analysis for GPUs. We then described classical loop compiler optimising transformations, these laid the foundations for modern compiler transformations for GPUs. Finally we surveyed the use of machine learning techniques for compiler optimisations.

The following chapters present the contributions of this thesis. The next one describes the compiler passes that we implemented in this work.

Chapter 4

Compiler Analyses and Transformations

This chapter describes the design and the implementation of the compiler passes in this thesis. We present all the stages of coarsening: divergence analysis, transformation, thread remapping and the tools that we implemented to support them. We also present the `SymEngine` tool for analysing coalescing of memory accesses. This chapter is structured as follows: Section 4.1 introduces the coarsening transformation presenting all its stages and providing an example. Section 4.2 explains what divergence analysis is and why it is useful in the context of coarsening. Section 4.3 describes the implementation of the coarsening pass. Section 4.4 presents the technique used to customize the compiler tool-chain, so to have a single portable compiler for all the GPU architectures we use. Section 4.5 shows how the iteration space is transformed after the application of the transformation. Section 4.6 presents `SymEngine`. This tool symbolically executes OpenCL kernels to extract information about coalescing of memory accesses. We describe its design and implementation and we also provide results of its accuracy. Section 4.7 concludes the chapter.

4.1 Introduction

Thread-coarsening is a transformation designed for data parallel programs. It increases the amount of computation performed by a single thread by replicating the instructions in the function body. To preserve the semantics of the original program it also reduces the number of work-items running concurrently on the computing device. The transformation of the kernel body takes place at compile-time, while the reduction of the number of work-items occurs at run-time. The stages of the transformation are listed in figure 4.1. Both the divergence analysis and coarsening have been implemented to work on LLVM-IR. The compiler pass is available as open-source [83].

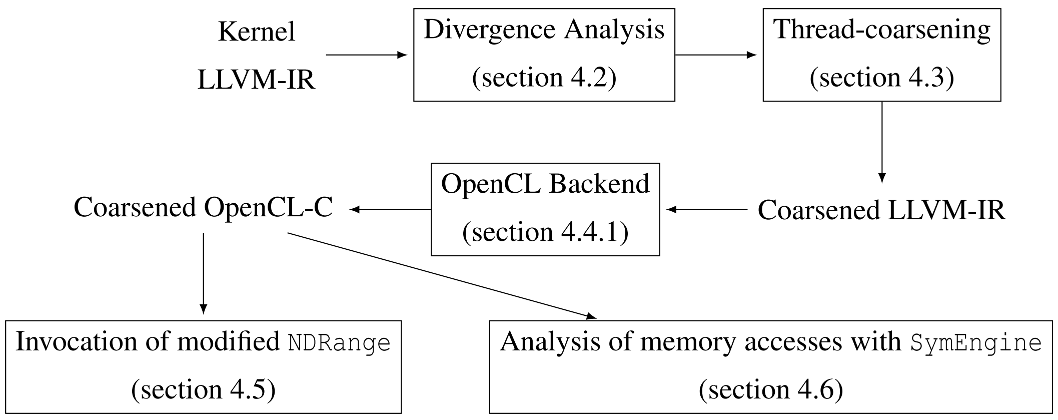


Figure 4.1: Stages of the coarsening transformation with references to sections with the description.

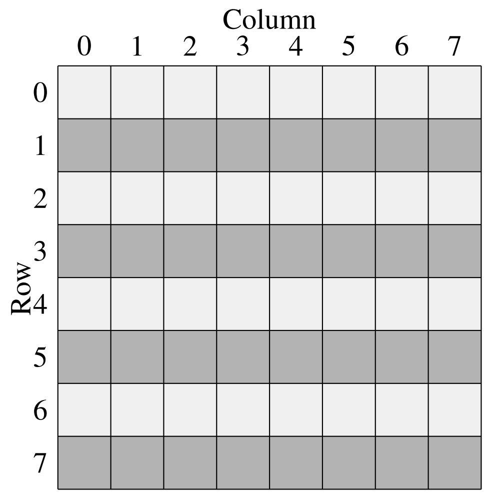
```
kernel void transposeMatrix(global float* input,
                            global float* output,
                            size_t width, size_t height) {
  size_t column = get_global_id(0);
  size_t row = get_global_id(1);
  size_t inputIndex = row * width + column;
  size_t outputIndex = height * column + row;
  output[outputIndex] = input[inputIndex];
}
```

(a) Original kernel code. Instructions that depend on *row* (the work-item-id along dimension 1) are underlined.

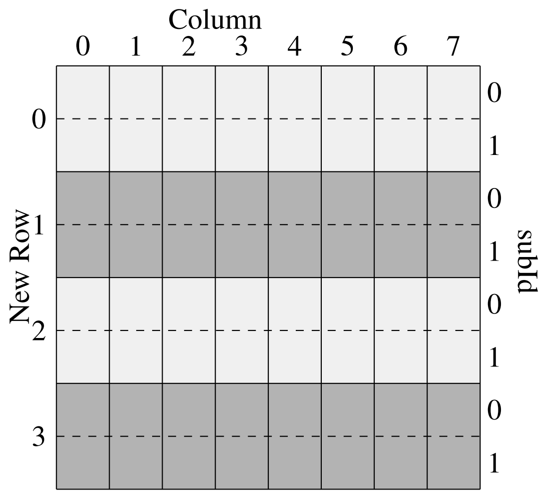
```
kernel void transposeMatrix(global float* input,
                            global float* output,
                            size_t width, size_t height) {
  size_t column = get_global_id(0);
  size_t row0 = 2 * get_global_id(1) + 0;
▶ size_t row1 = 2 * get_global_id(1) + 1;
  size_t inputIndex0 = row0 * width + column;
▶ size_t inputIndex1 = row1 * width + column;
  size_t outputIndex0 = height * column + row0;
▶ size_t outputIndex1 = height * column + row1;
  output[outputIndex0] = input[inputIndex0];
▶ output[outputIndex1] = input[inputIndex1];
}
```

(b) Kernel code after coarsening with factor 2. Replicated instructions are identified by ▶, while the computation of the remapping for the new instructions is highlighted in red.

Figure 4.2: Matrix transposition kernel before and after coarsening of factor 2 along direction 1.



(a) Original 2-Dimensional `NDRange` space for the kernel in figure 4.2a. Each cell represents a thread and it works on one element of the input matrix.



(b) `NDRange` space modified by the coarsening transformation. This is the iteration space for the kernel in figure 4.2b. Notice that the number of rows has been halved (since the coarsening factor is 2) and now each work-item works on two elements of the input matrix.

Figure 4.3: `NDRange` layout for the kernel in figure 4.2 before and after coarsening.

Thread-coarsening is a transformation designed for data parallel programs. It increases the amount of computation performed by a single thread by replicating the instructions in the function body. To preserve the semantics of the original program it also reduces the number of work-items running concurrently on the computing device. The kernel body is transformed at compile-time, while the reduction of the number of work-items occurs at run-time. The stages of the transformation are listed in figure 4.1. Both the divergence analysis and coarsening have been implemented to work on LLVM-IR. The compiler pass is available as open-source [83].

Figures 4.2a and 4.2b give an example of how thread-coarsening transforms an OpenCL kernel that performs matrix transposition. Being a two-dimensional kernel, coarsening can be applied in both directions. In this example, the coarsening transformation works on direction 1, thus each transformed work-item will work on two rows, as figure 4.3b shows. Each divergent instruction is replicated a number of times equal to the coarsening factor minus one. Replicated LLVM-IR instructions are cloned and inserted right after the original ones. In figure 4.2b the coarsened instructions are identified by the symbol ►.
The newly inserted instructions will work on the data points referenced by the work-items before the merging in the original iteration space as shown in figure 4.4. The $G_x \cdot G_y$ work-items

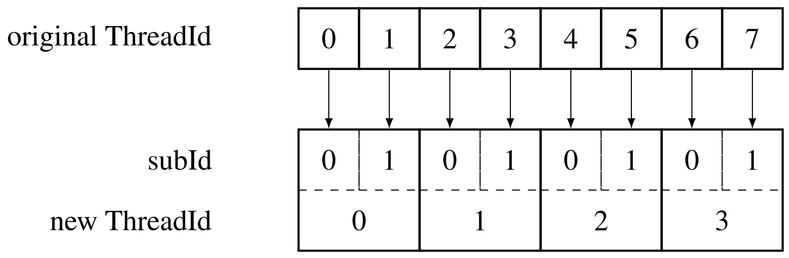


Figure 4.4: Mapping between the work-items in the original `NDRange` space (top of the figure), with the work-items in the transformed `NDRange` space (bottom).

in the original iteration space are identified by $origTid$, where $origTid \in [0, G_x \cdot G_y - 1]$. The symbols G_x and G_y are defined at page 9. The new iteration space resulting from coarsening with factor $factor$ is $newTid \times subId$ where $newTid \in [0, (G_x \cdot G_y - 1)/factor]$ and $subId \in [0, factor - 1]$ is the sub-index of the work-item being merged. Coarsening is defined by the following transformation:

$$origTid \mapsto newTid \times subId \tag{4.1}$$

with constraint

$$newTid \cdot factor + subId = origTid. \tag{4.2}$$

The coarsening transformation is controlled by three integer parameters:

Direction defines which direction to aggregate work-items in the iteration space. This option is meaningful only for multi-dimensional iteration spaces. In the formulas in the remainder of the chapter we call this parameter $direction$. In our experiments this parameter can have the following values: $[0, 1]$ depending on the dimensionality of the benchmark.

Factor defines how many times to replicate the body of the kernel, and, conversely, by how many times to shrink the `NDRange` space. In the formulas in the remainder of the chapter we call this parameter $factor$. In our experiments this parameter can have the following values: $[1, 2, 4, 8, 16, 32]$

Stride defines how many work-items to interleave between the ones that are merged together. This option is described in detail in section 4.3.2. In the formulas in the remainder of the chapter we call this parameter $stride$. In our experiments this parameter can have the following values: $[1, 2, 4, 8, 16, 32]$ Notice that a stride greater than 1 is only meaningful for a coarsening factor greater than 1.

The following sections describe in detail the individual steps of the transformation.

```
 1: function FindDivergentInstructionsForStraightLineCode(kernel, direction)
 2:     seeds=get_global_id(direction) ∪ get_local_id(direction)
 3:     return FindDivergentInstructionsGivenSeeds(kernel, direction, seeds)
 4: function FindDivergentInstructionsGivenSeeds(kernel, seeds)
 5:     divergentInsts = ∅
 6:     workingSet = seeds
 7:     while workingSet ≠ ∅ do
 8:         inst = pop(workingSet)
 9:         divergentInsts = divergentInsts ∪ inst
10:         users = ForwardSlicing(inst)
11:         workingSet = workingSet ∪ (users / divergentInsts)
12:     return divergentInsts
```

Figure 4.5: Pseudo-code of divergence analysis for straight line programs only. The algorithm analyses the body of the given kernel and returns the set of instructions which depend on the work-item-id of the given direction.

4.2 Divergence Analysis

This section describes divergences analysis and its use for thread-coarsening.

The goal of divergence analysis is to distinguish the instructions in the kernel body in two types: the ones which depend on the position of the work-item in the `NDRange` space and the ones which do not. In the example of figure 4.2a divergent instructions that depend on *row* (*i.e.*, g_y, refer to page 9) are underlined. The analysis is based on two simple ideas: the work-item-id for the coarsening direction is divergent and instructions that use divergent instructions are divergent themselves. The algorithm of the analysis of straight-line programs is given figure 4.5. First, the output of `get_global_id` and `get_local_id` for the coarsening direction is used to initialize the `workingSet`) of instructions. Each instruction in the set is then considered individually adding it to the output set `divergetInsts`). The instructions that use the current instructions are then added to the working set. This phase is called code-slicing [132] (see code line number 10). Execution terminates when the working set is empty. This algorithm can successfully identify data flow dependencies between instructions in the kernel body and the work-item-id. Notice that this is a conservative approach: we mark divergent all the instructions that could have divergent behaviour. The implementation is guaranteed to finish by the fact that a given divergent instruction is placed in the `workingSet` only once and never reinserted again.

Management of branches Program with branches need special care. Consider figure 4.6. Here the branch in BB_A is divergent, the blocks that are control-dependent on it and its immediate post dominator form a *divergent region*. The output of the phi-node in BB_D (which is the

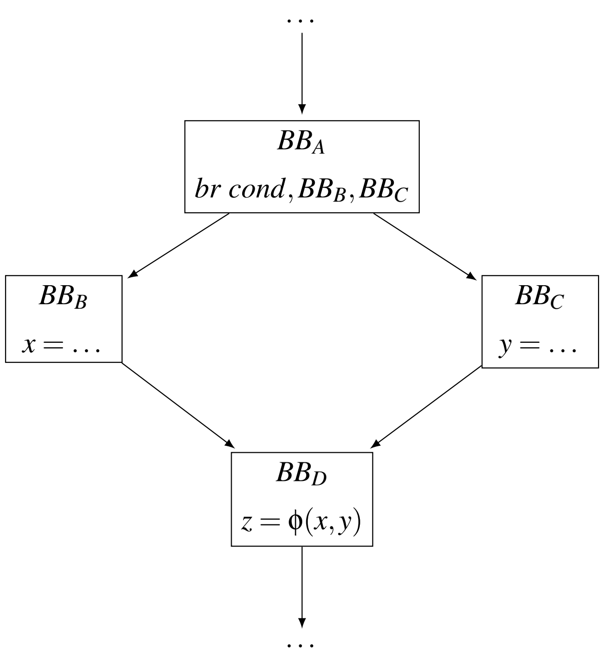


Figure 4.6: Assume that $cond$ is divergent, this implies that the branch in BB_A is divergent too, making this a *divergent region*. The phi-node in BB_D has to be marked divergent because it belongs to a block that is the immediate post dominator of a divergent branch.

```
function FindAllDivergentInstructions(kernel, direction)
    seeds=get_global_id(direction) $\cup$ get_local_id(direction)
    cfg = BuildCFG(kernel)
    divergentInsts = $\emptyset$
    while seeds $\neq \emptyset$ do
        currentDivergentInsts = FindDivergentInstructionsGivenSeeds(kernel, seeds)
        divergentInsts = divergentInsts $\cup$ currentDivergentInsts
        divergentBranches = GetBranches(currentDivergentInsts)
        cfg = MarkImmediatePostDominatorsAsDivergent(divergentBranches, cfg)
        phis = GetDivergentPhisFromDivergentBlocks(cfg)
        seeds = phis
    return divergentInsts
```

Figure 4.7: This iterative algorithms finds all the divergent instructions starting from the work-item ids. It first marks as divergent instructions that depend on the work-item id. It then marks phi-nodes in blocks that are immediate post-dominators of divergent branches as seeds for the next iteration. The algorithm stops when there are no more seeds.

```
function KernelCoarsening($kernel$, $direction$, $factor$)
    allDivergentInsts = FindAllDivergentInstructions($kernel$, $direction$)
    divergentBranches = GetBranches(allDivergentInsts)
    divergentRegions =
        GetDivergentRegions(divergentBranches)
    divergentInsts =
        GetInstsOutsideDivergentRegions(allDivergentInsts)
    for all inst in divergentInsts do
        for subId in $[(factor - 1)\ \dots\ 1]$ do
            newInst=ReplicateInst(inst, subId)
            InsertInst(inst, newInst)
    for all region in divergentRegions do
        for subId in $[(factor - 1)\ \dots\ 1]$ do
            newRegion=ReplicateRegion(region, subId)
            InsertRegion(region, newRegion)
```

Figure 4.8: Pseudo-code of thread-coarsening. The algorithm transforms $kernel$ replicating instructions and regions that are divergent along $direction$. Instructions and regions are replicated $factor - 1$ times, so to increase the amount of work performed by a single thread.

immediate-post dominator of BB_A) defines the values of the alive instructions of the divergent region. This phi-node should be marked as divergent, even though its arguments may not be. This is because its output depends on the result of the divergent branch in BB_A. The phi-node is divergent even though its arguments are not, since they might have been replaced by constants coming from BB_B and BB_C. We solve this by marking the immediate post dominators of a divergent branch as a divergent block. In the case in the example BB_D is the immediate post dominator of BB_A. Once we identify divergent blocks we mark their phi-nodes as divergent and use these as seeds for the next iteration of code slicing. The overall implementation is given in figure 4.7. Given a well-structured control flow graph the algorithm finds all divergent instructions in the program. Informally, a well-structured CFG is one that can be generated using `if-else` and `while` statements. The `-structurizecfg`[1] pass transforms an unstructured program into a structured one. The next stage in the compilation pipeline is the coarsening transformation.

4.3 Coarsening

Once that divergent instructions have been identified we can apply the coarsening transformation. Each divergent instruction is replicated a number of times equal to the coarsening factor minus one. The pseudo-code of our transformation is given in figure 4.8. Notice the function `replicateInst` at line 10. It clones the given instruction modifying the operands of the new one so to make it work on the data referenced by the given *subId*. The replicated instruction is then added to the kernel function appending it after the original instruction. This is done by `insertInst` at line 11.

4.3.1 Control Flow Management

Special care must be taken in the presence of control flow. A *non-divergent* (uniform) branch means that is taken or not taken by all the work-items in the `NDRange` space. With respect to coarsening, uniform branches can be safely maintained as their are: coarsening will replicate the divergent instructions in the region controlled by the branch without modifying the CFG. If instead the branch is *divergent* such approach is unsafe and does not preserve the semantics of the program. We manage these cases treating the whole CFG region controlled by a divergent branch as a single statement. All the basic blocks enclosed by the branch and its immediate post-dominator are called a *divergent region*. During coarsening divergent regions are cloned and placed right after the original ones. Divergent regions are identified starting from divergent branches (figure 4.8, line 5) and replicated along with divergent instructions (line 14). Region cloning presents two critical aspects:

- Unstructured control flow. Regions with multiple exit points cannot be safely copied. We solve this issue by structurising the CFG as a preprocessing step. This is done with the `-structurizecfg` LLVM pass.

- Region cloning is unsafe in the presence of synchronization (`barrier` function call) inside regions since this would break the semantics of the program. Our transformation instead is always safe since we clone only divergent regions. Here we exploit the property that the OpenCL specification ensures that barriers can only be placed in regions which behave in non-divergent fashion. Regions with barriers are therefore always treated as non-divergent.

Region replication has a significant negative effect on performance since it implies extensive code duplication. For more information about its impact on execution time see chapter 5.

[1]http://llvm.org/docs/doxygen/html/StructurizeCFG_8cpp.html

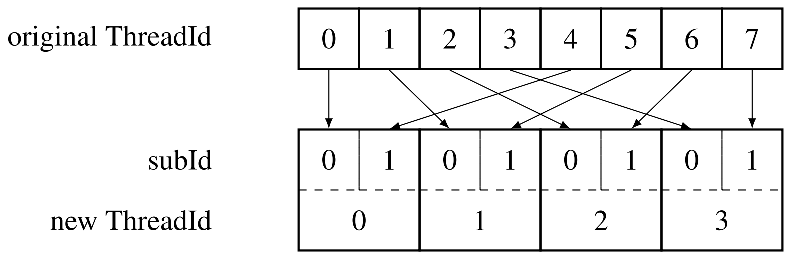


Figure 4.9: Mapping between the work-items in the original `NDRange` space (top of the figure), with the work-items in the transformed `NDRange` space (bottom) when a stride of 4 is applied.

4.3.2 The stride option

All the GPUs considered in this work have a warp-based architecture. This type of execution model greatly benefits from coalesced accessed to memory (see section 2.2.1). Coarsening programs that have coalesced accesses might make them lose this favourable property. For this reason, we propose an extension to basic coarsening which restores the original coalesced access pattern. We merge together non-consecutive work-items in the original space, we merge work-items that are separated by a stride. The number of work-items to interleave between merged work-items is the value of the stride parameter. The new constraint on the thread iteration space transformation from formula 4.1 is:

$$\lfloor \frac{newTid}{stride} \rfloor \cdot factor \cdot stride + newTid \cdot \bmod(stride) + subId \cdot stride = origTid \tag{4.3}$$

The effect of stride on the remapping is showed in figure 4.9. Figure 4.10 shows the memory locations accessed by 8 work-items divided into 2 warps when performing a coalesced load (`x = A[work_item_id]`) in 3 different cases: no coarsening (4.10a), coarsening by factor 2 and stride 1 (4.10b), and coarsening by factor 2 and stride 4 (4.10c).

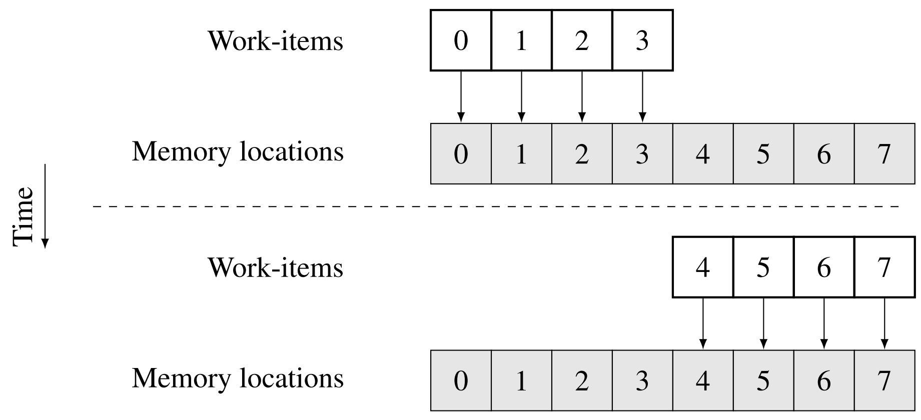
4.3.3 Limitations

In our implementation of the coarsening transformation we did not consider atomic instructions or volatile data. We do not support these language constructs and none of the benchmark we use in the next chapter have these features. We did not evaluated our transformation on texture memory, even tough our implementation should be able to support it.

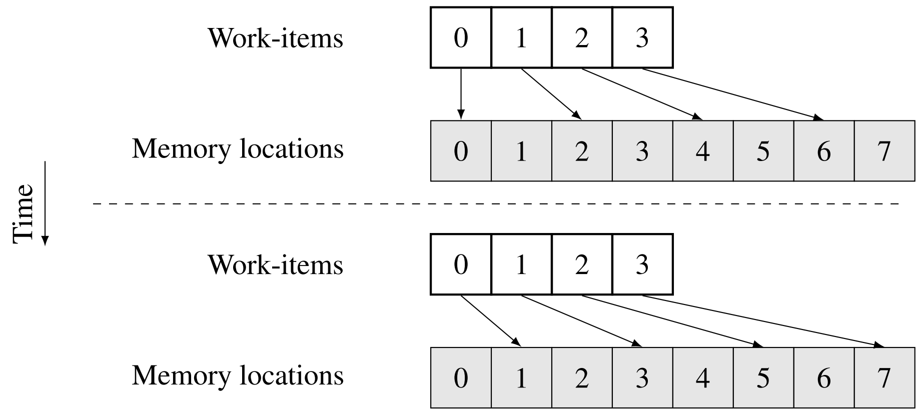
4.3.4 Effects of Coarsening on Program Structure

This section describes the effects of coarsening on the program shape with an example.

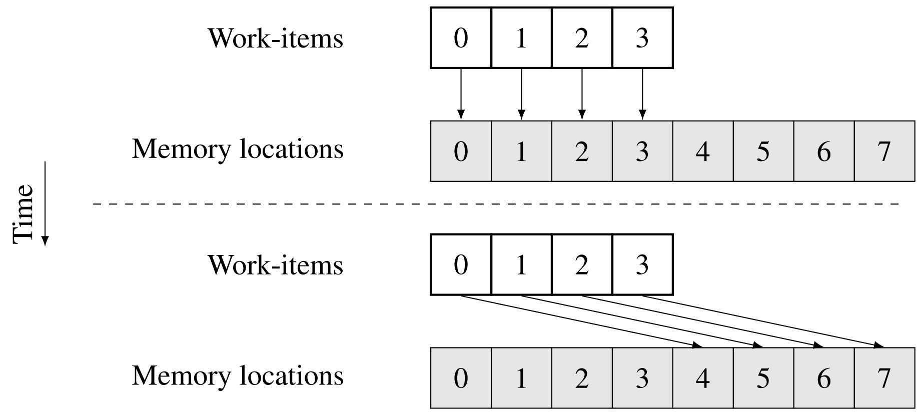
At the fundamental level coarsening attempts to reduce the amount of redundant instructions in OpenCL kernel code. This is done by reusing the computation of uniform instructions to process multiple output data points. Parallel kernels use a number of uniform integer instructions (such as loop iterations variables) to compute output values, usually divergent floating



(a) Original memory accesses for `x = A[work_item_id]`. Threads are divided in two warps with consecutive threads accessing consecutive memory locations.



(b) Memory accesses for the coarsened version of the load: `x1 = A[2 * work_item_id]; x2 = A[2 * work_item_id + 1];`. Accesses are no longer coalesced.



(c) Recovered coalesced access pattern using a stride of 4: `x1 = A[work-item-id % 4]; x2 = A[work-item-id % 4 + 4];`.

Figure 4.10: Three different memory access patterns from an originally coalesced load. No coarsening (a), coarsening without stride (b), coarsening with stride (c). In the original configuration (a) threads are divided into two warps: $[0,3]$ and $[4,7]$. When coarsening by factor two (b–c) the number of warps is halved to one.

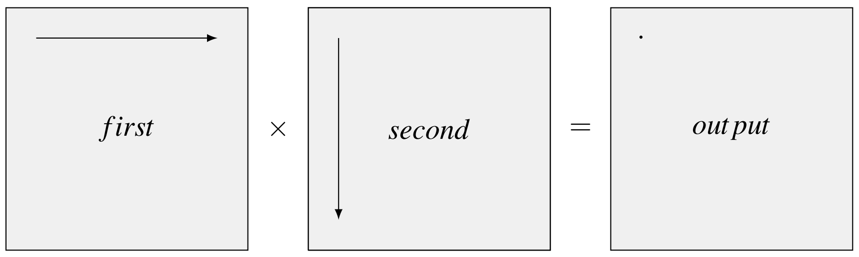
```
kernel matrixMultiplication(global const float* first,
                            global const float* second,
                            global float* output,
                            size_t size) {
  size_t row = get_global_id(1);
  size_t column = get_global_id(0);
  float result = 0.f;
  for (size_t index = 0; index < size; ++index) {
    result += first[row * size + index] *
              second[index * size + column];
  }
  output[row * size + column] = result;
}
```

(a) Original kernel code for a two-dimensional matrix multiplication benchmark. Notice that the `for` loop and the load from `second` do not depend on `row`.

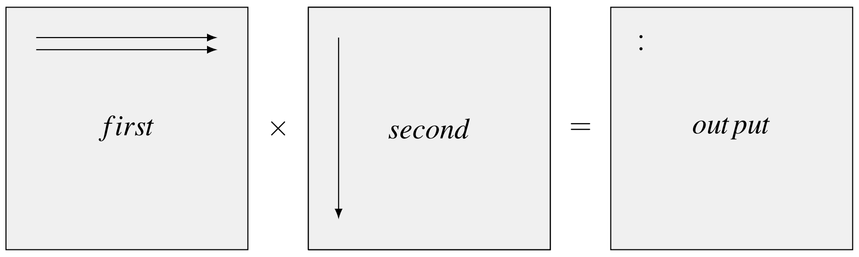
```
kernel matrixMultiplication(global const float* first,
                            global const float* second,
                            global float* output,
                            size_t size) {
  size_t row1 = 2 * get_global_id(1) + 0;
▶ size_t row2 = 2 * get_global_id(1) + 1;
  size_t column = get_global_id(0);
  float result1 = 0.f;
▶ float result2 = 0.f;
  for (size_t index = 0; index < size; ++index) {
    float secondValue = second[index * size + column];
    result1 += first[row1 * size + index] * secondValue;
▶   result2 += first[row2 * size + index] * secondValue;
  }
  output[row1 * size + column] = result1;
▶ output[row2 * size + column] = result2;
}
```

(b) Kernel code after coarsening with factor 2 along dimension 1. Replicated instructions are identified by ▶. The uniform value from `second` can be reused in a register, without issuing a second load.

Figure 4.11: Two version of matrix multiplication in OpenCL. The code computes $output = first \cdot second$.



(a) Memory accesses for the matrix multiplication program in figure 4.11a.



(b) Memory accesses for the matrix multiplication program in figure 4.11b. Notice that only one column is traversed by the matrix *second*.

Figure 4.12: Memory locations accessed by a single work-item for the two versions of matrix multiplication in figure 4.11, one is the original version (a), the other is the coarsened one (b). Notice that the original version stores a single value to the output matrix. This is identified by the dot.

point values. With coarsening we enable the possibility to reuse integer instructions for the computation of multiple floating point outputs. This is realized in multiple ways:

- Reuse of integer instructions such as loop iteration variables for loops that are not divergent.
- Reuse of uniform branches. When applying coarsening is possible to avoid the replication of branches that do not depend on the work-item-id. This reduces the overhead of branching.
- When two work-items that load the same value from memory are merged together they can reuse the loaded value in a register and avoid a redundant load instruction. [2] This case is exemplified in figures 4.11 and 4.12. Figure 4.11a shows the kernel body of a matrix multiplication example. The values loaded from `second` are uniform with respect to the coarsening direction: number 1. This avoids replicating the load from `second`. This is visually showed in the schemas in figure 4.12. The arrows represent the memory locations accessed by a work-item. In the default case a work-item traverses a column of `first` and a row of `second` writing a single memory location for `output`. In

[2] Refer to `http://blog.llvm.org/2009/12/introduction-to-load-elimination-in-gvn.html` for more information about redundant load elimination in LLVM.

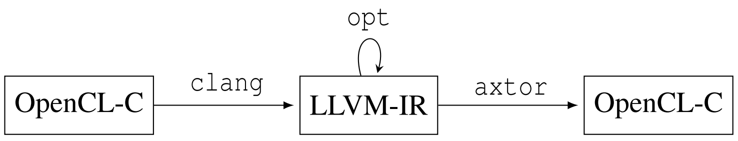


Figure 4.13: Schema of the use of axtor. `clang` generates an LLVM-IR module, `opt` runs optimisation passes on it and `axtor` translates it back to OpenCL-C code.

the coarsened case a work-item computes two output values using only one column of `second`. This property gives significant improvements on performance as we show in section 5.6.1.

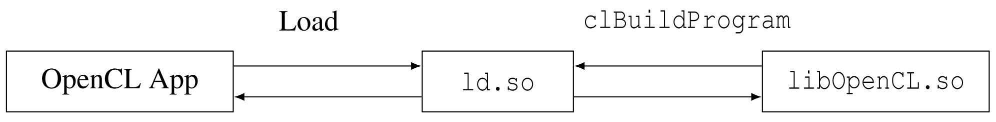
4.4 Compiler Set-up

This section describes the compiler infrastructure that we use to evaluate thread-coarsening.

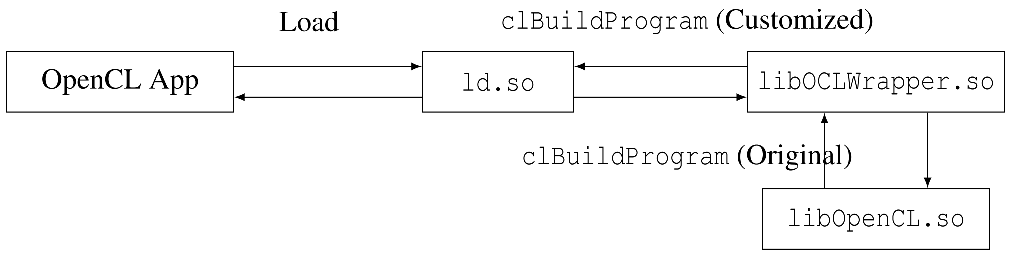
4.4.1 OpenCL Backend

GPUs by different vendors use different ISAs. This poses a problem for compiler portability, one of the main goals of this work. Extensive cross-architectural evaluation of compiler transformations for GPUs is only possible with a source-to-source compiler. At the time of this thesis, standard proposals for cross-platform intermediate representation (*e.g.* HSA [55] and SPIR [66]) are not yet available for consumer products. In 2012 Nvidia merged its PTX backend with the LLVM open-source repository. At the same time Nvidia released a mathematical library meant to be linked against OpenCL programs to provide the implementation of built-in functions [140]. This solution provides the full compiler support for Nvidia GPUs. We chose not to use this since it is of no help for AMD GPUs. Previous works that develop compiler transformations for multiple GPU architectures (such as [138]) employ the Cetus compiler [13]. This source-to-source compiler builds an IR that represents the abstract syntax tree of the program. Code transformation then effectively transform the AST. This type of approach for compiler transformations is limited, and makes analysis based on data flow and subsequent transformations hard to implement.

Axtor Our solution to this problem is to employ a so-called OpenCL backend, *i.e.*, a compiler pass that reads LLVM-IR and emits an equivalent piece of OpenCL-C code. Our compiler toolchain is depicted in figure 4.13. The original OpenCL-C source code is translated to LLVM-IR using the `clang` open-source C front-end. We then apply analysis and transformation passes on the LLVM-IR using the `opt` tool. The transformed LLVM program is then translated back to OpenCL-C using the `axtor` LLVM OpenCL-backend [95]. `axtor` has two main tasks:



(a) Normal loading of the `clBuildProgram` function from the `libOpenCL.so` shared library.



(b) Interception of the invocation of `clBuildProgram` by `libOCLWrapper.so` using `LD_PRELOAD`

Figure 4.14: Schemas representing a normal invocation of `clBuildProgram` using `libOpenCL.so` (a) and interception of the same invocation by preloading `libOCLWrapper.so` (b).

transform the unstructured control flow regions of the program into structured ones, and then translate LLVM-IR instructions into C statements. The transformed OpenCL program is then fed into the hardware vendor proprietary compiler for code generation. The OpenCL-C code generated by axtor is highly redundant, since it keeps the same shape of the SSA code. This is not actually an issue since the transformed code is then optimised by the vendor compiler.

4.4.2 Customization of the Compilation Process

To be able to test compiler options for OpenCL-C kernels we have to change the compilation process of device code. OpenCL-C programs are compiled by the `clBuildProgram` function which invokes the hardware vendor proprietary compiler. This compilation step can be customized without changing the OpenCL host application. This is possible thanks to the dynamic library loading mechanism used by the GNU/Linux operating system.

Hardware vendors ship OpenCL implementations in the form of a shared library called `libOpenCL.so`. At build time, the application developer has to link his C program against the OpenCL library. At run-time, the GNU/Linux loader (named `ld.so`) searches for OpenCL API functions in the dynamic libraries contained in the directories listed in the `LD_LIBRARY_PATH` environmental variable. Once that the appropriate function is found the loader executes it (see figure 4.14a). This setting can be changed at run-time by using the `LD_PRELOAD` environmental variable. When this variable is set to contain the path of a shared library, the loader looks for a dynamic symbol in that library first before looking anywhere else. Figure 4.14b shows how the preloading of the new shared library changes the function resolution process.

Here follows the procedure to change the compilation process of an OpenCL kernel:

- Implement a function called `clBuildProgram` with the same signature of the original. This function will perform the following operations:
  - Extract the program source code from the `cl_program` object in input.
  - Call the `clang` executable to generate LLVM-IR from OpenCL-C code.
  - Read from environmental variables the compiler options that define the transformation to apply to the code.
  - Call the `opt` executable to transform the LLVM-IR code.
  - Call the `axtor` executable to translate the LLVM-IR code back to OpenCL-C. `axtor` is presented in section 4.4.1.
  - Invoke the original `clBuildProgram` function to compile the transformed OpenCL-C code to machine target code.
- Compile the function into a shared library (in this example we call it `libOCLWrapper.so`)
- Set the environmental variable `LD_PRELOAD` to point to `libOCLWrapper.so` and set all the options controlling the kernel compilation process.
- Execute the original OpenCL program.

4.5 Invocation

In order to preserve the semantics of the original program it is necessary to change the number of work-items launched on the computing device. The iteration space has to be reduced by the thread-coarsening factor. This is done without recompilation of the program using `LD_PRELOAD` technique (see section 4.4.2). We wrote a library that provides a replacement for `clEnqueueKernel`. The new function reads both the input arguments given by to the original function and environmental variables for the coarsening *direction* and $factor$. The wrapper function then shrinks the `NDRange` space along *direction* by $factor$. The new `NDRange` space thus defined is then given in input to the OpenCL function by the hardware vendor to execute the kernel.

4.6 Memory Accesses Modelling

This section describes an analysis useful for characterizing the memory access pattern of OpenCL kernels. The tool we present here is used for the selection of the stride parameter in section 7.5. In particular we aim to estimate the number of memory transactions performed by the kernel. This corresponds to the number of cache lines transferred from main memory

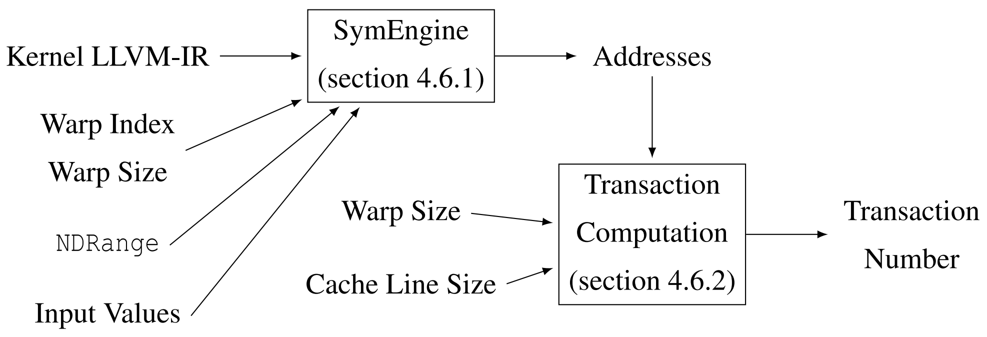


Figure 4.15: Overview of the computation of hardware memory transactions.

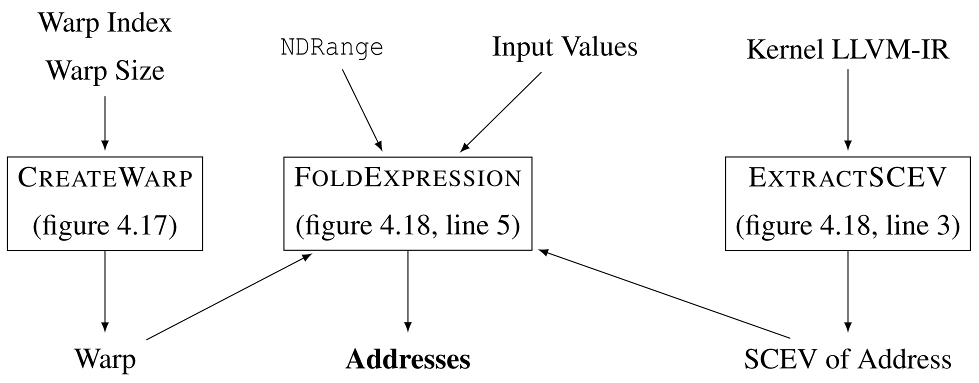


Figure 4.16: Components of the `SymEngine` tool.

to L1 cache or vice versa. See section 2.2.1 for a description of how GPU memory transactions work. To obtain this information it is necessary to estimate the addresses touched by a load or a store when making a memory request. This is done using a compiler pass that symbolically executes the code of the kernel function. We call the tool the performs this analysis `SymEngine`. An overview of the computation of the memory accesses is given in figure 4.15. In section 7.5 we present how we use the output of our memory modelling to select a suitable stride parameter.

4.6.1 `SymEngine`

The goal of `SymEngine` is to compute what addresses in a buffer are touched during the computation. `SymEngine` takes the following input parameters:

- LLVM-IR of the kernel function.
- Values of the integer kernel arguments. The values that are important for the task are the ones used in the computation of addresses, these are usually the sizes of the input and output buffers.
- The dimensions of the `NDRange` space $((G_x, G_y), (S_x, S_y))$.

```
1: function CreateWarp(($S_x$,$S_y$), $warpIndex$, $warpSize$)
2:     warp = []
3:     firstThreadIndex = $warpIndex \cdot warpSize$
4:     for all workItemIndex in [firstThreadIndex ... firstThreadIndex $+ warpSize$] do
5:         localY = workItemIndex / $S_x$
6:         localX = workItemIndex % $S_x$
7:         warp.append((localX, localY))
8:     return warp
```

Figure 4.17: Algorithm to compute which work-items compose a given warp. S_x and S_y are the dimensions of the work-groups.

- The warp to simulate. This is identified by two sets of coordinates: the first one identifies the work-group it belongs to and the second is the index of the warp in the work-group.

Figure 4.16 shows the stages to compute addresses starting from the inputs of `SymEngine`. The first step of `SymEngine` is to identify which work-items in the `NDRange` space are part of the warp to be simulated. The function CreateWarp in figure 4.17 shows how this is done for two-dimensional kernels. The algorithm starts by computing the index of the first work-item in the warp (line 3). Then, for all the $warpSize - 1$ subsequent work-items the algorithm computes the 2D position of the work-item in the work-group with dimensions S_x and S_y (lines 5 and 6). The function returns a list of all the 2D work-items in the warp.

The second stage is to compute what addresses the warp actually touches, this is implemented by the function ComputeAddresses in figure 4.18. The first step of the algorithm to extract a so-called Scalar Evolution representation (SCEV) from a memory instruction (ExtractSCEV in line 3). See section 2.3.1 for a description of SCEV. SCEV provides a mathematical representation of the computation of the address of a memory instruction. Then, for each work-item in the warp the SCEV expression is folded (FoldExpression in line 5) replacing all the values passed as input such as: work-item id, iteration space (S_x, S_y, G_x, G_y) and kernel arguments (line 5).

4.6.2 Transaction Computation

Once that we know which addresses are touched by the work-items in a warp we compute the number of transactions performed by the work when executing the memory instruction. Once that the addresses for the memory instruction of each kernel in the simulated warp the next step is the computation of the number of transactions. This is done with the algorithm in figure 4.19. ComputeTransactionNumber takes in input the $warpSize$ addresses accessed by the warp and the size of a cache line. It then computes in which cache line the addresses

```
1: function ComputeAddresses(memoryInst, warp, ($S_x$, $S_y$, $G_x$, $G_y$), kernelArguments)
2:     addresses = []
3:     scev = ExtractSCEV(memoryInst)
4:     for all workItem in warp do
5:         address = FoldExpression(scev, workItem, ($S_x$, $S_y$, $G_x$, $G_y$), kernelArguments)
6:         addresses.append(address)
7:     return addresses
```

Figure 4.18: Algorithm to compute the set of addresses touched by *warp* when executing *memoryInst*.

```
1: function ComputeTransactionNumber(addresses, cacheLineSize)
2:     cacheLines = addresses / cacheLineSize
3:     cacheLinesNumber = GetNumberOfUniqueValues(cacheLines)
4:     return cacheLinesNumber
```

Figure 4.19: Algorithm to compute the number of transactions made by a warp that is accessing the given addresses and with a GPU of the given cache line size.

fall by dividing them by the cache-line size. The number of transaction is then the number of unique cache lines touched.

To keep the presentation simple, the algorithms reported in this section do not deal with unknown addresses. Indirect accesses to memory or addresses computed using complex arithmetic make the construction of SCEV expressions fail (line 3 of figure 4.18). We manage these cases with a fall back that returns the maximum number of transactions. This number is equal to the *warpSize*, as if all the threads in a warp make individual transactions. This is an reasonable approximation since it is unlikely that indirect accesses to memory show coalescing properties.

The computation of the transaction number can work in two modes. The first one simulates the execution of a single warp. In this mode, the estimate of the total number of transactions made by the application is the number of transactions per memory instruction for the simulated warp multiplied by the total number of warps in the `NDRange` space. The second mode works simulating all the warps in a work-group and then multiplying the results for the number of work-groups. This mode takes into account the shape of control flow graph. In specific, we build the program dependency graph (PDG) of the application [43] and we annotate each node (a basic block) with a label. This label is a SCEV expression obtained combining the conditions that control the execution of the parents of the block in the PDG. The label is evaluated by each work-item that symbolically executes the block. If the folding of the SCEV returns `true` the memory instructions in the block are symbolically executed and the number of transactions

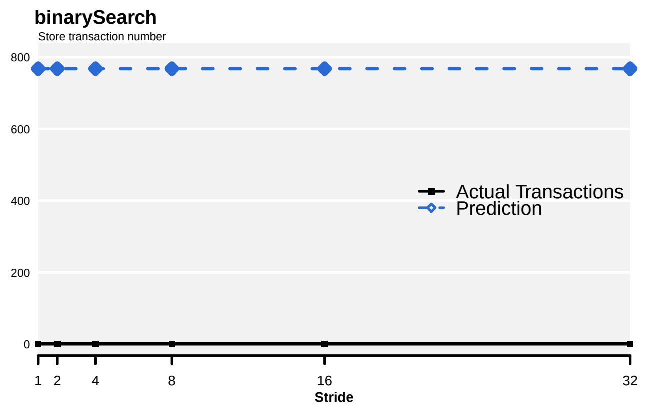


Figure 4.20: Comparison between actual and predicted transaction number for `binarySearch` varying the stride parameter on *Fermi*.

is computed. If the folding returns `false` (meaning the block is not executed by the current work-item) then the block is skipped. Baghsorkhi *et al.* [14] use this same methodology for their GPU simulator to statically decide what warps execute what blocks in a CUDA program. For the purposes of the prediction of the stride factor the first mode of execution is accurate enough to give high prediction accuracy for most cases. We employ the second method the `reduce` kernel from the AMD-SDK benchmark suite. This program computes the sum of all the values in an array. Efficient parallel reductions employ a hierarchical method that makes a single work-item per work-group perform a store to the output buffer. This is done using a branch like the following:

```
if(local_work_item_id == 0) {
  outputBuffer[position] = ... ;
}
```

This branch can be effectively predicted by `SymEngine` and account for the store only for the first work-item in the work-group.

4.6.3 Evaluation

This section provides experimental data to validate accuracy of the output of `SymEngine`. Table 4.1 reports the correlation between our prediction for loads and stores transactions and the actual number collected with the Nvidia profiler on the *Fermi* device for all the kernels that we use in our experiments. The meaning of the correlation coefficient is described in section 2.7.1. The multiple versions of the benchmark have been computed varying the coarsening factor and the stride. The correlation is above 0.9 in 29 cases out of 34, usually very close to 1. This signifies the `SymEngine` is capable of statically approximating the execution-time memory accesses of OpenCL kernels. There are a few notable exceptions: `binarySearch`, `dwtHaar1D` and `convolution`. The issue with `binarySearch` is that the whole program only makes a store

Kernel	Source	Store Correlation	Load Correlation
`binarySearch`	AMD SDK	NA	0.985
`blackscholes`	Nvidia	1.000	1.000
`convolution`	AMD SDK	1.000	-0.313
`dwtHaar1D`	AMD SDK	-0.737	1.000
`fastWalsh`	AMD SDK	0.995	0.995
`floydWarshall`	AMD SDK	0.903	0.982
`mriQ`	Parboil	1.000	0.998
`mt`	AMD SDK	1.000	1.000
`mtLocal`	AMD SDK	1.000	1.000
`mvCoal`	Nvidia	1.000	1.000
`mvUncoal`	Nvidia	1.000	1.000
`nbody`	AMD SDK	0.995	0.602
`reduce`	AMD SDK	1.000	1.000
`sgemm`	Parboil	1.000	1.000
`sobel`	AMD SDK	1.000	1.000
`spmv`	Parboil	0.900	0.733
`stencil`	Parboil	1.000	0.988

Table 4.1: Correlation between transaction number prediction by `SymEngine` for loads and stores and the actual number collected with the Nvidia profiler on *Fermi*.

of a single integer value to main memory: the position of the searched value. This implies that the number of transactions is always 1, irrespective of the coarsening factor or stride. For a set of constant values the variance is 0, this means that the correlation, according to formula 2.6 cannot be computed. For `binarySearch` we provide figure 4.20. The black line at the bottom of the plot represents the actual number of transactions (always 1), while the blue line at the top is our prediction. Since we cannot statically predict the outcome of the branch that controls the store, we assume that all the work-items in the `NDRange` space execute it. This leads to an overestimation of the actual value, nonetheless we correctly estimate the constant value of the counter. The `convolution` kernel computes at execution time the addresses to load from. Such computation cannot be modelled correctly by `SymEngine`. For `dwtHaar1D` the problem lies in the lack of support in LLVM SCEV infrastructure to deal with shift operators (<<, >>) use in the computation of the addresses to load from. We will see how this failure in the modelling leads to a miss-prediction for the stride parameter for `dwtHaar1D` in section 7.5.

4.7 Summary

This chapter has described the compiler passes that we implemented in this work. We presented the stages of the coarsening transformation and the tools that we used to implement it. Finally, we introduced `SymEngine`, a symbolic execution engine for the analysis of memory access.

Next chapter presents a technique to automatically analyse the effects of coarsening on performance through hardware performance counters.

Chapter 5

Performance Counters Analysis

This chapter analyses performance improvements and degradations given by thread-coarsening on 17 OpenCL benchmarks and four GPUs. We show that the same compiler configuration has different effects on run-time performance for different devices We introduce a technique based on regression trees to automatically analyse the hardware profiler counters collected during the evaluation of the compiler transformation. This gives us a way to quickly draw conclusions about what counters are affected the most by coarsening and their impact on performance.

Section 5.1 introduces the problem of performance portability and presents our approach to the solution. Section 5.2 motivates the work showing how the same compiler configurations affect performance differently on different devices. Section 5.3 describes the experimental set-up and the compiler options used in the chapter. Section 5.4 presents the performance results given by the exploration of the thread-coarsening space. Section 5.5 describes the technique based on regression trees we used to analyse hardware profiler counters. Section 5.6 presents the insights gained thanks to the construction of the regression trees on the four GPUs considered. Section 5.7 shows how the value of profiler counters change when changing the coarsening factor. Section 5.8 concludes the chapter.

5.1 Introduction

GPUs are the corner-stone of modern high performance computing. The success of these devices for general purpose computing has lead to the introduction into the market of many models by different vendors. In this context, OpenCL has been proposed as a common programming framework for heterogeneous devices and is now supported by all major hardware manufacturers.

OpenCL thus provides functional portability across a diverse range of platforms. Although *functionally* portable OpenCL is far from being *performance* portable. OpenCL programs designed and tuned for one system are unlikely to perform as well on another platform. This problem is particularly acute given the diversity of modern GPU architectures. Even hardware from the same vendor might show different characteristics from one generation to the next. AMD, for example, has changed their top-tier discrete GPUs from VLIW to SIMD cores [9] as we described in section 2.2.2

This heterogeneity makes performance analysis and compiler tuning for multiple architectures extremely challenging. This happens because the same compiler transformation can change performance differently on different GPU models. In this chapter we showcase this performance portability issues with respect of the coarsening transformation presented in chapter 4. We explain how coarsening changes the performance of 17 OpenCL applications on four GPUs by Nvidia and AMD. With the goal of analysing performance differences between program versions we run the multiple compiler configurations through Nvidia and AMD proprietary profilers. In this scenario, we then propose a methodology to assist the analysis of the profiler output. Our technique based on regression trees can select the most important counters that are affected by coarsening and their relationship with output performance. This methodology works effectively across hardware, helping the compiler developer to quickly understand the effects of coarsening on performance for a new GPU architecture.

In this chapter we make the following contributions:

- We provide an evaluation of the performance given by the coarsening compiler transformation on 17 benchmarks and four devices by Nvidia and AMD.
- We propose a technique based on regression trees to analyse the profiler counters collected on the four machines.
- We study the output of our performance analysis technique to understand the performance differences that we see across benchmarks and devices.

5.2 Motivation

In this section we present the performance variability of thread-coarsening when evaluated on a matrix transposition (`mt`) benchmark on four GPU architectures. The platforms are two AMD GPUs (*Cypress* and *Tahiti*) and two Nvidia GPUs (*Fermi* and *Kepler*). Figure 5.1 shows for each of these platforms the relative performance improvement achieved for different coarsening factors with a baseline of 1 (*i.e.*, no coarsening). As we can see, the impact of coarsening varies widely depending on the device. For the two AMD GPUs, we achieve the highest speedups around 3x with coarsening factor of 32. On the Nvidia *Fermi* GPU the optimal is achieved with coarsening factor 8 and leads to about 25% improvement. Nvidia *Kepler* GPU achieves a 20% improvement with a coarsening factor of 2; performance deteriorates with larger coarsening factors. This simple example illustrates the challenges encountered by programmers when

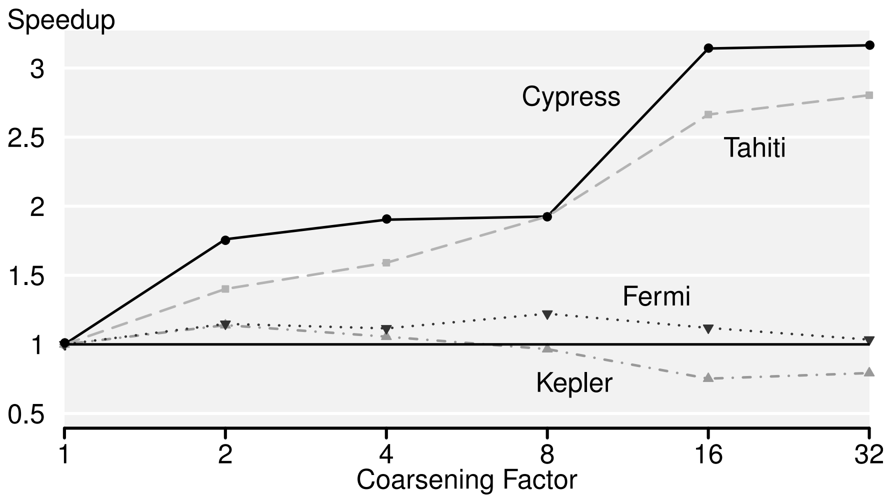


Figure 5.1: Speedup achieved by different coarsening factors on `matrixTransposition` on four platforms.

Name	**Model**	**GPU Driver**	**OpenCL Version**	**Linux Kernel**
Fermi	Nvidia GTX 480	304.54	1.1 CUDA 5.0.1	3.2.0
Kepler	Nvidia K20c	304.54	1.1 CUDA 5.0.1	3.1.10
Cypress	AMD HD5900	1084.4	1.2 AMD-APP 1084.4	3.1.10
Tahiti	AMD HD7970	1084.4	1.2 AMD-APP 1084.4	3.1.10

Table 5.1: OpenCL devices used for experiments.

porting OpenCL programs. Choosing a single coarsening factor for all the devices would lead to suboptimal performance even among devices from the same vendor. In section 5.6 we provide an explanation for the performance differences we record on our devices.

5.3 Experimental Set-up

This section presents the experimental set-up for this chapter.

5.3.1 Devices

For our experiments we used four devices from two vendors: Nvidia and AMD. Table 5.1 presents the hardware set-up for each device and lists the names that we use in the chapter to identify them. From Nvidia we used GTX480 and Tesla K20c. The first one is based on the Fermi architecture [100] with 15 streaming multiprocessors (SM) and 32 CUDA cores for each. The Tesla K20c instead, ships the Kepler architecture [101]. K20c has 13 SMs each with 192 CUDA cores. From AMD we used Radeon HD 5900 (*Cypress*) and HD7970(*Tahiti*). The first one has 20 SIMD cores each with 16 thread processors, a 5-way VLIW core. The Graph-

Parameter	**Possible values**
Coarsening factor	$\{1, 2, 4, 8, 16, 32\}$
Coarsening dimension	$\{0, 1\}$
Stride	$\{1, 2, 4, 8, 16, 32\}$
Local work-group size (for each dimension)	$\{$*Device min*$, \ldots, 32, 64, 128, \ldots,$ *Device max*$\}$

Table 5.2: Parameter space. For Nvidia GPUs the maximum size of a work-group is 1024 work-items per direction and 1024 in total. For AMD GPUs this value is 256.

Device	**1D**	**2D**
Fermi / *Kepler*	211	1650
Tahiti / *Cypress*	155	1082

Table 5.3: Configuration number of the four devices. The number of configurations depend on the dimensionality of the benchmark and the capabilities of the device.

ics Core Next architecture in *Tahiti* is a radical change for AMD. Each of the 32 computing cores contains 4 vector units (of 16 lanes each) operating in SIMD mode. This exploits the advantages of dynamic scheduling as opposed to the static scheduling needed by VLIW cores. Section 2.2 gives more details about the architecture of both GPUs.

5.3.2 Coarsening Parameter Space

Section 4.1 introduced the parameters that control the thread-coarsening compiler transformation. Here we list the range of values used for the evaluation of the transformation. The parameter space we considered is shown in Table 5.2. We have three parameters controlling the coarsening pass: factor, dimension and stride. The parameter in the last row, the work-group size, determines the total number of work-items, along each dimension, in a work-group. This maximum value is constrained by the device limitations on a per-kernel basis. We have exhaustively explored all combinations of our parameters leading to about 150-300 configurations for one dimensional kernels and 1000-2000 configuration for two dimensional kernels depending on the device limitation. These sum up to about $43'000$ configurations evaluated across all programs and devices.

Table 5.3 records the total number of configurations for one- and two-dimensional programs for our four GPUs.

	Program name	Source	`NDRange` size
1)	`binarySearch`	AMD SDK	64K
2)	`blackscholes`	Nvidia	256K
3)	`convolution`	AMD SDK	1M
4)	`dwtHaar1D`	AMD SDK	2M
5)	`fastWalsh`	AMD SDK	512K
6)	`floydWarshall`	AMD SDK	(6K x 6K)
7)	`mriQ`	Parboil	256K
8)	`mt`	AMD SDK	(4K x 4K)
9)	`mtLocal`	AMD SDK	(4K x 4K)
10)	`mvCoal`	Nvidia	256K
11)	`mvUncoal`	Nvidia	256K
12)	`nbody`	AMD SDK	64K
13)	`reduce`	AMD SDK	8M
14)	`sgemm`	Parboil	(512 x 512)
15)	`sobel`	AMD SDK	(512 x 512)
16)	`spmv`	Parboil	16K
17)	`stencil`	Parboil	(512 x 510 x 62)

Table 5.4: OpenCL applications with the corresponding `NDRange` sizes, expressed in number of work-items.

5.3.3 Benchmarks

We have used 17 benchmarks from various sources as shown in Table 6.3. The table also shows the global size that we used (expressed in total number of work-items). We made sure that for each benchmark we are able to change the work-group size at will, this involved changes in OpenCL host code to parametrize hard-coded constants. In the benchmarks selection process we avoided programs with features that our coarsening pass does not support, such as atomic instructions or texture memory. The baseline performance reported in the results section is that of the best work-group size as opposed to the default one chosen by the benchmark. This is necessary to give a fair comparison across platforms when the original benchmark is written specifically for one platform. All the performance numbers we report in section 5.4 are relative to the kernel-only execution time of the original unmodified code after being transformed by the `axtor` OpenCL-backend. This is done to factor out possible changes in the shape of the kernel introduced by `axtor`. Each experiment has been repeated 15 times for GPUs aggregating the results using the median.

5.4 Performance Results

The section presents the performance impact of the exploration of the coarsening transformation for the 17 benchmarks on the four architectures.

5.4.1 Speedups

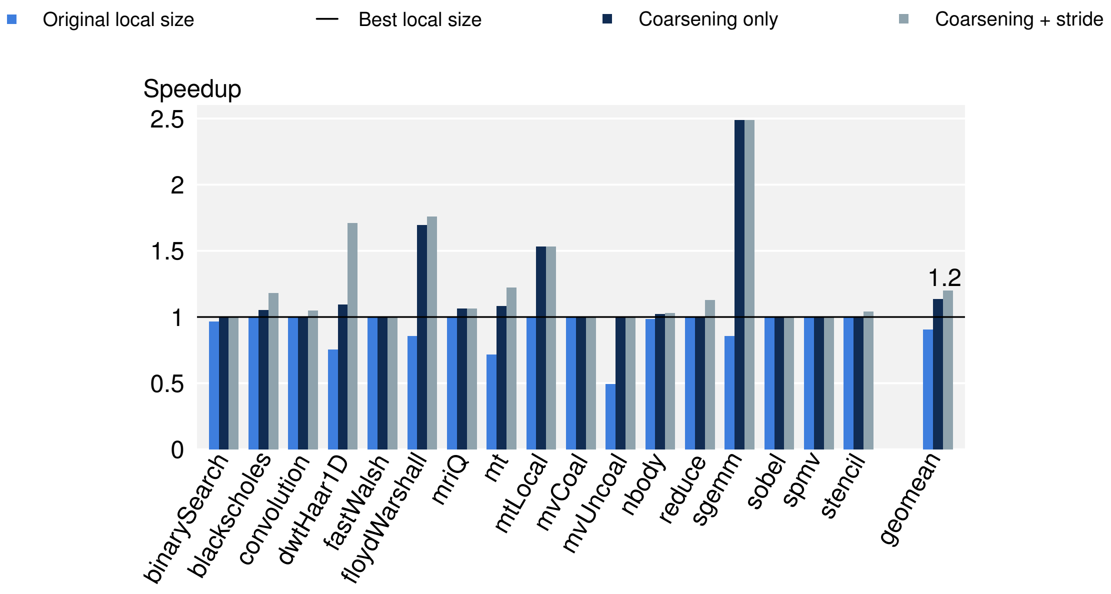
Figures 5.2 and 5.3 shows the maximum speedup obtained across all the transformation settings (see section 5.3.2) for each program and device. We fixed the baseline performance to the one of the original program (coarsening factor 1) with its best work-group size. This is a strict baseline and ensures a fair comparison across devices. In the figure the baseline is represented by the horizontal line at 1.0. The first bar reports the performance of the program run with the *original* work-group size. This is often lower than 1.0 because this parameter has to be tuned specifically for each platform as showed in [82].

The second bar shows the best speedup achievable when the application is *coarsened*. The third one shows the additional gain from applying a *stride* values different from 1 to remap work-items. Coarsening ensures significant speedups for most of the benchmarks across the five devices. Speedups range from a few percent up to 4.59 for `floydWarshall` on *Tahiti*.

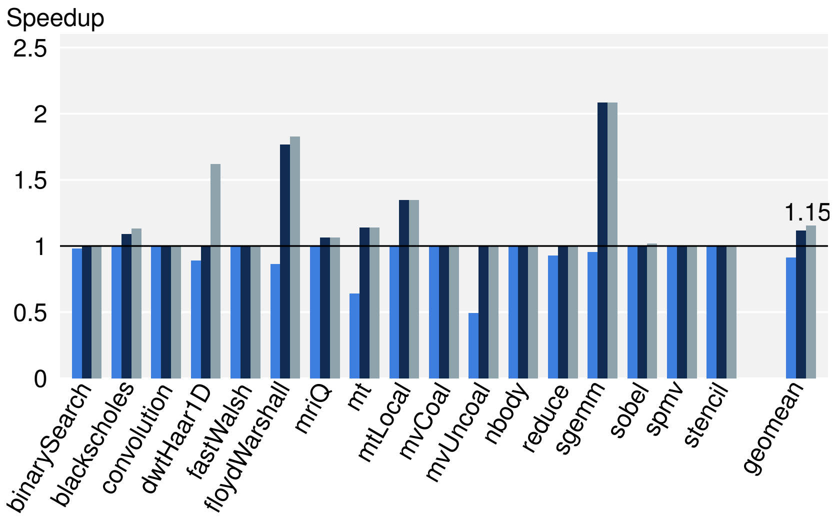
Platforms The two Nvidia GPUs (5.2a, 5.2b) show similar performance. The average improvement is 1.15x for *Kepler* and 1.2x for *Fermi*. The programs that improve the most after coarsening are `dwtHaar1D`, `floydWarshall`, `mt`, `mtLocal` and `sgemm`. Other kernels do not show significant improvements, in particular on Nvidia *Kepler*. This proves the need to thoroughly study performance so to select the compiler settings on a per-program basis. The two GPUs by AMD (5.3a, 5.3b) instead show more significant and diversified speedups averaging 1.5 for *Cypress* and 1.37 for AMD *Tahiti*. Differences in the architectures design are reflected in the speedup differences for benchmarks such as `dwtHaar1D`, `nbody` and `sgemm`.

Stride The stride optimisation is particularly effective for AMD GPUs giving an average improvement of 1.14 and 1.07 over the best coarsening-only configuration for *Cypress* and *Tahiti* respectively. On Nvidia devices the speedup given by the stride option is less consistent: on average 1.05 on *Fermi* and 1.03 *Kepler*. On Nvidia applications such as `dwtHaar1D`, `mt` and `reduce` show significant speedups up to 1.56. The stride values giving the best performance are the largest ones in our space (16 and 32). This is because on Nvidia and AMD 16 or 32 consecutive hardware threads need to access consecutive memory locations to achieve a coalesced memory transaction.

A detailed explanation of the overall behaviour of the benchmarks is given in section 5.6.

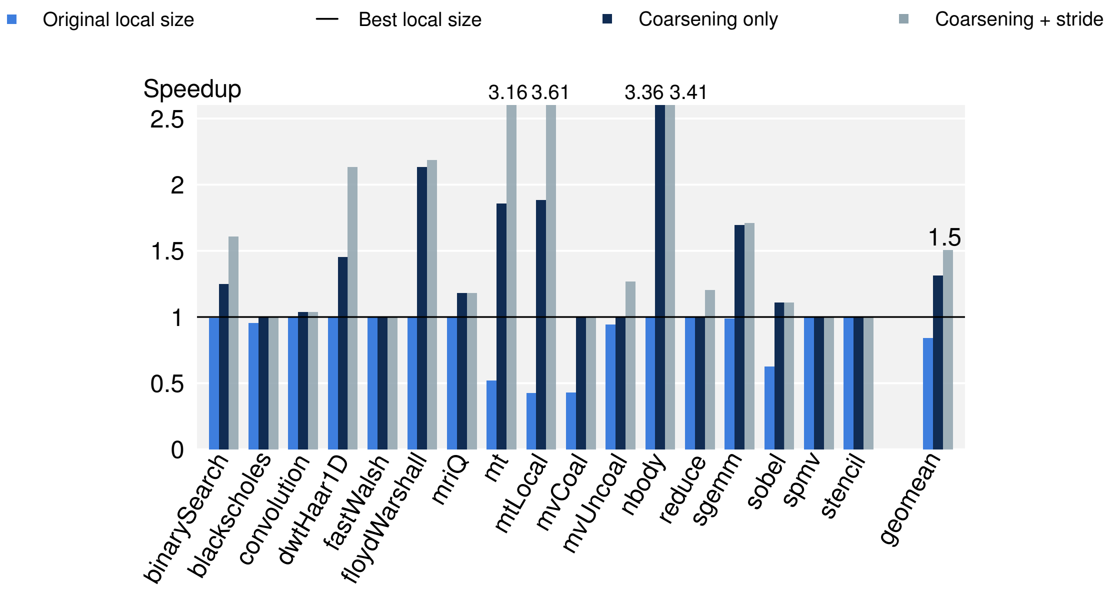


(a) *Fermi*

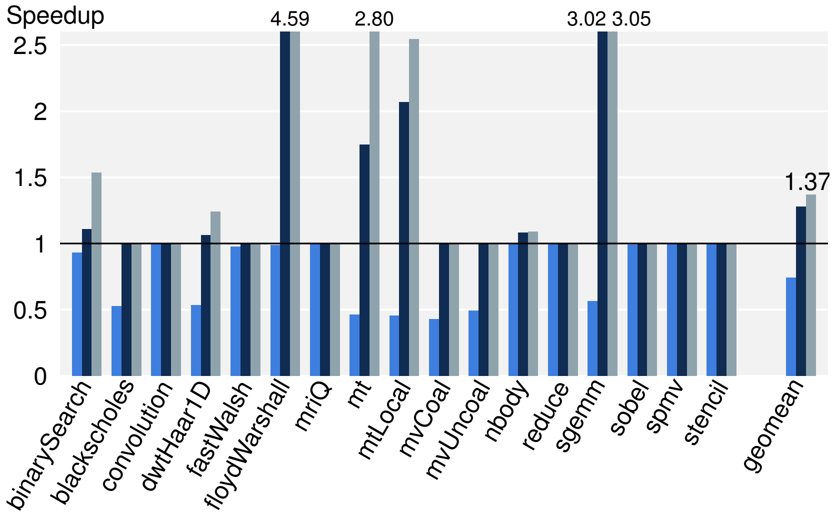


(b) *Kepler*

Figure 5.2: Speedup given by the best coarsening configuration for *Fermi* and *Kepler*. The baseline (the horizontal line at 1) is given by the execution time of the best work-group size for each program. The first bar represents the performance of the application run with the work-group size defined by the benchmark suite. The second one is the speedup achieved by the best coarsening configuration. The third bar represents the performance of the best coarsening plus stride configuration.



(a) *Cypress*



(b) *Tahiti*

Figure 5.3: Speedup given by the best coarsening configuration for *Cypress* and *Tahiti*. The baseline (the horizontal line at 1) is given by the execution time of the best work-group size for each program. The first bar represents the performance of the application run with the work-group size defined by the benchmark suite. The second one is the speedup achieved by the best coarsening configuration. The third bar represents the performance of the best coarsening plus stride configuration.

5.4.2 Effect of Coarsening Factor

We now consider the effect of the coarsening factor on execution time. Figures 5.4 and 5.5 show the maximum achievable speedup for each coarsening factor when setting the work-group size and stride to their best values, for all programs and architectures. The missing data points represent configurations that fail to run. This is usually due to the fact that the kernel function after the transformation would use a number of hardware registers larger than the maximum allowed by the device. This overall view shows variability of the best coarsening factor. The speedup curves of a single application look different across the four devices. For example `binarySearch` (first row) is not sensitive to any coarsening factor on Nvidia *Fermi* and *Kepler*, while it shows significant improvements on AMD *Cypress* and *Tahiti*. Furthermore, the optimal coarsening factor is 2 for *Cypress* and 8 for *Tahiti*. The technique we introduce in the next section automatically analyses this large amount of information providing useful indication on what counters are important to understand changes in execution time.

5.5 Performance Analysis Methodology

This section fist introduces the rationale behind the methodology that we chose to analyse performance. We then describe the technique in detail presenting its output and the insights that we gain.

State of the art techniques for performance evaluation consist in building analytical models by hand [53, 54, 116]. Such approaches fall short when dealing with an environment so heterogeneous as the one of GPU computing. Manually developing an analytical model for multiple architecture by different vendors is infeasible. This is also true because the performance of kernel functions depends on the GPU driver OpenCL compiler versions. A methodology that can quickly adapt to any device is therefore extremely useful in this context.

To analyse the effects of our transformations we propose using profiler counters coupled with regression trees [109]. Regression trees are a robust statistical analysis and offers two advantages:

- It is *platform-portable* since creating the performance model is fully automated and requires no human effort.
- The resulting model is *easy to visualize and understand*, and as such offers insights into the factors that affects performance.

5.5.1 Profiling

In order to understand the effects of thread-coarsening on performance we collect profiling information for each device. For all GPUs we used the proprietary OpenCL profiler from

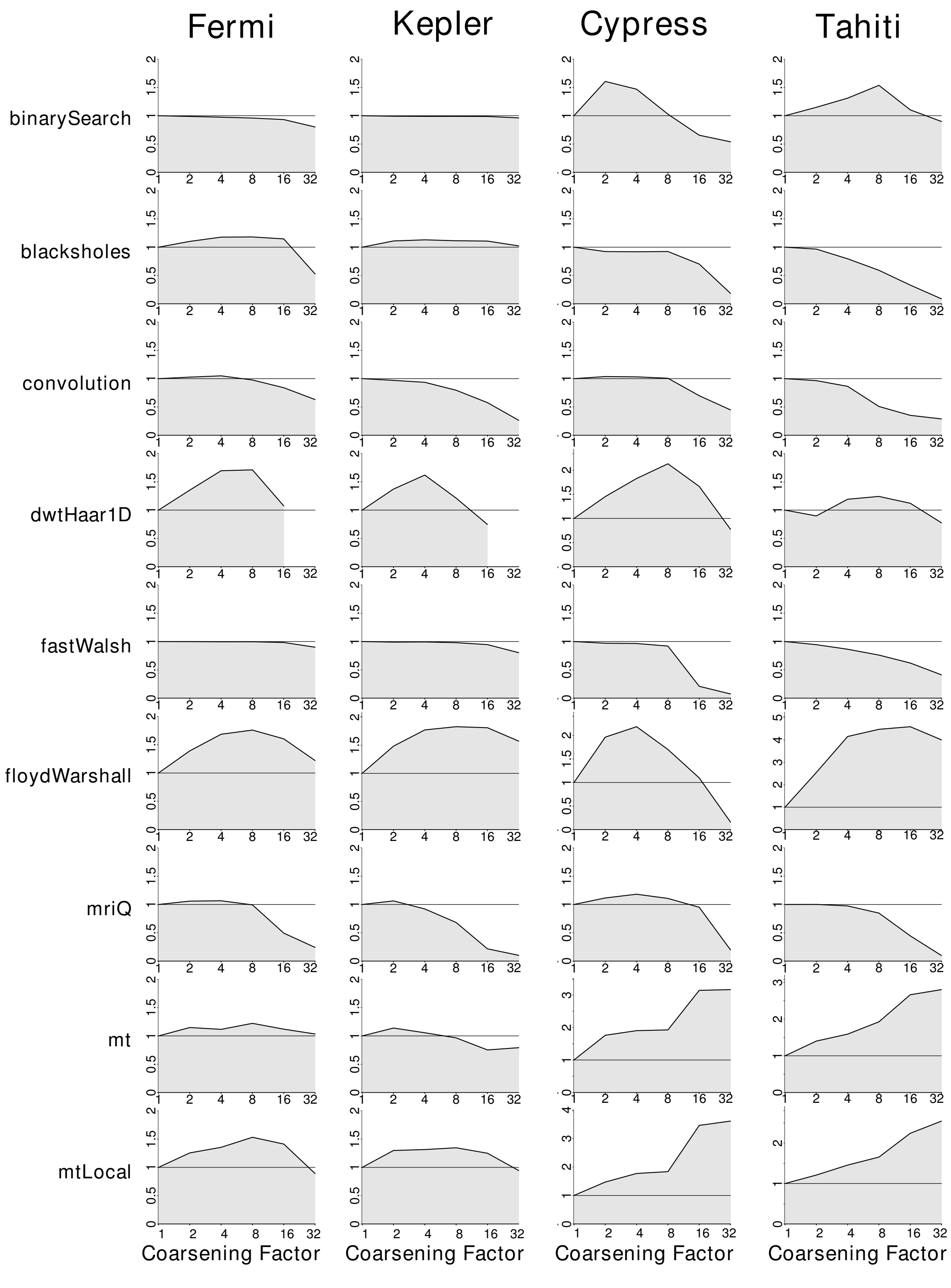


Figure 5.4: Maximum speedup for all the programs and devices as a function of the coarsening factor.

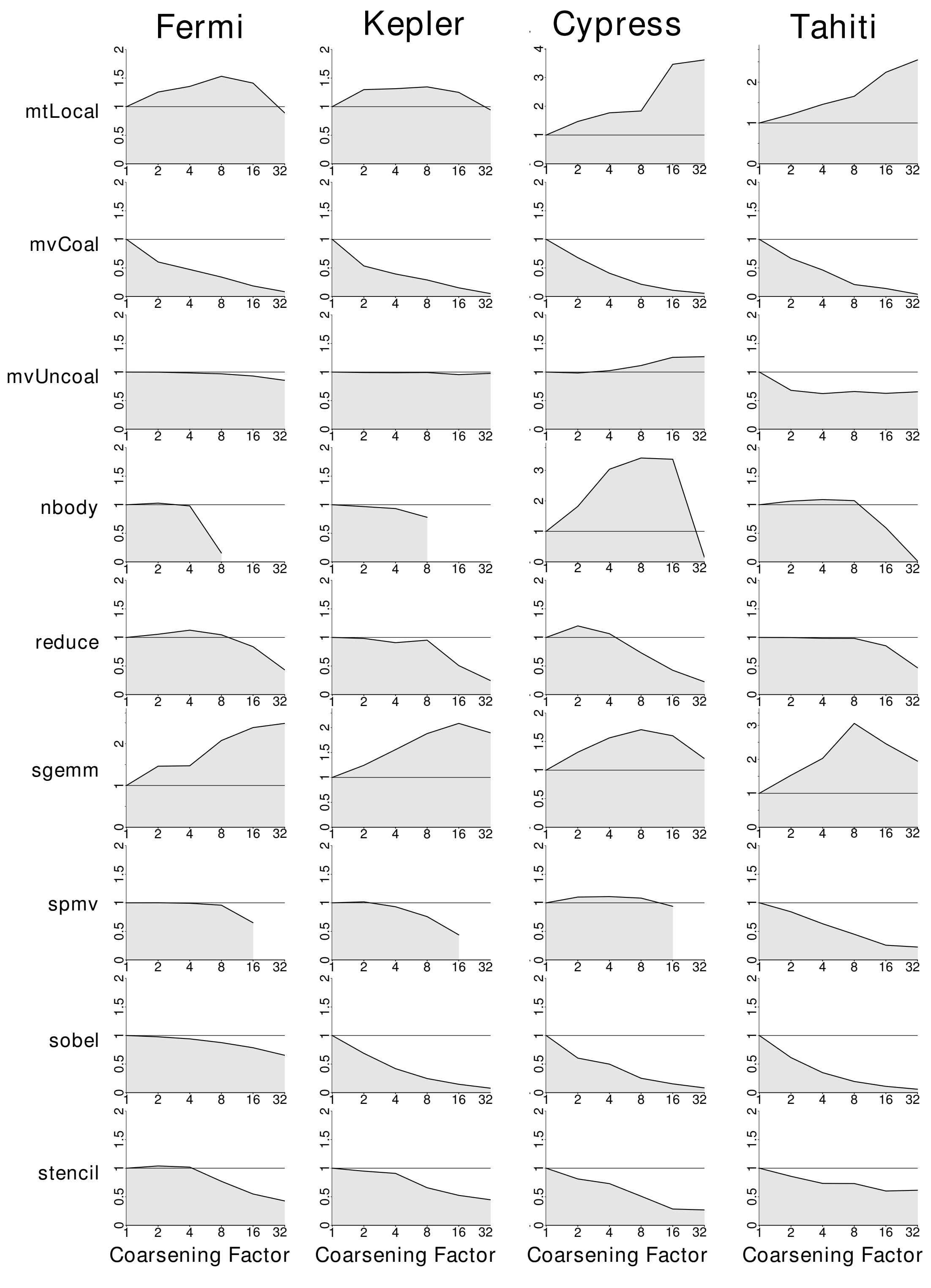


Figure 5.5: Maximum speedup for all the programs and devices as a function of the coarsening factor.

each vendor [10, 1]. Section 2.4 presents how to gather profiler counters and what type of information is available on the four devices. Appendix A contains the list of all the counters that we collected with the meaning of each of them.

Aggregation of Raw Counters The output of the Nvidia profiler is particularly detailed (refer to tables A.1, A.2, A.3 and A.4). This level of resolution makes it hard to extract useful information from the profiler output. Data has be aggregated so to reduce the amount of information to process. We used the following formulas to aggregate counters so to obtain a more significant representation of the data. Here we focus on counters related to the utilization of the memory hierarchy.

- L1 Cache.
  - The Nvidia profiler distinguishes cache accesses that are directed to global memory from the ones directed to local memory (a partition of global memory where registers are spilled). We ignore this distinction and we aggregate the two types of counters in a single value:
    - $\text{l1Hit} = \text{l1\_global\_load\_hit} + \text{l1\_local\_load\_hit}$
    - $\text{l1Miss} = \text{l1\_global\_load\_miss} + \text{l1\_local\_load\_miss}$
  - $\text{l1HitRate} = 100 \cdot \text{l1Hit} / (\text{l1Hit} + \text{l1Miss})$
  - The total number of memory requests is defined as the number of hits in L1 plus the number of misses.
    $\text{globalLoadTransactions} = \text{l1\_global\_load\_miss} + \text{l1\_global\_load\_hit}$
- L2 Cache
  - L2 hits and misses are distinguished based on the memory partition they are assigned to. We ignore this distinction by adding the counters across all the partitions.
    - $$\begin{aligned}\text{l2ReadHit} = {} & \text{l2\_subp0\_read\_l1\_hit\_sectors} + \\ & \text{l2\_subp1\_read\_l1\_hit\_sectors} + \\ & \text{l2\_subp2\_read\_l1\_hit\_sectors} + \\ & \text{l2\_subp3\_read\_l1\_hit\_sectors}\end{aligned}$$
    - $$\begin{aligned}\text{l2ReadMiss} = {} & \text{l2\_subp0\_read\_sector\_misses} + \\ & \text{l2\_subp1\_read\_sector\_misses} + \\ & \text{l2\_subp2\_read\_sector\_misses} + \\ & \text{l2\_subp3\_read\_sector\_misses}\end{aligned}$$

    - $*$ l2WriteMiss = l2_subp0_write_sector_misses + l2_subp1_write_sector_misses + l2_subp2_write_sector_misses + l2_subp3_write_sector_misses
    - $*$ l2ReadQueries = l2_subp0_read_l1_sector_queries + l2_subp1_read_l1_sector_queries + l2_subp2_read_l1_sector_queries + l2_subp3_read_l1_sector_queries
    - $*$ l2WriteQueries = l2_subp0_write_l1_sector_queries + l2_subp1_write_l1_sector_queries + l2_subp2_write_l1_sector_queries + l2_subp3_write_l1_sector_queries
    - $*$ l2WriteHit = (l2WriteQueries − l2WriteMiss)
- l2ReadHitRate = 100 · l2ReadHit / (l2ReadMiss + l2ReadHit)
- l2WriteHitRate = 100 · l2WriteHit / (l2WriteMiss + l2WrriteHit)

- dramWrites = fb_subp0_write_sectors + fb_subp1_write_sectors
- dramReads = fb_subp0_read_sectors + fb_subp1_read_sectors

Given the large amount of counters, it is hard to establish their relationship with performance. Our methodology takes in input all this information and produces a synthetic result that can be easily read.

5.5.2 Tree Construction

The overall goal of our technique is to correlate changes in the values of profiler counters to changes in execution time performance (*i.e.*, speedup). Let $s_{p,cf}$ be the speedup achieved on program p with coarsening factor $cf \in \{1,2,4,8,16,32\}$ and the best possible stride and work-group size. The speedup are expressed for each program with respect to the uncoarsened version, which means $s_{p,1} = 1 \ \forall p$. Let $\vec{f}_{p,cf}$ be the vector of profiler counters extracted from the same run.

We want to find a mapping between profiler counters $\vec{f}_{p,cf}$ and speedup $s_{p,cf}$ to understand what factors affect performance and how. To achieve this, we record the execution time and profiler counters for all coarsening factors on all our programs for each platform.

The regression trees describe the mapping $\vec{f}_{p,cf} \rightarrow s_{p,cf}$ acting as a broad model for the speedups in figures 5.2 and 5.3. Once constructed, the trees can then be used to explain the observed run-time performance behaviour. Tree construction involves recursively partitioning the data into sets with similar speedup $s_{p,cf}$ based on the profiler counters $\vec{f}_{p,cf}$. To increase

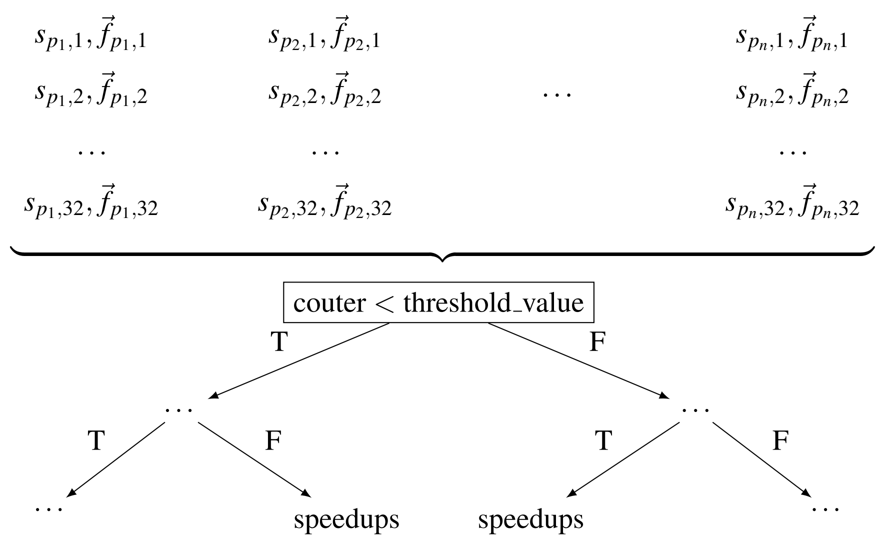

$s_{p_1,1}, \vec{f}_{p_1,1}$
$s_{p_1,2}, \vec{f}_{p_1,2}$
$s_{p_1,32}, \vec{f}_{p_1,32}$
$s_{p_2,1}, \vec{f}_{p_2,1}$
$s_{p_2,2}, \vec{f}_{p_2,2}$
$s_{p_2,32}, \vec{f}_{p_2,32}$
$s_{p_n,1}, \vec{f}_{p_n,1}$
$s_{p_n,2}, \vec{f}_{p_n,2}$
$s_{p_n,32}, \vec{f}_{p_n,32}$
couter < threshold_value
T
F
speedups
speedups

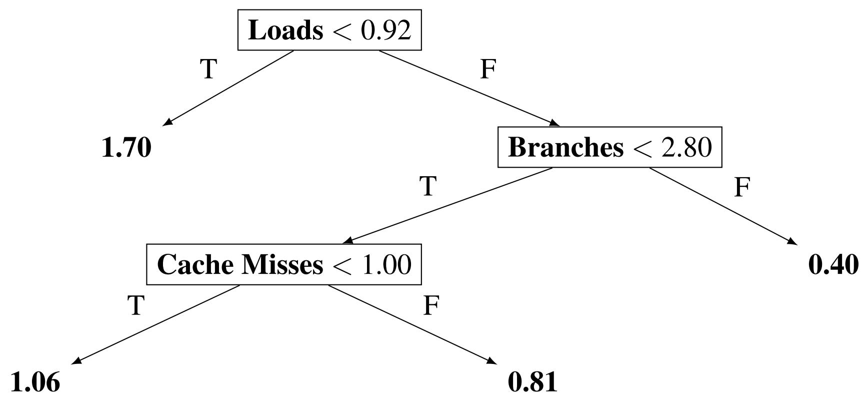
Figure 5.6: Input data used to construct a regression tree. $s_{p_n,1}$ is the speedup given by program n compiled with coarsening factor 1. $\vec{f}_{p_n,1}$ is the set of program counters collected from program n compiled with factor 1.

the accuracy of the model we used in the construction of the trees the top best 5 performing configurations for each coarsening factor. This process is outlined in figure 5.6. We used the R package *tree* (version 1.0-33) using deviance as impurity metric. Deviance is a measure of the variance of the values in the leaves of a tree. The algorithm of the tree construction tries to minimise it, having leaves as uniform as possible. The trees are then pruned using cross-validation ensuring that the expected speedup has a correlation of at least 0.8 with the curves showed in figures 5.4 and 5.5 and error about 10% for all the GPUs. The trees for each platform are shown in figure 5.7. Each node represents the relative increase or decrease of a profiler counter over its value for the baseline configuration. The right child of a node satisfies the condition of that node while the left one does not. Based on the inherent property of a regression tree, the conditions near the top of the tree are the ones that are best at distinguishing bad performance from good performance. These regression trees provide a concise and visual model to explain the performance of all the applications on a device.

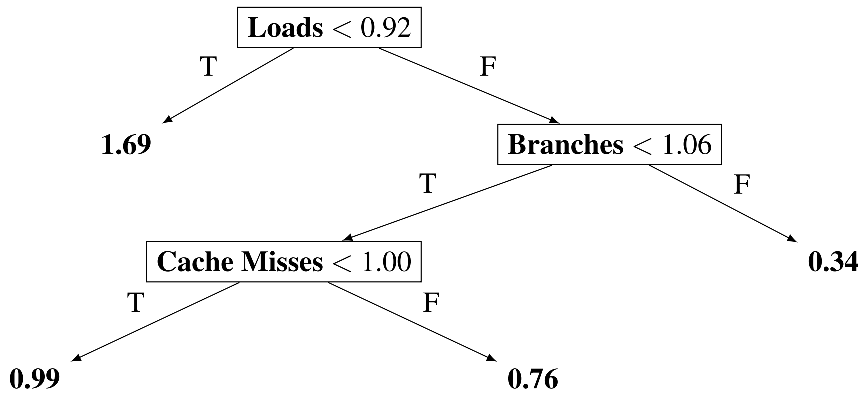
5.6 Per-device analysis

5.6.1 Nvidia *Fermi* GPU

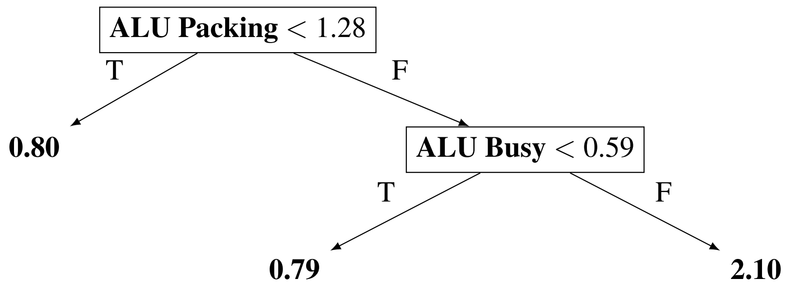
As can be seen in figure 5.7a, the number of memory loads is the most important counter to consider. A reduction in the number of loads leads to a significant performance improvement: 1.70x on average. This is the case for 2 applications: `sgemm` and `floydWarshall`. Consecutive



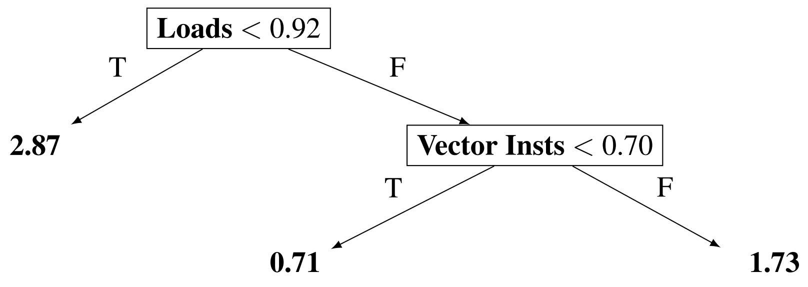
(a) Output regression tree for *Fermi*.



(b) Output regression tree for *Kepler*.

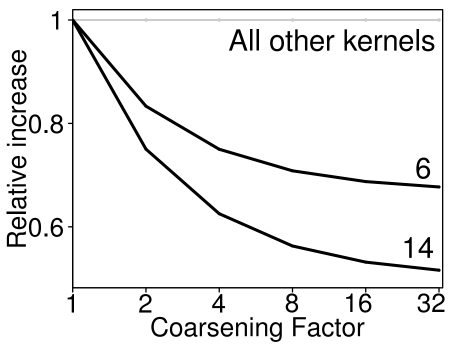


(c) Output regression tree for *Cypress*.

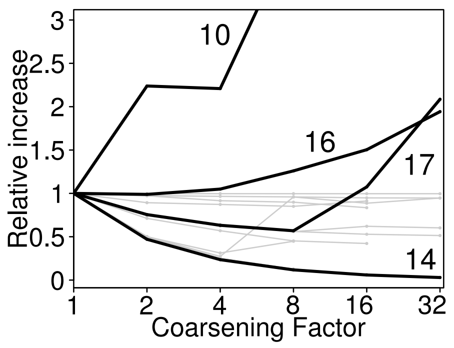


(d) Output regression tree for *Tahiti*.

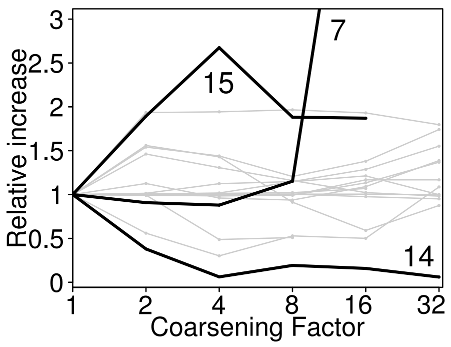
Figure 5.7: Regression trees for the four GPU architectures. The labels of the nodes identify the profiler counter used to split the data. In each node the counter is compared against its relative increase over its value in the default configuration. The numbers in the laves represent the expected speedup of configurations in the specified partition.



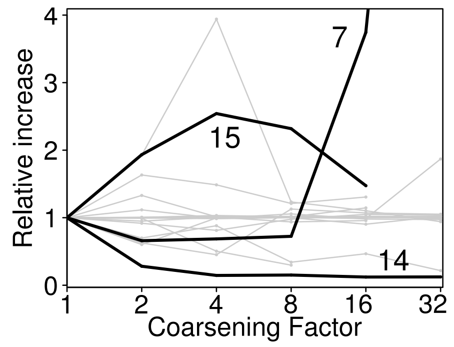
(a) Loads (all devices)



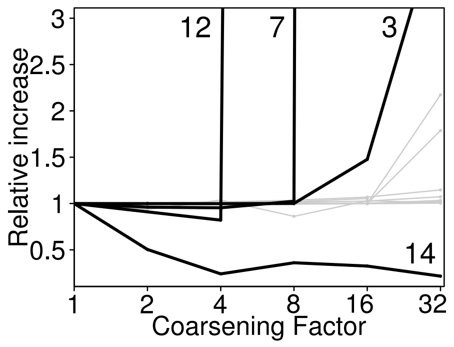
(b) Branches (all devices)



(c) Cache Misses (*Fermi*)

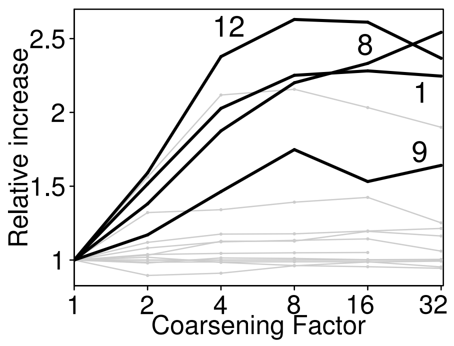


(d) Cache Misses (*Kepler*)

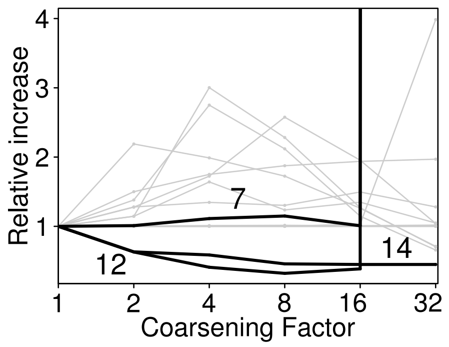


(e) L2 Cache Misses (*Fermi*)

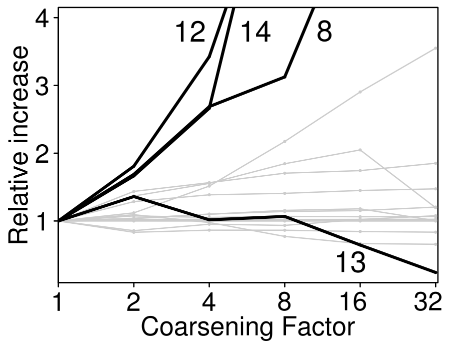
Figure 5.8: Relative value of hardware profiler counters as a function of the coarsening factor for all devices (a, b) and for Nvidia devices (c, d, e). All values are normalized with respect to the value of the counter for coarsening factor 1. Each line represents a kernel. The thick black lines correspond to the benchmarks referenced in the text, the numbers beside them identify the kernels according to table 6.3.



(a) ALU Packing (*Cypress*)



(b) Cache misses (*Tahiti*)



(c) Vector Instructions (*Tahiti*)

Figure 5.9: Relative value of hardware profiler counters as a function of the coarsening factor for AMD devices. All values are normalized with respect to the value of the counter for coarsening factor 1. Each line represents a kernel. The thick black lines correspond to the benchmarks referenced in the text, the numbers beside them identify the kernels according to table 6.3.

work-items of these two applications were originally loading the same values from memory. When coarsening, *redundant load elimination* successfully remove the unnecessary LLVM-IR instruction, enabling the reusage of the loaded value in registers. This optimisation effectively reduces the total number of loads executed. This behaviour is exemplified for matrix multiplication (`sgemm`) in section 4.3.4.

Going down one level, the second most important counter is the number of branches. Control flow is critical for GPU performance [135, 32]. If the output of a branch diverges within a warp all its work-items will execute both paths of the branch thus leading to a lower core occupancy. Configurations in the lower right corner, where the number of branches has increased and the number of loads has not been reduced, show significant slowdowns. In this region fall `spmv`, `mvCoal` and `stencil` with slowdowns on average 0.4x. The bodies of these three kernels are enclosed in divergent regions that check that work-items do not overflow buffer boundaries. These regions are entirely replicated by the coarsening pass (as explained in section 4.3.1). This leads to an increase in the relative number of branches and hence lower occupation. Finally, the last counter selected concerns the cache utilization (in this case it is *l2CacheMisses*). As we show in figures 5.8 and 5.9 coarsening changes how the kernel utilizes the GPU memory hierarchy. If the working set of the benchmark does not increase we experience a moderate improvement: 1.06x. On the other hand trashing the cache leads to major performance drops. More details and examples on this in section 5.7.3.

5.6.2 Nvidia *Kepler* GPU

As described in section 2.2.1 the *Kepler* architecture is an evolution of *Fermi* maintaining the same design principles. This explains the similarities between the two trees in figures 5.7a and 5.7b. The number of loads and branches are still the most important counters. In this case though the tree shows higher sensitivity to branches. This is due to the change in the instruction scheduling policy from a dynamic one in *Fermi* to a static one in *Kepler* [101]. While on *Fermi* a relative increase of 2.8 of the number of branches leads to a significant slowdown (0.4x of the original performance) this threshold is lowered to 1.06 for *Kepler*. Overall the expected speedups given by the tree reflect the lower performance improvement achievable on *Kepler* with respect to *Fermi*. The reason for this difference is the higher number of computing cores mounted on a *Kepler* streaming multiprocessor (192 vs 32) which ensures better performance to applications running a higher number of threads even if they execute redundant operations.

5.6.3 AMD *Tahiti* GPU

As seen on the Nvidia devices a reduction in the number of loads leads to a significant improvement on *Tahiti* too: 2.87x on average. Descending on the right branch of the root the second most important counter is related to the utilization of the vector unit mentioned in section 2.2.2.

For benchmarks `mt`, `mtLocal` and `nbody` the increase in data-dependence-free instructions in a single thread allows a better utilization of the vector lanes of the GPU cores. On the other hand kernels like `mvCoal`, `mvUncoal`, `spmv`, `stencil` show a reduction in the vector utilization due to the divergent regions they contain. These limit the scope of vectorization reducing the utilization of the SIMD lanes.

5.6.4 AMD *Cypress* GPU

The cores of the *Cypress* GPU are based on a VLIW architecture. The *ALUPacking* counter measures the utilization of the 5-way VLIW cores and is the most important counter. The increase in the core efficiency is due to a better utilization of the functional units thanks to the better scheduling now performed on the larger number of instructions available after coarsening. This beneficial effect of coarsening is similar to effect showed by unrolling and software pipelining applied to loops on classical CPU architectures [39]. Notice that the core utilization is more relevant for performance than a reduction in the number of loads which is not used to discriminate benchmarks on this device. On *Cypress* the importance of the efficient utilization of the computing cores is confirmed by the choice of *ALUBusy* as second most important counter. Sufficient utilization of the ALU leads to a 2.10x speedup.

5.7 Performance Counters

The previous analysis identifies what are the important hardware counters related to performance when applying coarsening. In this section, we now study how coarsening affects the counters selected by the trees and how these affect performance, while others such as cache misses are architecture specific. In figures 5.8 and 5.9 we plot the relative value of a profiler counter as a function of the coarsening factor. Counters are normalized with respect to the value of the counter of the baseline configuration (no coarsening): this value is always 1.

5.7.1 Memory Loads

We first look at the impact of coarsening on the number of memory loads in the program. This is platform-independent counter is important for both Nvidia devices and *Tahiti*. Considering figure 5.8a we see that for `floydWarshall` (6) and `sgemm` (14) the number of loads is decreases as the coarsening factor increases. This is due to the reuse in registers of values originally loaded by different work-items, thus leading to the elimination of the redundant loads. The possibility to reduce the number of loads is extremely beneficial in GPUs given that these devices are usually limited by memory latency.

5.7.2 Branches

Figure 5.8b shows the effect of coarsening on the total number of branches. Here we report a single plot representative for all GPU devices. Thread-coarsening generally reduces the number of executed branches. This is due the fact that our coarsening pass, relying on divergence analysis, preserves non-divergent branches replicating only the instructions they control (see section 4.3.1). This leads to an overall reduction of the number of branches with respect to the total number of instructions, consider for example `sgemm` (14). Important exceptions to the trend are `mvCoal` (10), `spmv` (16) and `stencil` (17) which all show divergent regions as explained in section 5.6. All these benchmarks show large performance degradations when coarsening.

5.7.3 Cache Utilization

Figures 5.8c-5.9b describe the cache utilization for the two Nvidia GPUs and *Tahiti*. As we can see from the general trend of the grey lines coarsening tends to put the cache under stress increasing the miss rate. This is due to an increase in the working size of each thread. Considering for example `sobel` (15) we see that an increase in the cache miss rate for both *Fermi* and *Kepler*: this justifies the performance drop we record in figures 5.4 and 5.5. Extreme cases are `mriQ` (7) and `nbody` (12) for which we see that coarsening by high factors (4 and 8 on *Fermi*, 16 on *Tahiti*) leads to a large increase in the number of misses which corresponds to significant performance penalties. The reason for this effect lies in the structure of the code: both these kernels have a non-divergent loop in their body. The coarsening pass replicates the instructions within the loop body thus increasing the loop working-set. After a certain threshold the advantage given in the reduction of the number of branches is outweighed by the penalty due of cache misses. An exception to the general trend is `sgemm` (14) for which the possibility to save load instructions also has a beneficial effect on the cache utilization.

5.7.4 Vector Instructions

Figure 5.9c shows the utilization of vector instructions for AMD *Tahiti*. This is the only architecture in which vector utilization is a significant factor for performance. Figure 5.9c shows the utilization of vector instructions on *Tahiti* when coarsening. For applications like `sgemm` (14) and `mt` (8) coarsening has a beneficial effect on the vector lanes utilization leading to significant speedups.

5.7.5 ALU Packing

The degree of utilization of the 5-way VLIW cores for the *Cypress* architecture is showed in figure 5.9a. Applications such as `nbody` (12), `mt` (8), `binarySearch` (1) and `mtLocal` (9) show an increase in the utilization of the cores. This means that the AMD compiler can schedule more efficiently the larger amount of data-dependency-free instructions available after coarsening. As we can see from figures 5.4 and 5.5 the listed applications greatly benefit from coarsening confirming the importance of improving the utilization of the arithmetic cores when tuning programs for *Cypress*.

5.7.6 Instruction Cache Behaviour

Notice that the counters made available by Nvidia and AMD do not include information about instruction cache. Therefore we cannot make statements about the effects that this component has on performance. Nevertheless, we can assume that coarsening has an overall negative impact on the instruction cache, since we are increasing the number of instructions executed per thread.

Summarising the results of our analysis we can conclude that coarsening works the best by reducing redundant computation. On GPUs the possibility to save loads and branches has proven to be effective for performance. Attention must be placed in the kernel cache utilization since coarsening usually increase the working set size. Architectures from AMD enjoy a better utilization of the vector units after coarsening thanks to the larger number of data-dependency free instructions now available to the compiler for scheduling.

5.8 Summary

In this chapter we proposed a methodology to rigorously analyse performance of multiple GPU models. We applied it to the study of the thread-coarsening transformation. We can significantly improve performance by exploring the coarsening parameter space (coarsening factor, stride and work-group size). We showed that the coarsening space is highly target dependent. The regression trees we introduced in this chapter are capable of relating changes of profiler counters to execution time variations. This is a novel platform-independent analysis technique that can effectively identify the root causes of performance variation across the coarsening space.

In the next chapter we study the effects of thread-coarsening across the problem input size space and we propose a way to speedup iterative compilation.

Chapter 6

Fast Tuning of the Coarsening Transformation

This chapter presents a study on the effect of thread-coarsening on performance across multiple input sizes. We want to evaluate how the best coarsening configuration changes across problem sizes. Analysing how the utilization of the GPU varies across workload sizes we identify the saturation point as the smallest problem size for which the device is fully utilized. We show that the saturation point is the smallest input size for which the optimal coarsening factor gives high speedups for a much larger input sizes as well. We exploit this property to improve the performance of iterative compilation.

Section 6.1 introduces the chapter describing the goal of studying the how coarsening affects performance across problem input sizes. Section 6.2 motivates the work showing how the utilization of the GPU and the relative performance of transformation configuration change across input sizes. Section 6.3 presents the experimental set-up and the compiler options space used in the chapter. Section 6.4 shows experimental data presenting the computational throughput for all the benchmarks in our set. Section 6.5 presents the technique that we use to exploit hardware saturation so to speed-up iterative compilation. Section 6.6 shows the improvements that we obtain by applying our technique. Section 6.7 concludes the chapter.

6.1 Introduction

This chapter analyses the performance of the coarsening optimisation space across multiple problem input sizes. We do this with the goal of reducing the search-time for iterative compilation. In our context iterative compilation aims at finding a configuration for the coarsening transformation that improves performance over the baseline and approaches the maximum attainable. Our search space is made of the coarsening factor, stride, coarsening direction and work-group size.

To reduce search time, application programmers often evaluate many different compiler transformations on problem sizes smaller than their target. The aim is to have quick feedback on what are the most effective optimisation options. This methodology assumes that compiler configurations that work well on a small problem size will also work well on the target size, which is usually much larger. The testing problem size must be chosen with care: too small and it might not match the behaviour of the bigger target size, too large and it makes tuning too expensive. Finding the right problem size on which to perform tuning is a crucial issue.

To tackle this problem we introduce the concept of *saturation* of hardware resources for graphics processors. Saturation is achieved when an application is run with a problem input size (*i.e.*, number of work-items) sufficiently large to fully utilize hardware resources. Programs running in such a region of the problem size space usually scale according to the complexity of the problem and show performance stability across program optimisations. This means that code transformations effective for a small problem size in the saturation zone are usually effective for extremely large problem sizes too. We develop a search strategy that exploits the Minimum Saturation Point (MSP) to reduce the total tuning time necessary to find good optimisation settings.

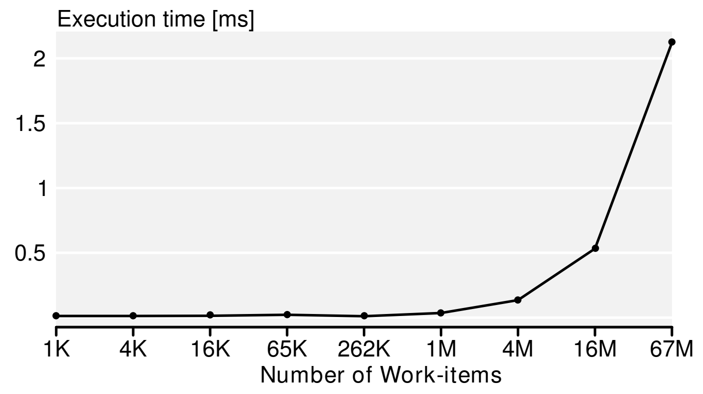
In this work we describe the property of saturation in terms of the *throughput* metric: the amount of work performed by a parallel program per unit of time, showing how this reaches stability when the hardware saturates.

This chapter makes the following contributions:

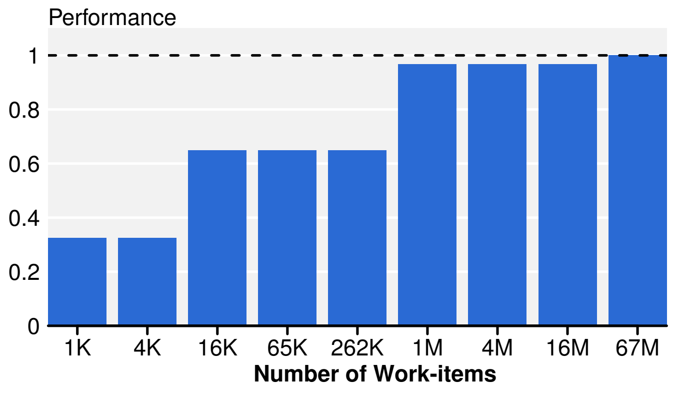
- We show the existence of saturation for 16 OpenCL benchmarks on three GPU architectures: *Fermi* and *Kepler* by Nvidia and *Tahiti* by AMD.
- We show that the best configuration for the coarsening configuration remains stable for problem sizes larger than the saturation point.
- We show that the hardware saturation can be successfully exploited to speedup iterative tuning of coarsening transformation leading to a reduction of search space of an order of magnitude.

6.2 Motivation

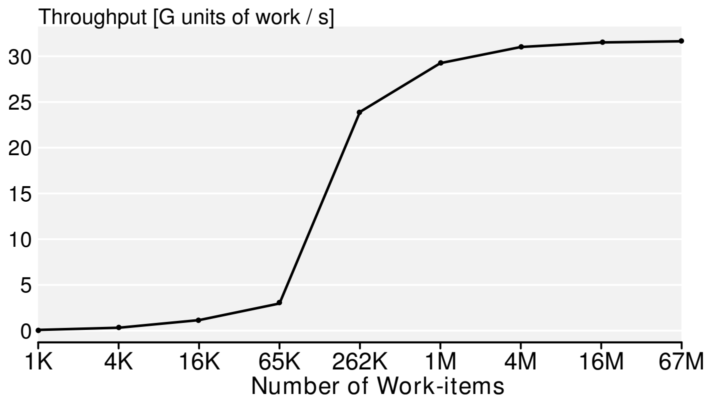
This section introduces the problem by characterizing the coarsening optimization space across input sizes. As an example we use the `floydWarshall` program running on the AMD *Tahiti* GPU and the optimisation space of the thread-coarsening transformation. Figure 6.1 shows the behaviour of our benchmark as a function of its input size. For OpenCL workloads there is direct mapping between the input size and the overall number of work-items instantiated. We measure the problem size using the total number of work-items.



(a) Kernel execution time (in milliseconds) as a function of the input size (*i.e.*, total number of work-items).



(b) Normalized performance attainable performing the search on the input size at the bottom of the bar and evaluating the best performing configuration found on the largest one (67M). For example the best performing configuration for input size 1K when evaluated for 67M work-items achieves only 30% of the maximum attainable by tuning coarsening directly on input size 67M. The barplot reaches 1 for 67M by construction.



(c) Throughput as a function of the problem input size.

Figure 6.1: Execution time (a), relative performance (b) and throughput (c) of the program `floydWarshall` running on AMD *Tahiti* as a function of the input size (expressed as total number of work-items). Notice the log-scale of the x-axes. The throughput is expressed in billions units of work over seconds, see section 6.4.

Figure 6.1a shows the execution time of the application as a function of the input size. The largest data size, with 67 million work-items is the input size we want to optimise for. Unsurprisingly this leads to the longest execution time. If we were to tune this application by evaluating multiple versions, the time required to search the optimisation space would be directly proportional to the execution time. It would be preferable to tune the application on the smallest possible input size.

However, the performance of the best optimisation settings for a given problem size does not transfer across input sizes as shown in figure 6.1b. In the barplot we show the relative performance achieved by the best performing configuration found on one input size when evaluated on the target one (67 million work-items). Performance is computed as the ratio between the speedup achieved by the search and the maximum attainable. The best optimisation settings for the smallest input size achieves only a third of the performance of the best one corresponding to the largest input size. This means, that if the application were tuned on smallest input size, it would run three times slower than the best achievable on the large input size. A good trade-off is an input size of 1 million work-items achieving performance close to the largest input size. This represents a savings of 67× in search time (67 million / 1 million). The question is how we can find the optimal input size without having to perform the search for all input sizes in the first place.

One way to solve this problem is to look at the throughput metric, which is the number of operations executed per second (the formal definition is given in section 6.4). Figure 6.1c shows the throughput of the application as a function of the problem size. As we can see, the throughput starts low and increases with the problem size. It flattens out at around 1 million work-items which corresponds to the saturation of the hardware resources. We call this point the *minimum saturation point* (MSP), because passed this point throughput remains constant. Coincidentally, the MSP is smaller than the target input size (67 million work-items) and leads to performance on par with the best achievable when searching on the largest input size.

The remainder of the chapter characterizes hardware saturation using the throughput metric and shows that it can be used to reduce tuning time.

6.3 Experimental Set-up

This section presents the experimental set-up for this chapter.

6.3.1 Devices

The devices used for the experiments are the same as chapter 5. The only differences are driver versions and operating system version. Refer to section 5.3.1 for a description.

Name	**Model**	**GPU Driver**	**OpenCL version**	**Linux kernel**
Fermi	Nvidia GTX 480	304.54	1.1 CUDA 5.0.1	3.2.0
Kepler	Nvidia K20c	331.20	1.1 CUDA 5.0.1	3.7.10
Tahiti	AMD HD7970	1084.4	1.2 AMD-APP 1084.4	3.1.10

Table 6.1: OpenCL devices used for experiments.

Parameter	**Possible values**
Coarsening factor	$\{1, 2, 4, 8, 16, 32\}$
Coarsening dimension	$\{0, 1\}$
Stride	$\{1, 2, 4, 8, 16, 32\}$
Work-group size (for each dimension)	$\{$*Device Min*$, \ldots, 32, 64, 128, \ldots,$ *Device Max*$\}$

Table 6.2: Parameter space. For Nvidia GPUs the maximum size of a work-group is 1024 work-items per direction and 1024 in total. For AMD GPUs this number is 256.

6.3.2 Coarsening Parameter Space

We tuned the transformation by considering all combination of our parameters leading to about 150-300 configurations for one dimensional kernels and 1000-2000 configuration for two dimensional kernels depending on the device limitation and the problem size. These add up to about $160'000$ configurations evaluated across all input sizes, programs and devices. This large evaluation of the coarsening transformation is made possible by the our portable compiler tool-chain described in chapter 4.

6.3.3 Benchmarks

In this chapter, we use 16 OpenCL benchmarks from various sources, as shown in Table 6.3. The only missing benchmark from the programs in chapter 5 is `spmv`, for which the input size cannot be changed automatically. In the case of programs from the *Parboil* benchmark suite we used the *opencl_base* version. The table reports the range of sizes for the input problem, these are reported in terms of the total number of launched work-items. For the rest of the chapter we consider the largest problem size as the target one, *i.e.*, the one that we want to optimise the program for. We used three platforms described in Table 6.1. In all our experiments we only measure the kernel execution time. To reduce measurement noise each experiment has been repeated 50 times aggregating the results using the median.

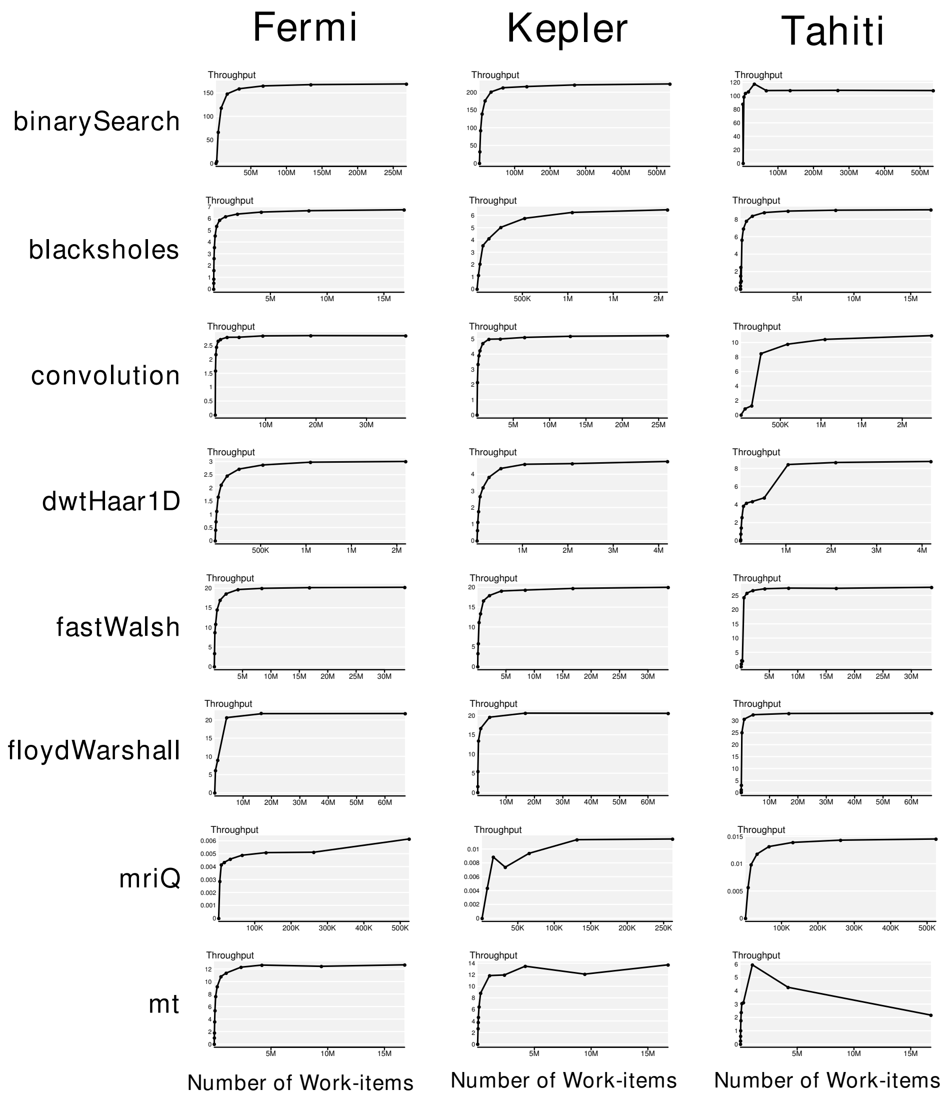


Figure 6.2: Throughput, defined as billions units of work processed per unit of time (see formula 6.1), as a function of the total number of work-items (*i.e.*, the problem input size) for all the programs and the three devices.

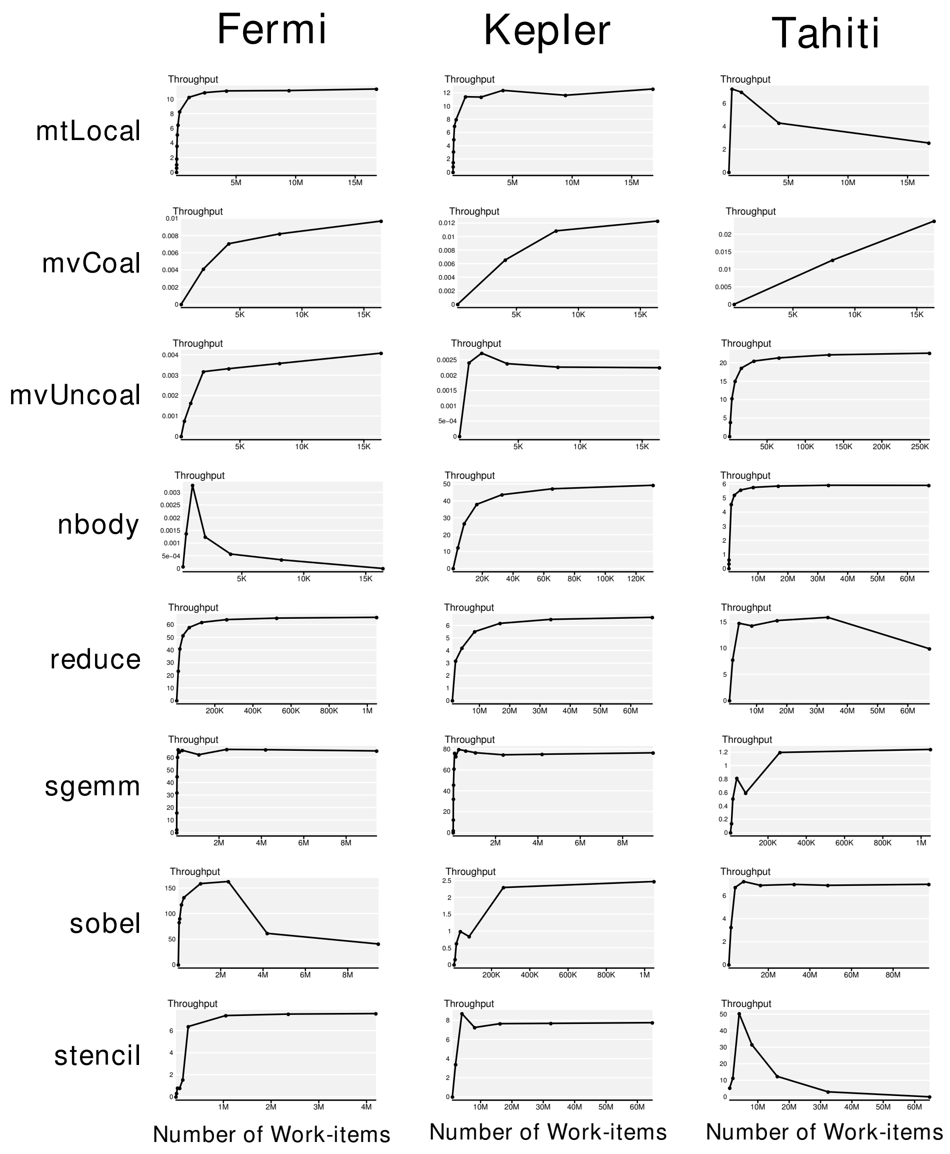


Figure 6.3: Throughput, defined as billions units of work processed per unit of time (see formula 6.1), as a function of the total number of work-items (*i.e.*, the problem input size) for all the programs and the three devices.

Program name	**Source**	**`NDRange` sizes limits**
`binarySearch`	AMD SDK	$[1M \dots 268M]$
`blackscholes`	Nvidia SDK	$[2K \dots 16M]$
`convolution`	AMD SDK	$[16K \dots 37M]$
`dwtHaar1D`	AMD SDK	$[32 \dots 2M]$
`fastWalsh`	AMD SDK	$[32K \dots 33M]$
`floydWarshall`	AMD SDK	$[9K \dots 67M]$
`mriQ`	Parboil	$[1K \dots 524K]$
`mt`	Nvidia SDK	$[1K \dots 16M]$
`mtLocal`	Nvidia SDK	$[1K \dots 16M]$
`mvCoal`	Nvidia SDK	$[256 \dots 16K]$
`mvUncoal`	Nvidia SDK	$[256 \dots 16K]$
`nbody`	AMD SDK	$[1K \dots 262K]$
`reduce`	AMD SDK	$[262K \dots 67M]$
`sgemm`	Parboil	$[256 \dots 9M]$
`sobel`	AMD SDK	$[4K \dots 1M]$
`stencil`	Parboil	$[1M \dots 97M]$

Table 6.3: OpenCL programs used as benchmarks with the corresponding range of input sizes we used for experiments

6.4 Throughput and Hardware Saturation

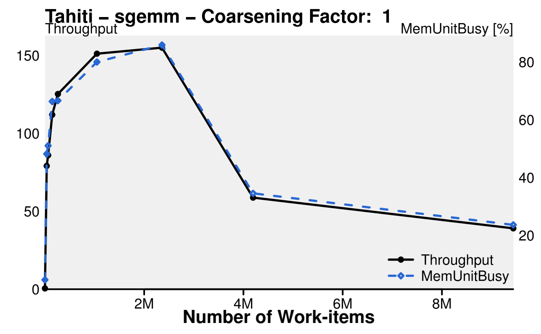
This section defines the concept of hardware saturation and presents its features.

Graphics processors are highly parallel machines containing a large amount of computing cores. To harness the computing power of GPUs applications must run large amount of work-items to fully exploit all the hardware resources. Programs running a small number of work-items might show a behaviour which is not representative of larger ones due to under-utilization of the hardware resources. We describe this using the notion of throughput which we formally defined as :

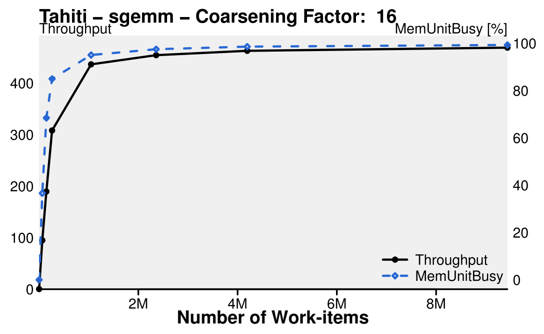
$$throughput = \frac{units\_of\_work}{execution\_time} \tag{6.1}$$

where $units\_of\_work$ is a metric which depends on the algorithmic complexity of the application. In case of linear benchmarks, this corresponds to the number of input elements processed by the kernel: $units\_of\_work = input\_size$. In case of non-linear programs the input size is scaled according to the complexity. For example the `nbody` program is quadratic in complexity and $units\_of\_work = (input\_size)^2$. Note that in our benchmarks the total number of work-items is a linear function of the $input\_size$.

Figures 6.2 and 6.3 show the throughput as a function of the total number of running work-items for each benchmark and our three platforms. Consider, for example, the top-left plot



(a) Coarsening Factor 1.



(b) Coarsening Factor 16.

Figure 6.4: Plots showing the correlation between throughput and *MemUnitBusy* hardware counter for `sgemm` on *Tahiti* using coarsening factor 1 and 16. *MemUnitBusy* is the percentage of time in which the memory unit is busy processing memory requests. We can see that the coarsened configuration does not suffer of low memory utilization leading much higher throughput. Throughput is expressed in billions of units of work per second.

in figure 6.2, `binarySearch` running on *Fermi*. This graph clearly shows that the throughput increases along with the problem size (*i.e.*, total number of work-items) until 50 million work-items. Passed this size the throughput reaches a plateau and stabilizes at around 180 billion $units\_of\_work$ per second. We call this region of the problem input size space *saturation* region. From the shape of the plot we can make two observations. The first one is that a program executed with a small number of work-items behaves differently then when executed with a large number of work-items. Small input sizes show low throughput, this means that they are in proportion slower than large input sizes. Second, the fact that the throughput stabilizes for sufficiently large input sizes shows that we hit the hardware saturation limit of the GPUs and that applications scale according to the theoretical complexity of the algorithm.

6.4.1 Outliers Analysis

The *Tahiti* column in figures 6.2 and 6.3 show a number of outliers which do not show a plateau. Examples of these are: `mvUncoal`, `sgemm`, `stencil`, `mt` and `mtLocal`. Being liner algebra applications, these programs are usually memory bound [129]. We investigate these cases using performance counters. Figure 6.4 reports the results of our analysis for `sgemm`. The top sub-figure shows the throughput performance along with the value of the performance counter *MemUnitBusy* as a function of the total number of work-items. *MemUnitBusy* is the hardware counter that measures the percentage of execution time that the memory unit is busy processing loads and stores. We can see that there is a high correlation between the two curves. This signifies that performance for large input sizes is limited by the memory functional unit, which is unable to handle efficiently a large amount of requests coming from different work-items. As shown in figure 6.4b the coarsening transformation mitigates this problem, leading to much higher performance. The throughput curve still follows the hardware counter with no decrease for higher input sizes. A similar analysis can be performed for programs such as `mvUncoal` where the uncoalesced memory access pattern highlights the problem even more. The `mvUncoal` benchmark is described in more detail in section 6.6.

6.5 Throughput-Based Tuning

Taking advantage of the saturation properties described earlier, this section introduces a methodology to accelerate iterative compilation.

6.5.1 Tuning Across Input Sizes

We now show how the best compiler settings found for a given input size perform on the largest input size that we used in our experiments. Figures 6.5 and 6.6 shows this performance on all benchmarks and devices for the full compiler transformation space descried in section 6.3. For example a value of 1 for input size x means that the best performing coarsening configuration for x is the best for the target size as well. A value of 0.5 means that the best configuration on x gives half of the maximum performance when evaluated on the target. By definition the line reaches 1 for the largest input size.

The overall trend for performance is to increase as the input size on which we perform the search gets close to the target one. After a certain input size, performance reaches a plateau and stabilizes in most cases. Small input sizes tend to have low and unpredictable performance while large input sizes have performance close to that of the largest input size. This means that for input sizes in that saturate the device performance reaches stability. This can be clearly seen when comparing figures 6.2 and 6.3 with figures 6.5 and 6.6; when throughput saturates, so does the performance.

6.5.2 Throughput and Coarsening Factor

Before introducing our search technique, we consider the impact of the coarsening factor on the throughput as shown in figure 6.7. This figure shows the throughput for five different values of the factor parameter (labelled on the right). The relative performance of the different coarsening factors changes across input sizes (*i.e.*, the lines in the plot overlap and cross) until saturation is reached between 1M and 4M work-items. For all coarsening factors, the throughput reaches a plateau and stabilizes for sizes larger than 4M work-items. In this example the factors 8 and 16 are the best parameters for problem sizes of about 4 million work-items remaining stable up to 67 million work-items.

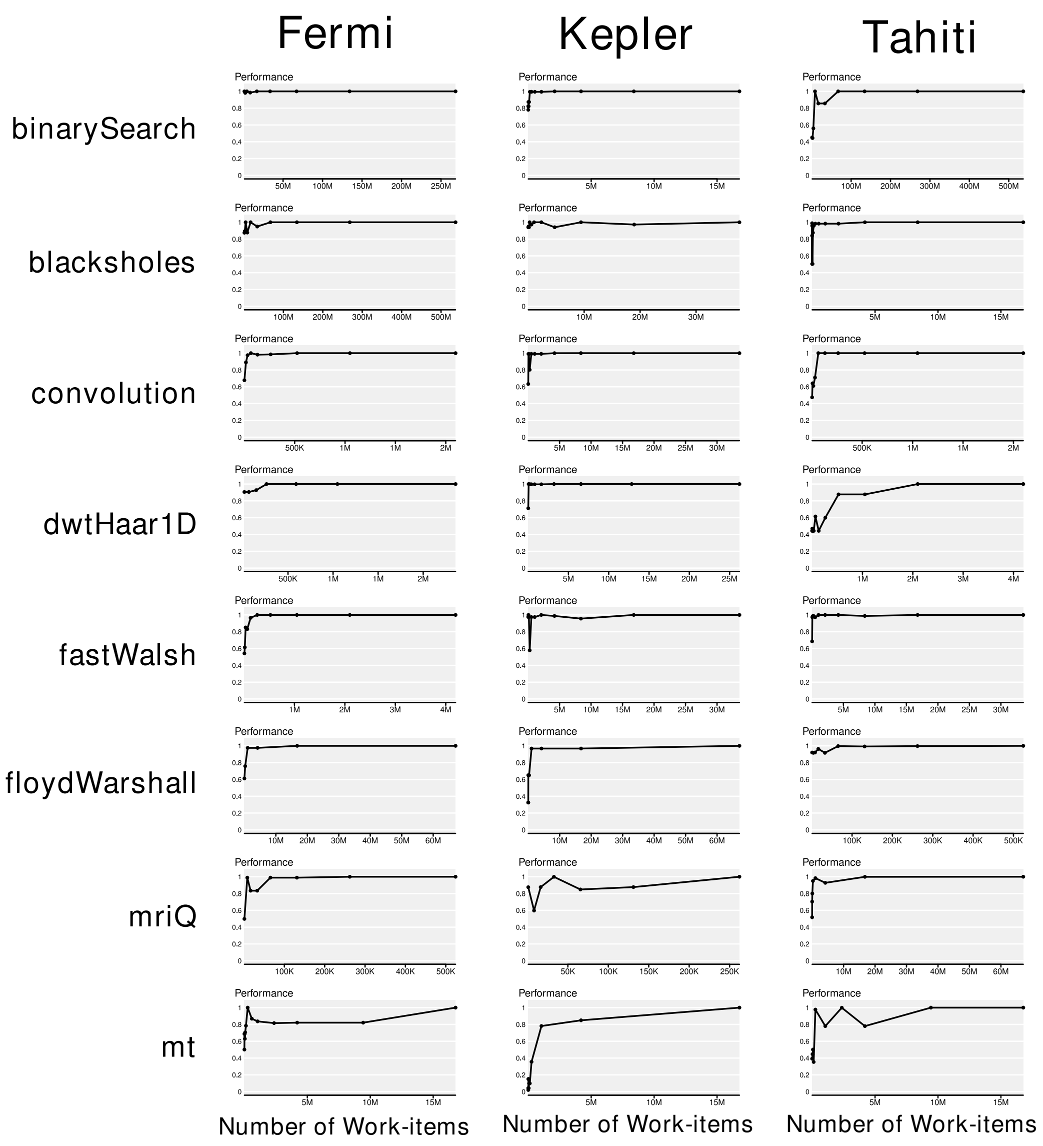


Figure 6.5: Performance of the best optimisations settings found for a given input size evaluated on the largest input size. The value that is plotted is the relative proportion of the maximum performance attainable. A value of 1 means that the maximum performance is achieved.

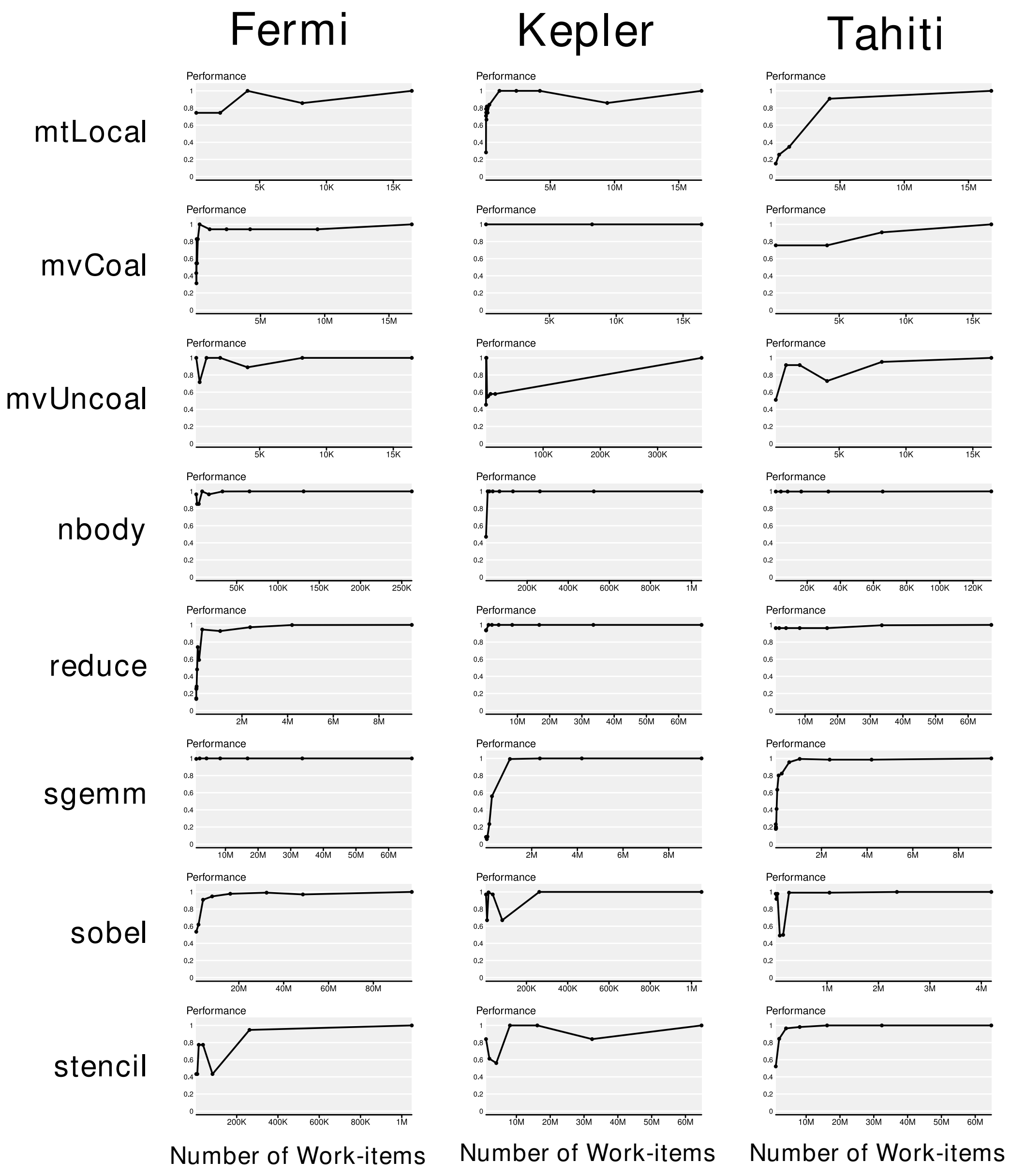


Figure 6.6: Performance of the best optimisations settings found for a given input size evaluated on the largest input size. The value that is plotted is the relative proportion of the maximum performance attainable. A value of 1 means that the maximum performance is achieved.

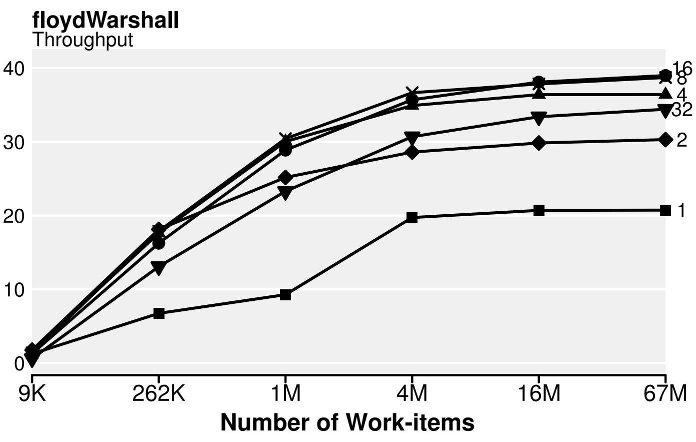


Figure 6.7: Throughput as a function of problem input size (*i.e.*, the total number of work-items) for five different values of the coarsening factor (on the right) for `floydWarshall` running on *Kepler*. Note the log-scale of the x-axis, this to highlight the throughput of small input sizes. Throughput is expressed in billions of units of work per second. The relative performance of the curves remains stable for problem sizes larger than 4 million threads.

This signifies that performing a search for the best parameter on an input size at the left of the saturation point will probably lead to a bad choice when evaluating on the largest input size. On the contrary, performing beyond the saturation should lead to good choices when evaluated on the largest input size.

6.5.3 Throughput Based Input Size Selection

We now propose a tuning technique that takes advantage of the saturation plateau that exists for both performance and throughput. Our method is the following. We first build the throughput curve by running the benchmark using the default compiler parameters on multiple input sizes within the range given in table 6.3. Once the throughput curve is built, we select the smallest input size that achieves a throughput within a given threshold of the maximum one. The threshold is used to deal with noise in the experimental data and small fluctuations in execution time around the throughput plateau. The resulting selected input size is our Minimum Saturation Point (MSP). These are presented in table 6.4 for all benchmarks and devices. The last step consist of searching the coarsening space for an input size equal to the MSP. Following this methodology, we expect that the best optimisation settings found at the MSP leads to performance in par with the best for the largest input size. We refer to this searching technique as MSP-Tuning.

The next section presents the results obtained applying the proposed tuning technique.

	Program name	Fermi	Kepler	Tahiti
1)	`binarySearch`	8M	16M	8M
2)	`blackscholes`	1M	1M	1M
3)	`convolution`	262K	802K	1M
4)	`dwtHaar1D`	262K	524K	1M
5)	`fastWalsh`	2M	2M	1M
6)	`floydWarshall`	4M	4M	1M
7)	`mriQ`	524K	131K	65K
8)	`mt`	2.3M	4M	1M
9)	`mtLocal`	1M	1M	262K
10)	`mvCoal`	16K	16K	16K
11)	`mvUncoal`	16K	1K	1K
12)	`nbody`	32K	65K	131K
13)	`reduce`	1M	1M	2M
14)	`sgemm`	36K	65K	1M
15)	`sobel`	262K	262K	1M
16)	`stencil`	4M	4M	4M

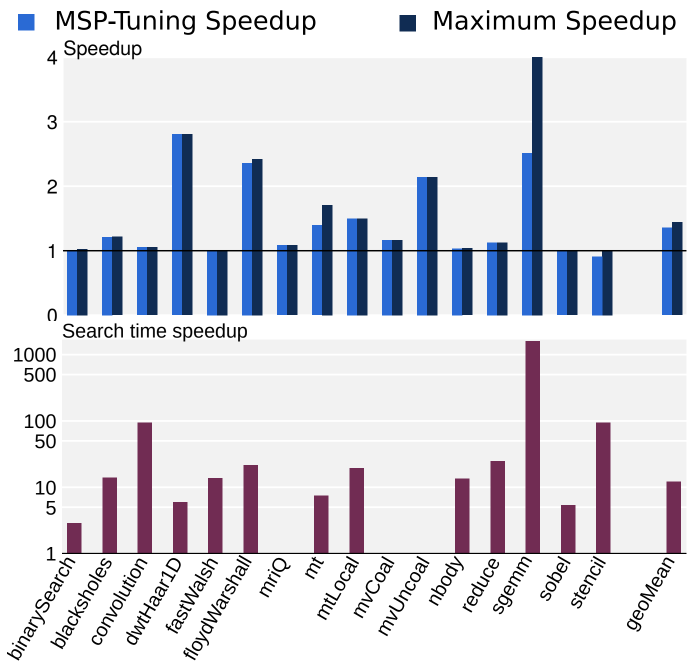
Table 6.4: Minimum Saturation Point identified by the technique presented in section 6.5.3 for all the benchmarks and architectures. MSP is expressed in number of work-items.

6.6 Results

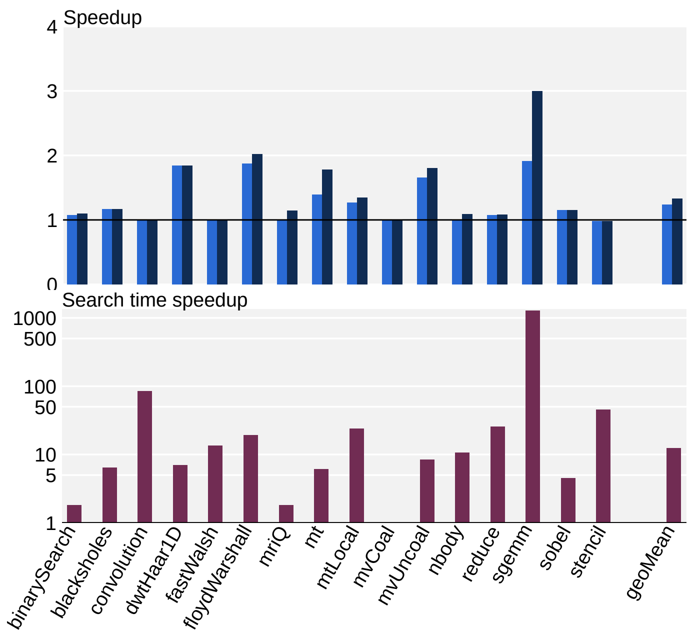
This section provides results for the speedups in kernel execution time and in search time given by MSP-Tuning.

Definition of Search Speedup In our experiments the baseline to compute kernel speedups is represented by the execution time of the application run without applying the coarsening transformation (*i.e.*, coarsening factor 1 and stride 1) and with the default work-group size by the benchmark suite. In all our experiments the problem size we are optimising for is the largest of the range we record in table 6.3, we call this the *target* problem size. Figure 6.8 and 6.9 summarizes the results of MSP-Tuning described in section 6.5.3 for our three OpenCL devices. Consider the top barplot in each subfigure. It reports the speedup over the baseline attainable using MSP-Tuning compared to the maximum speedup attainable with thread-coarsening. In these results we chose to use a threshold of 10% for the identification of the MSP. The second barplot reports the speedup in the search optimal configuration. The speedup is computed using this formula:

$$Search\_speedup = \frac{Time_{target}}{Time_{MSP} + Time_{throughput}} \tag{6.2}$$

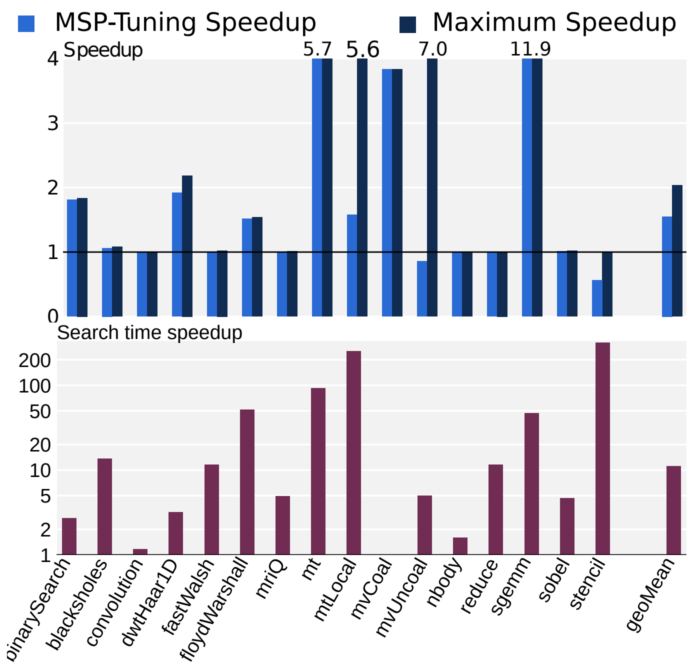


(a) *Fermi*



(b) *Kepler*

Figure 6.8: Summary of the tuning results for Nvidia GPUs. In each subfigure, the top plot represents the kernel execution speedup attainable with MSP-Tuning in comparison with the maximum available speedup given by coarsening. The bottom plot shows the search-time speedup given by MSP-Tuning, notice that in this plot the y-axis is in log-scale.



(a) *Tahiti*

Figure 6.9: Summary of the tuning results for *Tahiti*. The top plot represents the kernel execution speedup attainable with MSP-Tuning in comparison with the maximum available speedup given by coarsening. The bottom plot shows the search-time speedup given by MSP-Tuning, notice that in this plot the y-axis is in log-scale.

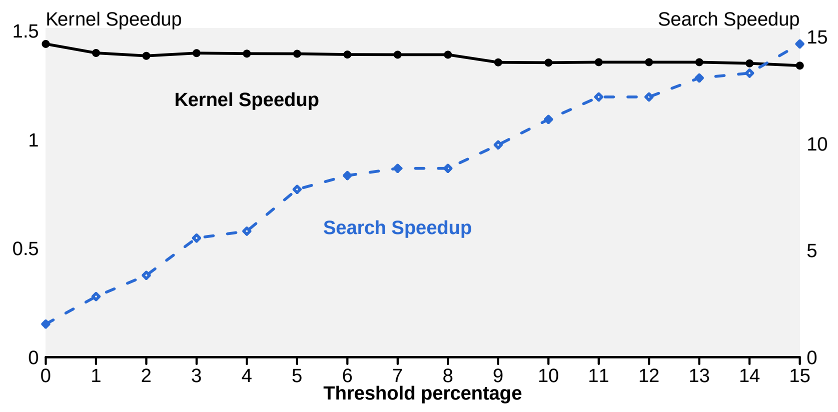
Here $Time_{target}$ is the time spent exhaustively searching for the best configuration on the *target* input size, $Time_{MSP}$ is the search time on the MSP input size and $Time_{throughput}$ is the time spent building the throughput curve.

6.6.1 Result Description

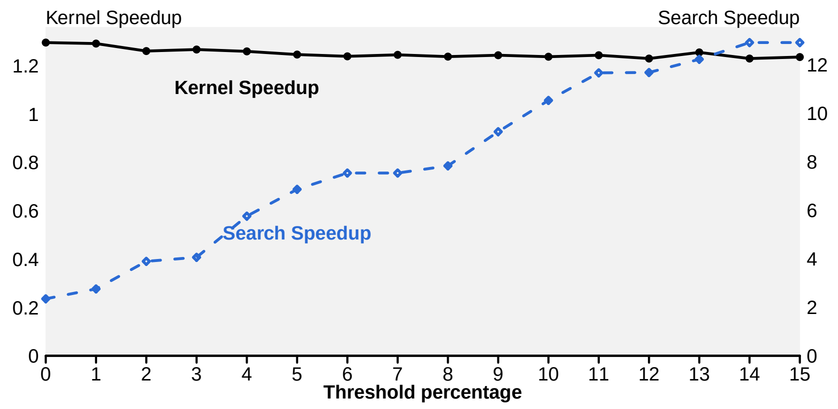
The kernel speedups achieved on *Fermi*, *Kepler* and *Tahiti* are respectively: 1.35×, 1.24× and 1.55×. Which represent 82%, 73% and 53% of the maximum performance. On the other hand the search time speedup given by MSP-Tuning is about one order of magnitude for the three devices. To understand the results in search time we have to take into account the algorithmic complexity of the application. A program like `sgemm` ensures large tuning savings (even in the order of thousands) thanks to its cubic complexity: halving the input size leads to a factor of eight in tuning time saving. The overall results are similar for the two Nvidia GPUs, *Fermi* and *Kepler* demonstrating the similarities of the two architectures. On the other hand the Tahiti results show significant differences with higher speedups on average. The search fails to achieve good performance on `mtLocal`, `mvUncoal` and `stencil` due to the erratic shape of throughput and performance lines for these benchmarks, see figures 6.2, 6.3 and 6.5, 6.6.

Of particular interest is the application `mvCoal`. This is a matrix-vector multiplication benchmark from the Nvidia SDK. Considering the search time speedup barplots (and comparing tables 6.3 and 6.4) we can see that no reduction on the input size is attainable, *i.e.*, the selected MSP corresponds to the largest input size. This is because no saturation plateau is reached, look at the corresponding plot in figures 6.2 and 6.3. The reason for this anomaly lies in the structure of the algorithm: one thread processes one row of the input matrix and the whole input vector producing a single element of the output. Thus we have only a thread for each matrix row: a little number of work-items with respect to amount of data processed. Scaling the problem to larger sizes is prohibitive, running 16K work-items means to work with about 1GB of data, scale to 32K means to work with 4GB, hitting device hardware constraints. In summary our target GPUs cannot full express the throughput potential of `mvCoal` due to limitations in the available memory resources. Similar considerations can be made for `mvUncoal`, a different version of the same program.

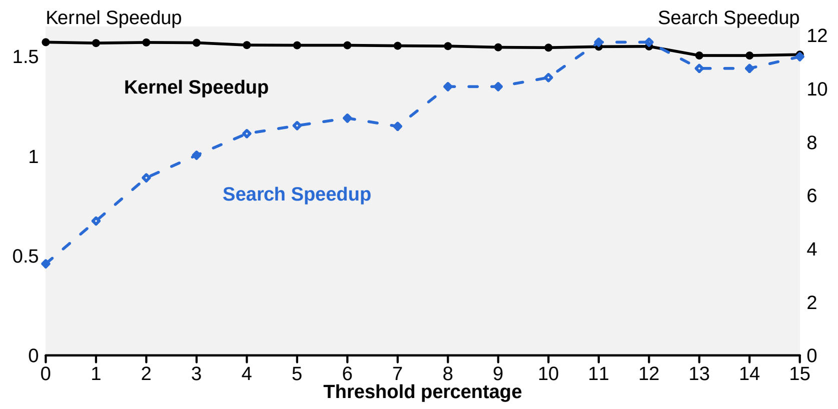
Noise Threshold Figure 6.10 shows how the choice of the noise threshold described in section 6.5.3 affects the performance of our strategy. For the values of percentages between 0 and 15 it plots the achievable kernel speedup by implementing the search technique and the saving in search time. As expected the search speedup increases with the threshold percentage, this because the higher threshold values allows the selection of smaller input sizes as MSP. The kernel speedup slightly decreases increasing the percentage. Our search technique works because the speedup given by MSP search increases faster than the decrease of the kernel speedup.



(a) Fermi



(b) Kepler



(c) Tahiti

Figure 6.10: Plots showing the attainable kernel speedup and the search-time speedup as a function of the noise threshold. A threshold of 0% means that we select as MSP the input size giving the maximum throughput. We can see that by increasing the threshold we can improve the search speedup with small degradation of the kernel speedup.

Based on this data, we select a threshold of 10%, leading to marginal performance degradation with respect to 0% and more than 10$\times$ improvement in search speed across platforms.

6.7 Summary

This chapter has introduced a way to improve the run-time performance of iterative compilation by leveraging the properties of modern GPU architectures. In particular we exploit hardware saturation. Saturation is reached when the device runs a problem large enough (in number of work-items) so to fully utilize its hardware resources. We have provided experimental evidence of saturation on three devices from Nvidia and AMD. We have showed that the thread-coarsening compiler transformation has stable performance across problem sizes that saturate hardware resources. We then propose a technique to identify the lower–bound of the saturation area in the problem input space (called MSP). We use this input size for fast tuning of the parameters of the coarsening transformation without relying on iterative compilation.

Next chapter proposes a methodology to select a coarsening factor and a stride that can improve performance over the baseline.

Chapter 7

Selection of Coarsening Parameters

This chapter presents a technique to select the coarsening factor and a methodology to select the stride parameter. The coarsening factor is chosen by developing a neural network model that can decide whether to coarsen a program or not. The stride is selected relying on symbolic execution of the kernel function by our `SymEngine` tool. Section 7.1 introduces the problem defining the goal of our prediction. Section 7.2 motivates the work specifically showing that the coarsening factor parameter has to be chosen specifically for each program and architecture. Section 7.3 describes the set-up for our experiments. Section 7.4 characterizes the optimisation space for the coarsening and stride parameter. Section 7.5 presents our policy to select the stride parameter. Section 7.6 describes the methodology that we used to train the neural network model for the coarsening factor prediction. Section 7.7 presents the overall results for our prediction methodologies. Section 7.8 concludes the chapter.

7.1 Introduction

Graphical Processing Units are widely used for high performance computing. They provide cost-effective parallelism for a wide range of applications. The success of these devices has lead to the introduction of a diverse range of architectures from many hardware manufacturers. OpenCL is the industry-standard language for GPUs that offers program portability across accelerators of different vendors: a single piece of OpenCL code is guaranteed to be executable on many devices.

A uniform language specification, however, still requires programmers to manually optimise kernel code to improve performance on each target architecture. This is a tedious process, which requires knowledge of the hardware, and must be repeated each time the hardware is updated. This is also true for the process of tuning compiler heuristics. This problem is particularly acute for GPUs which undergo rapid hardware evolution. The solution to this problem is a cross-architectural optimiser capable of achieving performance portability.

This chapter studies this issue focusing on the optimisation of the thread-coarsening compiler transformation described in chapter 4. Thread-coarsening merges together two or more work-items, increasing the amount of work performed by a single work-item, and reducing the total number of work-items instantiated. Selecting the best coarsening factor *i.e.*, the number of work-items to merge together, is a trade-off between exploiting thread-level parallelism and avoiding execution of redundant instructions. Making the correct choice leads to significant speedups on all our devices. Our data show that picking the optimal coarsening factor is difficult since most configurations lead to performance decrease and only careful selection of the coarsening factor gives improvements. Selecting the best parameter requires knowledge of the particular device since different GPUs have different optimal factors

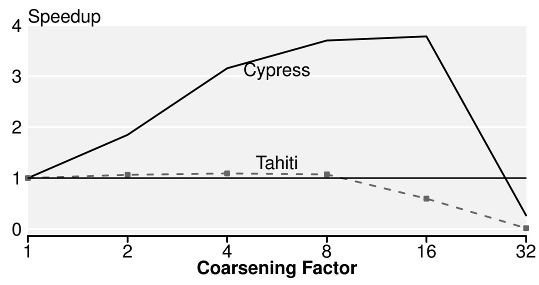
In the previous chapter we showed how to speedup iterative compilation for the tuning of the coarsening transformation. In this new piece of work we aim at selecting the coarsening factor using an automated machine learning technique. We build our model based on a neural network that decides whether it is beneficial to apply coarsening. The inputs to the model are static code features extracted from the parallel OpenCL code. These features include, among the others, branch divergence and instruction mix information. A different model is trained for each of the four GPUs we use for experiments. This means that our training methodology is the same for the four devices, the resulting model is different since it adapts itself to the characteristics of the target GPU. We also introduce a technique to predict the stride parameter. Our proposal is based on symbolic execution of OpenCL kernel with the goal of extracting information about coalescing of memory accesses. The tool that perform the analysis is called `SymEngine` (presented in section 4.6)

In summary this chapter makes the following contributions:

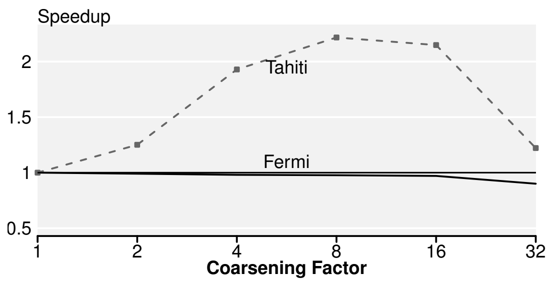
- We provide a characterization of the optimisation space for the coarsening factor and the stride for our four GPU architectures.
- We propose a way to select an appropriate stride factor that relies on the static computation of memory transactions made by `SymEngine`.
- For each of the four devices we train a neural network model to predict whether to apply coarsening or not.
- We show our performance results for our 17 benchmarks and four architectures.

7.2 Motivation

This section motivates the work explaining the importance of determining whether to coarsen, and by how much. Figure 7.1 shows the speedup given by thread-coarsening over the baseline (no coarsening) as a function of the coarsening factor for two benchmarks on different GPUs.



(a) `nbody` run on *Cypress* and *Tahiti*



(b) `binarySearch` run on *Fermi* and *Tahiti*

Figure 7.1: Speedup achieved with thread-coarsening as a function of the coarsening factor for `nbody` and `binarySearch`

The first figure (7.1a) shows the performance of `nbody` on two AMD architectures: *Cypress* and *Tahiti*. Even though these two devices are from the same vendor, they have significant architectural differences [9]. Cypress cores are VLIW while Tahiti cores have processing elements working in SIMD fashion. Such differences are reflected in the different effects that the coarsening transformation has on the benchmarks. The higher degree of ILP made available by coarsening in the `nbody` benchmark is successfully exploited by the VLIW cores of *Cypress* ensuring a significant speedup. The same does not apply to *Tahiti* where no improvement can be obtained and we record a significant slowdown for higher factors.

Differences across architectures also emerge for `binarySearch`, figure 7.1b. Here, we make a comparison between the AMD *Tahiti* and the Nvidia *Fermi* architectures. On the first device `binarySearch` ensures an improvement of 2.3x over the default for coarsening factor 8. On Nvidia, however, no improvement can be obtained.

From these simple examples it is clear that optimising coarsening is an important problem. The coarsening factor must be determined on a per-program and per-device basis. Classical hand-written heuristics are not only complex to develop, but are likely to fail due to the variety of programs and ever-changing OpenCL devices available.

7.3 Experimental Set-up

This section presents the experimental set-up for this chapter.

7.3.1 Parameter Space

In this work we address the tuning of the *coarsening factor* and the *stride* parameters. The first one controls how many work-items to merge together (see section 4.3), while the second defines how many work-items to interleave between thread to merge together (see section 4.3.2). In our experiments, we selected values $1, 2, 4, 8, 16, 32$ for both coarsening factor and stride. We

	Program name	Source	`NDRange` size
1)	`binarySearch`	AMD SDK	268M
2)	`blackscholes`	Nvidia SDK	16M
3)	`convolution`	AMD SDK	6K
4)	`dwtHaar1D`	AMD SDK	2M
5)	`fastWalsh`	AMD SDK	33M
6)	`floydWarshall`	AMD SDK	(8K x 8K)
7)	`mriQ`	Parboil	524K
8)	`mt`	AMD SDK	(4K x 4K)
9)	`mtLocal`	AMD SDK	(4K x 4K)
10)	`mvCoal`	Nvidia SDK	16K
11)	`mvUncoal`	Nvidia SDK	16K
12)	`nbody`	AMD SDK	131K
13)	`reduce`	AMD SDK	67M
14)	`sgemm`	Parboil	(3K x 3K)
15)	`sobel`	AMD SDK	(1K x 1K)
16)	`spmv`	Parboil	262K
17)	`stencil`	Parboil	(3K x 510 x 62)

Table 7.1: OpenCL applications with the reference number of work-items.

limited our evaluations to powers of two since these are the most widely applicable parameter values, given the reference input sizes for the considered benchmarks.

7.3.2 Benchmarks

We use 17 benchmarks from various sources as shown in table 7.1. The table also shows the global size (total number of work-items) used. The baseline performance reported in the results section is that of the best work-group size (obtained with a search) as opposed to the default one chose by the programmer. This is necessary to give a fair comparison across devices the original benchmark is written specifically for one GPU. Each experiment has been repeated 50 times aggregating the results using the median. Chapter 6 has shown that the choice for the optimal coarsening factor is consistent across input sizes. For this reason we restricted our analysis to the input size (*i.e.*, total number of work-items running) for the original kernel listed in table 7.1.

Name	Model	GPU Driver	OpenCL version	Linux kernel
Fermi	Nvidia GTX 480	304.54	1.1 CUDA 5.0.1	3.2.0
Kepler	Nvidia K20c	331.20	1.1 CUDA 5.0.1	3.7.10
Cypress	AMD HD5900	1124.2	1.2 AMD-APP 1124.2	3.1.10
Tahiti	AMD HD7970	1084.4	1.2 AMD-APP 1084.4	3.1.10

Table 7.2: OpenCL devices used for our experiments.

7.3.3 Devices

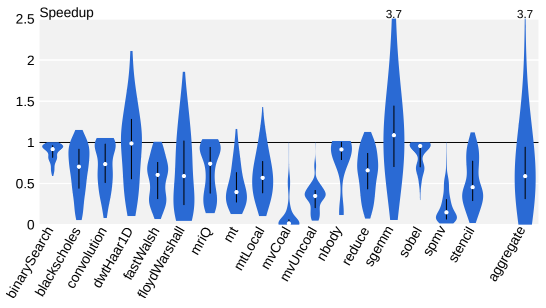
The devices used for the experiments are listed in table 7.2 and the same as for chapter 5. The only differences are driver versions and operating system version. Refer to section 5.3.1 for a description.

7.4 Optimisation Space Characterization

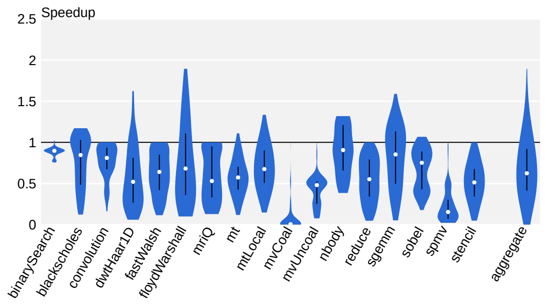
In the motivating example, in figure 7.1, we showed that different applications behave differently with respect to coarsening on different devices. In this section we expand on this topic and present the overall optimisation space.

7.4.1 Distribution of Speedup

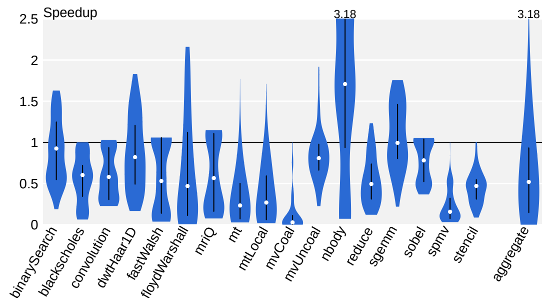
The violin plots in figure 7.2 show the distribution of the speedups (over no coarsening) achievable by changing the coarsening factor and the stride parameter. The width of each violin corresponds to the proportions of configurations with a certain speedup. The white dot denotes the median speedup, while the thick black line shows where 50% of the data lies. These plots highlight differences and similarities in the spaces of different applications. Intuitively, violins with a pointy top correspond to benchmarks difficult to optimise: configurations giving the maximum performance are few. This is the case for `mt` and `mtLocal` (matrix multiplication and matrix multiplication with local memory). At the same time, these two programs show similar shapes among the two Nvidia and AMD devices. On the other hand `nbody` shows different shapes among the four architectures: a large improvement is available on *Cypress* while no performance gain can be obtained on the other three devices. Interesting cases for analysis are benchmarks such as `mvCoal` and `spmv`. They have an extremely pointy top with and the whole violin is below 1, meaning no performance improvement is possible on these benchmarks with coarsening. Consider now the aggregate violin in the final column of each subfigure. This shows the distribution of speedups across all programs. In all the four cases the white dots lies well below the 1, near 0.5 in most cases. This result means that on average, the coarsening transformation will slow programs down. This fact coupled with the different



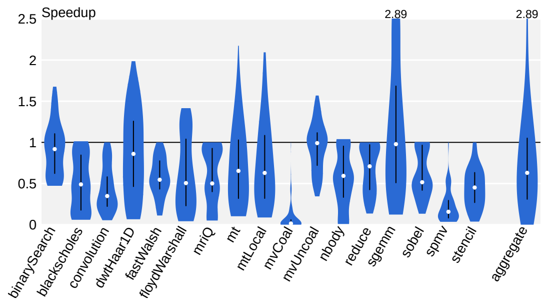
(a) *Fermi*



(b) *Kepler*



(c) *Cypress*



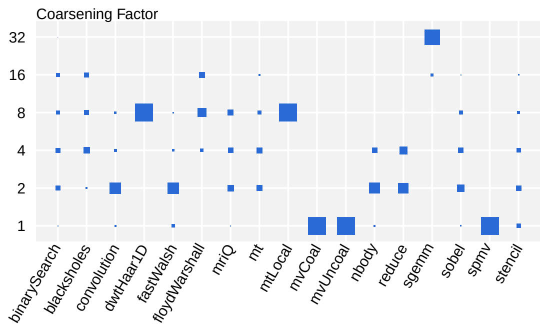
(d) *Tahiti*

Figure 7.2: Violin plots showing the distribution of speedups for all the benchmarks and devices. The shape of the violin corresponds to the speedup distribution. They indicate on how hard it is to tune a program. The thick black line shows where 50% of the data lies. The white dot is the position of the median.

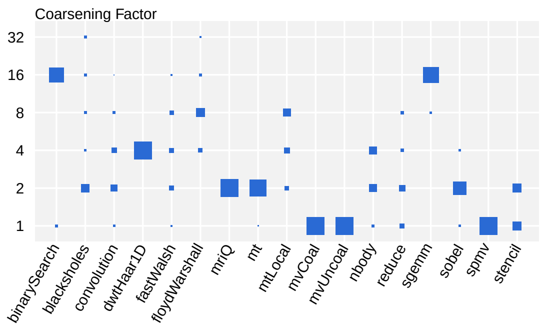
optimisation distributions shows how difficult choosing the right coarsening factor and stride is. The heuristic in section 7.5 and the model presented in section 7.6 tackle the problem of selecting an appropriate configuration for each program on the four machines.

7.4.2 Coarsening Factor Characterization

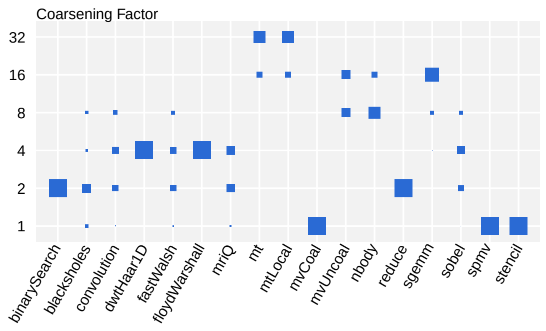
One of the objectives of this work is the determination of the best thread-coarsening factor. In this section we quantify how much this parameter varies across program and device. Figure 7.3 shows Hinton diagrams for each target device. For each benchmark and coarsening factor it plots a square. The size of the square is proportional to the percentage of configurations having the given coarsening factor among the top 5% performing ones. Intuitively: the larger the square is, the more likely it is that the corresponding coarsening factor will perform well. From these plots it is immediately clear that no single factor can give good performance for all the programs. On the two Nvidia devices (*Fermi* and *Kepler*) the optimal coarsening factor tends to be among 1, 2, or 4, with few exceptions. On the two AMD architectures (*Tahiti* and *Cypress*) instead the best coarsening factor are often larger (4 or 8).
In section 7.7 we show the speedup we can achieve by using a single coarsening factor for all the programs on a specific device. We compare our prediction model against this baseline.



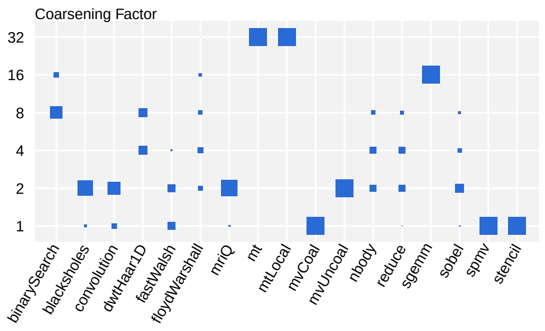
(a) *Fermi*



(b) *Kepler*

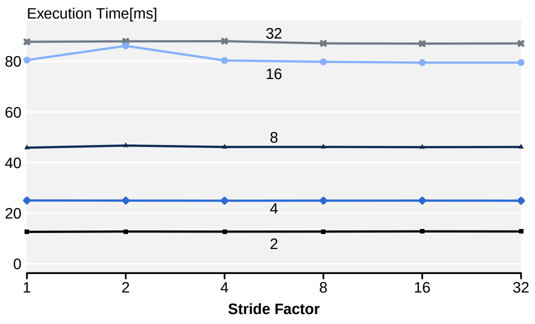


(c) *Cypress*

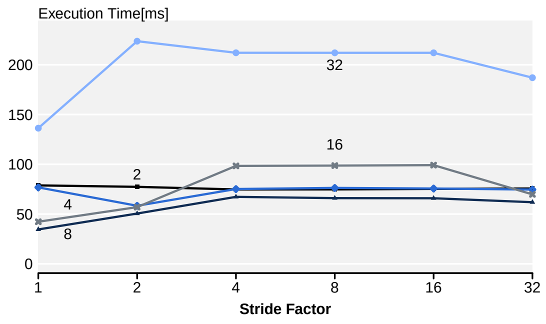


(d) *Tahiti*

Figure 7.3: Hinton diagrams showing the percentage of configurations with the given coarsening factor in the performing top 5% configurations. Intuitively, the larger the square is the more likely it is for the given coarsening factor to be the fastest.



(a) *Fermi*: the stride parameter has no effects on performance.



(b) *Cypress*: large strides have detrimental effect on performance. The minimum execution time is for stride 1.

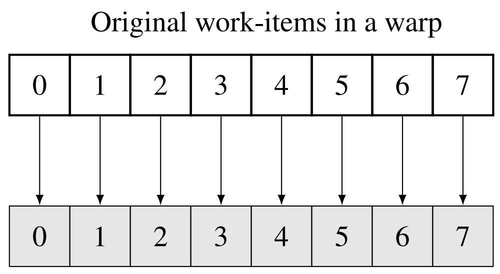
Figure 7.4: Execution time (in milliseconds), for the `mvUncoal` kernel running on *Fermi* and *Cypress* varying the stride parameter for multiple coarsening factors. The labels close to the lines are the corresponding coarsening factor.

7.4.3 Stride Factor Characterization

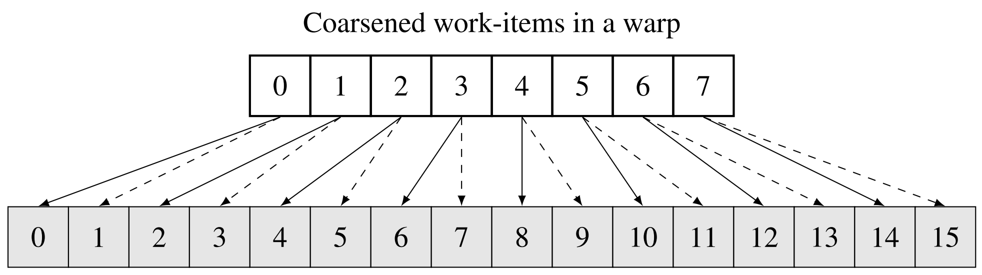
This section describes how the performance of our benchmarks is affected by the stride parameter.

The goal of our analysis of the stride parameter is to find ways to simplify the problem of selecting an appropriate value for it. We start by making two considerations. First, the stride parameter is only useful for benchmarks that show coalesced memory accesses in their original version. So, if a kernel does not have this property a stride larger than 1 might have a negative effect, if any, on performance. For example, consider figure 7.4. It plots the execution time of the `mvUncoal` kernel (which does not have coalesced accesses) running on *Fermi* and *Cypress* when varying the stride parameter. For *Fermi* we see no impact on performance by varying the stride parameter, while for *Cypress* we see that large stride slows the kernel down. The second idea is that that real benefit of the stride parameter is achieved when all the accesses by a memory instruction in a warp are consecutive. This property is achieved only for a coarsening factor as large as the warp size: 32 hardware threads on Nvidia and 16 on AMD (a quarter of a wavefront). We can see this in figure 7.5. Subfigure 7.5a shows a fully coalesced access pattern (all the memory request are contiguous). The schema then shows three different memory access patterns for the same memory operation after coarsening by factor 2. Subfigures b and c show the memory access pattern for stride 1 and 2. None of them achieves full coalescing: the accessed memory locations are not contiguous. Only the last configuration (7.5d) reaches coalescing, this is because the selected stride equals the width of the warp (8 in this example).

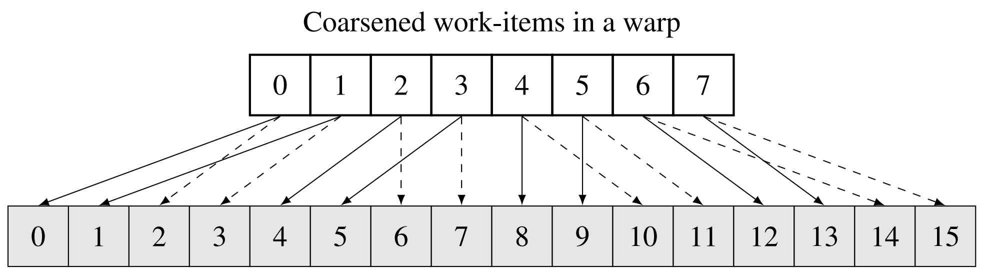
Using these two considerations we can simplify the search to just two stride values: 1 and 32. Figure 7.6 shows, for all 4 devices, a comparison between the maximum speedup



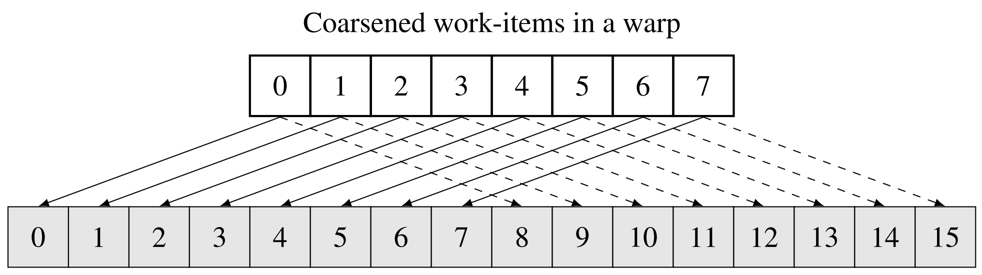
(a) Memory accesses of the original kernel



(b) Memory accesses for a stride of 1. The coalescing property is lost.



(c) Memory accesses for a stride of 2. Full coalescing of memory accesses is not achieved.



(d) Memory accesses for a stride of 8. Memory accesses are now completely coalesced (the data elements accessed by the warp in a single transaction are all consecutive).

Figure 7.5: Comparison of memory accesses for a kernel with warp size 8 coarsened by factor 2. White squares represent the work-items in a warp, while the grey squares represent memory locations. Arrows are accesses to memory locations. Solid arrows represent the accesses made by the original memory operation, dashed arrows represent the accesses made by the replicated one. All the accesses for solid arrows happen a at the same time, while dashed ones are a subsequent transaction.

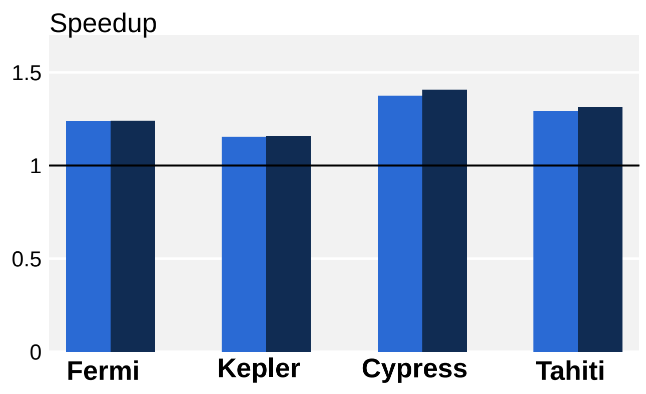
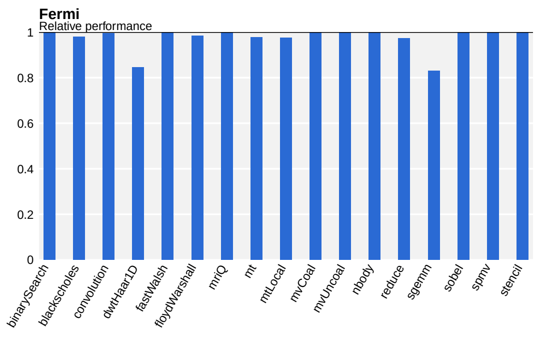


Figure 7.6: Comparison between the maximum performance attainable, second bar, with the performance given by restricting the choice for the stride to just 1 and 32, first bar.

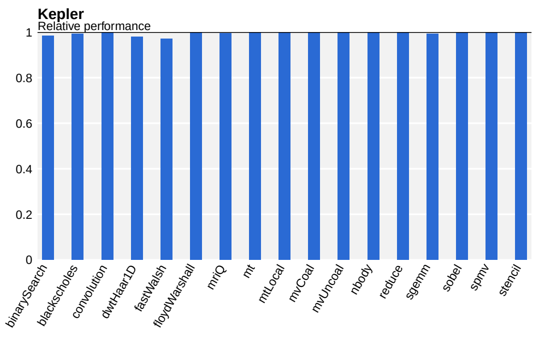
attainable in our original parameter space (the bar on the right) and the maximum speedup attainable restricting the choice of the stride to 1 or 32 (the bar on the left). We can see that the average performance penalty coming from this simplification is negligible. For *Fermi*, *Kepler*, *Cypress* and *Tahiti* using just two stride values gives 99.2%, 97.7%, 91.8%, 93.2% of the original performance on average across the benchmarks. Our optimisation policy, described in section 7.5, discriminates between stride values 1 and 32 on a per benchmark-basis.

7.4.4 Relationship between Coarsening Factor and Stride

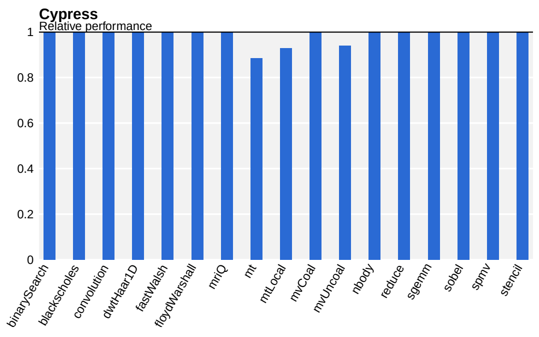
In this section we analyse the relationship between the coarsening factor and the stride parameter. We want to understand whether the choice for the optimal coarsening factor changes in relationship to the stride. We performed the following analysis: for each program on each device we select the coarsening factor which gives the best performance *on average* across all the stride values: we call it cf_{avg}. We then evaluate the performance attained by cf_{avg} for all stride values. If cf_{avg} performs better with respect to the other coarsening factors for each stride value this means that the relative performance of this coarsening factor is actually stable across stride values. This implies that the choice for the optimal factor is independent of the stride. Figure 7.7 reports the results of this analysis for each kernel and device. In these plots we show the average relative performance of cf_{avg} with respect to other coarsening factors for all the stride values. This number is usually close to 1, meaning that the cf_{avg} is not only the best coarsening factor *on average*, but also the *absolute* best. We can see that, by selecting the coarsening factor independently of the stride, we lose at maximum 17% of the maximum performance available on `sgemm` and `dwtHaar1D` on *Fermi*. From this we can conclude that the choice for the optimal coarsening factor is independent of the stride. This simplifies the optimisation problem, since we can decouple it in two simpler problems: the choice of the stride and the choice of the factor.



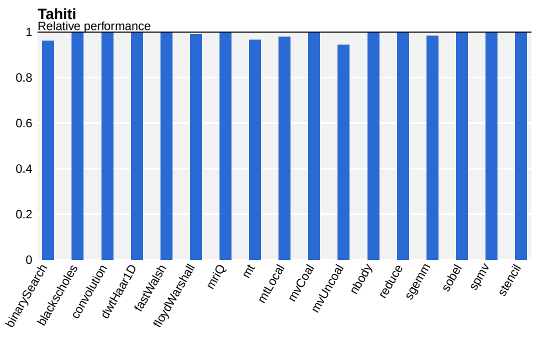
(a) *Fermi*



(b) *Kepler*



(c) *Cypress*



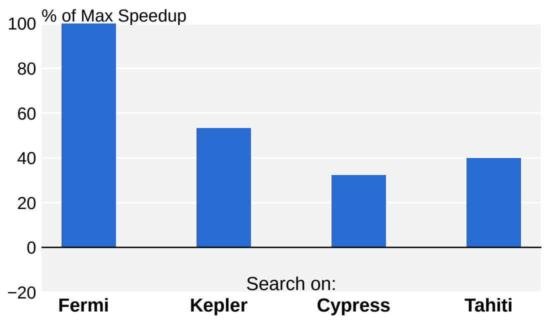
(d) *Tahiti*

Figure 7.7: Average performance of a single coarsening factor (called cf_{avg}) compared against other factors across all stride values.

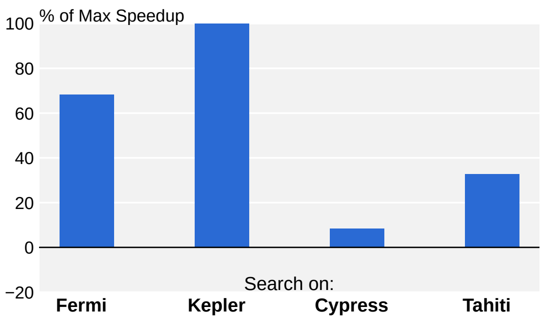
7.4.5 Cross-architecture optimisation portability

In this section we investigate if knowing the best coarsening factor and stride for one architecture can be successfully exploited to a new device. This study fulfils two purposes: (1) it characterizes the diversity of the devices that are evaluated in this work, (2) it studies if an optimiser tuned for a device would be effective when ported to another one. For this task we explored all coarsening factors and strides on all the kernel and devices selecting the best for each. Given two devices A and B we then evaluate for each kernel the performance of the best configuration of A (*Search-Device*) on B (*Target-Device*). Figure 7.8 reports the results of this analysis for all combinations of A and B. Our baseline (0%) is the performance of the unmodified kernel on the target device.

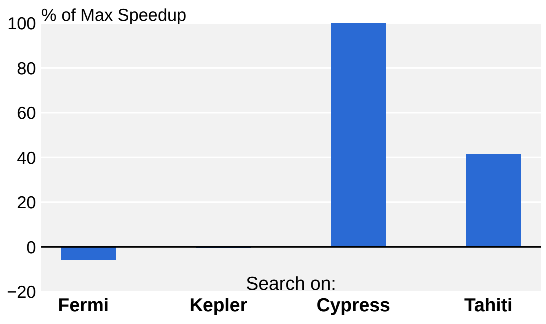
From these plots we can see, for example, that performance transfers effectively from *Fermi* to *Kepler* (first bar of figure 7.8b), where we reach 70% of the maximum speedup. Surprisingly the opposite is not true (see the second bar of figure 7.8a), where 52% is attainable. Its important to notice that porting performance across Nvidia and AMD leads to bad performance on average. For example the best configurations from *Fermi* and *Kepler* when evaluated on *Cypress* lead to performance degradations (of -6% and -0.2%). Similarly, bad performance is given when porting from *Cypress* and *Tahiti* to *Kepler*.



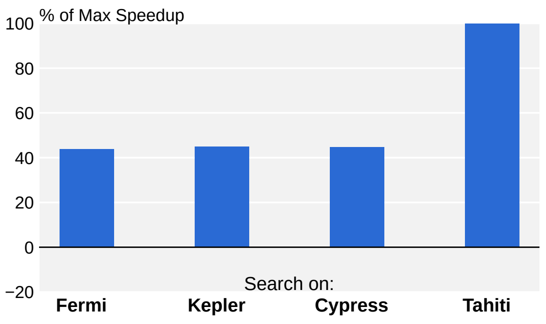
(a) Target device: *Fermi*



(b) Target device: *Kepler*



(c) Target device: *Cypress*



(d) Target device: *Tahiti*

Figure 7.8: The bars in these plots report the percentage of the maximum performance available on the target device (reported in the caption) achieved when porting the best configuration found on the search device (at the bottom of the bars). The baseline (meaning a performance improvement of 0%) is the performance of the unmodified kernel on the target device. When *Target-Device* and *Search-Device* are the same the bar reaches 100% by construction. For example, the *Cypress* bar in figure (a) shows that 37% of the maximum performance of *Cypress* is achievable when using the best configuration coming from *Fermi*. Instead the *Fermi* bar in figure (c) shows a performance degradation of 6%. Notice that the bar for *Kepler* in plot 7.8c shows a performance degradation of -0.2%, that is the reason why no bar is visible.

Consider that these results represent the upper bound of how an optimiser specialized for one architecture would perform when applied to another one. Any realistic compiler heuristic when applied to an unseen program, would perform considerably worse than this. This shows that performance portability is not guaranteed when going across vendors. This justifies the need to specialize the choice of the coarsening configuration on a per-device basis relying on a flexible optimiser. The goal of the model is to decide a coarsening factor and a stride for each kernel on a specific architecture.

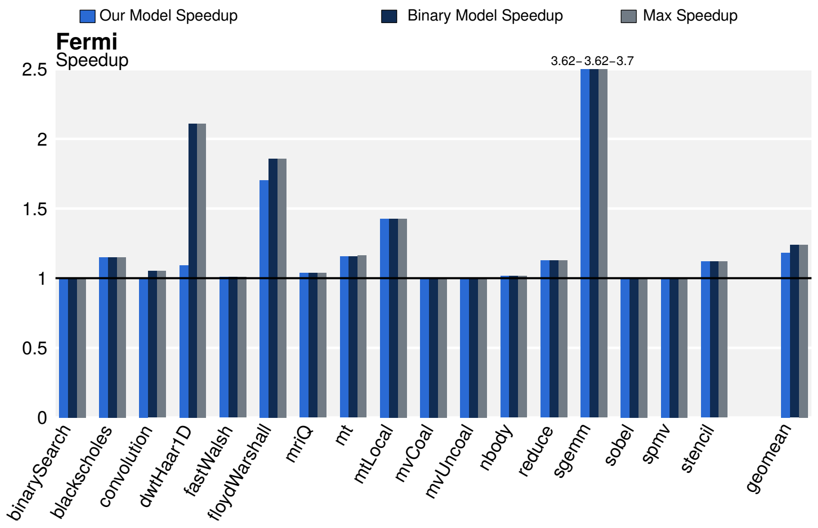
7.5 Stride parameter selection

This section describes the technique that we use to select the stride parameter.

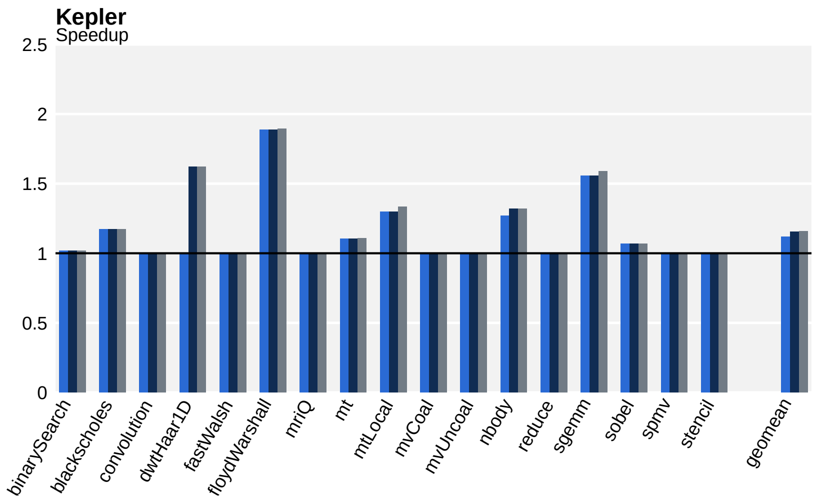
Our policy for the selection of the stride value relies on our understanding of the coalescing of memory access and the way that the stride parameter changes the shape of the code, see section 7.4.3. The core idea is that kernels showing coalesced accesses to memory will benefit from a large stride value (32), independently of the coarsening factor. Kernels that do not have coalesced accesses will either be insensitive to the stride or will benefit by a small stride (1). Our technique can be then summarized as follows. We symbolically execute the kernel with `SymEngine` to estimate which memory locations are touched by a kernel. Based on this information we compute the number of transactions performed by each memory instruction as we presented in section 4.6.2. If the kernel shows coalesced accesses, meaning the number of transactions, per load, is 1, select stride 32. If, instead, the kernel shows uncoalesced accesses, meaning the number of transactions per load cannot be statically determined by `SymEngine` (or it is high, close to 32), select stride 1. Even though `SymEngine` has been verified on Nvidia GPUs similar optimisation principles apply for AMD hardware. For this reasons we apply the same heuristic on AMD devices as well.

7.5.1 Performance Evaluation

Figures 7.9 and 7.10 show the performance results that we obtain applying our methodology to select the stride parameter. The three bars represent respectively: the speedup given by our heuristic, the maximum speedup given by restricting the choice to just 1 and 32 and the maximum speedup over all the search space. The baseline for this experiment is the performance of the application run with coarsening factor 1 and the optimal work-group size for each kernel. On Nvidia 7.9 we can see our method achieves the maximum performance on the majority of programs. The notable exception is `dwtHaar1D`. As we describe in section 4.6.3 we fail to analyse the store access pattern for this kernel. Our method in these cases is to default the stride parameter to 1. This choice prevents us from gaining most of the performance available for `dwtHaar1D`. Similar considerations apply for AMD devices (figure 7.10). Here, benchmarks

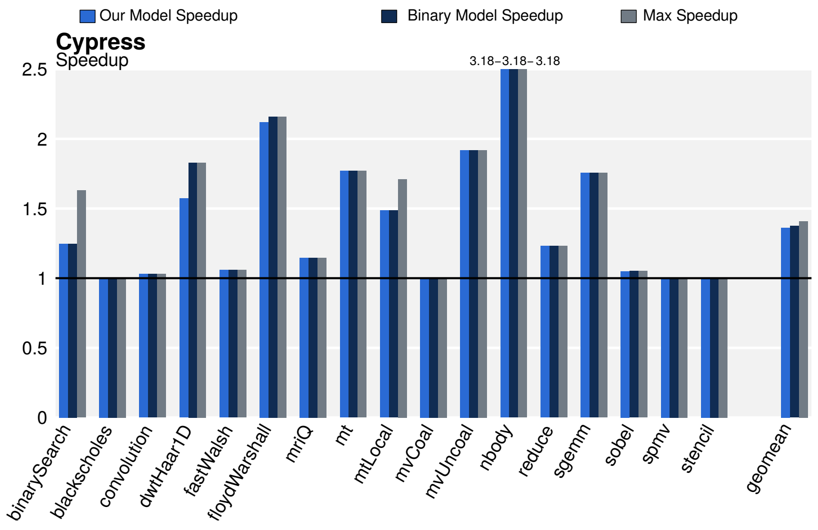


(a) *Fermi*

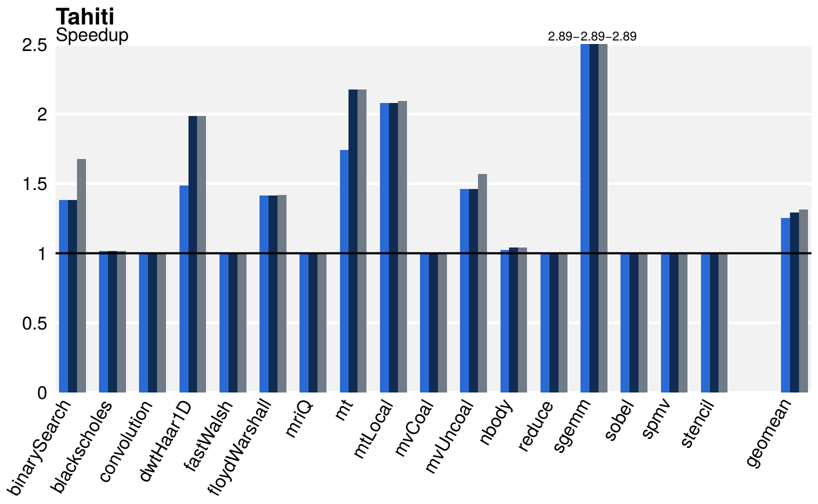


(b) *Kepler*

Figure 7.9: Barplots showing the performance of our method to predict the stride parameter for the Nvidia devices. The first bar shows the performance of our method. The second bar shows the maximum performance achievable restricting the choice of the optimal stride to 1 or 32. The third bar shows the maximum speedup attainable overall.



(a) *Cypress*



(b) *Tahiti*

Figure 7.10: Barplots showing the performance of our method to predict the stride parameter for the AMD devices. The first bar shows the performance of our method. The second bar shows the maximum performance achievable restricting the choice of the optimal stride to 1 or 32. The third bar shows the maximum speedup attainable overall.

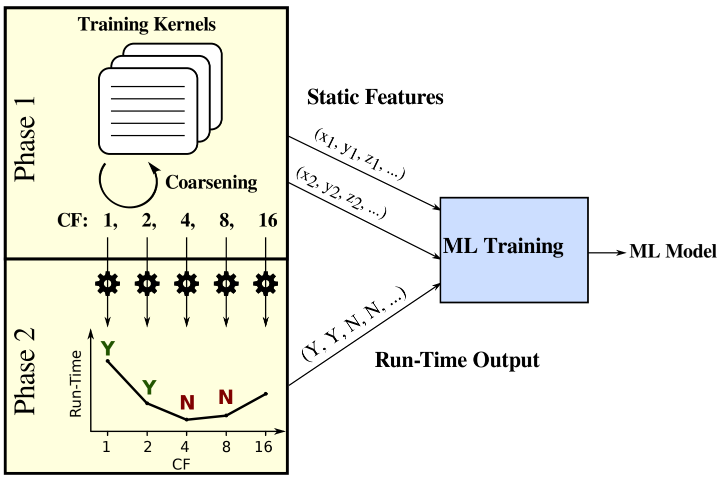


Figure 7.11: Model training overview. In Phase 1 we collect static features of the kernel functions. In Phase 2 we run the testing programs collecting the binary output variables.

like `binarySearch` and `mtLocal` have optimal stride factors different from 1 or 32 (identified by the fact that the second bar does not reach the maximum). In this cases our simplification prevents us from reaching the maximum performance. Overall our heuristics gives the following performance: 75.7%, 75.6%, 88.4% and 80.5% *Fermi*, *Kepler*, *Cypress* and *Tahiti* respectively.

7.6 Neural Network Model Description

7.6.1 Training

This section describes the training and the evaluation of our neural network model.

In this work we use Leave One Out Cross Validation for the evaluation of performance results. See section 2.7.2 for a description of this technique.

Figure 7.11 gives a visual representation of the training process. OpenCL kernels from the training set are compiled and run with the six different coarsening factors. During the compilation step (Phase 1) a compiler-analysis pass collects static features of each different version of the code. The full list of these features is given in section 7.6.3. In Phase 2 we run all the different versions of the program arranging the results by increasing coarsening factor. The binary training output variable is computed answering the following question: *Is further coarsening going to improve the performance of the current version?* Answering this question on each version of the kernel function will generate a tuple of binary values. This idea of reducing the classification problem to a binary decision has the advantage of simplifying the task and increase the amount of training data available, since now every coarsened version of the program can be used as training point. This technique is called called *data augmentation*, refer

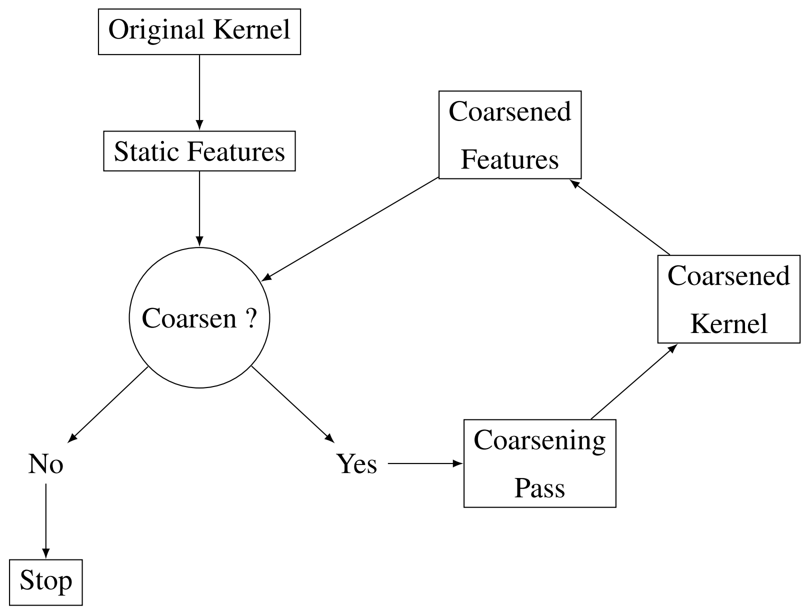


Figure 7.12: High-level view of the use of our neural network iterative model. Static source code features of the program are extracted from the original kernel code and fed into the model which decides whether to coarsen or not. If yes, new features are computed and the model is queried again. From a high level point of view the model simply computes the best coarsening factor from static program characteristics. At the end of the process the program is compiled using the predicted factor.

to [71] for another example. Then, for each kernel the 5 sets of features and the 5-tuple of binary variables are used for training the machine learning model. In the case of neural networks, training consists in updating the weights on the edges of the network in such a way that the error between the prediction and the expected result is minimized (see section 2.6.2) This process is repeated for each target architecture independently. We thus obtain four different models for the four GPUs we use.

7.6.2 Neural Network Model

Our model is built using a neural network for classification with one hidden layer. neural networks [19] are a supervised learning algorithm that classify data-points assigning to each of them the expected probability to belong to a given class. Given a previously unseen program the model determines whether coarsening should be applied or not. If the answer is yes, the model is applied again to the coarsened version of the program to determine if we should keep going or stop coarsening. In the latter case, the model is called a third time deciding if the program should be coarsened one more time and so on. Figure 7.12 gives an overview of how the model is applied. Such model relies on the peculiar shape of the coarsening performance curve, which shows improvement up until the optimal coarsening factor. Further coarsening leads to performance degradation. This common behaviour is shown for all the benchmarks in figures 5.4 and 5.5. Note that we actually compile the program multiple times for the extraction of static features for programs with intermediate coarsening factors.

We choose the iterative predictor to implement a conservative approach. By iteratively querying the model in multiple intermediate steps we make sure to limit the application of thread-coarsening only to those applications which truly benefit from it. This is a beneficial property, since large coarsening factors are rarely optimal (consider the Hinton diagrams in figure 7.3).

7.6.3 Program Features

This section describes the selection of the static code features used for prediction. The model is based exclusively on static characteristics of the target OpenCL kernel function. These characteristics, called features, are extracted using an analysis pass that works on the LLVM-IR. Note that, since our goal is to develop a *portable* optimiser, we don't use any architecture-specific feature. We apply our function analysis at the LLVM-IR level because this is the lowest level of abstraction that is portable across architectures. In particular, we do not collect any information after instruction scheduling or register allocation, since these are target-specific phases. We first describe the candidate features. This is followed by the process used to reduce these to a smaller and more useful subset.

Feature Name	Description
BasicBlocks	Number of Basic Blocks
Branches	Number of Branches
DivInsts	Number of Divergent Instructions
DivRegionInsts	Number of Instructions in Divergent Regions
DivRegionInstsRatio	Ratio between the Number of instructions inside Divergent Regions and the Total number of instructions
DivRegions	Number of Divergent Regions
TotInsts	Number of Instructions
FPInsts	Number of Floating point Instructions
ILP	Average ILP per Basic Block
Int/FP Inst Ratio	Ration between Integer and Floating Point Instructions
IntInsts	Number of Integer Instructions
MathFunctions	Number of Mathematical Builtin Functions
MLP	Average MLP per Basic Block
Loads	Number of Loads
Stores	Number of Stores
UniformLoads	Number of Loads that do not depend on the Coarsening Direction
Barriers	Number of Barriers

Table 7.3: Candidate static features used by the machine learning model.

7.6.4 Candidate Features

The full list of candidate features is given in table 7.3. We selected these based on the results of chapter 5, where we analysed which hardware performance counters are affected by the coarsening transformation. Our analysis based on regression trees identified the number of executed branches, memory utilization (in terms of memory instructions and cache utilization) and instruction level parallelism (ILP) as useful information to characterize thread-coarsening. We approximate these *dynamic* counters using *static* code characteristics counterparts, such as the total number of kernel instructions, static number of branches and divergent regions and static ILP. We employ 17 candidate static features.

Absolute Features Divergent control flow has a strong impact on performance for graphics processors because it forces the serialization of instructions that would normally be executed concurrently on the different lanes of a warp [102]. The degree of thread divergence in a kernel is measured using the results of *divergence analysis*. We count the total number of divergent

1) Delta instructions
2) Delta divergence
3) Arithmetic Intensity
4) Instructions
5) Divergence
6) Delta Arithmetic Intensity
7) Instruction Parallelism

Table 7.4: Final features selected after the application of Principal Component Analysis.

instructions (for an explanation of what divergent instructions are see section 4.2), the total number of divergent regions (CFG regions controlled by a divergent branch) and their relative size with respect to the overall number of instructions in the kernel. Another counter related to the complexity of the control flow of the kernel is the total number of blocks. We also compute the average per-block theoretical ILP and Memory Level Parallelism (MLP) at the LLVM-IR level. To compute the static ILP and MLP we followed the methodology presented in the work by Sim *et al.* [116]. For ILP, we schedule the instructions in each basic block as early as possible and we compute the depth of the schedule. The ratio between the total number of instructions in the block and the depth of the schedule is our value for ILP. MLP is defined as the average number of memory instructions between a load of a value and the use of that same value. It is used to estimate how many outstanding memory requests the program can sustain.

Relative Features In addition to the absolute values for the features we also employ the relative difference between two program versions: before and after the application of coarsening. The relative increase of a feature value after coarsening is computed according to the following formula: $\frac{(feature_{After} - feature_{Before})}{feature_{Before}}$, where $feature_{Before}$ and $feature_{After}$ are the values of the feature before and after coarsening. In the remainder of the chapter delta features are identified by *Delta* added to the feature name.

7.6.5 Feature Selection

This section describes how we select features starting from the candidate ones.

Principal Component Analysis The large number of features (absolute plus deltas) is automatically reduced using the standard technique of Principal Component Analysis.

It is important to notice that we use the same set of features in table 7.3 for all devices without manual intervention. Principal Components Analysis minimises the cross-correlation of the input features by linearly aggregating those features that are highly correlated and therefore redundant. After application of PCA the 7 final selected features in table 7.4 describe 95%

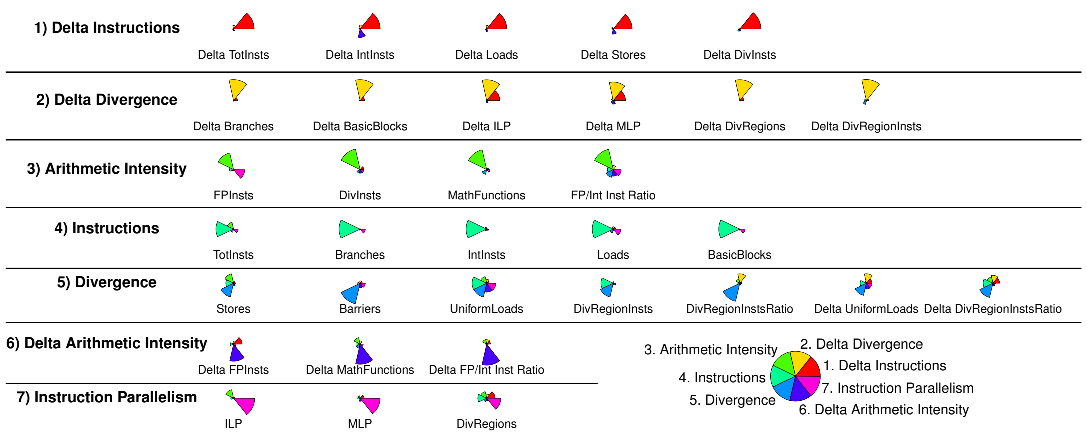


Figure 7.13: Result of the PCA analysis with Varimax rotation. The seven resulting components are labelled on the left and sorted according to their relative importance (the amount of variance of the feature space covered by each of them). On the right-end side each pie chart depicts the contribution of each original feature to a given output principal component. Features are laid out in rows according to the component they contribute to.

of the variance of the original space on average across the four devices.

With the goal of providing an understanding of the importance of each feature we applied to the results of naive PCA a space transformation called Varimax rotation [88] (see section 2.6.3). This transformation makes sure that each feature of the original space contributes either strongly or weakly to each new principal component. Such property makes it easier to interpret the output of PCA and allows us to label each principal component with a meaningful name based on the features that contribute to it. A similar approach to PCA has been followed by [65] for prediction of execution time of parallel applications in heterogeneous environments. See section 2.6.3 for a description of PCA analysis.

Figure 7.13 shows the result of the PCA with Varimax rotation when reducing the space from 34 candidate features to the 7 components. The new principal components are organized by rows and labelled on the left. Each pie chart shows which component a given original feature contributes to. Components are sorted by rows according to their relative importance: how much of the feature variance they cover. So the most important component is labelled *Delta Instructions*. This component groups together features describing by how much the number of instructions in the kernel body increases while coarsening.

The second most important component (*Delta Divergence*) describes the increase in the amount of divergence. It is interesting to notice that the two most important components describe the *difference* of characteristics in the feature space rather then their absolute value. Absolute features account for a smaller amount of variance of the feature space. Section 7.7.4 describes how two representative features of the first two components are related to the results of the coarsening transformation and how they are used for prediction. This section presents the evaluation of our machine learning predictor.

7.7 Model Evaluation

This section presents the performance results for our iterative neural network model. All the speedups reported here are relative to the execution time of the original application executed without applying the coarsening transformation, *i.e.*, coarsening factor 1.

7.7.1 Unique-Factor Model Description

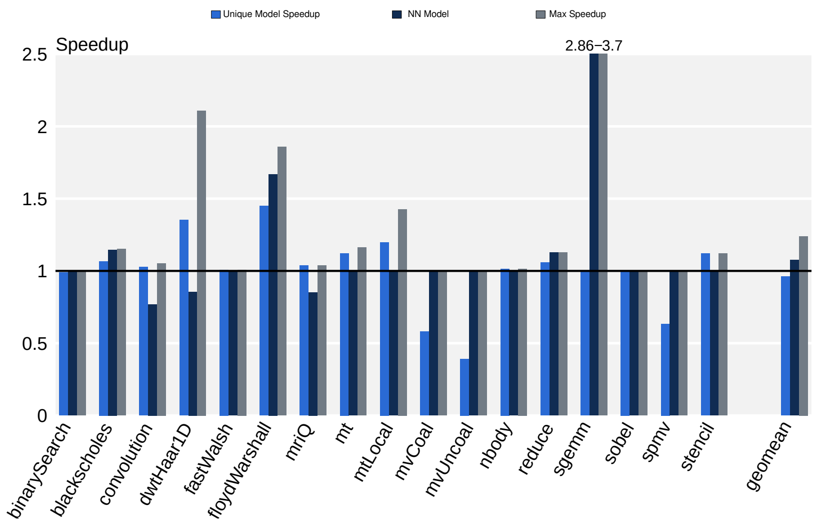
To offer a rigorous evaluation of our machine learning model we compare its performance against a simple heuristic. Given a set of training programs with the corresponding speedups for each coarsening factor we compute the single best factor on average for that device. The chosen value is then applied on the testing program (not present in the original training set). This optimisation policy makes the assumption (often wrong) that coarsened programs behave similarly to one-another and, therefore, similar optimisation parameters give comparable performance. We employed this simple average-based policy because of lack of similar previous work against which to compare our model. In the following sections we call this simple model *Unique Model*. Our machine learning based predictor is called *NN Model*.

7.7.2 Performance Evaluation

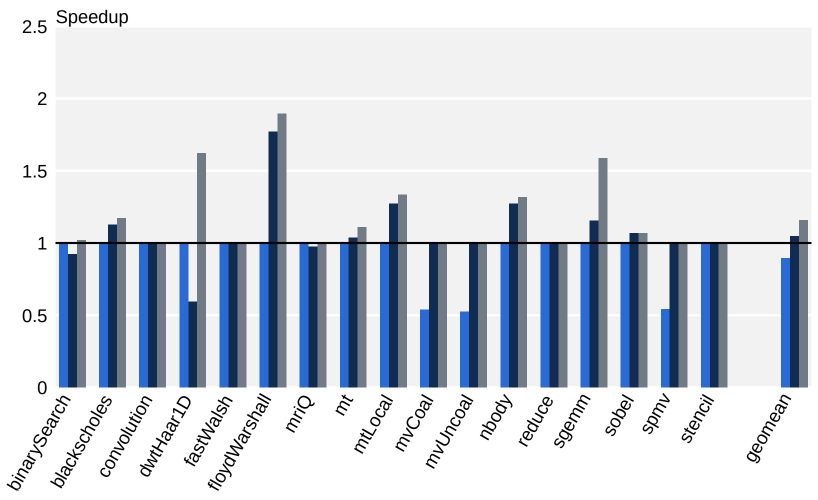
Figures 7.14 and 7.15 shows the results of our experiments for the four target devices. We report three bars: one for the performance of *Unique Model*, the second one for *NN Model* and the final one shows the maximum speedup attainable with coarsening.

Performance of *Unique Model* We first observe that *Unique Model* performs poorly on all the devices. On the two Nvidia it results in a slowdown on average, while on AMD its gives a marginal improvement over the baseline. This difference of *Unique Model* across the two vendors can be explained considering that on Nvidia the performance profile is quite irregular. Either a benchmark greatly benefit from coarsening, such as `sgemm` and `floydWarshall` or it does not at all, as in the case of `spmv` and `stencil`. Having these two types of effects makes it hard to estimate performance based on the average trend without considering the inherent properties of each benchmark. On AMD, *Unique Model* records on par with the baseline since coarsening gives on average higher improvements (1.53x and 1.54x) and a larger number of kernels benefit from it.

Divergent Kernels Applications such as `mvCoal`, `spmv` and `stencil` are penalized by coarsening on all the devices, *i.e.*, the best factor is 1. The reason for this is the divergent behaviour of

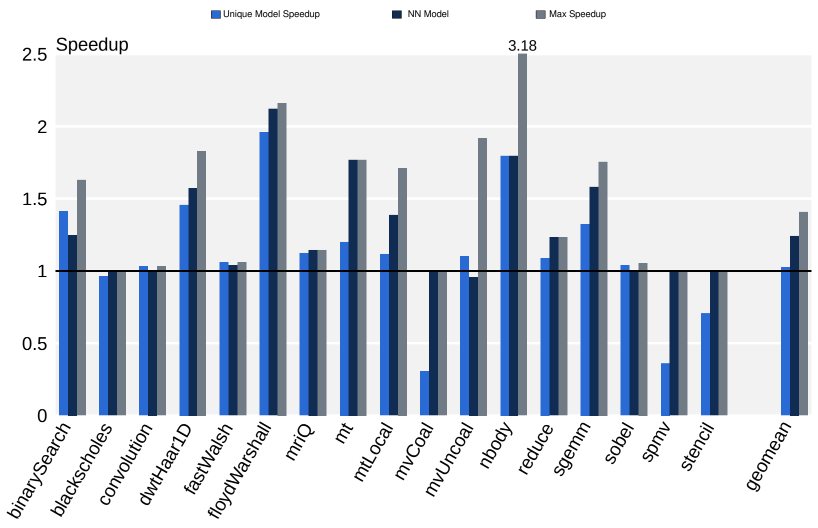


(a) *Fermi*

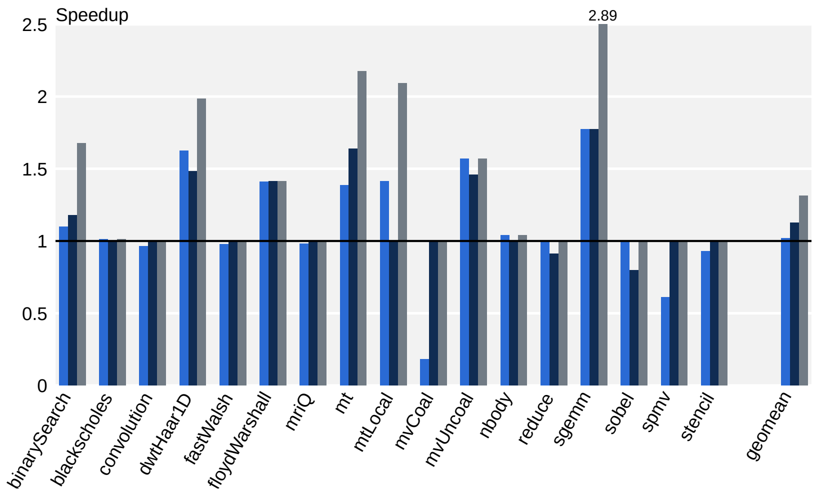


(b) *Kepler*

Figure 7.14: Barplots summarising the results. The first bar represents the speedup given by *Unique Model*. The second bar is the speedup given by the *NN Model*. The third bar is the maximum speedup attainable with thread-coarsening.



(a) *Cypress*



(b) *Tahiti*

Figure 7.15: Barplots summarising the results. The first bar represents the speedup given by *Unique Model*. The second bar is the speedup given by the *NN Model*. The third bar is the maximum speedup attainable with thread-coarsening.

these applications. These kernels are enclosed into divergent regions checking for the bounds of the iteration space. Such pattern is particular detrimental to the baseline performance on GPUs and coarsening makes this problem worse. *Unique Model* leads to significant slowdowns on these kernels since they deviate from the norm. The features *DivRegions* and *DivRegionInsts* model the degree of divergence in the kernel body. Using this information *NN Model* successfully predicts coarsening factor 1 for all the benchmarks for which *Unique Model* leads to slowdowns. This positive result is a consequence of the conservative design policy of our iterative model (section 7.6.2).

Uniform Kernels On the other hand *NN Model* enables coarsening effectively for benchmarks such as `sgemm` and `floydWarshall` on all architectures. These two applications show high speedups with coarsening. This is because they benefit from the reduction in redundant memory operations that coarsening ensures. Such characteristic is expressed in the feature *UniformLoads*. In chapter 5.7.1 we propose an explanation for this behaviour.

The two applications for which *NN Model* fails to achieve significant improvements are `mt` and `mtLocal`. On AMD in particular these two kernels benefit from coarsening but the model does not take advantage of the possible gain. The reason lies in the particular shape of the code of these two programs. They contain few instructions, just loads and stores. Thus coarsening leads to the replication of all the instructions in the function body. Consequently this makes the feature *Delta TotInst* and *Delta Divergent Insts* have high values. Usually this signifies that coarsening should be disabled.

The other outlier in our benchmarks is `dwtHaar1D` for which we loose performance without exploiting the large speedup available on Nvidia devices. The reason for this loss is a wrong modelling of the access pattern and a consequent miss prediction of the stride factor. We explain this in detail in section 7.5.1

In summary, *NN Model* gives an average performance improvement of 1.08x, 1.05x, 1.24x, 1.13x on *Fermi*, *Kepler*, *Cypress* and *Tahiti* respectively, which is 31%, 30%, 60% and 40% of the maximum performance available. The largest performance penalty (aside for `dwtHaar1D`) is for `sobel` running on *Tahiti* and it is about 20%. If we were able to model the access patter of the outlier `dwtHaar1D` correctly and consequently improve its performance we would be able to achieve 50%, 70%, 62% and 42% of the maximum on *Fermi*, *Kepler*, *Cypress* and *Tahiti* respectively.

7.7.3 Accuracy of prediction

Tables 7.5 and 7.6 show the predicted coarsening factor against the optimal one for all programs and devices. From this table we confirm the diversity of the best optimisation factor as described by the Hinton diagrams in figure 7.3. We also see that *NN Model* overestimates the

Kernel	**Fermi**		**Kepler**	
	Predicted	**Best**	**Predicted**	**Best**
`binarySearch`	1	1	4	16
`blackscholes`	4	8	2	4
`convolution`	4	4	1	1
`dwtHaar1D`	4	8	4	4
`fastWalsh`	1	2	1	1
`floydWarshall`	4	16	4	8
`mriQ`	8	4	4	2
`mt`	1	8	4	2
`mtLocal`	1	8	4	4
`mvCoal`	1	1	1	1
`mvUncoal`	1	1	1	1
`nbody`	1	2	2	2
`reduce`	4	4	1	1
`sgemm`	8	32	2	16
`sobel`	1	1	2	2
`spmv`	1	1	1	1
`stencil`	1	8	1	1

Table 7.5: The table reports the best coarsening factors compared against the ones predicted by *NN Model* for all programs on Nvidia Devices.

Kernel	Cypress		Tahiti	
	Predicted	Best	Predicted	Best
binarySearch	2	2	8	8
blackscholes	1	1	1	2
convolution	4	4	1	1
dwtHaar1D	4	4	2	4
fastWalsh	8	4	1	1
floydWarshall	4	4	4	16
mriQ	4	4	1	2
mt	4	32	4	32
mtLocal	4	32	1	32
mvCoal	1	1	1	1
mvUncoal	4	8	2	2
nbody	2	16	4	4
reduce	2	2	2	4
sgemm	4	8	2	16
sobel	1	4	8	4
spmv	1	1	1	1
stencil	1	1	1	1

Table 7.6: The table reports the best coarsening factors compared against the ones predicted by *NN Model* for all programs on AMD Devices.

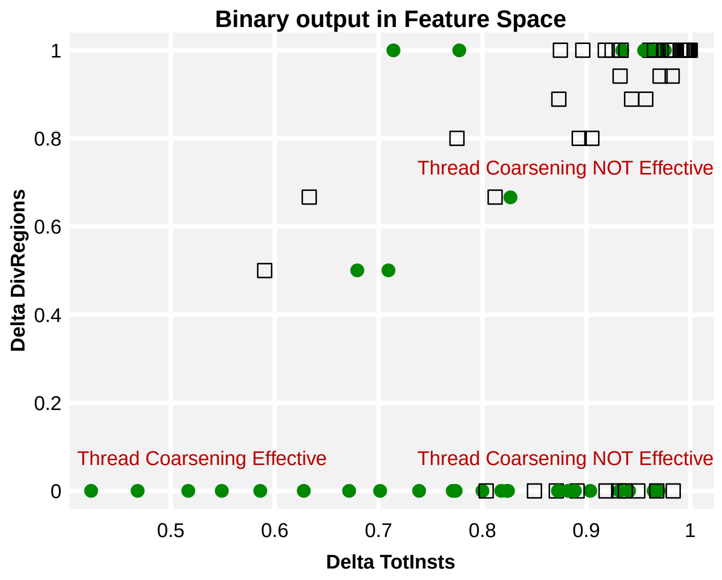


Figure 7.16: All our training data-points laid out according to two of the most relevant features: *Delta TotInsts* and *Delta DivRegions*. A dot identifies that coarsening is successful, a square that is not. The red labels highlights different regions of the feature space.

factor only in 4 cases out of 68, and always predicts *no coarsening* (*i.e.*, factor 1) when the optimal is one.

7.7.4 Neural Network Model Analysis

This section provides an understanding of how the *NN Model* predicts the coarsening factor relying on the distribution of the training points in the feature space. Figure 7.16 shows all the points in our data set, *i.e.*, all kernels for all the coarsening factors, arranged in the two dimensional space of *Delta TotInsts* and *Delta DivRegions* for AMD *Cypress*. The plot shows a dot for each benchmark for which coarsening is effective and a square when it is not, these are our training data. From this plot we clearly see that *Delta DivRegions* (on the y-axis) splits the data points in two clusters: one for which its value is 0 and one for which its value is positive. For the first cluster other features will discriminate these points in a higher-dimensional space. In the first cluster fall kernels such as `mt`, `sgemm` or `mriQ` which do not have any divergent region. The second cluster instead contains `mvCoal`, `stencil`, `spmv`, kernels whose body is enclosed by divergent branches. The feature on the x-axis instead represent the overall difference in instructions in the kernel body before and after coarsening.

From the distribution of dots and squares we notice the following trends. (1) Configurations at the bottom of the graph (with low *Delta DivRegions*) tend to benefit from coarsening. Not having divergent regions in the original code means that coarsening is likely to be effective. The opposite also holds, since configurations in the top half of the plot usually do not benefit from the transformation. (2) Configurations on the right-hand side of the plot are likely not to benefit from coarsening. A sharp increase in the instruction number signifies a high level of divergence leading to poor performance.

7.8 Summary

This chapter addressed the problem of developing a methodology to optimise the thread-coarsening transformation. We first presented an evaluation of the complexity of the problem by characterising the optimisation space. We showed that the optimal choice for the coarsening factor is hard to reach and device-specific. By relying on our previous work on performance characterization and analysis of the coarsening transformation we select relevant code features that characterize the effects that the coarsening transformation has on run-time. We first introduced a technique to select the stride parameter using symbolic kernel execution with `SymEngine`. We then developed a machine learning model that can predict a coarsening factor that improves performance on four GPU architectures.

Next chapter concludes the thesis by summarising the results and by providing a critical analysis of all the chapters.

Chapter 8

Conclusion

This thesis presented the design and implementation of the thread-coarsening compiler transformation along with techniques to analyse its effect on performance and predict effective parameter configurations. Chapter 4 presented the implementation details of the compiler passes that we developed in this work. Chapter 5 presented the effects of coarsening on run-time performance. We introduced a technique based on regression trees to automatically identify which counters are affected by the coarsening transformation and to study their relationship with run-time performance. Chapter 6 studied how the performance of coarsened kernels changes across problem input sizes. We showed the existence of a hardware saturation point and how to exploit it for fast iterative compilation. Chapter 7 introduced techniques to select the coarsening factor and the stride.

8.1 Contributions of the Work

This section summarizes the contributions of the four technical chapters of the thesis.

8.1.1 Compiler Analyses and Transformations

Chapter 4 presented the design choices and the implementation details of the thread-coarsening compiler pass and the `SymEngine` kernel analyser. As a first contribution we described all the stages of the coarsening transformation. This is the first implementation of coarsening to be implemented in LLVM and available as open-source. Previous works in the field relied on source-to-source compilers or manual application of the transformation. We showed that by employing divergence analysis it is possible to coarsen only divergent instructions, thus mitigating the overhead of redundant (uniform) computations. The second contribution is the introduction of `SymEngine`, a symbolic execution engine for the analysis of memory access patterns or OpenCL kernels. We showed that through symbolic execution it is possible to accurately predict the number of hardware transactions made by a kernel on an Nvidia GPU.

8.1.2 Performance Counters Analysis

In Chapter 5 we provided an evaluation of the coarsening transformation for 17 OpenCL programs and 4 GPUs, two Nvidia and two AMD. We showed that the same program might behave differently across devices making the task of compiler tuning for heterogeneous environments challenging. With the goal of understanding how coarsening impacts performance we analyse the changes of hardware profiler counters varying the coarsening factor. The results of profiling are analysed using regression trees that relate changes in the values of counters to execution-time speedups. We demonstrated how regression trees can effectively identify the root cause of performance differences for multiple coarsening configurations.

8.1.3 Fast Tuning of the Coarsening Transformation

Chapter 6 presented an evaluation of the coarsening optimisation space across input sizes. First, we analysed the absolute performance achieved by an OpenCL kernel using the throughput metric (number of output data elements computed per unit of time). We experimentally prove that the throughput is low for small problem input sizes and increases for large ones. The throughput reaches a plateau for problem sizes larger than a threshold that we call minimum saturation point (MSP). Along with raw performance analysis, we evaluated the coarsening search space for each of the input sizes, recording the best configuration for each. We then showed that the optimal configuration for the MSP performs well for much larger problem sizes. Based on this observation, we introduce a technique to improve the performance of iterative compilation. We explore the coarsening space for the input size corresponding to the MSP and we port the best configuration found to the much larger target size. With this method we improve performance over the benchmark baseline and we speedup up the search by 10x.

8.1.4 Selection of Coarsening Parameters

Our last technical Chapter, number 7, tackled the problem of selecting the coarsening factor and the stride parameters. We first gave a characterization of the optimisation space: we showed that fast configurations are rare and change across kernels and across devices. We then provided experimental data describing how the coarsening factor and the stride can be predicted independently. Our heuristics for the determination of the stride parameter uses `SymEngine` to compute the number of hardware transactions. If the original kernel shows coalesced memory accesses (the number of transactions per memory instruction is minimized) we select a large stride to preserve this favourable property. The determination of a suitable coarsening factor is done using a neural network for classification. The model distinguishes between kernels that are suitable for coarsening and kernels that are not. By iteratively querying the model we determine when it is not profitable any more to coarsen the kernel. We our combined technique gives 31%, 30%, 60% and 40% of the maximum performance available on *Fermi*, *Kepler*, *Cypress* and *Tahiti*, improving over a simpler optimisation policy that selects a unique value for the stride and coarsening factor for all the programs.

8.2 Critical Analysis

This section critically discusses the work in this thesis.

8.2.1 Compiler Analyses and Transformations

In our implementation for the coarsening transformation we leveraged the LLVM compiler infrastructure, working on LLVM intermediate representation. This has posed the problem of generating code for a variety of GPU architectures. We found the solution using the `axtor` OpenCL backend: it generates OpenCL-C kernels from LLVM-IR modules. This introduces an extra step in the compilation pipeline which might not be tolerable in a production environment. In our evaluation results we factor out this extra step by using as baseline the performance of the coarsened kernel after it has been transformed by `axtor`. In section 8.3 we propose future extension to overcome this problem.

8.2.2 Performance Counters Analysis

The primal objective of our regression based technique is to be portable across devices, so to provide a single methodology capable of explaining performance on a wide range of vendors. This meant that we chose to settle for a solution less accurate then hand-tuned models for specific GPUs. The insights provided by our methodology as still useful and hard to obtain without detailed knowledge of the device hardware. Another important point to consider about our regression trees is that they rely on the performance counters available on each device. We showed that Nvidia GPU SDK provide a detailed profiler with low-level information in particular about the memory system. The same is not true for AMD where the available counters are much fewer and much more coarse grained. These are inherent limitations for performance analysis. An important cost associated with performance analysis is the execution time of profile runs. Our methodology requires multiple profile runs for each coarsening factor for all the devices under study. Depending on the accuracy of the profilers this process might take a considerable amount of time. We judge this cost to be affordable since our technique is meant for compiler engineers. These users can afford to spend more time in understanding program performance than the end-users of the compiler.

8.2.3 Fast Tuning of the Coarsening Transformation

In chapter 6 we analysed how the application throughput saturates when increasing the problem input size. We also showed examples of applications deviating from this norm. For these outliers our policy of MSP tuning does not achieve the same results as for the other applications. Ideally, these outliers should be identified in advance. Take as reference the `mvCoal` kernel. This program cannot reach saturation due to the large amount of data used per thread. One way of detecting in advance this type of outliers is to compute the proportion between the number of threads running concurrently and the amount of data touched by each thread as a function of the problem size.

8.2.4 Selection of Coarsening Parameters

The main limitation of our prediction methodology is the small number of benchmarks available for training and performance evaluation. We worked on a small set of programs since OpenCL benchmarks usually target a specific architecture applying code optimisations and we manually selected programs that where easy to port to new devices. We can see the effects of this limitation in section 7.7 where miss predicting the stride factor of `dwtHaar1D` leads to a significant overall performance loss. Using a small set of programs forced us to use leave one out cross-validation, meaning testing the model on one benchmark at the time an training on the others. If more benchmarks were available, we could have applied K-fold cross-validation, using multiple programs for testing. Section 8.3 proposes a way to overcome this limitation.

8.3 Future Work

Improve the compilation toolchain As we explained in section 8.2.1 the introduction of the `axtor` OpenCL backend introduced a further step in the compilation toolchain. This can be avoided lowering the transformed LLVM-IR to a portable cross-platform representation like SPIR-V [69]. This new version of the SPIR language has been designed for both Vulkan (graphics) and OpenCL (compute) workloads and therefore likely to find widespread application.

Input size analysis A possible way to improve our proposal to speedup iterative compilation is a quicker identification of the minimum saturation point, without the exploration of all the input sizes of interest. This could be done evaluating the throughput of the application for small input sizes only, and then use regression to obtain an estimate of the curves in figures 6.2 and 6.3.

Better characterization of GPU programs The accuracy of trained models depends on the capability of the program features to characterize their run-time behaviour and to distinguish different benchmarks. Static features might fall short of this task. A possible solution for this problem would be to run the LLVM-IR version of the body of the OpenCL kernel function using the LLVM interpreter engine [108]. A modified version of the execution engine would collect dynamic counts for LLVM instructions. This would enable the use of dynamic and yet platform agnostic program information.

Automatic benchmark generation As we have seen in this work, the process of compiler tuning is highly dependent on the quality and the amount of programs used to select the optimisation parameters. Benchmarks must represent realistic workloads and, at the same time, cover the optimisation space grasping all the nuances of modern GPUs. Just relying on standard benchmark does not fully solve the problem since they might be tuned for a specific device or thought for a specific case of use. To solve this problem we propose to automatically synthesize training programs. This methodology would generate a set of random OpenCL-C programs inspired by standard OpenCL benchmarks and fit a model to predict their performance on a target device. Based on the accuracy of the machine learning model our infrastructure would generate another set of benchmarks. The process can be repeated until a predefined level of prediction accuracy is reached. This would help avoid the problem of over-fitting on a small problem sample size.

8.4 Summary

This chapter concludes the thesis. We summarized the contributions of the work, we provided a critical analysis of the work showing its limitations and presented directions for future research.

Appendix A

This appendix lists the hardware performance counters from our four GPUs. Tables A.1 and A.2 refer to Nvidia *Fermi*, tables A.3 and A.4 refer to Nvidia *Kepler*, table A.5 refers to AMD *Cypress* and table A.6 refers to AMD *Tahiti*.

Counter Name	**Description**
regperworkitem	Hardware registers used per work-item
inst_issued	Instructions issued
inst_executed	Instructions executed
l2_subp0_read_sector_queries	Queries from L1 going to slice 0 of L2
l2_subp1_read_sector_queries	Queries from L1 going to slice 1 of L2
l1_global_load_hit	Global memory load hits in L1
l1_global_load_miss	Global memory load misses in L1
l1_local_load_hit	Hits in L1 coming from local memory requests (in Nvidia terminology local memory is where registers are spilled)
l1_local_load_miss	Misses in L1 coming from local memory requests
l1_shared_bank_conflict	Bank conflicts in shared memory
l1_local_store_hit	Store hits in local memory
l1_local_store_miss	Store misses in local memory
l2_subp0_read_sector_misses	Read misses in sector 0 of L2
l2_subp1_read_sector_misses	Read misses in sector 1 of L2
l2_subp0_write_sector_queries	Write requests in sector 0 of L2
l2_subp1_write_sector_queries	Write requests in sector 1 of L2
l2_subp0_write_sector_misses	Write misses in sector 0 of L2
l2_subp1_write_sector_misses	Write misses in sector 1 of L2

Table A.1: Raw performance counters collected for *Fermi* (Part1)

Counter Name	**Description**
l2_subp0_read_hit_sectors	Read hits in sector 0 of L2
l2_subp1_read_hit_sectors	Read hits in sector 1 of L2
gld_inst_8bit	8 bit global load instructions
gld_inst_16bit	16 bit global load instructions
gld_inst_32bit	32 bit global load instructions
gld_inst_64bit	64 bit global load instructions
gld_inst_128bit	128 bit global load instructions
gst_inst_8bit	8 bit global load instructions
gst_inst_16bit	16 bit global load instructions
gst_inst_32bit	32 bit global load instructions
gst_inst_64bit	64 bit global load instructions
gst_inst_128bit	128 bit global load instructions
local_load	Loads from local memory (see note above about local memory)
local_store	Stores to local memory
fb_subp1_read_sectors	Read requests to partition 0 of DRAM
fb_subp0_read_sectors	Read requests to partition 1 of DRAM
warps_launched	Total warps launched
threads_launched	Total work-items launched
active_cycles	Number of cycles in which at least one warp is active
active_warps	Accumulated number of warps active per cycle in a multiprocessor. At each cycle this value is incremented by the number of active warps
branch	Branches
divergent_branch	Divergent Branches (which result is not uniform across the work-items)
fb_subp0_write_sectors	Write requests to partition 0 of DRAM
fb_subp1_write_sectors	Write requests to partition 1 or DRAM
global_store_transaction	Transactions to store data
gld_request	Load instructions per warp
gst_request	Store instructions per warp
shared_load	Loads from shared memory
shared_store	Stores to shared memory
threads_launched	Threads launched during profiling

Table A.2: Raw performance counters collected for *Fermi* (Part 2)

Counter Name	Description
regperworkitem	Hardware registers used per work-item
inst_executed	Instructions issued
branch	Branches
divergent_branch	Divergent Branches (which result is not uniform across the work-items)
l1_global_load_hit	Global memory load hits in L1
l1_global_load_miss	Global memory load misses in L1
l1_local_load_hit	Hits in L1 coming from local memory requests (in Nvidia terminology local memory is where registers are spilled)
l1_local_load_miss	Misses in L1 coming from local memory requests
l2_subp0_write_l1_sector_queries	Write requests in sector 0 of L2 coming from L1
l2_subp1_write_l1_sector_queries	Write requests in sector 1 of L2 coming from L1
l2_subp2_write_l1_sector_queries	Write requests in sector 2 of L2 coming from L1
l2_subp3_write_l1_sector_queries	Write requests in sector 3 of L2 coming from L1
l2_subp0_read_l1_sector_queries	Read requests in sector 0 of L2 coming from L1
l2_subp1_read_l1_sector_queries	Read requests in sector 1 of L2 coming from L1
l2_subp2_read_l1_sector_queries	Read requests in sector 2 of L2 coming from L1
l2_subp3_read_l1_sector_queries	Read requests in sector 3 of L2 coming from L1
l2_subp0_read_l1_hit_sectors	Hits in sector 0 of L2 coming from L1
l2_subp1_read_l1_hit_sectors	Hits in sector 1 of L2 coming from L1
l2_subp2_read_l1_hit_sectors	Hits in sector 2 of L2 coming from L1
l2_subp3_read_l1_hit_sectors	Hits in sector 3 of L2 coming from L1
l2_subp0_write_sector_misses	Write misses in sector 0 of L2
l2_subp1_write_sector_misses	Write misses in sector 1 of L2
l2_subp2_write_sector_misses	Write misses in sector 2 of L2
l2_subp3_write_sector_misses	Write misses in sector 3 of L2
l2_subp0_read_sector_misses	Read misses in sector 0 of L2
l2_subp1_read_sector_misses	Read misses in sector 1 of L2
l2_subp2_read_sector_misses	Read misses in sector 2 of L2
l2_subp3_read_sector_misses	Read misses in sector 3 of L2

Table A.3: Raw performance counters collected for *Kepler* (Part 1).

Counter Name	Description
l1_local_store_hit	Store hits in local memory
l1_local_store_miss	Store misses in local memory
gld_request	Load instructions per warp
gst_request	Store instructions per warp
local_load	Loads from local memory
local_store	Stores to local memory
fb_subp0_read_sectors	Read requests to partition 1 or DRAM
fb_subp1_read_sectors	Read requests to partition 1 or DRAM
fb_subp0_write_sectors	Write requests to partition 0 or DRAM
fb_subp1_write_sectors	Write requests to partition 1 or DRAM
ihared_load	Loads from shared memory
shared_store	Stores to shared memory
shared_load_replay	Re-executions of loads due to bank conflics in shared memory
shared_store_replay	Re-executions of stores due to bank conflics in shared memory
warps_launched	Total warps launched
threads_launched	Total work-items launched
active_cycles	Number of cycles in which at least one warp is active
active_warps	Accumulated number of warps active per cycle in a multiprocessor. At each cycle this value is incremented by the number of active warps.
thread_inst_executed	Instructions executed
global_store_transaction	Transactions to store data
threads_launched	Threads launched during profiling

Table A.4: Raw performance counters collected for *Kepler* (Part 2).

Counter Name	**Description**
VGPRs	Vector general purpose registers used per work-item
SGPRs	Scalar general purpose registers used per work-item
FCStacks	Control-flow stacks used by the kernel
Wavefronts	Wavefront executed
ALUInsts	Average number of ALU instructions per work-item
FetchInsts	Average number of global memory loads per work-item
WriteInsts	Average number of global memory stores per work-item
LDSFetchInsts	Average number of local memory loads per work-item
LDSWriteInsts	Average number of local memory stores per work-item
ALUBusy	Percentage of time ALU instructions are processed
ALUFetchRatio	Ratio between number of load and ALU instructions
ALUPacking	Efficiency (in percentage) of the packing of ALU instructions in VLIW processing units.
FetchSize	Amount of data fetched from global memory
CacheHit	Percentage of loads that hit in cache
FetchUnitBusy	Percentage of time the load unit is acitve
FetchUnitStalled	Percentage of time the load unit is stalled
WriteUnitStalled	Percentage of time the store unit is stalled
FastPath	Amount of data written to global memory through the fast path
CompletePath	Amount of data written to global memory through the complete path
PathUtilization	Percentage of data written through complete or fast path over the toal amount of data written to global memory
LDSBankConflict	Percentage of time local memory is stalled due to bank conflicts

Table A.5: Raw performance counters collected for *Cypress*

Counter Name	Description
VGPRs	Vector general purpose registers used per work-item
SGPRs	Scalar general purpose registers used per work-item
FCStacks	Control-flow stacks used by the kernel
Wavefronts	Wavefront executed
VALUInsts	Average number of vector ALU instructions per work-item
SALUInsts	Average number of scalar ALU instructions per work-item
VFetchInsts	Average number of global memory vector loads per work-item
SFetchInsts	Average number of global memory scalar loads per work-item
VWriteInsts	Average number of global memory vector stores per work-item
LDSInsts	Average number of local memory loads and stores per work-item
VALUUtilization	Percentage of active vector ALU instructions executed per work-item
VALUBusy	Percentage of time vector ALU instructions are processed
SALUBusy	Percentage of time scalar ALU instructions are processed
FetchSize	Amount of data fetched from global memory
CacheHit	Percentage of loads that hit in cache
MemUnitBusy	Percentage of time the memory unit is acitve
MemUnitStalled	Percentage of time the memory unit is stalled
WriteUnitStalled	Percentage of time the write unit is stalled
LDSBankConflict	Percentage of time local memory is stalled due to bank conflicts
WriteSize	Total amount of data written to video memory

Table A.6: Raw performance counters collected for the *Tahiti*

Bibliography

[1] NVIDIA Corporation, NVIDIA Profiler
http://docs.nvidia.com/cuda/profiler-users-guide/, 2011. (Cited on pages 31, 33, and 74.)

[2] Parboil Benchmark Suite. http://impact.crhc.illinois.edu/parboil.php, 2011. (Cited on page 34.)

[3] The OpenACC Application Programming Interface, v2.0, 2013. (Cited on page 34.)

[4] Nvidia Corporation, Parallel Thread Execution ISA
http://docs.nvidia.com/cuda/pdf/ptx_isa_4.2.pdf, 2015. (Cited on page 14.)

[5] M. Abrash. *Michael Abrash's Graphics Programming Black Book*. 1997. (Cited on page 35.)

[6] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M.F.P. O'Boyle, J. Thomson, M. Toussaint, and C.K.I. Williams. Using Machine Learning to Focus Iterative Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2006. (Cited on page 39.)

[7] AMD Inc. AMD Graphics Cores Next (GCN) Architecture, June 2012. (Cited on page 15.)

[8] AMD Inc. AMD Graphics Cores Next (GCN) Architecture Whitepaper
https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf, 2012. (Cited on page 16.)

[9] AMD Inc. AMD Accelerated Parallel Processing OpenCL Programming Guide, 2013. (Cited on pages 15, 64, and 105.)

[10] AMD Inc. AMD APP Profiler
http://developer.amd.com/tools/heterogeneous-computing/
amd-app-profiler/, 2013. (Cited on page 74.)

[11] O. Bachmann, P.S. Wang, and E.V Zima. Chains of Recurrences - A Method to Expedite the Evaluation of Closed-Form Functions. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation (ISSAC)*, 1994. (Cited on page 18.)

[12] D.F. Bacon, S.L. Graham, and O.J. Sharp. Compiler Transformations for High-performance Computing. In *ACM Computing Surveys*, 1994. (Cited on page 35.)

[13] H. Bae, D. Mustafa, J. Lee, Aurangzeb, H. Lin, C. Dave, R. Eigenmann, and S.P. Midkiff. The Cetus Source-to-Source Compiler Infrastructure: Overview and Evaluation. In *International Journal of Parallel Programming (IJPP)*, 2013. (Cited on pages 37 and 54.)

[14] S.S. Baghsorkhi, M. Delahaye, S.J. Patel, W.D. Gropp, and W.W. Hwu. An Adaptive Performance Modeling Tool for GPU Architectures. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2010. (Cited on pages 31 and 60.)

[15] S.S. Baghsorkhi, I. Gelado, M. Delahaye, and W.W. Hwu. Efficient Performance Evaluation of Memory Hierarchy for Highly Multithreaded Graphics Processors. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2012. (Cited on page 31.)

[16] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009. (Cited on page 32.)

[17] K. Beyls and E.H. DHollander. Reuse Distance as a Metric for Cache Behavior. In *Proceedings of IASTED International Conference on. Parallel and Distributed Computing and Systems (PDCS)*, 2001. (Cited on page 33.)

[18] Johnie Birch. Using the Chains of Recurrences Algebra for Data Dependence Testing and Induction Variable Substitution. In *Ms Thesis, Florida State University*, 2002. (Cited on page 18.)

[19] C.M. Bishop. *Neural Networks for Pattern Recognition*. 1995. (Cited on pages 22 and 120.)

[20] C.M. Bishop. *Pattern Recognition and Machine Learning*. 2006. (Cited on page 23.)

[21] F. Bodin, T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle, and E. Rohou. Iterative Compilation in a Non-Linear Optimisation Space. In *In Proceedings of the Workshop on Profile and Feedback-Directed Compilation*, 1998. (Cited on page 39.)

[22] R. Bordawekar, U. Bondhugula, and R. Rao. Believe It or Not!: Mult-core CPUs Can Match GPU Performance for a FLOP-intensive Application! In *Proceedings of the International Conference on Parallel Architectures and Compilation (PACT)*, 2010. (Cited on page 30.)

[23] M. Boyer, K. Skadron, and W. Weimer. Automated dynamic analysis of CUDA programs. In *Third Workshop on Software Tools for MultiCore Systems*, 2008. (Cited on page 33.)

[24] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *Proceedings of ACM SIGGRAPH*, 2004. (Cited on page 29.)

[25] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. Evidence-based Static Branch Prediction using Machine Learning. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1997. (Cited on page 40.)

[26] D. Callahan, J. Cocke, and K. Kennedy. Estimating Interlock and Improving Balance for Pipelined Architectures. In *Journal of Parallel and Distributed Computing*, 1988. (Cited on page 35.)

[27] J.H Clark. The Geometry Engine: A VLSI Geometry System for Graphics. In *Proceedings of ACM SIGGRAPH*, 1982. (Cited on page 28.)

[28] B. Coutinho, D. Sampaio, F.M.Q. Pereira, and W. Meira. Divergence Analysis and Optimizations. In *Proceedings of the International Conference on Parallel Architectures and Compilation (PACT)*, 2011. (Cited on page 38.)

[29] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1991. (Cited on page 17.)

[30] M. Deering, S. Winner, B. Schediwy, C. Duffy, and N. Hunt. The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics. In *Proceedings of ACM SIGGRAPH*, 1988. (Cited on page 28.)

[31] E. Demers. Evolution of AMD graphics. Presented at AMD Fusion Developer Summit, 2011. (Cited on page 15.)

[32] G. Diamos, B. Ashbaugh, S. Maiyuran, A. Kerr, H. Wu, and S. Yalamanchili. SIMD Re-convergence at Thread Frontiers. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2011. (Cited on page 80.)

[33] G.F. Diamos, A.R. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A Dynamic Optimization Framework for Bulk-synchronous Applications in Heterogeneous Systems. In *Proceedings of the International Conference on Parallel Architectures and Compilation (PACT)*, 2010. (Cited on pages 32 and 38.)

[34] R. Dolbeau, F. Bodin, and G.C. de Verdiere. One OpenCL to Rule Them All? In *Proceedings of the International Workshop on Multi-/Many-core Computing Systems (MuCoCoS)*, 2013. (Cited on page 34.)

[35] J.J. Dongarra and A.R. Hinds. Unrolling Loops in FORTRAN. In *Software – Practice and Experience*, 1979. (Cited on page 35.)

[36] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M.F.P. O'Boyle, and O. Temam. Fast Compiler Optimisation Evaluation Using Code-Feature Based Performance Prediction. In *Proceedings of International Conference on Computing Frontiers (CF)*, 2007. (Cited on page 40.)

[37] C. Dubach, T.M. Jones, E.V. Bonilla, G. Fursin, and M.F.P. O'Boyle. Portable Compiler Optimisation Across Embedded Programs and Microarchitectures Using Machine Learning. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2009. (Cited on page 40.)

[38] C. Dubach, T.M. Jones, and M.F.P. Boyle. Exploring and Predicting the Architecture/Optimising Compiler Co-Design Space. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2008. (Cited on page 32.)

[39] J.R. Ellis. Bulldog: A Compiler for Vliw Architectures. In *PhD Thesis, Yale University*, 1985. (Cited on pages 35 and 81.)

[40] N. Farooqui, K. Schwan, and S. Yalamanchili. Efficient Instrumentation of GPGPU Applications Using Information Flow Analysis and Symbolic Execution. In *Proceedings of the Workshop on General Purpose Processing Using GPUs (GPGPU)*, 2014. (Cited on page 38.)

[41] N. Fauzia, L. Pouchet, and P. Sadayappan. Characterizing and Enhancing Global Memory Data Coalescing on GPUs. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2015. (Cited on page 33.)

[42] Paul Feautrier. Dataflow Analysis of Array and Scalar References. 1991. (Cited on page 36.)

[43] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The Program Dependence Graph and Its Use in Optimization. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1987. (Cited on pages 31 and 59.)

[44] H. Fuchs and J. Poulton. Pixel-Planes: a VLSI-Oriented Design for a Raster Graphics Engine. In *Proceedings of VLSI Design*, 1981. (Cited on page 28.)

[45] W.W.L. Fung, I. Sham, G. Yuan, and T.M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2007. (Cited on page 32.)

[46] D. Grewe, Z. Wang, and M.F.P. O'Boyle. Portable Mapping of Data Parallel Programs to OpenCL for Heterogeneous Systems. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2013. (Cited on page 34.)

[47] S.L. Grimm and J.G. Stadel. The GENGA Code: Gravitational Encounters in N-body Simulations with GPU Acceleration. In *The Astrophysical Journal, Volume 796, Issue 1, article id. 23, 16 pp.*, 2014. (Cited on page 30.)

[48] T. Grosser. *Enabling Polyhedral Optimizations in LLVM*. 2011. (Cited on page 18.)

[49] T.D. Han and T.S. Abdelrahman. Reducing Branch Divergence in GPU Programs. In *Proceedings of the Workshop on General Purpose Processing Using GPUs (GPGPU)*, 2011. (Cited on page 36.)

[50] T.D. Han and T.S. Abdelrahman. Reducing Divergence in GPGPU Programs with Loop Merging. In *Proceedings of the Workshop on General Purpose Processing Using GPUs (GPGPU)*, 2013. (Cited on page 36.)

[51] M.J. Harris. Fast Fluid Dynamics Simulation on the GPU. In *GPU Gems, pp. 637665. Addison-Wesley*, 2004. (Cited on page 30.)

[52] M.J. Harris. Optimizing Parallel Reduction in CUDA. 2007. (Cited on page 30.)

[53] S. Hong and H. Kim. An Analytical Model for a GPU Architecture with Memory-Level and Thread-Level Parallelism Awareness. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2009. (Cited on pages 31 and 71.)

[54] S. Hong and H. Kim. An Integrated GPU Power and Performance Model. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2010. (Cited on pages 31 and 71.)

[55] HSA Foundation. HSA Programmer's Reference Manual: HSAIL Virtual ISA and Programming Model, 2013. (Cited on page 54.)

[56] AMD Inc. ATI CTM Guide. Technical Reference Manual, 2006. (Cited on page 2.)

[57] E. Ipek, S.A. McKee, R. Caruana, B.R. de Supinski, and M. Schulz. Efficiently Exploring Architectural Design Spaces via Predictive Modeling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006. (Cited on page 32.)

[58] T.B. Jablin, P. Prabhu, J.A. Jablin, N.P. Johnson, S.R. Beard, and D.I. August. Automatic CPU-GPU Communication Management and Optimization. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2011. (Cited on page 2.)

[59] W. Jia, K.A. Shaw, and M. Martonosi. Stargazer: Automated Regression-Based GPU Design Space Exploration. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2012. (Cited on page 32.)

[60] W. Jia, K.A. Shaw, and M. Martonosi. Starchart: Hardware and Software Optimization Using Recursive Partitioning Regression Trees. Proceedings of the International Conference on Parallel Architectures and Compilation (PACT), 2013. (Cited on page 32.)

[61] R. Karrenberg and S. Hack. Whole-Function Vectorization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2011. (Cited on pages 33 and 38.)

[62] R. Karrenberg and S. Hack. Improving Performance of OpenCL on CPUs. In *International Conference on Compiler Construction (CC)*, 2012. (Cited on page 38.)

[63] R Karrenberg, M. Kosta, and T. Sturm. Presburger Arithmetic in Memory Access Optimization for Data-Parallel Languages. In *Proceedings of the International Symposium on Frontiers of Combining Systems (FroCoS)*, 2013. (Cited on page 33.)

[64] A. Kerr, E. Anger, G. Hendry, and S. Yalamanchili. Eiger: A Framework for the Automated Synthesis of Statistical Performance Models. In *Workshop on Performance Engineering and Applications (WPEA)*, 2012. (Cited on page 33.)

[65] A. Kerr, G. Diamos, and S. Yalamanchili. Modeling GPU-CPU Workloads and Systems. In *Proceedings of the Workshop on General Purpose Processing Using GPUs (GPGPU)*, 2010. (Cited on pages 32 and 123.)

[66] Khronos Group. The SPIR Specification, Standard Portable Intermediate Representation, Version 1.2, 2014. (Cited on page 54.)

[67] Khronos Group. OpenCL: The Open Standard for Parallel Programming of Heterogeneous Systems. `http://www.khronos.org/opencl/`, 2015. (Cited on pages 2, 6, and 34.)

[68] Khronos Group. OpenGL: The Industry's Foundation for High Performance Graphics `http://www.khronos.org/opencl`, 2015. (Cited on page 28.)

[69] Khronos Group. SPIR-V Specification (Provisional), 2015. (Cited on page 135.)

[70] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi. Evaluating Performance and Portability of OpenCL Programs. In *Proceedings of the International Workshop on Automatic Performance Tuning (WAPT)*, 2010. (Cited on page 34.)

[71] A. Krizhevsky, S. Ilya, and G.E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems (NIPS)*, 2012. (Cited on pages 30 and 120.)

[72] M.D. Lam, E.E. Rothberg, and M.E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1991. (Cited on page 35.)

[73] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2004. (Cited on page 17.)

[74] H. Leather, E. Bonilla, and M.F.P. O'Boyle. Automatic Feature Generation for Machine Learning Based Optimizing Compilation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2009. (Cited on page 40.)

[75] B.C. Lee and D.M. Brooks. Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006. (Cited on page 32.)

[76] B.C. Lee, D.M. Brooks, B.R. de Supinski, M. Schulz, K. Singh, and S.A. McKee. Methods of Inference and Learning for Performance Modeling of Parallel Applications. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2007. (Cited on page 41.)

[77] V.W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A.D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and

GPU. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2010. (Cited on page 30.)

[78] Y. Lee, V. Grover, R. Krashinsky, Stephenson M., S.W. Keckler, and K. Asanovic. Exploring the Design Space of SPMD Divergence Management on Data-Parallel Architectures. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2014. (Cited on page 39.)

[79] Y. Lee, R. Krashinsky, V. Grover, S.W. Keckler, and K. Asanovic. Convergence and Scalarization for Data-Parallel Architectures. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2013. (Cited on page 38.)

[80] E. Lindholm, M.J. Kilgard, and H. Moreton. A User-programmable Vertex Engine. Proceedings of ACM SIGGRAPH, 2001. (Cited on page 28.)

[81] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. In *IEEE Micro*, 2008. (Cited on page 29.)

[82] Y. Liu, E.Z. Zhang, and X. Shen. A Cross-Input Adaptive Framework for GPU Program Optimizations. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2009. (Cited on pages 37 and 68.)

[83] A. Magni. Thread Coarsening Compiler Pass Source Code. `http://github.com/HariSeldon/coarsening_pass`, 2014. (Cited on pages 3, 42, and 44.)

[84] A. Magni. SymEngine Source Code. `http://github.com/HariSeldon/symengine`, 2015. (Cited on page 3.)

[85] A. Magni, C. Dubach, and M.F.P. O'Boyle. A Large-Scale Cross-Architecture Evaluation of Thread-Coarsening. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2013. (Cited on page 3.)

[86] A. Magni, C. Dubach, and M.F.P. O'Boyle. Automatic Optimization of Thread-Coarsening for Graphics Processors. In *Proceedings of the International Conference on Parallel Architectures and Compilation (PACT)*, 2014. (Cited on page 4.)

[87] A. Magni, C. Dubach, and M.F.P. O'Boyle. Exploiting GPU Hardware Saturation for Fast Compiler Optimization. In *Proceedings of the Workshop on General Purpose Processing Using GPUs (GPGPU)*, 2014. (Cited on page 4.)

[88] B.F.J. Manly. *Multivariate Statistical Methods: A Primer, Third Edition*. 2004. (Cited on pages 24, 33, and 123.)

[89] W.R. Mark, R.S. Glanville, K. Akeley, and M.J. Kilgard. Cg: A System for Programming Graphics Hardware in a C-like Language. In *Proceedings of ACM SIGGRAPH*, 2003. (Cited on page 29.)

[90] D.G. Merrill, M. Garland, and A.S. Grimshaw. Scalable GPU Graph Traversal. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2012. (Cited on page 30.)

[91] D.G. Merrill and A.S. Grimshaw. Parallel Scan for Stream Architectures. In *Technical Report CS2009-14, Department of Computer Science, University of Virginia*, 2009. (Cited on page 30.)

[92] D.G. Merrill and A.S. Grimshaw. Revisiting Sorting for GPGPU Stream Architectures. In *Proceedings of the International Conference on Parallel Architectures and Compilation (PACT)*, 2010. (Cited on page 30.)

[93] Microsoft Corporation. Direct3D `https://msdn.microsoft.com/library/windows/apps/hh452744`, 2015. (Cited on page 28.)

[94] Mike Houston. Anatomy of AMDs TeraScale Graphics Engine, 2008. (Cited on page 14.)

[95] S. Moll. *Decompilation of LLVM IR*. 2011. (Cited on page 54.)

[96] A. Monsifrot, F. Bodin, and R. Quiniou. A Machine Learning Approach to Automatic Production of Compiler Heuristics. In *Artificial Intelligence: Methodology, Systems, Applications (AIMSA)*, 2002. (Cited on page 40.)

[97] S.K. Murthy. Automatic Construction of Decision Trees from Data: A Multi-Disciplinary Survey. *Data Mining and Knowledge Discovery*, 1997. (Cited on page 21.)

[98] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. In *Proceedings of ACM Queue*, 2008. (Cited on pages 2 and 29.)

[99] C. Nugteren, G.-J. van den Braak, H. Corporaal, and H. Bal. A Detailed GPU Cache Model Based on Reuse Distance Theory. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2014. (Cited on pages 32 and 33.)

[100] Nvidia Corp. Nvidia's Next Generation CUDA Compute Architecture: Fermi `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`, 2009. (Cited on pages 11, 13, and 65.)

[101] Nvidia Corp. Nvidia's Next Generation CUDA Compute Architecture: Kepler `http://www.nvidia.com/content/PDF/kepler/`

`NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`, 2012. (Cited on pages 65 and 80.)

[102] Nvidia Corp. CUDA Toolkit Documentation `http://docs.nvidia.com/cuda/`, 2015. (Cited on page 121.)

[103] J.D ”Owens, D. Luebke, N. Naga Govindaraju, M. Harris, J. Kruger, A. Lefohn, and T.J.” Purcell. A Survey of General-Purpose Computation on Graphics Hardware. In *Proceedings of the Computer Graphics Forum*, 2007. (Cited on page 29.)

[104] Z. Pan and R. Eigenmann. Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2006. (Cited on page 39.)

[105] S.G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich. OptiX: A General Purpose Ray Tracing Engine. In *Proceedings of ACM SIGGRAPH*, 2010. (Cited on page 30.)

[106] S.J. Pennycook, S.D. Hammond, S.A Wright, J.A Herdman, I. Miller, and S.A Jarvis. An Investigation of the Performance Portability of OpenCL. In *Journal of Parallel and Distributed Computing*, 2013. (Cited on page 34.)

[107] B.T. Phong. Illumination for Computer Generated Pictures. In *Proceedings of the Communication of the ACM*, 1975. (Cited on page 28.)

[108] J. Price and McIntosh-Smith S. *Oclgrind: An OpenCL SPIR Interpreter and OpenCL Device Simulator*. 2014. (Cited on page 136.)

[109] B.D. Ripley. *Pattern Recognition and Neural Networks*. 1996. (Cited on pages 21 and 71.)

[110] B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Global Value Numbers and Redundant Computations. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, 1988. (Cited on page 17.)

[111] S. Rul, H. Vandierendonck, J. D’Haene, and K. De Bosschere. An Experimental Study on Performance Portability of OpenCL Kernels. In *Proceedings of the Symposium on Application Accelerators in High-Performance Computing (SAAHPC)*, 2010. (Cited on page 34.)

[112] S. Ryoo, C.I. Rodrigues, S.S. Baghsorkhi, S.S. Stone, D.B. Kirk, and W.W. Hwu. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2008. (Cited on page 36.)

[113] S. Ryoo, C.I. Rodrigues, S.S Stone, S.S Baghsorkhi, S. Ueng, J..A. Stratton, and W.W. Hwu. Program Optimization Space Pruning for a Multithreaded GPU. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2008. (Cited on page 36.)

[114] M. Samadi, A. Hormati, M. Mehrara, J. Lee, and S. Mahlke. Adaptive Input-Aware Compilation for Graphics Engines. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2012. (Cited on page 37.)

[115] R.N. Sanchez, J.N. Amaral, D. Szafron, M. Pirvu, and M. Stoodley. Using Machines to Learn Method-Specific Compilation Strategies. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2011. (Cited on page 41.)

[116] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc. A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2012. (Cited on pages 31, 71, and 122.)

[117] M. Stephenson and S. Amarasinghe. Predicting Unroll Factors Using Supervised Classification. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2005. (Cited on page 40.)

[118] M. Stephenson, S. Amarasinghe, M. Martin, and U. O'Reilly. Meta Optimization: Improving Compiler Heuristics with Machine Learning. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2003. (Cited on page 40.)

[119] K. Stock, L. Pouchet, and P. Sadayappan. Using Machine Learning to Improve Automatic Vectorization. In *Transactions on Architecture and Code Optimization (TACO)*, 2012. (Cited on page 40.)

[120] R.W. Swanson and L.J. Thayer. A Fast Shaded-polygon Renderer. In *Proceedings of ACM SIGGRAPH*, 1986. (Cited on page 28.)

[121] Guangming Tan, Linchuan Li, Sean Triechle, Everett Phillips, Yungang Bao, and Ninghui Sun. Fast Implementation of DGEMM on Fermi GPU. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2011. (Cited on page 30.)

[122] I. Tanasic, L. Vilanova, M. Jordà, J. Cabezas, I. Gelado, N. Navarro, and W. Hwu. Comparison Based Sorting for Systems with Multiple GPUs. In *Proceedings of the Workshop on General Purpose Processing Using GPUs (GPGPU)*, 2013. (Cited on page 30.)

[123] T. Tang, X. Yang, and Y. Lin. Cache Miss Analysis for GPU Programs Based on Stack Distance Profile. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, 2011. (Cited on page 33.)

[124] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. Multi2Sim: A Simulation Framework for CPU-GPU Computing. In *Proceedings of the International Conference on Parallel Architectures and Compilation (PACT)*, 2012. (Cited on page 32.)

[125] S. Unkule, C. Shaltz, and A. Qasem. Automatic Restructuring of GPU Kernels for Exploiting Inter-thread Data Locality. In *International Conference on Compiler Construction (CC)*, 2012. (Cited on pages 3 and 38.)

[126] R.A. Van Engelen. Symbolic Evaluation of Chains of Recurrences for Loop Optimization. Technical report, 2000. (Cited on page 18.)

[127] R.A. Van Engelen. Efficient Symbolic Analysis for Optimizing Compilers. In *International Conference on Compiler Construction (CC)*, 2001. (Cited on page 18.)

[128] V. Volkov. Better Performance at Lower Occupancy, 2010. (Cited on page 37.)

[129] V. Volkov and J. Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2008. (Cited on pages 37 and 92.)

[130] R. Vuduc, J.W. Demmel, and J.A. Bilmes. Statistical Models for Empirical Search-Based Performance Tuning. 2004. (Cited on page 39.)

[131] Z. Wang and M.F.P. O'Boyle. Partitioning Streaming Parallelism for Multi-cores: A Machine Learning Based Approach. Proceedings of the International Conference on Parallel Architectures and Compilation (PACT), 2010. (Cited on page 41.)

[132] M. Weiser. Program Slicing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 1981. (Cited on page 46.)

[133] M. Wolf and M. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 1991. (Cited on page 35.)

[134] H. Wu, G. Diamos, T. Sheard, M. Aref, S. Baxter, M. Garland, and S. Yalamanchili. Red Fox: An Execution Environment for Relational Query Processing on GPUs. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2014. (Cited on page 30.)

[135] H. Wu, G. Diamos, J. Wang, S. Li, and S. Yalamanchili. Characterization and Transformation of Unstructured Control Flow in Bulk Synchronous GPU Applications. 2012. (Cited on page 80.)

[136] P. Xiang, Y. Yang, M. Mantor, N. Rubin, L.R. Hsu, and H. Zhou. Exploiting Uniform Vector Instructions for GPGPU Performance, Energy Efficiency, and Opportunistic Reliability Enhancement. In *Proceedings of the International Conference on Supercomputing (ICS)*, 2013. (Cited on page 39.)

[137] S. Yan, C. Li, Y. Zhang, and H. Zhou. yaSpMV: Yet Another SpMV Framework on GPUs. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2014. (Cited on page 30.)

[138] Y. Yang, P. Xiang, J. Kong, M. Mantor, and H. Zhou. A Unified Optimizing Compiler Framework for Different GPGPU Architectures. In *Transactions on Architecture and Code Optimization (TACO)*, 2012. (Cited on pages 2, 3, 37, and 54.)

[139] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU Compiler for Memory Optimization and Parallelism Management. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2010. (Cited on page 37.)

[140] Yaun Lin. Building GPU Compilers with libNVVM, 2013. (Cited on page 54.)

[141] X. Ye, D. Fan, W. Lin, N. Yuan, and P. Ienne. High Performance Comparison-Based Sorting Algorithm on Many-Core GPUs. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2010. (Cited on page 30.)

[142] E.Z. Zhang, Y. Jiang, Z. Guo, and X. Shen. Streamlining GPU Applications on the Fly: Thread Divergence Elimination Through Runtime Thread-data Remapping. In *Proceedings of the International Conference on Supercomputing (ICS)*, 2010. (Cited on page 36.)

[143] E.Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly Elimination of Dynamic Irregularities for GPU Computing. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011. (Cited on page 37.)

[144] Y. Zhang and J.D. Owens. A Quantitative Performance Analysis Model for GPU Architectures. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2011. (Cited on page 31.)

[145] E.V. Zima. On Computational Properties of Chains of Recurrences. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation (ISSAC)*, 2001. (Cited on page 18.)