Improving the Programmability of GPU Architectures

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de rector magnificus prof.dr.ir. C.J. van Duijn, voor een commissie aangewezen door het College voor Promoties, in het openbaar te verdedigen op woensdag 30 april 2014 om 16:00 uur

door

Cedric Nugteren

geboren te Dordrecht

Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

voorzitter:	prof.dr.ir. A.C.P.M. Backx
1^e promotor:	prof.dr. H. Corporaal
2^e promotor:	prof.dr.ir. H.E. Bal (Vrije Universiteit Amsterdam)
leden:	prof.dr. P.H.J. Kelly (Imperial College London)
	dr. A. Cohen (École Polytechnique)
	dr.ir. A.L. Varbanescu (Universiteit van Amsterdam)
	prof.dr.ir. G. de Haan
	prof.dr. J.J. Lukkien

PhD thesis

Improving the Programmability of GPU Architectures

point.one

This work was supported by the Dutch government in their Point-One research program within the Morpheus project PNE101003 and carried out at the TU/e.

Advanced School for Computing and Imaging

This work was carried out in the ASCI graduate school.
ASCI dissertation series number 295.

HiPEAC COMPILATION ARCHITECTURE

This PhD-trajectory included a 4-month HiPEAC sponsored internship at ARM.

Printing: Printservice Technische Universiteit Eindhoven

A catalogue record is available from the Eindhoven University of Technology Library. ISBN: 978-90-386-3599-6

Table of contents

Preface

In front of you lies the pinnacle of a PhD-student's hard work: the thesis. However, hidden from the reader's eyes is the path that has led to this result. How was the subject chosen? What was left out? What were the difficult parts? Which part of the process could be improved? Prospective and current PhD-students are encouraged to read this preface, as well as any others interested in the process that led to this thesis.

In retrospect, the subject of this work was chosen in 2008 by browsing through the first CUDA documents for my master thesis work. Because CUDA and accelerating scientific workloads on GPUs was new at the time, mapping application X on GPU Y was still considered a scientific contribution. I therefore spent the first months of my PhD accelerating a histogram computation on a GPU, which has led to a publication at the GPGPU workshop. The results of this work encouraged me to investigate whether the GPU architecture could be improved. But how to do any experiments without a simulator or model at hand? The lack of a simulator forced me to search for a different research question, eventually leading to the question "how can a compiler improve the programmability of GPU architectures?".

My advisors pointed me to the work of Wouter Caarls, describing skeleton-based compilation for embedded systems. Without a compiler-background and without substantial literature research (*don't try this at home*), I started the development of my own skeleton-based compiler, focussing on GPUs instead. Although this would eventually lead to BONES (the compiler presented in chapter 4 of this thesis), a more thorough background study would have saved time and work (including my 'deprecated' SAMOS and GPGPU papers). To increase the scientific value of my compiler work, and motivated by the lack of structure in skeleton-classes, I started developing a program code classification. This classification is now known as algorithmic species, presented in chapter 3. This direction of research was further motivated by a 1-day visit to Imperial College London and by the engineering effort involved to further improve the BONES compiler.

As happened with many topics I have worked on during my PhD-thesis, I moved BONES and algorithmic species quickly to the background after I was encouraged to pursue new ideas. Especially after my 4-month internship with ARM, I was motivated to work on other topics: creating a detailed cache model of a GPU and addressing the problem of locality-aware thread scheduling. The results of these topics have led to chapter 5, a loosely connected part of this thesis.

This illustrates the curvy road that was taken to arrive at this final result. Consequently, as is the case with many PhD-theses, this work was mostly based on publications in the last year of the PhD. Because of the variety of topics investigated (from compilers to architecture to performance modelling), several publications had to be left out. For example, this thesis does not include my roofline model-based work nor the earlier mentioned work on histogram acceleration. Overall, I am satisfied with the broad scope of topics I was able to work on during my time as a PhD-student and the result it has led to. I hope that this thesis will be a valuable read for anyone interested in GPUs, compilers, processor architecture and performance modelling.

Name-tags collected at conferences, workshops and symposia.

"Imagine a banana. Or anything curved. Actually, don't, cause it's not curved or like a banana. Forget the banana!"

- The Doctor describing conceptual space (Doctor Who, 2011)

Chapter 1
Introduction

Digital technology has become a core part of today's daily life. For example, think of the ubiquity of personal devices such as smart-phones, e-readers, digital cameras and televisions. As another example, consider the pervasiveness of deeply embedded technology, such as electronics in cars, medical equipment and home appliances. Or, from another perspective, think of our daily use of services such as online encyclopaedias, social networks, digital music and on-line shopping. At the heart of all these devices and applications is the microprocessor, a digital system's central processing unit (CPU). In 2014, the amount of microprocessors has already superseded the world population, and is still growing: 2.6 billion ARM-based microprocessors were shipped in the first quarter of 2013 alone[1].

Users of microprocessor-based digital technology have become increasingly aware of the power and energy usage of technology. As an example, consider battery-powered devices such as smart-phones and tablets. When browsing shops for a new smart-phone or tablet, the expected battery life is typically included in a side-by-side comparison. Users even limit the use of such devices to preserve battery life. On the other side of the power and energy spectrum is the energy usage in data-centres (hosting our *cloud* services) and supercomputers. Such systems draw several megawatts of power, requiring specialised energy infrastructures and cooling solutions. The energy cost for such systems has already become the dominant operating expense, and is expected to grow further [60].

The power and energy issues of modern digital technology have a strong relationship with the power and energy issues of the microprocessors themselves. Throughout history, power and energy have not always been as important as they

[1]Source: *ARM Holdings PLC results Q1 2013.*

are now. Since the first microprocessors in the early 1970s, design has mostly focused on performance improvements. Only recently have power and energy come into play, rapidly gaining terrain over performance: since the early 2000s, microprocessor design is driven by *energy efficiency* rather than performance [61].

The shift from performance to energy efficiency as the main metric of microprocessor design is directly related to Moore's law and the end of Dennard scaling. Dennard's scaling theory [49] showed that by reducing the dimensions and electrical characteristics of transistors (a chip's building blocks), proportional gains in terms of density, speed and energy efficiency are enabled. This is known as *process scaling*. For the first 30 years of microprocessor design, process scaling has resulted in a doubling of the amount of transistors (Moore's law [103]) and an increase of 40% in switching frequency every 18 months, while maintaining constant power consumption. Because of physical limitations, Dennard' scaling theory has since 2004 failed to sustain the predicted exponential gains in frequency and energy efficiency. Thus, although the amount of transistors is still growing, increasing performance is not a simple matter of increasing the frequency anymore. More importantly, an increased amount of transistors now also consumes an increased amount of power, explaining the need for energy efficient designs.

The ever increasing transistor count together with the limited operating frequency has led to the design of *multi-core* processors. Rather than increasing the frequency and designing larger microprocessors, the additional transistors given by Moore's law are used to duplicate the existing microprocessor *core*. This has led to commodity multi-core processors with 2, 4, or 8 cores on a single chip. Recently, specialised *many-core* processors with 256 or even more cores per chip were introduced. On the other hand, the energy efficiency issues have led to the design of specialised microprocessors, capable of operating efficiently only within a limited *domain*. Such designs have been created traditionally for the *embedded systems* community, but have gained increasing momentum in the *general purpose* world [61]. Examples of specialised architectures are digital signal processors (DSPs), very-large instruction word (VLIW) processors, application-specific instruction-set processors (ASIPs), and graphics processing units (GPUs).

GPUs have rapidly gained terrain since 2008 as energy efficient microprocessors applicable to a broad range of problems. However, developing efficient GPU programs is not straightforward. The goal of this thesis is therefore to improve the *programmability* of GPUs. This section will introduce the GPU and the problem statement in detail, and will give an overview of the contributions of this thesis.

1.1 Using GPUs as accelerators

Graphics processing units (GPUs) are an example of specialised microprocessors or '*accelerators*': they are designed specifically to accelerate the rendering of 2D and 3D-graphics. The GPU's primary market is the game industry, enabling games to render increasingly complex 3D-scenes every year. Although the first 3D-graphics accelerators were introduced in the 1990s, the term 'GPU' was not used until NVIDIA's introduction of the *GeForce 256* in 1999 [95]. This was the first GPU to include a hardware *transform and lighting* (T&L) engine, setting the trend for GPUs to take over almost all of the graphics work from the system's main microprocessor (the CPU). The introduction of the *GeForce 3* in 2001 marks the next step in the evolution: GPUs were from then on able to run small custom programs known as *shaders* on a 3D scene's vertices and pixels. Eventually, starting from NVIDIA's *GeForce 8 series* in 2006, the logic executing these *vertex shaders* and *pixel shaders* was unified [95]. This resulted in the GPU's architecture as we know it today: many small programmable units executing small programs (the shaders) independently to alter vertex or pixel data[2].

Along with the unification of the two types of shaders, NVIDIA introduced the CUDA framework [95]. With this framework, the GPU's shaders can be programmed with non-graphics programs. This allows GPUs to accelerate other types of applications, so-called *compute* applications or GPGPU[3] workloads. Because of the specialised, domain-specific architecture of the GPU, many applications will perform poorly. Still, a significant fraction of the most demanding parts of applications can benefit from GPUs [66]. Example domains include linear algebra (e.g. matrix-multiplication, Cholesky decomposition) [71], image processing (e.g. edge detection, histogram equalisation) [101], computational biology (e.g. genomics, neuroscience) [141], statistics (cluster analysis, support vector machines) [142], and physics (e.g. fluid dynamics, turbulence simulation) [30]. These domains all have a high resemblance to graphics processing: they perform a large amount of mostly independent and similar small tasks on large data-structures. In other words, they have a high data-level parallelism.

Now, several years after the introduction of CUDA, using GPUs for compute applications is widespread. It has even become a market to consider for GPU designers, as more recent architectures are no longer designed with graphics as a single application in mind. For example, NVIDIA's latest GPUs (codenamed *Fermi* and *Kepler*) incorporate special features only useful for compute applications, such as scratchpad memories and dynamic parallelism [110]. Even low-power mobile GPUs such as the ARM Mali have recently been re-designed to support compute applications, demonstrating the importance of GPU-compute.

1.2 Key aspects of the GPU architecture

The focus of this thesis lies on the programmability of GPUs for compute applications. This topic is introduced in this section by discussing the key aspects of

[2]ATI/AMD has followed NVIDIA's evolution of T&L, shaders and a unified architecture.
[3]Although misleading, GPGPU is an acronym for general purpose computing on GPUs.

the GPU architecture[4], i.e. the main architectural features [65, 95] that make the GPU a specialised architecture: 1) the fine-grained multi-threading, 2) the single-instruction multiple-thread (SIMT) execution model, 3) the abundance of parallelism, and 4) the high bandwidth to memory.

Fine-grained multi-threading

Programs executing on microprocessors consist of a sequence of *instructions* to perform. Some instructions have a short *latency* (e.g. an addition) and some a long latency (e.g. a memory access). In case there are *dependences* between instructions, these latencies directly affect the total execution time of a program in a basic microprocessor architecture. Therefore, most advanced CPUs are designed to reduce the latency of instructions, for example through techniques such as instruction level parallelism (ILP), branch prediction, out-of-order execution, or the use of cache memories.

Because of highly parallel target workloads, a GPU provides a radically different solution: after issuing a (long or short latency) instruction, it switches to another stream of instructions. These streams are named *threads*; continuously switching between them results in *fine-grained multi-threading*. To maintain performance while switching threads, GPUs are equipped with a large *register-file* that stores the context of many GPU threads, enabling context-switching at no cost. Fine-grained multi-threading simplifies a core significantly (saving chip area and power), because techniques as used in CPUs to reduce the instruction latency are no longer necessary (assuming a sufficient number of independent threads).

Single-instruction multiple-thread execution model

A significant fraction of a microprocessor's chip area and power are spent merely to support the execution of instructions. Even very basic microprocessors have overheads such as fetching and decoding instructions, and controlling the execution *pipeline*. To increase the ratio of chip area and power spent on the computations themselves, microprocessors can be equipped with multiple *processing elements*. This allows a single instruction to execute multiple times in parallel on different data, creating a single-instruction multiple-data (SIMD) architecture. For example, the addition of two vectors of 8 elements can be performed on a 4-wide SIMD architecture with 2 vector-add instructions as opposed to 8 scalar-adds.

Similar to SIMD architectures, a GPU is also equipped with multiple processing elements. However, they are programmed in a single-instruction multiple-thread (SIMT) fashion: threads are used instead of special vector instructions. Because the majority of the pipeline is still only able to handle a single instruction, threads executing on different processing elements at the same time must perform the same instruction. Therefore, threads are grouped into units of scheduling that

[4]The introduction discusses an NVIDIA/AMD-like desktop GPU architecture. Other GPUs might not feature all the discussed aspects.

are executed in lock-step: *warps* in NVIDIA terminology or *wavefronts* in AMD terminology. The SIMT model can lead to underutilisation of the processing elements when threads in a warp encounter control flow divergence, for example when an if-statement is only taken by a subset of the threads.

Parallelism

Since the GPU cores are relatively simple (e.g. no out-of-order execution), a chip is typically equipped with multiple SIMT cores (in the range of 1-32 for current GPUs). The thread execution model is extended to support multiple cores, but with a limitation: threads executing on different cores do not have the ability to synchronise their execution nor to share data through a local memory. GPU programs (named *kernels*) therefore require a high degree of *parallelism*: threads must be independent of each other in the sense that they can be executed in any order without modifying the semantics of the program. For graphics processing this is a given: vertices and pixels of a frame can be computed in parallel.

High memory bandwidth

The main data memory of most microprocessors is located on a different chip: *off-chip* from the microprocessor's point of view. The latency to access this data is therefore typically one to two orders of magnitude higher than performing computations. Off-chip memories (e.g. SDRAMs such as DDR3) are therefore engineered to minimise latency. As discussed before, fine-grained multi-threading makes GPUs less susceptible to memory latencies. Therefore, GPUs do not require the minimised latency of conventional SDRAMs: the design of GPU memories (e.g. GDDR5) is not driven by latency minimisation. Instead, the design is driven by frequency maximisation, aiming for a high memory *bandwidth* and *throughput*. Still, GPUs might include *on-chip* memories to reduce traffic to off-chip memories rather than to reduce latencies. On-chip memories in GPUs come in two flavours: programmer-managed (*scratchpad*) and hardware-managed memory (*cache*).

Example GPU and kernel execution

To complete the background on GPU architecture, an abstract view of an example GPU (figure 1.1) and the specifications of a real GPU are given. The figure shows a 2-core GPU with a per-core local memory (cache and scratchpad) and a second level (L2) shared cache. Each core has a single instruction fetch and decode unit, a single thread issue and schedule unit, 8 processing elements, and several load-store units (LSUs) and special function units (SFUs). A large per-core register-file can store the context of many threads, enabling fine-grained multi-threading.

As an example, consider the NVIDIA GeForce GTX580, a 2011 high-end *Fermi* desktop GPU. It has 16 cores, each with 32 processing elements running at 1.5GHz. The off-chip GDDR5 SDRAM is clocked at 1.0GHz and has a maximum throughput of 192GB/s through a quad-pumped 384-bits bus. The GPU includes

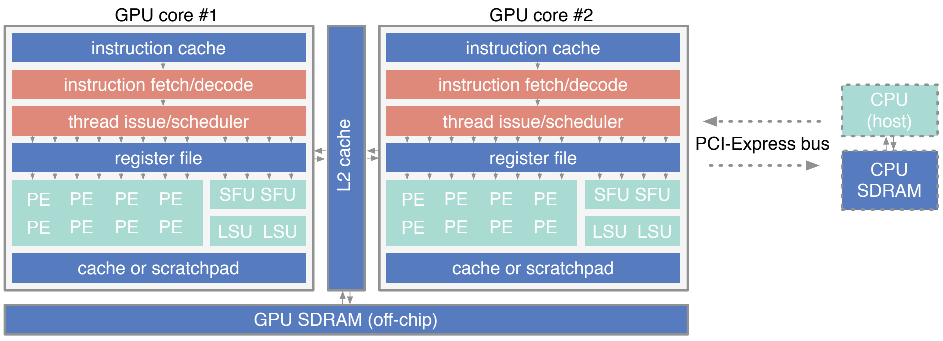


Figure 1.1: Example GPU with 2 cores and 16 processing elements (abbreviated as PE). It can for example be connected to a host CPU through a PCI-Express bus.

a 768KB L2 cache and a 64KB configurable on-chip memory per core. Its peak power consumption is 244W. The GPU uses a *host* CPU as a control processor.

Consider the case where 256K integers need to be copied from one array to another. A GPU kernel for this example could assign each thread with reading an integer from the input array and writing it to the output array (see figure 2.6 for a similar example). The programmer could divide this workload for example in 256 blocks of 1024 threads (see also section 2.3.1). Each core would be assigned a single block at first. These 1024 threads will execute in a fine-grained multi-threading fashion as warps (e.g. 32 threads) on the processing elements of a GPU's core. After a core completes a block entirely, it will proceed to the next.

1.3 Problem statement

Since multi-core processors and specialised architectures such as the GPU are expected to continue the exponential growth of performance and energy efficiency, we need to ensure that all features of the architecture are fully exploited. As a simple example, consider the parallelism of an application. If the application contains a sequential part that accounts for 25% of the execution time, it becomes impossible to reach a speed-up higher than a factor 4, even with an infinite amount of cores (Amdahl's law). Even if the application has no sequential part, it remains the task of the programmer to identify the parallel parts and to construct an efficient implementation. In general, programming has become increasingly challenging over the past decade due to the *heterogeneity* and *parallelism* in processors [21, 61]. Moreover, implementing efficient code has also become increasingly important due to the end of Dennard scaling: the continuation of Moore's law will no longer lead to 'free' performance gains [130].

In particular, developing efficient GPU programs is not straightforward [21, 61]. Programmers are faced with among others new programming languages, multiple levels of parallelism, and a multi-level memory hierarchy. For example, the programmer has to organise threads and data in such a way that data is accessed in a memory and cache-friendly manner. Alternatively, the programmer might decide to create local data copies in the on-chip scratchpad memories, which requires explicit instructions to be distributed as work over the threads. GPU programmers face many more such issues, illustrated by the relatively large design space of a straightforward histogram computation [9]. To design efficient GPU programs, programmers are in many cases required to have detailed architectural and application knowledge, and need to invest a significant amount of time.

The goal of this thesis is to improve the *programmability* of GPUs by both 1) generating efficient GPU source code automatically using a source-to-source compiler, and 2) by automatically ordering threads to maximise data-locality. We define programmability to be inversely proportional with the effort required by a programmer to implement an efficient solution. Several metrics are intertwined with programmability (see also figure 1.2): 1) *performance*: the speed or energy efficiency of a program, 2) *portability*: the generality in terms of correctness and performance across different microprocessors, and 3) *productivity*: the effort and knowledge required to design and maintain program code.

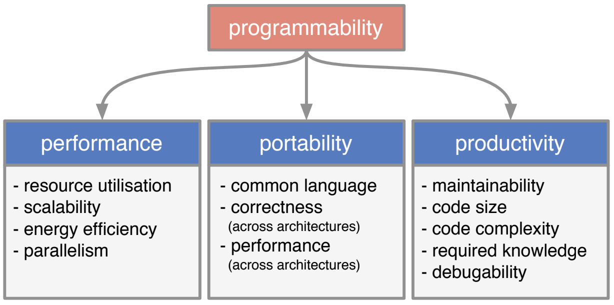


Figure 1.2: Programmability and related metrics: portability, productivity and performance.

1.4 Contributions and thesis outline

This thesis focusses on addressing the programmability of GPUs and is based on the publications [1, 2, 3, 4, 5, 6, 7]. Although the focus of this thesis lies on GPUs, we underline that many contributions are also applicable to other processors, in particular those with a parallel programming model. This thesis makes the following main contributions:

- Chapter 2 motivates the thesis by discussing current microprocessor design trends and future trends. These trends are illustrated with an example 'future' GPU [1]. Furthermore, approaches to address the programmability of GPUs are discussed, including an overview of programming models.

- To be able to reason efficiently about a wide variety of applications, chapter 3 discusses classifications of program code. First, a survey of existing classifications is performed. Next, a new classification named '*algorithmic species*' is presented, classifying loop nests based on memory access patterns. Two versions of algorithmic species are presented: one based on the polyhedral model [5], and one based on array reference characterisations [4].
- Chapter 4 discusses code generation. To improve the programmability of GPUs, a source-to-source compiler is presented that transforms sequential code into efficient GPU code [3]. The compiler uses *algorithmic skeletons* as pre-optimised templates for specific classes of program code, corresponding to algorithmic species as introduced in chapter 3. Furthermore, the compiler's abilities to optimise host-accelerator data transfers [6] and to perform kernel fusion are discussed. Additionally, a comparison between several compiler approaches is presented.
- Chapter 5 motivates changing the GPU's thread scheduling mechanism to improve programmability by automatically maximising data-locality. To obtain insight into cache behaviour, a detailed model of a GPU's cache is introduced first [2]. Furthermore, a case for locality-aware thread scheduling is made, evaluating among others cache performance [7].

This thesis does not contain a separate related work section: it is discussed per chapter. Chapter 6 concludes the thesis and presents proposals for future work.

1.5 Context of this work

This work is performed within the context of the *Morpheus* project, a national Point-One programme project. The project focuses on vital sign monitoring using (non-intrusive) cameras for both baby's at home and neonatals in hospitals. Vital sign monitoring includes respiration-rate and heart-rate monitoring (using photoplethysmography), and posture and sleep analysis. The project's aim is to develop advanced vital sign monitoring 'vision' algorithms and to map them onto an energy efficient platform. A GPU is chosen to accelerate the highly parallel vision algorithms and to achieve this energy efficiency.

Although the GPU-acceleration part of the Morpheus project seems to be a one-time effort, it can in fact benefit from a GPU with improved programmability (e.g. through source-to-source compilation). First of all, GPU code has to be developed for multiple algorithms. Furthermore, the code has to be rewritten multiple times because the algorithms themselves are still in development during the project. Finally, a more general approach allows this work to be used beyond the Morpheus project, improving not just the energy efficiency of a specific set of algorithms, but the programmability of GPUs in general.

"Never underestimate the bandwidth of a station wagon full of tapes hurtling down the highway."

- Andrew S. Tanenbaum (1996)

Chapter 2
Motivation and outlook

If we plan to spend time and money to improve the insulation of our house to protect against the cold winter, we first need to make sure it is not already well insulated. Moreover, we need to make sure insulation is the solution we want: we could alternatively invest in warmer clothes to achieve the same goal. And, are we really sure that the cold winter is coming?

It should be clear by now that this thesis does not cover housing insulation, but rather aims to improve the programmability of GPUs. Regardless of the change of topic, our motivation follows the same line of reasoning. Therefore, this chapter investigates: 1) *necessity* (do we need to protect ourselves against the cold), 2) *opportunity* (can we improve our insulation), and 3) the *alternatives* (should we buy warmer clothes instead).

Because the programmability issues with current GPUs are well discussed in the literature [61], they are only briefly discussed. This chapter rather focuses on the programmability issues of *future* architectures. Although we cannot be certain about the future of microprocessor design and programming, it is possible to present an outlook by analysing past trends and predictions. Such an analysis of trends and predictions is presented in sections 2.1 and 2.2, and illustrated by discussing an example 'future' GPU in section 2.4. Furthermore, section 2.3 discusses alternatives to the compiler-approach presented in this thesis.

2.1 Current trends

Chapter 1 already discussed the exponential transistor growth (Moore's law) and the recent diminishing returns in frequency and energy consumption (the end of Dennard scaling). This section takes a closer look at the current trends of multi-core, many-core, and the memory wall.

Throughout this section, two sets of historical data on microprocessors that have been constructed using multiple sources are presented. The first set contains technical specifications of more than 1400 Intel microprocessors ranging all the way back to the release of the 80386 in 1985. The data was collected from Intel's ARK database [78] and extended with information from Wikipedia [143]. The second set contains the specifications of more than 500 of NVIDIA's 3D-accelerators (1990-1999) and GPUs (1999-2013). This data was taken from TechPowerUp's GPU database [134] and the GPU hardware museum [74].

2.1.1 The multi-core and many-core decade

In 2005, Herb Sutter authored 'The free lunch is over' [130]. Although this may sound like a catering announcement with bad news for students, it is in fact a study on the implications of the end of Dennard scaling. Based on historical CPU data (1970-2004), Sutter argued that CPU frequency and performance per clock will not increase as much as they did before. The free '*performance lunch*' for sequential applications is over: newer microprocessors will no longer trivially improve performance of existing applications. Sutter concluded his work with four implications for software developers: 1) applications will have to become *concurrent* to benefit from advances in microprocessor technology (i.e. parallelism), 2) applications are more likely to become CPU-bound rather than memory-bound in the future, 3) creating efficient programs will become increasingly important, and 4) programming languages will be forced to deal with concurrency.

Several years later, it becomes clear that Sutter's free lunch is indeed over. As of 2013, many desktop, laptop, and even smart-phone microprocessors have two or more processor cores. In a recent article, David Patterson reflects on the problems multi-cores have brought along for programmers [111]. He foresees three possible scenarios for 2020: 1) limit the core count on multi-cores to maintain programmer productivity and core utilisation, 2) only a few types of applications will continue to benefit from core scaling, or 3) advances in compilers or programming languages will allow trivial scaling of applications onto multi-cores. Although Patterson roots for the latter scenario, he is afraid it is not the most likely.

In other work, Shekhar Borkar proposes to design microprocessors with hundreds of small cores: a *many-core* processor [35]. The many-core design is motivated by Pollack's rule: performance increase of a single core is roughly proportional to the square root of its complexity. Therefore, spending area and power on many small cores is more energy efficient for parallelisable applications. Examples of recently introduced many-cores are the Kalray MPPA (with 256 cores) and the Intel Xeon Phi (with 60 cores).

To illustrate the current trends of technology scaling, our own data is presented in figures 2.1 and 2.2. These figures plot four properties of each of the microprocessors (CPUs and GPUs) in our database on a single logarithmic axis:

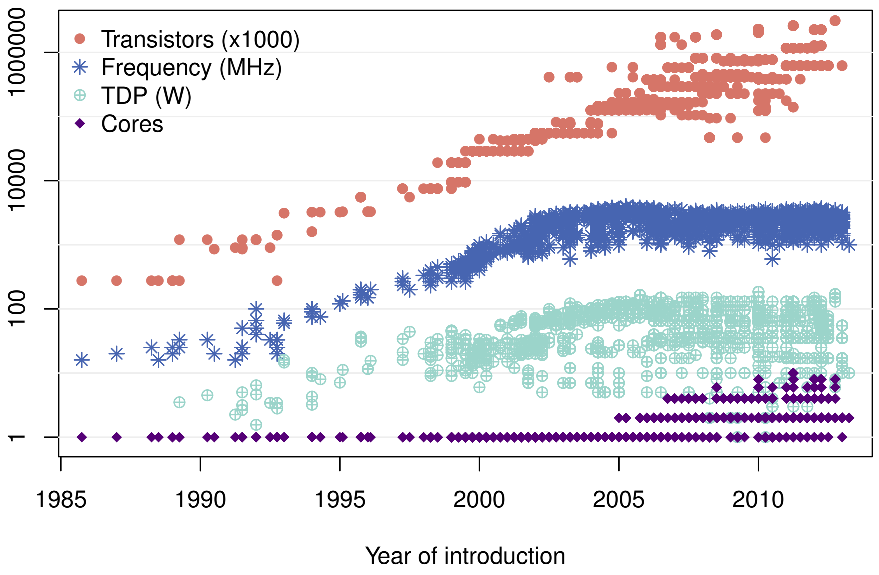


Figure 2.1: Historical data on 1403 Intel microprocessors showing the scaling of a chip's transistors, clock frequency, power dissipation, and core count.

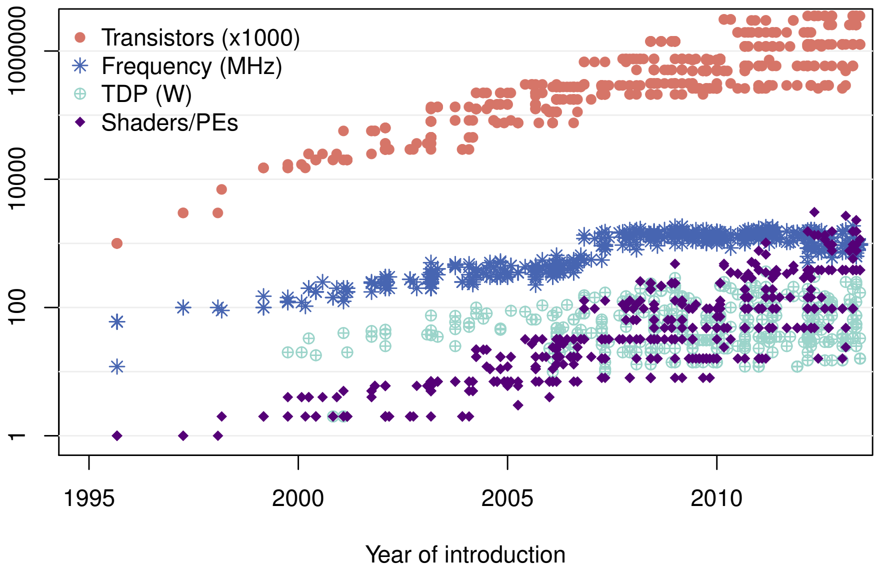


Figure 2.2: Historical data on 566 NVIDIA GPUs showing the scaling of a chip's transistors, clock frequency, power dissipation, and processing element count.

1) the number of transistors, 2) the nominal clock frequency, 3) the maximal power dissipation or *thermal design power* (TDP), and 4) the number of cores (for CPUs), processing elements or vertex and pixel shaders (for GPUs).

Figure 2.1 shows data on Intel CPUs. From the figure, we can clearly observe the continuation of Moore's law: the number of transistors grows exponentially. The end of the free performance lunch can also be identified: power dissipation reached its maximum at around 2004, ending the exponential growth of clock frequency. We also see industry's response: the number of cores started increasing from around 2005. Additionally, we observe that designs have started covering wider ranges. For example, in 2010-2011 Intel released CPUs with 100M to 2B transistors, 600MHz to 3.6GHz, 1 to 10 cores, and a TDP of 3W to 185W.

Figure 2.2 shows the scaling trends of NVIDIA 3D-accelerators and GPUs starting at 1995. Similar to the CPU trends, we also observe Moore's law and the end of frequency and TDP scaling. Additionally, a sudden clock frequency jump in 2007 can be seen. The jump is a result of the G80 design, introducing two clock domains: a high-frequency domain for the PEs and a half-frequency domain for the remainder. Finally, we note the proportional growth of the number of cores with the number of transistors: Pollack's rule and highly-parallel workloads have continued to motivate simple PE designs.

2.1.2 The memory wall

Still, there are other trends to consider besides multi-core and many-core. In 1995, Wulf and McKee observed the '*memory wall*': the growing disparity between the microprocessor's performance and the off-chip memory bandwidth and latency [148]. Both microprocessors and off-chip memories follow an exponential trend, but the exponent for memories is significantly lower. This results in an exponentially growing *gap* between both. So, even more than utilising the microprocessor efficiently, it becomes increasingly important to use the memory efficiently: data re-use should be exploited as much as possible, for example by caching in local on-chip memories.

To illustrate the memory wall and verify Wulf and McKee's predictions, our own microprocessor performance and memory bandwidth data is presented in figures 2.3 and 2.4. Because our database does not contain performance data for a single benchmark, microprocessor performance is estimated. This estimate assumes an instruction throughput of 1 instruction per cycle (IPC) per thread. In reality, the IPC can be higher or lower, depending for example on the ILP and the cache configuration. For CPUs, the clock frequency is multiplied with the number of cores to obtain an estimate of peak performance for *scalar* instructions. This number is further multiplied with the SIMD-width to obtain an estimate of the peak performance for *vector* instructions. For GPUs, an estimate of the peak performance is computed by multiplying the number of processing elements with the clock frequency. To obtain the memory bandwidth, the memory's clock frequency is multiplied with the bus width and the appropriate SDRAM scaling

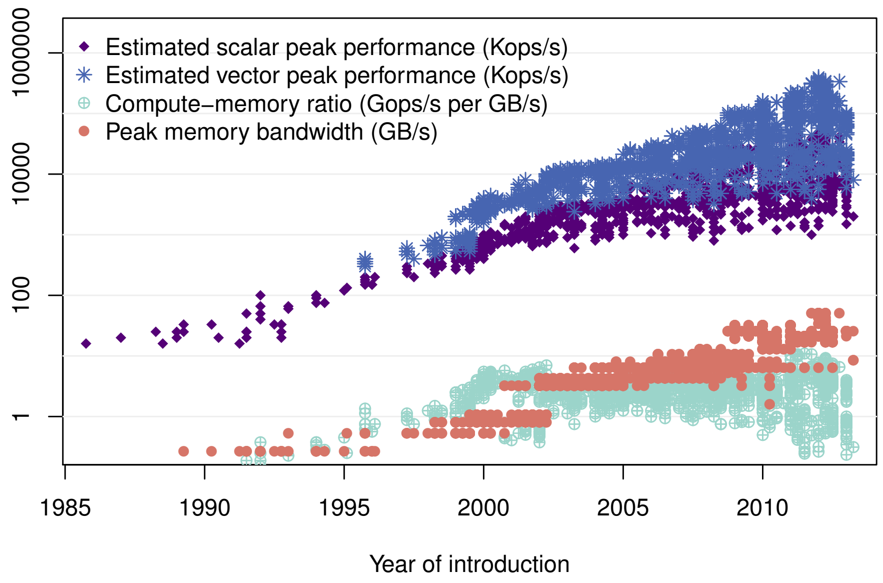


Figure 2.3: Historical data on 1403 Intel microprocessors showing the scaling of an estimate of peak performance (both scalar and vector) and peak memory bandwidth.

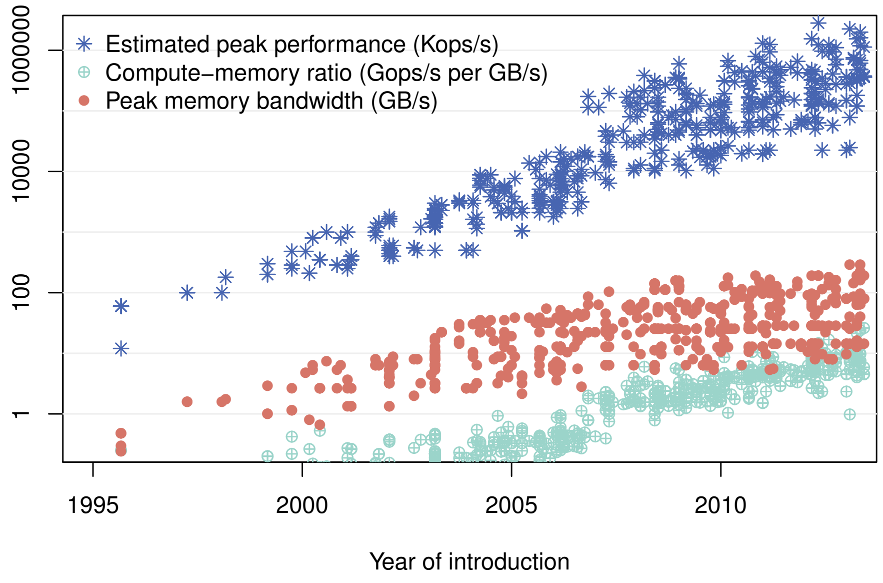


Figure 2.4: Historical data on 566 NVIDIA GPUs showing the scaling of an estimate of peak performance and peak memory bandwidth.

factor (1 for SDR, 2 for DDR/(G)DDR2/(G)DDR3, and 4 for GDDR5). Note that GPU workloads are typically more bandwidth hungry.

Figure 2.3 shows the estimated peak performance and memory bandwidth for Intel CPUs. The figure also plots the compute-memory ratio: the ratio between the microprocessor's peak performance (vector instructions) and the off-chip memory bandwidth. Between 1990 and 2000 growth of the compute-memory ratio is exponential, following Wulf and McKee's predictions (not considering latency). However, the growing gap has stabilised after 2000, with techniques such as *double-pumping* and *multi-channel memories* pushing the bandwidth further. Emerging technologies such as eDRAM and 3D stacking [33] are expected to continue the growth of memory bandwidth, allowing processor performance to scale further without increasing the compute-memory ratio.

For GPUs, the historical compute-memory ratio is different. Figure 2.4 shows that the peak performance outgrows the memory bandwidth, leading to a growth in the compute-memory ratio. In other words, Wulf and McKee's predictions in terms of memory bandwidth still hold. The reason for this is the GPUs tighter coupling (compared to CPUs) of performance growth to Moore's law. While CPUs are spending a large portion of the available transistors on supporting structures (e.g. caches), GPUs simply increase the number of processing elements (possible for highly parallel workloads), leading to a higher peak performance.

From our data, we conclude that the trend of an increasing gap between microprocessor and memory performance is indeed visible for GPUs. However, for CPUs, the compute-memory ratio has already hit a 'wall' at around the year 2000. This is caused by a combination of two factors: 1) when applications become memory bandwidth limited, CPU designers will rather spend transistors on larger caches than on a higher peak performance, and 2) the end of the free performance lunch has made it less straightforward to increase performance for applications with limited parallelism. In any case, for both CPUs and GPUs, the memory wall still plays a major role in microprocessor and application design.

2.1.3 Implications to programmability

The multi-core and many-core trends have severely impacted the programmability of current microprocessors. To exploit the full performance potential, programmers have to partition their applications into multiple concurrent threads, which might require communication and synchronisation at multiple levels. Furthermore, depending on the architecture, the programmer has to manage the sharing or privatisation of data structures. In some cases, designing a concurrent version of an application requires a complete change of the underlying algorithms. For example, an efficient algorithm with limited parallelism can be replaced by a less efficient but highly parallel algorithm to improve overall performance.

Similarly, the memory wall has impacted the programmability of current microprocessors. To counter the memory wall, microprocessors have been equipped with on-chip caches and scratchpad memories. However, scratchpad memories have to be explicitly managed by the software: it is a compiler's or programmer's task to orchestrate data movement to and from these memories. Moreover, even though caches are hardware-managed, programs must be designed with cache friendly memory access patterns and must be scheduled in a cache-aware manner for optimal performance. Other aspects related to the memory wall affecting the programmability are for example burst accesses, pre-fetching, and non-uniform memory access times.

Related to the increased required programming effort is the decreased *portability*: program code can for example be fine-tuned for a specific degree of parallelism or for a specific memory hierarchy. In fact, porting the code to another architecture could require significant modifications [124]. Furthermore, concurrent execution makes it significantly more difficult to find *bugs* in program code [31]. Moreover, as program code deals more and more with performance aspects such as concurrency and data orchestration (and less with the actual functionality), code maintainability and programmer productivity can further decrease.

As an example, consider the computation of a histogram, a common task in image processing. A histogram `H` can be computed by performing unit votes on locations equal to values of the input `A`. This can be written in C code as `H[A[i]]++`, with i being a loop index iterating over all elements of `A`. Implementing such a small program (1 line of C code) on a GPU requires very detailed architectural knowledge and significant programming effort, as demonstrated in [9] and [16]. Although all iterations can be performed in any order (and thus in parallel), the updates to `H` are required to be done atomically. Therefore, the computation is implemented in the form of a reduction tree: threads first compute their local histogram, after which the threads are synchronised and the histograms are aggregated pair-wise. This is performed iteratively using a tree structure, resulting in reduced parallelism after each step. Other examples of aspects to consider when implementing a histogram computation on a GPU are on-chip memory bank-conflicts and padding, coalesced memory accesses, the granularity of parallelism, and the use of hardware or software atomic operations.

2.2 The prospect of dark silicon

So far we have seen the current trends and their impact on the programmability of microprocessors. The work performed in this thesis is further motivated by discussing predictions of the (near) future trends and their implications to programmability. In particular, the focus lies on the trends of dark and dim silicon.

2.2.1 Dark and dim silicon

The previous section has already discussed the end of Dennard scaling and the multi-core and many-core decade it has brought us. However, a more careful study on the end of Dennard scaling identifies another (near future) trend, named '*the* *utilisation wall*' or '*dark silicon*'. This trend predicts that microprocessors will (limited by power dissipation) not be able to switch all transistors at a given time: a portion of the chip will be 'dark'. In fact, analytical models such as Esmaeilzadeh's [57] predict that already 50% of the chip will be dark by 2018.

Let us take a closer look at technology scaling to understand the reason behind dark silicon [133]. Our explanation is based on a scaling factor S between two technology nodes. For example, going from feature sizes of 45nm to 32nm corresponds to $S = 1.4$. Scaling by a factor S results in a transistor count increase of S^2 (scaling in two dimensions) and a switching frequency increase of a factor S. Together, this gives us a performance potential of S^3. However, this potential can only be exploited if a constant power budget is maintained. Previously, with Dennard scaling, scaling by a factor S also resulted in a factor S reduced transistor capacitance and a factor S reduced switching voltage (or S^2 reduced power). Together, this resulted in a S^3 power scaling, equal to the performance potential. However, the end of Dennard scaling limits the reductions in supply voltage, leaving us short a factor S^2. Thus, although there is a performance potential of S^3 (2.7 going from 45nm to 32nm), only a factor S (1.4 from 45nm to 32nm) can be exploited if a constant power budget needs to be maintained.

Summarising, there is a gap of S^2 (where S is the technology scaling factor) between the theoretical performance potential (S^3) and the power-neutral performance potential (S). If the chip size remains constant, this gap forces parts of microprocessors to be switched off (*dark silicon*) or operate at a significantly reduced clock frequency (*dim silicon*) in order to make the power budget [133]. In fact, existing microprocessors already start to include dark and dim silicon. For example, let us take a look at a *die* photo of an Intel Atom Penwell processor, as given in figure 2.5. Some of the components are unused and switched off (*dark*) in most scenarios, e.g. the video decoder and the image signal processor. Other components are significantly less power hungry (*dim*) compared to the main CPU and GPU core on the chip, e.g. the display controller and the audio processor.

2.2.2 Implications to computer architecture

To predict the implications of dark silicon for performance growth Esmaeilzadeh et al. [57] constructed a detailed analytical model consisting of three sub-models: 1) an ITRS-based technology scaling model, 2) a core power-performance scaling model based on empirical data, and 3) a multi-core and many-core architectural scaling model. Even in the most optimistic case, they find only an average 7.9x speed-up over 10 years (23% per year) for the PARSEC benchmark suite. A more conservative approach suggests only an average 3.7x speed-up over 10 years for the same benchmark suite.

If Esmaeilzadeh's predictions are correct, dark silicon will severely impact the growth of microprocessor performance. However, this might lead to new innovations in computer architecture and drive a new class of microprocessor architectures that 'spend' chip area to 'buy' energy efficiency. In recent work,

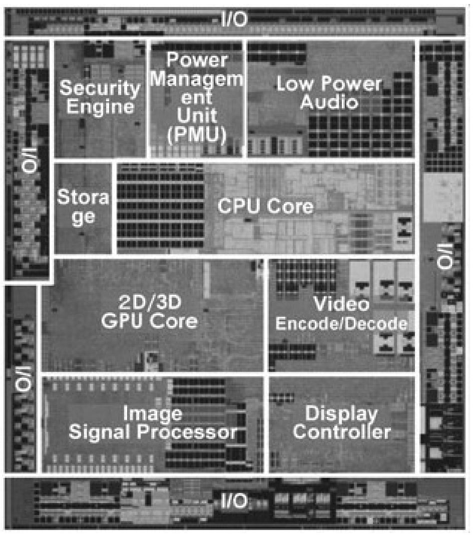


Figure 2.5: Die photo of an Intel Atom Penwell processor. The different components (microprocessors and other logic) are identified and labelled. Photograph: © Hiroshige Goto (2012).

Michael Taylor has identified four key techniques [133] as reactions to dark silicon:

1. **Shrinking silicon.** A logical reaction to unused silicon is to make chips smaller, reducing costs. However, making chips exponentially smaller will quickly turn silicon cost into a negligible fraction of the overall chip cost (e.g. packaging, testing, marketing), leading to a lack of incentive to scale to smaller transistors. Other issues with smaller chips are related to thermal hotspots and a reduced number of I/O pins. Taylor argues that chips will only shrink in size if no other way of using silicon can be found.

2. **Dim silicon.** Another possible reaction to dark silicon is to design general purpose logic that either runs at a low clock frequency or is used infrequently. Possible techniques are near-threshold voltage implementations of wide SIMD processors, coarse-grained reconfigurable arrays (CGRAs), larger caches, or computational sprinting. Some of these techniques are already available in current microprocessors. An example is Intel's Turbo Boost, a form of computational sprinting where the power budget is temporarily exceeded after which a 'dark' period follows. Others, such as CGRA, have not yet gained much popularity outside academia. With CGRAs, multiplexing costs of microprocessor pipelines are avoided by explicitly laying out the computational data-path in space [69, 99].

3. **Specialised co-processors.** Because area is becoming cheaper, a chip can be equipped with many domain-specialised co-processors, each designed to

be significantly more energy efficient for a specific domain compared to a general purpose processor. In such a design, applications are required to migrate to different co-processors, executing where it is most efficient. The Intel Atom Penwell of figure 2.5 already shows this in some form. However, Taylor expects the number of co-processors to scale much further, such as done in the GreenDroid architecture [68].

4. **New classes of circuits.** Finally, Taylor identifies a 'deus ex machina': the possibility of a breakthrough in semiconductor technology. Such a breakthrough will need to fundamentally change the way we build transistors to reshape computer architecture as we know it today.

We note that from Taylor's four key techniques, only dim silicon and specialised co-processors are directly related to computer architecture. Thus, from a computer architecture perspective, Taylor concludes that we will move towards more energy efficient and dynamic microprocessor designs (dim silicon), and towards an increased amount of specialised co-processors or accelerators.

2.2.3 Implications to programmability

While programmability is not yet recovered from the multi-core and many-core decade, dark silicon is already looming on the horizon. If Taylor's predictions of increasingly dynamic and specialised microprocessor designs are correct, programmability will definitely take another hit. In this section, we illustrate this by discussing some examples taken from Taylor's predicted shift towards dim silicon and specialised co-processors [133].

As discussed, dim silicon might lead to the adoption of CGRAs [69]. Such processors allow the data-path layout to be programmed, requiring programmers to work with programming paradigms significantly different from traditional microprocessors. Another example are low-power wide SIMD processors, requiring programs to be tailor made to wide vector data-types. In some cases, such low-power designs might lead to special restrictions imposed upon the application, lowering programmer productivity [123].

Furthermore, major challenges lie in the programmability of specialised co-processors [57]. Currently, there are already significant programmability issues with accelerators such as GPUs [61]. The future might bring many of these accelerators, each with their own ISA (instruction set architecture - e.g. x86-64 or ARM), their own form of parallelism (vectors, threads, data-flow, tasks), their own memory space (virtual or physical), their own memory hierarchy, and so on. This will lead to a computing environment where a significant programming effort will be required to optimise a given application for processors composed of multiple of such accelerators.

2.3 Addressing programmability issues

So far, we have mostly discussed the *necessity* to address programmability: it has become a major challenge and is expected to remain so in the near future. This section glances over a wide range of existing approaches to address programmability issues, identifying both *alternatives* and *opportunity.* Each of the remaining chapters of this thesis will further discuss specifically related approaches in-depth.

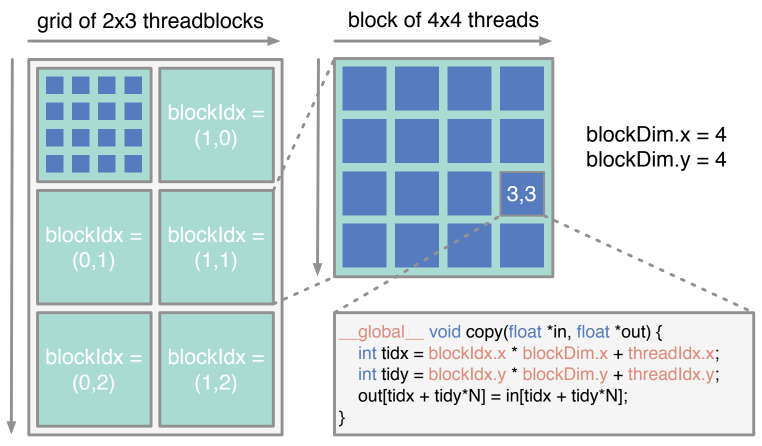
2.3.1 Programming languages and frameworks

Many existing programming languages and frameworks raise the level of abstraction to address the current programmability issues. This section discusses some of the most known or promising GPU-compute languages and frameworks (see also figure 2.7 for an overview). First, the traditional GPU-compute programming frameworks (mid-level) are discussed, followed by an overview of low-level intermediate languages. Finally, several high-level languages are discussed.

Traditional GPU-compute programming frameworks

NVIDIA's CUDA is one of the most well known GPU-compute frameworks [107], even though it is vendor specific. The CUDA-C language, derived from C++, is used to express the *kernel*: a small program that is executed for each thread. CUDA defines a grid of threadblocks, each containing a set of threads (see also the example given in figure 2.6). This hierarchy can be organised in a 1D, 2D or 3D fashion. The use of local 'shared' scratchpad memory and synchronisation barriers is possible only within a threadblock. In current GPUs, this requires an entire threadblock to be mapped onto a single GPU core. A CPU *host* needs to be present to launch the kernel and to orchestrate data movement to and from the GPU's off-chip memory. The CUDA framework has gradually evolved to become more high-level (e.g. through the addition of the Thrust library), but still retained its low-level fine-tuning possibilities for expert programmers. CUDA is well documented, there is a large user-base, and there are many libraries and tools available, including integration with languages such as Python and Matlab.

OpenCL is the answer of the Khronos group to CUDA. In contrast to CUDA, OpenCL is an open standard and is currently supported by many different platforms, including processors other than GPUs such as Intel CPUs, the Cell Broadband Engine, and Altera's ZinQ FPGAs. OpenCL and CUDA-C kernels are very comparable, but OpenCL's explicit kernel compilation approach (programmers have to call the JIT-compiler manually through the API) and multi-platform support imply large amounts of boilerplate code. Furthermore, because OpenCL is a standard, it lags behind CUDA in terms of features. OpenCL and CUDA can show comparable performance on NVIDIA GPUs [58]. However, OpenCL's portability does not imply *performance portability* across different platforms [112].



```
__global__ void copy(float *in, float *out) {
    int tidx = blockIdx.x * blockDim.x + threadIdx.x;
    int tidy = blockIdx.y * blockDim.y + threadIdx.y;
    out[tidx + tidy*N] = in[tidx + tidy*N];
}
```

Figure 2.6: An example CUDA thread configuration, showing a 2x3 grid of 6 threadblocks, each with 16 threads arranged in a 4x4 matrix. With the example CUDA-C kernel code, each thread copies a single value from an input into an output array.

On a similar level as CUDA and OpenCL are DirectCompute and RenderScript. DirectCompute is part of Microsoft's DirectX and is only available for Windows. DirectCompute's kernels are programmed in HLSL (high-level shading language), similar to GLSL (the OpenGL shading language). RenderScript is Google's compute framework for Android devices. It is very similar to OpenCL, but mostly supports execution on mobile devices, such as ARM CPUs and Imagination Technology's PowerVR GPUs.

Low-level intermediate languages

Because the ISA can change slightly among different generations of NVIDIA GPUs, the PTX intermediate language was introduced as a compilation target [107]. To target the actual ISA, just-in-time (JIT) compilation is used: PTX code is compiled just before the GPU program is executed. The PTX language is an assembly-style language containing the kernel program only. Although it is not common to express programs in PTX, it is possible.

In response to NVIDIA's PTX, Khronos has recently released specifications for OpenCL SPIR, an OpenCL standard portable intermediate representation. SPIR is derived from LLVM's intermediate representation (LLVM-IR), making it straightforward to adopt for existing LLVM-based compilers.

Another low-level language is HSAIL, an intermediate language designed by the HSA Foundation. The ultimate goal of this foundation is to address the programmability issues of GPUs, CPUs, other devices, and combinations thereof. HSAIL is a virtual ISA supporting all of C++ and targeting a wide range of devices. It is meant as a compilation target for OpenCL, C++AMP and others.

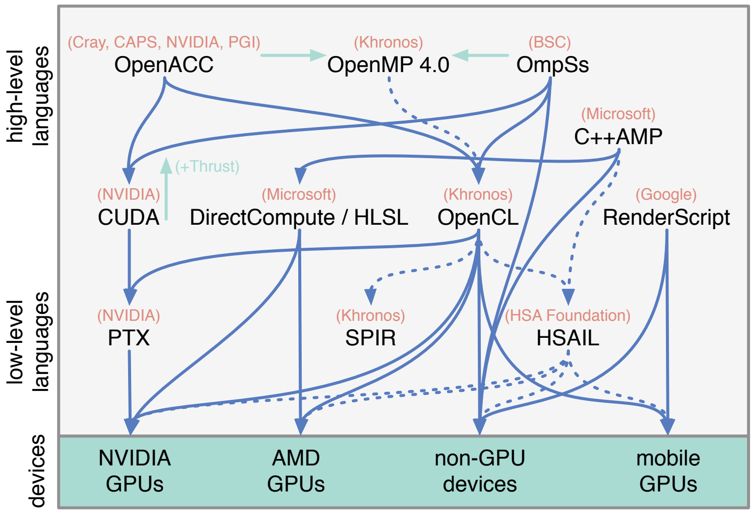


Figure 2.7: Overview of some of the most known and promising GPU-compute languages and frameworks. Blue arrows indicate compilation paths: solid for current and dashed for announced paths. The light green horizontal arrows indicate the merging of the OpenACC and OmpSs directives into the OpenMP 4.0 standard.

High-level languages and frameworks

Microsoft's C++AMP is a high-level language targeted at CPUs and GPUs (through HLSL). It is implemented as a library on top of C++ and requires little modifications to enable GPU acceleration. C++AMP manages CPU-GPU data transfers implicitly, improving programmability. C++AMP is therefore promising, additionally because the low-level HSAIL is designed to support its features.

OpenACC is a standard for compiler directives for GPUs, backed by among others CAPS, PGI and NVIDIA. OpenACC's directives enable straightforward GPU acceleration, although optimal performance is not always achievable [141]. Most of the directives are being pushed into the upcoming OpenMP 4.0 standard.

OmpSs is a set of GPU compiler directives from the Barcelona Supercomputing Center (BSC) influenced by OpenMP and StarSs [54]. OmpSs has properties similar to OpenACC, and parts are also being pushed into OpenMP 4.0.

A final example are domain-specific languages (DSLs). These languages are typically high-level and tuned for a specific application domain (e.g. mathematics or image processing). Examples are the OP2 [30] and OptiML DSLs [14].

2.3.2 Architectural support for programmability

Another way to address the programmability issues is to provide *architectural support*: the microprocessor is adjusted to include some of the programmer's or compiler's work into hardware. Such architectural support can come at the expense of energy, chip area or performance. In this section, we illustrate the efficiency versus programmability trade-off by discussing examples of architectural support for programmability in GPUs: 1) on-chip cache memories, 2) relaxed *memory coalescing* requirements, and 3) dynamic warp formation.

Firstly, let us discuss caches. On-chip memories can reduce traffic to off-chip memories, and decrease latencies for load and store operations. Most older (pre-Fermi) GPUs are equipped with programmer-managed on-chip memory (the scratchpad), while more recent GPUs (e.g. NVIDIA's Fermi and Kepler) also include hardware-managed on-chip memory (caches). With a scratchpad memory, it is the programmer's task to explicitly perform loads and stores to ensure that specific data-structures reside in the on-chip memory. In contrast, when using a cache, data-movement is transparent to the programmer: a pre-defined replacement policy determines which data-elements are stored on-chip. A cache can thus improve programmability: the programmer is no longer required to manage the on-chip memory. However, programmer-managed data-movement can in many cases outperform a cache's replacement policy. Nevertheless, this trade-off is more complicated. For example, a cache can still outperform a scratchpad, for example in the following cases: 1) the memory accesses are dynamic, i.e. they change from run to run, or 2) the overhead of additional scratchpad data-movement instructions (address calculation, loads, stores) is high.

Secondly, we discuss increasingly relaxed memory coalescing requirements. Many GPUs can unite (or coalesce) multiple memory accesses into a single larger access [107]. This '*memory coalescing*' is a necessary requirement for most GPUs to benefit from the high memory bandwidth. As a consequence, GPU programmers prioritise meeting the memory coalescing requirements over other optimisations. Architectural modifications have relaxed these requirements gradually over the last years [107], improving the GPU's programmability at the cost of increased hardware complexity. For example, NVIDIA GPUs now require only two instead of 32 individual memory accesses when a warp's accesses are misaligned. Furthermore, the L1 cache's recombination logic can mitigate the performance loss of earlier GPUs for non-sequential (shuffled) accesses.

Finally, we briefly mention dynamic warp formation. As discussed in section 2.4.3, the GPU's default static formation of warps is not necessarily optimal: a thread's properties can change over time. Therefore, it has been proposed [63, 102, 105] to address this problem dynamically in hardware. This is again a form of architectural support to improve programmability, as programmers have to spend less effort to for example reduce branch divergence.

2.3.3 Iterative compilation

Although modern compilers are powerful, they are not able to perform all optimisations required to generate efficient code. Performing such optimisations is left as a task for the programmer, or, as discussed earlier, might be solved by specific programming languages or architectural support. Another approach to solve these issues of programmability is to make compilation *iterative*. Iterative compilers learn over time through a feedback-loop from the impacts of their own decisions [53, 64]. We illustrate this by highlighting a number of compilers: 1) a machine-learning infrastructure for GCC, 2) a machine-learning compiler for polyhedral transformations, and 3) a GPU matrix-multiplication auto-tuner.

Milepost GCC [64] is an example of a compiler that is able to learn optimisations for a specific processor architecture based on a feedback-loop. The compiler (an extension to GCC) uses machine learning to automatically learn from compilation runs, taking into account the correlation between an application's features, run-time behaviour, and the chosen optimisations. Milepost GCC is constantly evolving, as it learns from a growing on-line database of compilation runs.

The second example is an iterative compiler performing polyhedral transformations [109]. The *polyhedral model* is a powerful framework to perform loop transformations (see also section 3.2.1). However, due to the complexity and variety of modern microprocessors and the wide extent of possible transformations (and combinations thereof), it is far from trivial to find a set of transformations that lead to improved performance. The iterative compiler first performs a training phase in which sets of transformations are applied and performance data is collected. From the initial training data a performance model is generated using machine learning algorithms. Based on the trained performance model, the compiler is able to select a set of useful polyhedral transformations.

Another example of iterative compilation is an *auto-tuner* designed specifically for the matrix-multiplication kernel on GPUs [94]. Since matrix-multiplication forms the basis for many linear algebra computations, it has been heavily fine-tuned for many different GPUs. This process of fine-tuning can be automated using an auto-tuning framework, in which many variants of the code are generated and empirically evaluated on hardware. In some cases, auto-tuning even outperforms manual tuning, illustrating the vastness of the optimisation space.

2.4 Example: An adaptive GPU architecture

To further illustrate the impact of trends such as dark silicon on the GPU architecture, seven opportunities for architectural improvements enabled by the discussed trends are discussed. These opportunities are enabled by three architectural parameters which are discussed individually in the following sections: 1) the number of active threads, 2) the ratio between the compute power and the off-chip memory bandwidth, and 3) the number of processing elements in a core and the size of a scheduling unit (*warp*).

We use NVIDIA's G80 GPU [95] as an example GPU throughout this section, although most desktop GPU architectures have a relatively similar high-level design. The G80 architecture has up to 16 cores, each containing 8 processing elements (PEs). PEs in a core share an instruction cache, instruction fetch and decode stage, a thread scheduler, a register-file, and an on-chip scratchpad.

2.4.1 Parameter 1: The number of active threads

One of the characteristics of a GPU is its ability to hide pipeline and off-chip memory latencies through fine-grained multi-threading. If sufficient parallelism is available in the application and the GPU's register-file is large enough, these latencies can be hidden entirely. The register-file is required to store the context for each of the *active threads* to enable zero-overhead context switching. For example, for the G80 with 8 PEs per core, the pipeline latency for basic ALU operations is 22 cycles [107] and the off-chip memory latency is on average ± 600 cycles [107]. Assuming that all instructions are dependent on the previous instruction, at least $22 \cdot 8 = 176$ active threads per core are needed to hide the pipeline latencies. With off-chip loads only, at least $\pm 600 \cdot 8 = \pm 4800$ active threads are needed. In practice, 256 or 512 active threads per core on the G80 is enough: they do not only contain memory accesses and their flow-dependences are limited.

Trade-offs

Allowing a large active thread count requires a large register-file, consuming power and occupying chip area. On the other hand, when the active thread count is not high enough to hide the latencies, performance can drop significantly as PEs are becoming idle. The optimal value for the active thread count is therefore dependent on many factors: the pipeline depth, the off-chip memory characteristics, the number of PEs per core, the thread scheduling mechanism, and the application. Because the application is typically unknown at microprocessor design time, the trade-off becomes a dynamic problem. Performing analysis of typical workloads can help to get insight in the statistical properties of occurrences of off-chip loads and dependencies.

Opportunities for improvements

Two opportunities for architectural improvements with respect to the number of active threads are identified: 1) a dynamically-sized register-file, and 2) a latency-aware thread-scheduler.

To save power when a small number of active threads is sufficient, we propose to add a **dynamically-sized register-file** to each core. Instruction sequences (e.g. at kernel-level) can be analysed statically where possible and dynamically otherwise to determine how many registers are required to accommodate a sufficient number of active threads. In case a high number of registers is required, another component (e.g. a cache) can be turned-off to maintain constant power (i.e. it becomes 'dark'). In this way, the register-file can benefit from power savings for workloads that only require a low number of active threads (as shown for a hierarchical register-file in [67]). This is in contrast to current GPUs that greedily schedule the largest number of threads possible.

Furthermore, a **latency-aware thread-scheduler** can reduce the required number of active threads. A GPU schedules warps that are ready for execution in a round-robin fashion: instruction per instruction [107]. For example, in the program of listing 2.1, instruction 1 will first be executed for all threads, followed by instructions 2, 3, and 4. However, since instruction 4 is dependent on the result from an off-chip load, a large number of active threads are required to hide the latency of instruction 3. In contrast, a latency-aware scheduler would give preference to scheduling instructions 1, 2 and 3 for a subset of the active warps first, such that the long latency of instruction 3 can be hidden by executing instructions 1, 2 and 3 of a second subset of active threads. To hide pipeline latencies as well, the subset size must be set equal or larger to the pipeline latency. This idea can be implemented in the form of two-level warp scheduling [105]. Two-level warp scheduling groups warps into fetch groups and schedules normally within such a group. However, execution can only switch to another fetch group when all warps in a group are idle. This technique showed a performance increase of 19% on average for a set of benchmarks [105].

```
instruction 1:    $r0 ← 2
instruction 2:    $r1 ← $r0 * 4
instruction 3:    $r2 ← load[$r1]
instruction 4:    $r3 ← $r2 * $r2
```

Listing 2.1: Example pseudo-assembly GPU code.

2.4.2 Parameter 2: Compute-memory ratio

The GPU is able to achieve a high theoretical throughput [61]. This is accomplished by two means: 1) by providing a large number of processing elements, and 2), by providing a high bandwidth to off-chip memory. While the first enables high instruction throughput, typically measured in giga-*floating point* operations per second (GFLOPS), the second enables high data throughput, measured in gigabytes/s (GB/s). The ratio between the two (GFLOPS versus GB/s) is the *compute-memory ratio*: an important design parameter for GPUs.

	2dconv	**2mm**	**3dconv**	**3mm**	**atax**	**bicg**
oper.int. [flops/byte]	1.35	0.50	1.60	0.51	0.50	0.50
limit on M2090	mem	mem	mem	mem	mem	mem
	corr	**covar**	**fdtd-2d**	**gemm**	**gesummv**	**grams**
oper.int. [flops/byte]	0.50	0.50	2.26	0.67	0.25	0.65
limit on M2090	mem	mem	mem	mem	mem	mem
	mvt	**syr2k**	**syrk**	—	**backp**	**bfs**
oper.int. [flops/byte]	0.50	0.70	0.67	—	3.41	0.94
limit on M2090	mem	mem	mem	—	mem	mem
	gauss	**hotspot**	**kmeans**	**nw**	**particle**	**path**
oper.int. [flops/byte]	2.08	27.33	4.15	8.79	1.99	14.33
limit on M2090	mem	comp	mem	comp	mem	comp

Table 2.1: Operational intensities for the PolyBench/GPU (2dconv → syrk) and Rodinia (backp → path) benchmarks, and their limits (compute or memory-bound) for a Tesla M2090.

An application can be classified as being either *compute-bound* (limited by instruction throughput) or *memory-bound* (limited by off-chip memory bandwidth). To determine which limit applies, the metric *operational intensity* can be used, measured in operations per byte [144]. To illustrate this, operational intensities for two benchmark suites are shown in table 2.1: PolyBench/GPU [117] and Rodinia [125]. We observe that benchmarks from PolyBench/GPU are on average more memory intensive, while Rodinia benchmarks are more compute intensive.

A visualisation of the compute-memory ratio can be obtained by plotting the *roofline model*, a very abstract performance model for microprocessors in general [144]. The roofline model gives the maximum achievable performance for a specific application based on its operational intensity and the hardware's compute-bound (instruction throughput) and memory-bound (memory bandwidth). An example is given in figure 2.8 for an NVIDIA Tesla M2090 GPU. This figure plots the operational intensities from the discussed benchmarks (table 2.1) as horizontal lines. Additionally, it shows the averages of the two benchmark suites.

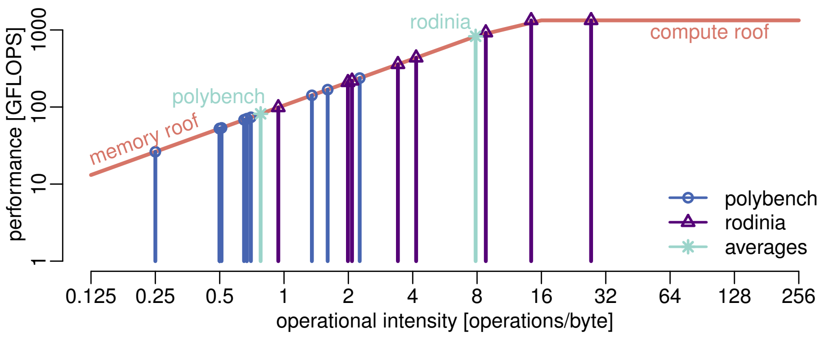


Figure 2.8: Roofline model for an NVIDIA Tesla M2090 GPU with data points from the PolyBench/GPU and Rodinia benchmark suites.

Trade-offs

Similar to the active thread count parameter, designing a GPU's roofline is a dynamic problem. Performing application analysis can again help to solve the compute-memory ratio trade-off for the average case, but will still result in over-dimensioned hardware in practice: the variance among applications is large (see table 2.1 for example).

Increasing the peak instruction throughput can be achieved by adding more processing elements, increasing their clock frequency, or improving the IPC (instructions per cycle) of individual processing elements. On the other hand, increasing the peak off-chip memory bandwidth can be achieved by increasing the clock frequency of the memory or increasing the width of the memory bus.

Opportunities for improvements

A dynamic compute-memory ratio (or: roofline) can be created to improve the overall power efficiency of GPUs, enabled by discrete fixed modes of operation. Two techniques to create a dynamic roofline are distinguished: roofline-aware DVFS (dynamic frequency and voltage scaling) and dynamic compute core disabling. Additionally, we identify the need to increase the operational intensity architecturally.

Dynamic frequency scaling can be used to reduce the clock frequency of the GPU core or off-chip memory, linearly lowering the compute or memory roofline. For example, most benchmarks from PolyBench/GPU (see table 2.1) allow halving the compute frequency without loosing performance[1], while (theoretically) saving up to a factor of two in terms of power. Furthermore, voltage scaling might be applied in addition, lowering the voltage in combination with the frequency for cubic gains in power ($P = \alpha \cdot f \cdot C \cdot V^2$). **Roofline-aware DVFS** can therefore be seen as an opportunity for architectural improvement: a dynamic roofline can save power while maintaining performance. Preliminary studies have shown that roofline-aware DVFS is able to increase energy efficiency by up to 58% while maintaining performance [8].

Similarly, linear power gains can be obtained by temporarily powering down entire cores for memory-bound applications. This **dynamic compute core disabling** is enabled by the GPU's modular architecture and can be applied at kernel-level granularity. As before, this lowers the roofline, saving power without compromising performance. Similarly, memory banks could be powered down for compute-bound applications. However, the memory consumes relatively little power compared to the microprocessor core [93] and imposes constraints on memory size and location.

Furthermore, we also identify the need to architecturally **increase the operational intensity** of applications. As the growth of compute performance is predicted to outgrow memory bandwidth growth despite emerging technologies such as 3D-stacking [85], an increasing number of GPU kernels will hit the memory wall in the near future. To maintain performance growth, future architectures will need to improve and better exploit data-reuse by increasing the size of register-files, caches and scratchpad memories.

2.4.3 Parameter 3: Core and warp sizing

GPUs typically contain multiple cores, each with multiple processing elements. The G80 architecture for example can scale up to 16 cores, with each core containing 8 PEs. On each core, instructions are executed as warps: groups of threads executing in lock-step. In table 2.2 we list several NVIDIA GPU families along with their core and warp sizes. We identify these sizes as third design parameter.

[1]Note that this is according to the roofline model; the peak memory bandwidth might not be achieved for low core clock frequencies due to a too low memory request rate.

	PEs per core	warps issued	warp size	release year
G80 (Tesla)	8	1 per core per cycle	32	2006
GT200 (Tesla)	8	1 per core per cycle	32	2008
GF100 (Fermi)	32	2 per core per cycle	32	2010
GF104 (Fermi)	48	4 per core per cycle	32	2010
GK104 (Kepler)	192	8 per core per cycle	32	2012
GK110 (Kepler)	256	8 per core per cycle	32	2013

Table 2.2: Warp and core sizes for various NVIDIA GPU families.

A GPU core contains a set of PEs that share common components (e.g. local memories and pipeline stages) which are inaccessible by other PEs in the GPU. For programming models such as CUDA and OpenCL, a *threadblock* or *workgroup* typically maps in its entirety onto a single core. In a G80 core, PEs share among others an instruction cache, an instruction fetch and decode stage, a scratchpad memory and a texture cache.

Since PEs in a core share the first few pipeline stages with other PEs, they are required to execute the same instruction at the same time. A core thus forms a natural fit to execute instructions from a single warp. Nevertheless, as shown in table 2.2, the warp size does not necessarily match to the number of PEs per core. In fact, on the G80, the 8 PEs in a core temporarily schedule the execution of a single warp, i.e. using 4 clock cycles for 32 threads [95]. Warps are thus a means to create virtual cores in time. Energy is saved by creating two clock domains: only a single instruction needs to be fetched and decoded every 4 cycles. Newer GPUs have more complicated designs, as they are able to issue multiple warps per core per cycle [107].

Trade-offs

To highlight the trade-offs of core and warp sizing, two extremes are discussed: a core size equal to the total number of PEs, and a core size of 1. For clarity, we assume in these cases a warp size equal to the core size and a single clock domain.

Many advantages can be identified for GPUs with a single large core. With the G80 design as a starting point, the following main advantages are found: 1) a significant area reduction and power saving by sharing various stages of the pipeline (e.g. instruction fetch, instruction decode, branch logic), 2) a single (larger) scratchpad memory can replace the existing smaller memories, 3) more inter-thread communication is possible through the execution of potentially larger threadblocks or workgroups, and 4), an increase in coalesced memory accesses by recombining threads [90].

At the other extreme is a GPU with a single PE per core. The main advantages for such a configuration are: 1) every PE can execute independent of others, i.e. there is no branch divergence penalty, and 2) a core takes a small fraction of the total chip area, making routing (e.g. of the clock tree) relatively easy.

The advantages for both of these cases are disadvantages for the other. This creates a trade-off with many factors, some of which are dynamic: an optimal value can only be determined at run-time. The warp size can furthermore be adjusted to create virtual cores, sharing many of the same trade-offs. However, instead of reducing the chip area when creating larger cores, a larger warp size will allow a reduction in clock frequency of the first pipeline stages, as only a single instruction needs to be decoded per warp.

Opportunities for improvements

Two opportunities to address the trade-offs for core and warp sizing are identified: run-time core fusion and dynamic warp formation and sizing.

Because a large core has advantages in terms of energy efficiency, it is appealing to design a GPU with such a configuration. However, this can result in a severe penalty in case of divergent kernels: one or more orders of magnitude depending on the GPU and the kernel. To still be able to accommodate such kernels, we propose to split a larger core in several smaller cores at run-time, creating an adaptive configuration which can be changed at kernel-granularity (based on static analysis or profiling). To be able to enable **run-time core fusion**, the hardware needs to be able to accommodate the smallest core size and thus include for example an instruction fetch and decode stage for every core [80]. Dark silicon justifies the additional area costs by power gating components when the GPU is configured for non-divergent kernels. Overall, a large core will significantly improve power efficiency for non-divergent kernels by saving power and increasing performance through e.g. increased opportunities for memory coalescing.

Secondly, **dynamic warp formation and sizing** is identified as an opportunity for improvement. Currently, warps are formed statically based on thread indexing. However, several works have proposed to re-combine warps at run-time [63, 102, 105], either to improve memory coalescing or to reduce branch divergence. Apart from dynamic warp formation, warp sizing can also play a major role to improve energy efficiency, as discussed in [90].

2.4.4 Discussion

This section identified opportunities for architectural improvements, including: dynamic register-file sizing, latency-aware scheduling, roofline-aware DVFS, dynamic compute core disabling, run-time core fusion, and dynamic warp sizing. Most of these are motivated by trends such as the memory wall and dark silicon. Common to these opportunities is the dynamism and workload-adaptiveness: setting architectural parameters at run-time enables a higher energy efficiency.

The discussed opportunities have shown possible directions of architecture design. Of course, there are many other directions to explore. An example is the GPU-CC architecture [37], which addresses the memory wall and energy efficiency issues by enabling a second 'pipelined' configuration of a GPU's PEs.

2.5 Summary

This chapter has discussed the current trends in microprocessor design, including the continuation of Moore's law, the end of Dennard scaling, and the recent multi-core and many-core decade. We have also illustrated and discussed the memory wall: the growing gap between microprocessor peak performance and memory bandwidth. We also discussed the near-future trend of *dark silicon* and its implications to microprocessor design and programmability. Both dark silicon and the current trends motivate our work: programmability of specialised microprocessors such as GPUs is a major issue and is expected to become even more important as increased specialisation is used to achieve energy efficiency.

We also gave an overview of some of the alternatives to the approach presented in the remainder of this thesis. We have discussed high-level programming languages, architectural support for programmability, and iterative compilations. Although each of the alternatives is promising, there is still room to further improve the programmability of specialised microprocessors, as will be demonstrated in this thesis.

"Who can explain why one species ranges widely and is very numerous, and why another allied species has a narrow range and is rare? Yet these relations [...] determine the present welfare, and, as I believe, the future success and modification of every inhabitant of this world."

- Charles Darwin (1859)

Chapter 3
Classifications of program code

Throughout history, many classifications have been proposed for various reasons. Classifications exist in many fields, including computer science, biology and chemistry. Although these classifications (or categorisations) have been applied to many different types of objects, they have a common goal: classes (objects of a similar kind or with common properties) allow objects to be recognised, differentiated and understood in a structured manner. In this chapter, we discuss classifications of program code: tools to help us recognise, differentiate and understand different types of program code. We see such a classification as a first step towards improving the programmability of specialised microprocessors such as the GPU. In other words, we first apply structure to our problem (this chapter) before we implement a solution (chapter 4). Examples of existing program code classifications are the Berkeley *dwarfs* [23], algorithmic skeletons [44], and the Galois system [114].

First and foremost, our program code classification (or *algorithm classification*) should help us address the discussed programmability issues: it needs to be suitable for our compiler-based approach. Still, we also keep in mind that a classification can also be a means to perform *performance prediction* or can be used as vocabulary for programmers. These different goals are characterised as:

- An algorithm classification can be seen as a way to facilitate the design and improve the quality of compilers. For example, a compiler can include an optimisation or transformation pass which is known to be suitable for a specific class of code. By feeding class information directly into compilers, their design can be focused on transformation and optimisation passes rather than analysis. The code analysis (extracting the classes) will still be required, but can now be cut-out and performed in a single common place, shared among many compilers. The extraction of classes should remain automated where possible, but should also have a manual or interactive mode such that programmers can feed additional information to the compiler.

- A classification can also be used for performance prediction, i.e. giving an estimate of the performance of program code on a specific microprocessor. An example of a performance model based on program code classes is the 'boat hull model' [11]. Since performance prediction is not the primary goal of this thesis, this topic is not further discussed. Still, we keep the goal in our mind, as it broadens the scope of applicability of an algorithm classification.

- An algorithm classification can also be a means for programmers to communicate and learn *design patterns*. For example, programmers can identify problems common to code of the same class, identify potential parallelism, or apply known design patterns and optimisation techniques. Furthermore, we also see an algorithm classification as a way to facilitate communication among programmers, presenting them with a common idiom in which they can describe their computational problems.

Having described the goals of classifications, five requirements are identified that need to be met in order to achieve these goals. The requirements are:

1. Classes should be **extracted automatically** from program code where possible. If they are not automatically extracted, the compilation process will involve manual work. Ultimately, this is not desirable, since the identification of classes can be error prone and places a heavy burden upon programmers, decreasing programmability.

2. A classification must be **intuitive** and easy to understand. Although classes could be extracted automatically, we also envisage a classification to be used manually: 1) as design patterns for programmers, and 2) for further manual classification or fine-tuning, e.g. the programmer might have additional information about input data ranges. Therefore, an easy to learn and descriptive algorithm classification is required.

3. We require algorithm classes to be **formally defined**, i.e. code belongs to a certain class if and only if a set of formal properties hold. Such a definition allows guarantees to be set on correctness, allows programmers to fully understand properties of classes, and facilitates automatic extraction of classes from program code.

4. An algorithm classification must be **complete** within set boundaries, i.e. a single algorithm should belong to at least one predefined class (possibly more than one if the classes overlap).

5. Classes must be **fine-grained**, capturing low-level algorithm details. Although a finer granularity is preferable, it must not come at the cost of the other requirements. We therefore require classes to include the most important aspects for performance, i.e. the structure and amount of parallelism, and information on data reuse and locality.

With the requirements set, an extensive survey of different types of existing algorithm classifications is performed (see section 3.1). However, a suitable classification for our purposes was not found. Therefore, our own program code classifications have been developed, of which figure 3.1 gives an overview. The figure shows that our first classification (A, see [17] and [12]) was inspired by classifications used for compilers based on 'algorithmic skeletons', most notably the work by Caarls et al. [39]. This initial classification was formalised in the work on 'algorithmic species' (B), by basing it on the *polyhedral model*, a mathematical representation of program code. Next, the underlying theory was revised to support a wider range of program code. The resulting updated version of algorithmic species (C) is based on array reference characterisations. Finally, based on the same theory, the granularity of algorithmic species is altered to obtain the finer-grained SPECIES+ classification (D).

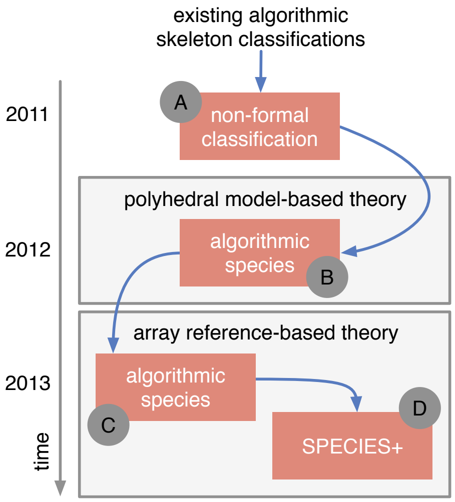

existing algorithmic skeleton classifications
2011
2012
2013
time
A
non-formal classification
polyhedral model-based theory
algorithmic species
B
array reference-based theory
algorithmic species
C
D
SPECIES+

Figure 3.1: A time-line of the algorithm classifications: A an earlier non-formalised classification inspired by algorithmic skeletons, B polyhedral-model based 'algorithmic species', C array reference-based 'algorithmic species', and D the more fine-grained SPECIES+.

This thesis discusses the three most recent versions of our algorithm classifications (B, C and D in figure 3.1). The reason to discuss multiple classifications is twofold: 1) they each have their own advantages and disadvantages, and 2) their succession illustrates some of the design choices made. The original algorithmic species theory is discussed in section 3.2, the revised theory in section 3.3 and the finer-grained SPECIES+ in section 3.4. However, before all this, let us take a look at existing algorithm classifications first.

3.1 A survey of algorithm classifications

With the goals and requirements for an algorithm classification set, we will look at existing algorithm classifications and discuss whether they meet our requirements (although they might have different goals).

Algorithm classifications have been designed in the past for many different purposes, which has resulted in a large body of existing work with many different properties. This work is grouped in four categories, ranging from high abstraction-level classifications (loosely coupled to program code) to low abstraction-level classifications (tightly coupled to program code). The following categories are identified, listed from high to low abstraction-level:

1. High abstraction-level classifications, such as the Berkeley *motifs* [23], pattern languages [87, 97, 98], and the Galois system [114].
2. Algorithmic skeletons [44] and related classifications, such as classical skeletons [41], contemporary skeletons [40, 55], and *idioms* [42].
3. Classifications based on directives or pragma's, typically tightly coupled to program code. Examples are the OpenACC and OmpSs directives.
4. Mathematical representations of code, such as Æcute [77], array regions [46], the polyhedral model [59], and the SUIF loop transformations [145].

The following sections discuss the most prominent work of each category, evaluating the goals and requirements. A number of classifications is also illustrated based on two running examples: matrix-vector multiplication (shown in figure 3.2) and a 2D Jacobi stencil computation (shown in figure 3.3). Table 3.1 and section 3.1.5 provide a summary of our findings. In this section, *italics* are used when introducing classification names and 'quotes' for class names.

3.1.1 High abstraction-level classifications

The 13 *motifs* (originally named *dwarfs*) from Berkeley [23] are an example of high abstraction-level classifications. Motifs are introduced as algorithmic methods to capture patterns of computation and communication. When classifying the two examples, we find that matrix-vector multiplication (figure 3.2) fits the 'dense

```
for (i=0; i<N; i++) {
  r[i] = 0;
  for (j=0; j<N; j++) {
    r[i] += M[i][j] * v[j];
  }
}
```

Figure 3.2: Matrix-vector multiplication ($\vec{r} = \mathbf{M} \cdot \vec{v}$).

```
for (i=1; i<N-1; i++) {
  for (j=1; j<N-1; j++) {
    M[i][j] = 0.2 * (A[i][j]
            + A[i-1][j] + A[i][j+1]
            + A[i+1][j] + A[i][j-1]);
  }
}
```

Figure 3.3: A 2D Jacobi stencil computation.

linear algebra' motif, which is characterised by computations on dense matrices or vectors. The stencil computation of figure 3.3 is classified under the 'structured grids' motif. Motifs are intended to be used for manual, high abstraction-level classification, yielding a coarse-grained intuitive classification, but lacking automated extraction, a formal definition, or a completeness guarantee.

Related to motifs is the work on pattern languages for parallelism [87, 97, 98]. Pattern languages are intended to guide programmers by providing descriptions of frequently occurring problems. They typically provide patterns at multiple levels, but often start at a high abstraction-level. An example is the pattern language OPL [87, 98] that uses motifs (named *computational patterns* in OPL) as a first classification step. A second step involves *structural patterns*, describing the interaction of computational patterns. Assuming that the output matrix **M** from the stencil computation (figure 3.3) is used as input to the matrix-vector multiplication (figure 3.2), the sequence of examples can be classified as a 'pipe-and-filter' structural pattern. The pattern language furthermore provides patterns for parallel programming at different abstraction levels (i.e. algorithm/implementation/execution). Both our examples use the 'data parallelism' *algorithm strategy*, the 'loop parallelism' *implementation strategy*, and the 'SIMD' *parallel execution pattern*. Although more detailed than motifs, OPL is still intended for manual classification purposes, making it unsuitable for our primary goal.

The *Galois* system [114] provides another high abstraction-level classification for manual uses. In contrast to other classifications, Galois is focused entirely on the classification of irregular algorithms (e.g. data dependent memory accesses or graph structures), making it orthogonal to our work.

3.1.2 Algorithmic skeletons and related classifications

Work on *algorithmic skeletons* [44] has led to a large number of algorithm classifications. A summary of *classical* skeletons is found in a skeleton survey [41], listing skeletons such as 'farm', 'pipe', 'fork-join', 'divide and conquer', 'client-server', and 'zip'. This survey of common algorithmic skeletons concludes with a general classification, capturing many skeletons from existing work. This classification is used to evaluate the examples. In the matrix-vector multiplication (figure 3.2), each computation `M[i][j] * v[j]` results in a partial result of a single element of vector $\vec{r}$, requiring recombination. This fits the 'recursively partitioned' or 'divide-and-conquer' skeleton. The stencil computation (figure 3.3) computes a result directly, making it fit the 'task queue' or 'farm' skeleton. Such classical skeletons are very intuitive, but provide no automation, no completeness guarantees, no formal definition, and are too coarse-grained to meet our goals.

Recent *contemporary* skeleton work [40, 55] uses lower abstraction-level classifications. Example skeletons are 'map', 'reduce', 'map-reduce', 'map-overlap', and 'map-array' [55] or 'pixel-to-pixel', 'neighbourhood-to-pixel', 'pixel-to-global', and 'bucket processing' [40]. Related to this contemporary skeleton work are *idioms* [42], a classification system defining 6 classes: 'stream', 'transpose', 'gather', 'scatter', 'reduction', and 'stencil'. When classifying the examples using contemporary skeletons and idioms, following results are found: the 2D Jacobi stencil computation of figure 3.3 classifies as 'map-overlap' [55], 'neighbourhood-to-pixel' [40], or as the equivalent 'stencil' [42]. However, these three classification techniques are unable to classify the full matrix-vector multiplication example, although the example can still be classified partially: the computation in the inner-loop j can be classified as 'reduce' [55], 'scalar reduction' [40] or 'reduction' [42]. Compared to classical skeletons, contemporary skeletons and idioms are already a better fit for our goals: they are formally defined in some cases (e.g. [40]), and occasionally provide tools for automation (e.g. [42]). Nevertheless, we cannot identify a single skeleton classification which fulfils all requirements, lacking aspects such as completeness and a fine granularity.

3.1.3 Directive-based classifications

Compiler directives such as *OpenACC* and *OmpSs* are tightly coupled to program code. Although directives are not strictly considered algorithm classifications, they do have the possibility to capture information of program code. OpenACC for example is used by various compilers to specify regions of code to be offloaded to accelerators. It is used by the PGI Accelerator [146] and HMPP Workbench [52] compilers. As more OpenACC directives are added to program code, an increasing amount of information will become available to the compiler. For example, prefixing '`#pragma acc parallel loop`' to both examples (figures 3.2 and 3.3) will already set a specific 'class'. We conclude that directives, although related to algorithm classifications, are missing a real notion and definition of classes. Furthermore, they lack properties such as automated extraction.

3.1.4 Mathematical code representations

The final group of classifications do not introduce algorithm classes as such, but rather give a mathematical representation of algorithm code. Such mathematical representations typically work on loop nests and represent aspects such as iteration spaces, reuse distances, loop dependences and data locality in a mathematical formulation. Examples are *Æcute* [77], *array regions* [46], the *polyhedral* *model* [59], and the *SUIF loop transformation formulation* [145]. These formulations or models are often used for transformations such as loop tiling and skewing.

Æcute, a decoupled access/execute specification [77], is an algorithm classification that uses a mathematical formulation. It has been used successfully in several works (e.g. [101]). The specification contains a description of the iteration space I, a precedence relationship R to set the order of execution, a partition P to indicate sets of iterations preferably executed on a single processing element, and a set of memory locations that may be read (M_r) or written (M_w) for a given iteration. The Æcute specifications for the two examples in figures 3.2 and 3.3 are shown in equations 3.1 and 3.2 respectively. For similar examples and a detailed explanation we refer to [77]. Although Æcute provides a fine-grained, complete, and formally defined classification, it is currently not automated, and less intuitive compared to classifications with a higher abstraction-level. Furthermore, the formulation of the partition P requires knowledge of the target platform, making its identification less straightforward and, more importantly, not portable across different microprocessors.

$$\begin{aligned}
I &= \{(i,j) : 0 \leq i < N, 0 \leq j < N\} \\
R &= \{((i,j),(i,k)) : 0 \leq i < N, 0 \leq j < k < N\} \\
P &= \Big\{\{(i,j) \in I : a(k-1) \leq i < ak, b(l-1) \leq j < bl\} : \\
&\qquad 1 \leq k < \frac{N}{a}, 1 \leq l < \frac{N}{b}\Big\} \\
M_r(i,j) &= \{M[i][j], v[j] : (i,j) \in I\} \\
M_w(i,j) &= \{r[i] : (i) \in I\}
\end{aligned} \tag{3.1}$$

$$\begin{aligned}
I &= \{(i,j) : 1 \leq i < N-1, 1 \leq j < N-1\} \\
R &= \emptyset \\
P &= \Big\{\{(i,j) \in I : a(x-1) \leq i-1 < ax, b(y-1) \leq j-1 < by\} : \\
&\qquad 1 \leq x < \frac{N-2}{a}, 1 \leq y < \frac{N-2}{b}\Big\} \\
M_r(i,j) &= \{A[i+x][j+y] : (i,j) \in I, -1 \leq x, y \leq 1\} \\
M_w(i,j) &= \{M[i][j] : (i,j) \in I\}
\end{aligned} \tag{3.2}$$

The polyhedral model, another example of a mathematical code representation, captures the iteration space (similar to Æcute's I), the array references (similar to Æcute's M), and the iteration ordering (similar to Æcute's R) of static affine loop nests, i.e. loops with affine array accesses and static and affine loop control. For our examples, we discuss only the iteration space in the form of the domain description $\mathcal{D}$ and the array references in the form of the access function f. For brevity, we only show the domain description for the inner-loops and give

f only for the access `M[i][j]` in line 4 of figure 3.2 and for `A[i][j-1]` in line 5 of figure 3.3. Detailed descriptions for these examples can be found in section 3.2.1 and in [115]. The resulting expressions are equations 3.3 (matrix-vector multiplication) and 3.4 (stencil computation).

$$\mathcal{D} = \{(i,j) \mid 0 \leq i,j < N\} \qquad f(i,j) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \cdot \begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} \tag{3.3}$$

$$\mathcal{D} = \{(i,j) \mid 0 \leq i,j < N\} \qquad f(i,j) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \cdot \begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix} = \begin{pmatrix} i \\ j-1 \end{pmatrix} \tag{3.4}$$

Although it is not strictly an algorithm classification, the polyhedral model does satisfy many of our requirements. It provides automatic extraction from source code, guarantees completeness within pre-defined bounds, obtains the finest granularity possible (it captures all information from the original code), and has a formal base. However, due to the nature of the polyhedral model (or other mathematical code representations for that matter), there is no real notion of classes, making it unsuitable in its current form to directly fulfil our goals.

3.1.5 Evaluation of existing classifications

The results of the classification of the examples are evaluated for different classifications[1]. This is done by assessment with respect to the requirements defined at the beginning of this chapter: classes are required to be 1) automatically extracted, 2) intuitive, 3) formally defined, 4) complete, and 5) fine-grained. The results of the classification of the examples is given in the second and third column of table 3.1, the requirements are evaluated in the remaining columns. The evaluation for the existing classifications is discussed per requirement:

1. Two classifications are identified that do provide **automatic extraction** of classes from program code. This includes an idiom recogniser [42] and several polyhedral model extraction tools, e.g. PET [140]. The reason that most classifications do not provide such tools can be found in the fact that the classes are either not formally defined or are too abstract, making the design of such a tool challenging or even impossible.

2. A large number of classifications, including motifs, pattern languages and skeletons, use descriptive vocabulary as class names. This makes the classification **intuitive** and easy to understand for manual uses. In contrast,

[1]Although Æcute and the polyhedral model are not strictly classifications with individual classes, we do use these notations here to improve readability.

name	example	class	automatic	intuitive	formal def.	complete	granularity
motifs	1) 2)	Dense linear algebra Structured grids	×	✓	×	×	very coarse
OPL	1) 2)	Dense linear algebra, data/loop, SIMD Structured grids, data/loop, SIMD	×	✓	×	×	coarse
skeletons	1) 2)	Recursive partitioned or D&C Task queue or farm	×	✓	×	×	coarse
Enmyren et al.	1) 2)	Partial: reduce Map-overlap	×	✓	×	×	average
Caarls et al.	1) 2)	Partial: scalar reduction Neighbourhood to pixel	×	✓	✓	×	average
idioms	1) 2)	Partial: reduction Stencil	✓	✓	×	×	average
Æcute	1) 2)	See equation 3.1 See equation 3.2	×	×	✓	✓	very fine
polyhedral model	1) 2)	See equation 3.3 See equation 3.4	✓	×	✓	✓	very fine

Table 3.1: Overview of the survey of algorithm classifications. The leftmost columns show the classes for each classification for the two examples of figures 3.2 and 3.3. The rightmost columns show the evaluation of the set requirements for each classification.

the mathematical descriptions provided by Æcute and the polyhedral model are less suited to fulfil goals for which higher abstraction-level classifications (such as motifs, pattern languages, and skeleton classifications) were introduced. The polyhedral model, due to the lack of a notion of classes, will for example not be directly applicable to tasks requiring an intuitive and compact class description, e.g. manual classification or design patterns for programmers.

3. A single skeleton classification, the Æcute classification and the polyhedral model provide **formal definitions** of classes. The other classifications rely on textual descriptions and examples that could lead to errors, ambiguity, and lack of clarity.

4. Classifications such as pattern languages and algorithmic skeletons are often not **complete**. They can be extended with new classes as they evolve, but might still be unable to classify certain algorithms. In contrast, classifications using a mathematical description are complete, at least within pre-defined boundaries. For example, the polyhedral model will be able to generate a domain description, access description, and iteration schedule for every static affine loop nest.

5. The **granularity** of classifications increases gradually as they become less abstract and closer to program code. On one end are the motifs, grouping

algorithms into 13 abstract classes. On the other end are mathematical loop representations, describing data accesses and loop iterations in detail.

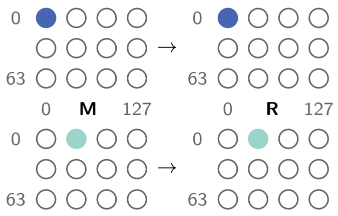
We have seen a large spectrum of classification techniques, each with their own benefits. However, none of the discussed classifications satisfies all requirements. Mathematical representations of program code such as Æcute and the polyhedral model come close, but lack a real notion of classes and thus fall short on intuitiveness, compactness, and ease of understanding. Other classifications such as skeletons or idioms do provide a descriptive classification, but fall short on automation, granularity, and completeness. We therefore conclude that a new classification is required to fulfil our requirements (and to meet the goals).

3.2 Algorithmic species

The previous section has shown that no existing algorithm classification fulfils all our requirements. This section therefore introduces a new classification: 'algorithmic species'. Algorithmic species[2] is a mathematically defined classification targeted at the earlier mentioned goals that builds upon the polyhedral model [59]. The classification defines classes (or: 'species') at a low abstraction-level, classifying array references and parallelism of (nested) static affine loops. The classes themselves are inspired by our earlier work [12] and [17], and by classifications used by compilers based on algorithmic skeletons [40, 55].

Prior to introducing the theory behind algorithmic species (section 3.2.2), we introduce several species informally by classifying the code examples of figures 3.4–3.8. In the figures on the right hand side of the code, different colours denote different iterations of the i-loop.

```
for (i=0; i<64; i++) {
  for (j=0; j<128; j++) {
    R[i][j] = 2 * M[i][j];
  }
}
```



M[0:63,0:127]|element → R[0:63,0:127]|element

Figure 3.4: An embarrassingly parallel algorithm (left), an illustration of the first two i-loop iterations (right), and its algorithmic species w.r.t. the i-loop (bottom).

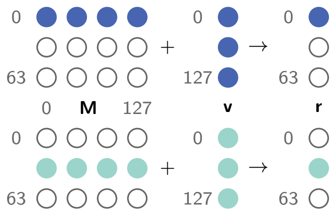
Figure 3.4 gives an example of a loop nest for which all loop iterations can be executed in any order while maintaining correctness. In the body of this example

[2]Algorithmic species (or 'species' in short) is both used as the name of the classification as well as to describe individual classes. Note that 'species' is both the plural and the singular form.

a single *element* from array `M` is read. After multiplying by 2, a single resulting *element* of array `R` is produced. Both arrays are accessed from indices 0 to 63 in the first dimension and from 0 to 127 in the second dimension. When this data is combined with the names of the arrays, the result as shown in the bottom of the figure is obtained. This result can be interpreted as: on every iteration of the first dimension (0 to 63) and second dimension (0 to 127) a different element is needed from the input array `M` to produce a different element of the output `R`.

A second example covers the matrix-vector multiplication code found in figure 3.5 (equal to figure 3.2 apart from the loop bounds). Here, production of a single element of `r` requires an entire row of array `M` and the complete array `v`. These accesses are identified as: *chunk* for the row access of `M` and *full* for the complete access of `v`. The resulting algorithmic species can be interpreted as: to produce a single element out of the total 64 elements in `r`, the entire array `v` and a chunk of data in the second dimension of `M` (a row) are needed.

```
  for (i=0; i<64; i++) {
S:  r[i] = 0;
    for (j=0; j<128; j++) {
T:    r[i] += M[i][j] * v[j];
    }
  }
```

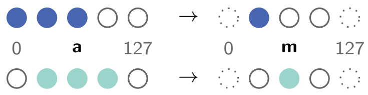


```
M[0:63,0:127]|chunk(-,0:127) ∧ v[0:127]|full → r[0:63]|element
```

Figure 3.5: Matrix-vector multiplication, in essence equal to figure 3.2 (left), an illustration of the first two *i*-loop iterations (right), and its algorithmic species w.r.t. the *i*-loop (bottom).

Next, let us take a look at the 1D stencil computation given in figure 3.6 (a 1D version of the 2D Jacobi stencil of figure 3.3). To produce a single *element* of array `m`, a *neighbourhood* of 3 elements from `a` is needed. A *chunk* access and a *neighbourhood* access differ from each other in the fact that the latter implies overlap between subsequent iterations, as is the fact for the stencil example. The full classification can be found in the bottom of the figure, in which the size of the neighbourhood is also given: ranging from -1 to 1.

```
for (i=1; i<128-1; i++) {
  m[i] = 0.33 * (a[i-1]+a[i]+a[i+1]);
}
```

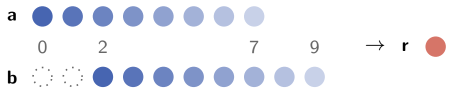


```
a[1:126]|neighbourhood(-1:1) → m[1:126]|element
```

Figure 3.6: A 1D version of the 2D Jacobi stencil computation from figure 3.3 (left), an illustration of the first two *i*-loop iterations (right), and its algorithmic species (bottom).

In the reduction example of figure 3.7, we can see that for every input *element* of arrays a and b a contribution to the result r is made. Since this is only a partial contribution, the result is denoted as *shared*. The classification also captures the offset access to array b by specifying the range 2–9.

```
for (i=0; i<8; i++) {
  r[0] += a[i] + b[i+2];
}
```

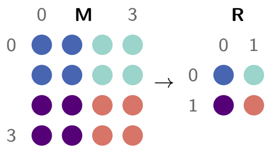


```
a[0:7]|element ∧ b[2:9]|element → r[0:0]|shared
```

Figure 3.7: An example of a reduction to a scalar value (left), an illustration of all 8 *i*-loop iterations (right), and its algorithmic species (bottom).

Finally, the example of figure 3.8 is classified. In this example, a 2x2 tile from M is required to produce a single output in R. The tile is classified as a two-dimensional *chunk* access, resulting in the classification as shown in the figure.

```
for (i=0; i<2; i++) {
  for (j=0; j<2; j++) {
    R[i][j] = M[2*i  ][2*j  ] +
              M[2*i+1][2*j  ] +
              M[2*i  ][2*j+1] +
              M[2*i+1][2*j+1];
  }
}
```



```
M[0:3,0:3]|chunk(0:1,0:1) → R[0:1,0:1]|element
```

Figure 3.8: An example of a 2D *chunk* access (left), an illustration of all 4 *i*-loop iterations (right), and its algorithmic species w.r.t. the *i*-loop (bottom).

3.2.1 Background: the polyhedral model

The polyhedral model [59] is a popular framework to perform code optimisations and parallelisation. It allows program code (most notably *loop nests*) to be represented as algebraic expressions, capturing the iteration spaces of loops, the array references, and the execution schedules. The strength of the polyhedral model is the availability of powerful transformations and analysis passes that can be applied to the program code's algebraic representations. Examples of transformations include loop tiling, skewing, re-ordering and fusion [115].

Because the polyhedral model is an abstract model, it is not able to represent all types of program code exactly: its algebraic expressions are an abstract representation to which restrictions apply, supporting only '*static affine loop nests*'. Static affine loop nests are loop nests for which: 1) loop control is static, 2) loop bounds are affine combinations of constants and static variables[3], 3) conditional

[3]Static variables are variables that remain constant throughout the execution of the loop.

statements are affine combinations of constants, loop variables and static variables, and 4) array references are affine combinations of constants, loop variables and static variables.

The polyhedral model is used as a basis to construct a formal theory for algorithmic species. This model is suitable because it gives us a strong formal basis and is (like our classification) focussed on loop nests in program code. Furthermore, there are already tools available to extract a polyhedral representation from program code (e.g. PET [140]). For our purposes, we use only the polyhedral's *domain descriptions*, capturing the iteration space, and its *access functions*, capturing the array references. Therefore, this section illustrates these domain descriptions $\mathcal{D}$ and access functions f, using matrix-vector multiplication as an example.

In the matrix-vector multiplication example (figure 3.5) two statements are identified: S in line 2 and T in line 4. For the statement S, the domain $\mathcal{D}_S$ is described as $\{i \mid 0 \leq i \leq 63\}$, which can be written in homogeneous coordinates as shown in equation 3.5. The array reference to array `r` in this statement is expressed as `r[`$f(\vec{t_S})$`]`, for which $\vec{t_S}$ is a vector[4] containing the loop iterators at statement S, any loop bound variables, and the constant 1. The access function $f(\vec{t_S})$ for this reference is given in equation 3.6.

$$\mathcal{D}_S = \mathbf{D}_S \cdot \vec{t_S} = \begin{bmatrix} 1 & 0 \\ -1 & 63 \end{bmatrix} \cdot \begin{pmatrix} i \\ 1 \end{pmatrix} \geq \vec{0} \tag{3.5}$$

$$f(\vec{t_S}) = \mathbf{F}_{S,r} \cdot \vec{t_S} = \begin{bmatrix} 1 & 0 \end{bmatrix} \cdot \begin{pmatrix} i \\ 1 \end{pmatrix} = (i) \tag{3.6}$$

Similar to statement S, the domain description $\mathcal{D}_T$ can be described for statement T as $\{(i,j) \mid 0 \leq i \leq 63 \ \wedge \ 0 \leq j \leq 127\}$ or as shown in equation 3.7. Furthermore, the reference to array `M[`$f(\vec{t_T})$`]` can be described as shown in equation 3.8. For the reference to array `M` a vector of length 2 is obtained, corresponding to the two-dimensional reference to array `M` in statement T. For brevity, the access functions for arrays `r` and `v` in statement T are omitted.

$$\mathcal{D}_T = \mathbf{D}_T \cdot \vec{t_T} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 0 & 63 \\ 0 & 1 & 0 \\ 0 & -1 & 127 \end{bmatrix} \cdot \begin{pmatrix} i \\ j \\ 1 \end{pmatrix} \geq \vec{0} \tag{3.7}$$

$$f(\vec{t_T}) = \mathbf{F}_{T,M} \cdot \vec{t_T} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \cdot \begin{pmatrix} i \\ j \\ 1 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} \tag{3.8}$$

Next, we discuss the references to another array `M` of an earlier example: the embarrassingly parallel program of figure 3.4. When the polyhedral model's domain description and access function are derived for the reference to `M` in this

[4]Although the vector $\vec{t}$ is typically named $\vec{x}$ in polyhedral representations, we use this notation to prevent confusion with a vector $\vec{x}$ introduced as part of algorithmic species. Similarly, $\mathbf{D}_S$ is used rather than $\mathbf{A}_S$ for the domain descriptions.

example, we observe that they are equal to those found for the matrix-vector multiplication example: equations 3.7 and 3.8. In other words, because the reference `M[i][j]` uses the same indices in both examples and the loop bounds are equal, the polyhedral model describes both references equally.

When evaluating these results with respect to the algorithmic species classifications of figures 3.4 and 3.5, we observe that the references to array `M` are classified distinctly: either as *element* or as *chunk*. The reason for a distinguished classification can be found in the fact that references to array `r` in the matrix-vector multiplication of figure 3.5 impose data dependences, limiting the parallelism in `M`. The polyhedral model does not expose this directly, because its access functions describe individual array references in isolation of others, whereas algorithmic species describes the references in relation to the entire loop nest. To be able to distinguish between the references to array `M` in these examples, a new representation of array references is introduced to formalise algorithmic species.

3.2.2 Polyhedral model-based algorithmic species

This section describes the polyhedral model-based theory of algorithmic species. The goal of this theory is to formally define the algorithmic species belonging to a given piece of code. As we have seen in the illustrating example, algorithmic species (or 'species' for short) are constructed from a combination of descriptive keywords called 'array access patterns'[5]. A total of 5 patterns are introduced, namely *element*, *chunk*, *neighbourhood*, *shared* and *full*. These access patterns are applied to individual array references, combining them will yield an algorithmic species, typically covering a complete loop nest. Key to our approach is using access patterns as building blocks, which allows us to form many different species using only a very limited set of access patterns.

Earlier, section 3.2.1 argued that the access functions provided by the polyhedral model do not directly lead to access patterns: two examples with the same access functions had different access patterns because of loop-iteration dependences. Therefore, we begin describing the species theory by defining different loop types and a different notation for access functions and domain descriptions.

Base loops and structure loops

The vector $\vec{t}$ as used in the polyhedral model is split to obtain two types of loops: *base loops* ($\vec{x}$) and *structure loops* ($\vec{y}$). Intuitively, structure loops can be seen as inner-loops distorting a one-to-one reference-to-iteration mapping of arrays in the outer base-loops, creating accesses of *structures* (e.g. rows, columns, tiles or neighbourhoods). After constructing a species (when the parallelisation requirements are met), base loops are loops containing iterations that can be

[5]Note that we do not use the word 'patterns' in 'array access patterns' to imply a relation with patterns in pattern languages.

safely executed in parallel on a shared memory microprocessor. In the matrix-vector multiplication example (figure 3.5), the j-loop is an example of a structure loop and the i-loop is an example of a base loop.

More formally, a structure loop is defined as a loop with at least one read and write containing either: 1) a read that is dependent on the loop iterator (i.e. occurs in the index expression) while all its writes are not, or 2), a write that is dependent on the loop iterator while all its reads are not. All other loops which contain at least an array reference are considered base loops. A procedure to derive this is given in algorithm 3.1.

ALGORITHM 3.1: Algorithm to derive base loops and structure loops.

```
   Input: Polyhedral description of all statements $\mathcal{S}$ in the loop body
   Output: Base loop vector $\vec{x}$ and structure loop vector $\vec{y}$
1  $\vec{x} = \emptyset; \vec{y} = \emptyset$
2  for each statement $S$ in $\mathcal{S}$ do
3      $\vec{t'}_S \leftarrow \vec{t}_S$ without constants
4      $n = length(\vec{t'}_S)$
5      for all array references $a$ in $S$ do
6          $\mathbf{F}'_{S,a} \leftarrow$ the first $n$ columns of $\mathbf{F}_{S,a}$
7      end
8      $\vec{R}_S \leftarrow$ projection of $\mathbf{F}'_{S,a}$ into a row vector, for which $a$ are reads
9      $\vec{W}_S \leftarrow$ projection of $\mathbf{F}'_{S,a}$ into a row vector, for which $a$ are writes
10     if $\vec{R}_S = \vec{0}$ or $\vec{W}_S = \vec{0}$ then
11         continue;
12     end
13     for $i$ in $[0 : n-1]$ do
14         if $(R_{S,i} \neq 0$ and $W_{S,i} = 0)$ or $(R_{S,i} = 0$ and $W_{S,i} \neq 0)$ then
15             $\vec{y} \leftarrow \vec{y} \bigcup \vec{t}_{S,i}$
16         end
17     end
18     $\vec{x} \leftarrow \vec{x} \bigcup (\vec{t}_S \bigcap \vec{y})$
19 end
   Result: $\vec{x}, \vec{y}$
```

The matrix-vector multiplication example (figure 3.5) is used to illustrate algorithm 3.1. First, the modified vectors $\vec{t'}$ are constructed by removing any constants from $\vec{t}$ and update the access matrices $\mathbf{F}$ accordingly, creating $\mathbf{F}'$. The results for statements S and T of the matrix-vector multiplication (figure 3.5) are shown in equations 3.9 (for S) and 3.10 (for T).

$$\vec{t'}_S = \begin{pmatrix} i \end{pmatrix}, \qquad \mathbf{F}'_{S,r} = \begin{bmatrix} 1 \end{bmatrix} \tag{3.9}$$

$$\vec{t'}_T = \begin{pmatrix} i \\ j \end{pmatrix}, \quad \mathbf{F}'_{T,r} = \begin{bmatrix} 1 & 0 \end{bmatrix}, \quad \mathbf{F}'_{T,M} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \mathbf{F}'_{T,v} = \begin{bmatrix} 0 & 1 \end{bmatrix} \tag{3.10}$$

Since statement S does not have any reads, no structure loops can be derived from S. As for statement T, we can proceed with algorithm 3.1 and construct the vectors $\vec{R}_T$ and $\vec{W}_T$. They are constructed by accumulating all values per column (projection) of all $\mathbf{F}'_{T,a}$ matrices for which a is a read ($\vec{R}_T$) or a write ($\vec{W}_T$). The results are shown in equation 3.11.

$$\vec{R}_T = \begin{pmatrix}1 & 1\end{pmatrix} + \begin{pmatrix}0 & 1\end{pmatrix} = \begin{pmatrix}1 & 2\end{pmatrix}, \qquad \vec{W}_T = \begin{pmatrix}1 & 0\end{pmatrix} \tag{3.11}$$

When we look at the loop iterator i, and thus at the first column of $\vec{R}_T$ and $\vec{W}_T$, we see no values equal to zero ($R_{T,0} = W_{T,0} = 1$). However, when looking at the second loop iterator j, we can find a read that is dependent on the loop iterator ($R_{T,1} \neq 0$) while all writes are independent ($W_{T,1} = 0$). Thus, j is identified as a structure loop. Therefore, $\vec{x} = (i)$ and $\vec{y} = (j)$ is the result of algorithm 3.1 for the matrix-vector multiplication example.

Domain descriptions and access functions

Now that base loops have been distinguished from structure loops, the polyhedral model's iteration domains $\mathcal{D}$ and access functions $f(\vec{t})$ are modified. In the original description, the domain description $\mathcal{D}$ and the access function $f(\vec{t})$ are given in the form of $\mathbf{D} \cdot \vec{t}$ and $\mathbf{F} \cdot \vec{t}$ respectively. Because the vector $\vec{t}$ is split into $\vec{x}$, $\vec{y}$ and a separate constant vector, a new definition of domain descriptions and access functions is required. These new descriptions will allow us to directly determine which of our 5 access patterns corresponds to a given array reference.

Because the loop iterators are split into two vectors, the domain descriptions are also split into two parts: one part describes the domain imposed by the base loops ($\mathcal{D}_x$) and another part describes the domain imposed by the structure loops ($\mathcal{D}_y$). To illustrate this, statement T of the matrix-vector multiplication code (see figure 3.5) is taken as an example. The domain description in the polyhedral model of our example (equation 3.7) are split into two new descriptions (equation 3.12).

$$\mathcal{D}_{x,T} = \begin{bmatrix}1 & 0\\ -1 & 63\end{bmatrix} \cdot \begin{pmatrix}i\\1\end{pmatrix} \geq \vec{0} \qquad \mathcal{D}_{y,T} = \begin{bmatrix}1 & 0\\ -1 & 127\end{bmatrix} \cdot \begin{pmatrix}j\\1\end{pmatrix} \geq \vec{0} \tag{3.12}$$

Next, the polyhedral model's access functions $f(\vec{t})$ are changed. Two new matrices are introduced: $\mathbf{A}$ represents the relation between the array indices and the base loop iterators $\vec{x}$, while $\mathbf{B}$ gives the relation between the array indices and the structure loop iterators $\vec{y}$. Additionally, $\vec{c}$ is introduced to set a constant offset. Together, this creates an access function $\vec{I}$ as shown in equation 3.13 capturing the array index, written as $\vec{I}_a$ for array reference a. Note that in this access function the loop iterators $\vec{x}$ and $\vec{y}$ are valid for an entire loop body instead of for an individual statement as is done for $\vec{t}$ in the polyhedral model.

$$\vec{I}_a = \mathbf{A}_a \cdot \vec{x} + \mathbf{B}_a \cdot \vec{y} + \vec{c}_a \tag{3.13}$$

To illustrate the new access function, we show examples for statement T of the matrix-vector multiplication (figure 3.5). We show the accesses to arrays `r` (equation 3.14), `M` (equation 3.15) and `v` (equation 3.16). In this example, we see that the accesses to arrays `r` and `v` are one-dimensional (matrices with a single row), while the reference to array `M` is two-dimensional (matrices with two rows).

$$\mathbf{A}_r = \begin{bmatrix}1\end{bmatrix}, \quad \mathbf{B}_r = \begin{bmatrix}0\end{bmatrix}, \quad \vec{c}_r = \begin{pmatrix}0\end{pmatrix}, \quad \vec{I}_r = \begin{bmatrix}1\end{bmatrix} \cdot \vec{x} + \begin{bmatrix}0\end{bmatrix} \cdot \vec{y} + \begin{pmatrix}0\end{pmatrix} = (i) \tag{3.14}$$

$$\mathbf{A}_M = \begin{bmatrix}1\\0\end{bmatrix}, \mathbf{B}_M = \begin{bmatrix}0\\1\end{bmatrix}, \vec{c}_M = \begin{pmatrix}0\\0\end{pmatrix}, \vec{I}_M = \begin{bmatrix}1\\0\end{bmatrix} \cdot \vec{x} + \begin{bmatrix}0\\1\end{bmatrix} \cdot \vec{y} + \begin{pmatrix}0\\0\end{pmatrix} = \begin{pmatrix}i\\j\end{pmatrix} \tag{3.15}$$

$$\mathbf{A}_v = \begin{bmatrix}0\end{bmatrix}, \quad \mathbf{B}_v = \begin{bmatrix}1\end{bmatrix}, \quad \vec{c}_v = \begin{pmatrix}0\end{pmatrix}, \quad \vec{I}_v = \begin{bmatrix}0\end{bmatrix} \cdot \vec{x} + \begin{bmatrix}1\end{bmatrix} \cdot \vec{y} + \begin{pmatrix}0\end{pmatrix} = (j) \tag{3.16}$$

With the new access function defined, the problem of distinguishing the array references of the matrices `M` in the examples of figures 3.4 and 3.5 is re-evaluated. Although the indices are the same for both examples (`M[i][j]`) and the polyhedral model gives the same description, the new access function does give a distinguishing description. The access function for array `M` in the matrix-vector multiplication example (figure 3.5) was already given in equation 3.15, while the access function for array `M` in the example of figure 3.4 is given in equation 3.17. The difference in representation is a result of the fact that $\vec{x}$ contains both loop iterators for the example of figure 3.4, but only the outer loop iterator for the matrix-multiplication example.

$$\mathbf{A}_M = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}, \mathbf{B}_M = \vec{c}_M = \begin{bmatrix}0\\0\end{bmatrix}, \vec{I}_M = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix} \cdot \vec{x} + \begin{bmatrix}0\\0\end{bmatrix} \cdot \vec{y} + \begin{pmatrix}0\\0\end{pmatrix} = \begin{pmatrix}i\\j\end{pmatrix} \tag{3.17}$$

Now, when the descriptions for array `M` in equations 3.15 and 3.17 are compared, different matrices **A** and **B** are found, allowing to distinguish between the *element* and *chunk* patterns. Next, it is shown how the matrices **A** and **B** are used to derive access patterns.

Deriving the access patterns

The total of five access patterns can be derived based on the domain descriptions and access functions. The patterns are given descriptive names to create an intuitive and easy to understand classification. Algorithm 3.2 formally defines these patterns based on the earlier defined matrices **A** and **B** and the domain descriptions ($\mathcal{D}_x$, $\mathcal{D}_y$) for a given array reference in a given statement. Intuitively, the algorithm defines the *element* pattern as references only dependent on base loops, the *full* pattern as references only dependent on structure loops, the *chunk* and *neighbourhood* patterns as references dependent on both types of loops[6], and the *shared* pattern as references independent of both base loops and structure loops. In other words, the patterns cover all four permutations of dependences on the two loop types. The additional fifth pattern is the result of the distinction between overlap among iterations for the *neighbourhood* and *chunk* patterns. In contrast to the *chunk* pattern, the *neighbourhood* pattern does allow (partial) overlap of array references among different iterations. Partial overlap is defined to exist when equation 3.18 holds.

$$\begin{aligned}\exists(\vec{x}_1,\vec{y}_1),(\vec{x}_2,\vec{y}_2) \quad \textit{such that} \quad \mathbf{A}_a\cdot\vec{x}_1+\mathbf{B}_a\cdot\vec{y}_1=\mathbf{A}_a\cdot\vec{x}_2+\mathbf{B}_a\cdot\vec{y}_2,\\ \textit{with } (\vec{x}_1\neq\vec{x}_2\vee\vec{y}_1\neq\vec{y}_2) \textit{ and } (\vec{x}_1,\vec{x}_2)\in\mathcal{D}_{x,S} \textit{ and } (\vec{y}_1,\vec{y}_2)\in\mathcal{D}_{y,S}\end{aligned} \tag{3.18}$$

ALGORITHM 3.2: Algorithm to derive the access patterns for an array.

```
Input: Access description matrices ($\mathbf{A}_a$, $\mathbf{B}_a$) and domain descriptions ($\mathcal{D}_{x,S}$,
       $\mathcal{D}_{y,S}$) for array reference $a$ in statement $S$
Output: Access pattern $\mathcal{P}_a$ for array reference $a$
1  if ($\boldsymbol{B}_a = 0$) then
2      if ($\boldsymbol{A}_a = 0$) then
3          $\mathcal{P}_a \leftarrow$ "shared"
4      else
5          $\mathcal{P}_a \leftarrow$ "element"
6      end
7  else
8      if ($\boldsymbol{A}_a = 0$) then
9          $\mathcal{P}_a \leftarrow$ "full"
10     else if (equation 3.18 holds, i.e. there is partial overlap) then
11         $\mathcal{P}_a \leftarrow$ "neighbourhood"
12     else
13         $\mathcal{P}_a \leftarrow$ "chunk"
14     end
15 end
Result: $\mathcal{P}_a$
```

The algorithmic species' access patterns allow base loops to be safely executed in any order, or even in parallel on a shared memory microprocessor. However, algorithm 3.2 does not include a check for dependences between iterations of base loops. Therefore, array references are further required to have no self-dependences. Self-dependences can exist within a single statement (e.g. `A[i+1] = A[i]`) or among different statements in a loop. Allen and Kennedy [19] state that, when there are no self-dependences, loops can be vectorised or parallelised. Their theory can be applied to species: no single array element is allowed to be

[6]Note that structure loops might be formed from manually unrolled 'loops' such as the accesses to array `a` in figure 3.6. This will be further discussed in section 3.2.3.

read ($\vec{I}_{a,r}$) and written ($\vec{I}_{a,w}$) or written twice in different base loop iterations. This is described formally in equation 3.19, which only holds if there are no dependences. Section 3.2.3 discusses implementation of this equation.

$$\begin{gathered}\nexists(\vec{I}_{a,r}(\vec{x}_1,\vec{y}_1) = \vec{I}_{a,w}(\vec{x}_2,\vec{y}_2)) \quad \textit{and} \quad \nexists(\vec{I}_{a,w}(\vec{x}_1,\vec{y}_1) = \vec{I}_{a,w}(\vec{x}_2,\vec{y}_2)),\\ \textit{with } (\vec{x}_1 \neq \vec{x}_2 \vee \vec{y}_1 \neq \vec{y}_2) \textit{ and } (\vec{x}_1,\vec{x}_2) \in \mathcal{D}_{x,S} \textit{ and } (\vec{y}_1,\vec{y}_2) \in \mathcal{D}_{y,S}\end{gathered} \tag{3.19}$$

Deriving access ranges of algorithmic species

So far, we have described new domain descriptions and access functions, which allowed us to define the algorithmic species' access patterns. This section enriches the species by including access ranges of the arrays as well as access ranges of the *chunk* and *neighbourhood* patterns.

To calculate the range of a chunk or neighbourhood access, information is required from the domain of the structure loop $\mathcal{D}_y$, the structure loop access matrix $\mathbf{B}$, and the constant offset vector $\vec{c}$. The procedure to derive the structure range is first illustrated step-by-step using statement T of the matrix-vector multiplication example (figure 3.5). The goal is to derive the access range and shape of the *chunk* access pattern for array `M`, which is a 1 by 128 row access. The domain of the structure loop is given in equation 3.20, with the loop bounds visible in the last column of $\mathbf{D}_{y,T}$. Due to the nature of the domain description, the value of the lower bound in $\mathbf{D}_y$ is negated. This value is inverted using a helper matrix $\mathbf{H}_{inv}$. Furthermore, a helper vector $\vec{h}$ is introduced, allowing selection of the last row. This allows the loop bounds to be found as shown in equation 3.21.

$$\mathcal{D}_{y,T} = \mathbf{D}_{y,T} \cdot \begin{pmatrix} j \\ 1 \end{pmatrix} \geq \vec{0} \quad \textit{with} \quad \mathbf{D}_{y,T} = \begin{bmatrix} 1 & 0 \\ -1 & 127 \end{bmatrix} \tag{3.20}$$

$$\vec{h} \cdot \mathbf{D}_{y,T}^{\top} \cdot \mathbf{H}_{inv} = \begin{pmatrix} 0 & 1 \end{pmatrix} \cdot \begin{bmatrix} 1 & -1 \\ 0 & 127 \end{bmatrix} \cdot \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{pmatrix} 0 & 127 \end{pmatrix} \tag{3.21}$$

To complete the access range calculation, we process the stride of the access between successive structure loop iterations ($\mathbf{B}$) and the access offset shifting the range with a constant amount ($\vec{c}$). The computation of the access range of the *chunk* pattern for array `M` in the example is shown in equation 3.22. The result can be interpreted as follows: each row corresponds to a dimension in the array reference, the first column denotes the start of the range, and the second the end. The ranges as defined for *chunk* and *neighbourhood* patterns are given with respect to the base loop iterators. As shown in equation 3.22 for the example, array `M` is accessed as a *chunk* only in the second dimension (i.e. a row access), ranging from 0 up to and including 127.

$$\mathbf{B}_M \cdot (\vec{h} \cdot \mathbf{D}_{y,T}^{\top} \cdot \mathbf{H}_{inv}) + \begin{pmatrix} \vec{c}_M & \vec{c}_M \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \cdot \begin{pmatrix} 0 & 127 \end{pmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 127 \end{bmatrix} \tag{3.22}$$

Formally, the access ranges $\mathcal{S}$ of chunk and neighbourhood patterns are computed as shown in equation 3.23. In this equation, $\mathbf{B}_a$ represents the array description for array a in statement S and $\mathbf{D}_{y,S}$ the domain description for S. The helper $\vec{h}$ is a vector of length $L+1$ with a 1 in the last position and zeroes elsewhere, where L is equal to the number of structure loop variables. The helper $\mathbf{H}_{inv}$ is equal in size to $\mathbf{D}_{y,S}^{\top}$ and formed of 2 by 2 diagonal matrices with -1 in the top left corner and +1 in the bottom right corner, as shown in the example of equation 3.21. Furthermore, $\mathbf{C}_a$ is a matrix with an equal number of columns as $\mathbf{D}_{y,S}^{\top}$, each containing the $\vec{c_a}$ vector.

$$\mathcal{S}_a = \mathbf{B}_a \cdot (\vec{h} \cdot \mathbf{D}_{y,S}^{\top} \cdot \mathbf{H}_{inv}) + \mathbf{C}_a \tag{3.23}$$

Finally, the access ranges $\mathcal{R}$ of the array references are discussed. The definitions and computations of access ranges are similar to those of the *chunk* or *neighbourhood* ranges, with the only difference that both matrices ($\mathbf{A}$ and $\mathbf{B}$) and both domain descriptions ($\mathbf{D}_x$ and of $\mathbf{D}_y$) are used. For example, the array access ranges of the arrays `r`, `M` and `s` in figure 3.5 are computed as shown in equations 3.24, 3.25, and 3.26 respectively.

$$\mathbf{A}_r \cdot (\vec{h} \cdot \mathbf{D}_{x,T}^{\top} \cdot \mathbf{H}_{inv}) + \mathbf{B}_r \cdot (\vec{h} \cdot \mathbf{D}_{y,T}^{\top} \cdot \mathbf{H}_{inv}) + \begin{bmatrix} \vec{c}_r & \vec{c}_r \end{bmatrix} = \begin{bmatrix} 0 & 63 \end{bmatrix} \tag{3.24}$$

$$\mathbf{A}_M \cdot (\vec{h} \cdot \mathbf{D}_{x,T}^{\top} \cdot \mathbf{H}_{inv}) + \mathbf{B}_M \cdot (\vec{h} \cdot \mathbf{D}_{y,T}^{\top} \cdot \mathbf{H}_{inv}) + \begin{bmatrix} \vec{c}_M & \vec{c}_M \end{bmatrix} = \begin{bmatrix} 0 & 127 \\ 0 & 63 \end{bmatrix} \tag{3.25}$$

$$\mathbf{A}_s \cdot (\vec{h} \cdot \mathbf{D}_{x,T}^{\top} \cdot \mathbf{H}_{inv}) + \mathbf{B}_s \cdot (\vec{h} \cdot \mathbf{D}_{y,T}^{\top} \cdot \mathbf{H}_{inv}) + \begin{bmatrix} \vec{c}_s & \vec{c}_s \end{bmatrix} = \begin{bmatrix} 0 & 127 \end{bmatrix} \tag{3.26}$$

Algorithmic species anatomy

With the access patterns and ranges defined, the complete algorithmic species can be constructed. Arrays a accessed using the *element*, *shared* or *full* patterns are constructed using the array name ($\mathcal{N}_a$), the access range ($\mathcal{R}_a$), and the access pattern ($\mathcal{P}_a$). For the access patterns *chunk* and *neighbourhood*, the additional structure range ($\mathcal{S}_a$) is included. The syntax used is given in equation 3.27 for both types.

$$\begin{array}{lll} \mathcal{N}_a[\mathcal{R}_a] \mid \mathcal{P}_a & if & \mathcal{P}_a \in \{\text{element}, \text{shared}, \text{full}\} \\ \mathcal{N}_a[\mathcal{R}_a] \mid \mathcal{P}_a(\mathcal{S}_a) & if & \mathcal{P}_a \in \{\text{chunk}, \text{neighbourhood}\} \end{array} \tag{3.27}$$

The individual access ranges (for both the arrays and the patterns) are constructed using commas (`,`) to separate dimensions and colons (`:`) to separate the start and end of a range. To complete the algorithmic species, multiple arrays are concatenated with a wedge ($\wedge$), while input and output arrays are separated by an arrow ($\rightarrow$).

Combining the described syntax, algorithmic species can be constructed as shown for the earlier discussed examples. However, the algorithmic species classification can also be used to classify inner loops. An example is the inner loop j (and its body) of the matrix-vector multiplication example (figure 3.5). In that case, i is considered as constant and the j loop is classified as:

```
M[i:i,0:127]|element ∧ v[0:127]|element → r[i:i]|shared
```

Summary of polyhedral model-based species

To conclude the polyhedral model-based theory of algorithmic species, the steps to derive species are summarised: 1) construct the domain descriptions and access functions in the polyhedral model (section 3.2.1), 2) derive the base loops and structure loops (algorithm 3.1), 3) compute the matrices $\mathbf{D}_{x,S}$, $\mathbf{D}_{y,S}$, $\mathbf{A}_a$, $\mathbf{B}_a$, and $\vec{c}$ (section 3.2.2), 4) apply the dependence check (equation 3.19), 5) derive the access patterns (algorithm 3.2), and 6), compute all access ranges (section 3.2.2). Because all these steps are deterministic formal steps, the resulting algorithmic species is formally defined to correspond to its program code and vice versa.

3.2.3 Automatic extraction of species

One of the goals for a classification of program code is to allow automatic extraction of classes where possible. Now that the theory behind algorithmic species has been introduced, the formal definitions can be followed to construct a tool to automatically identify algorithmic species in program code. This section introduces a C language *algorithmic species extraction tool* (ASET) to annotate species as pragma's in program code. Because the algorithmic species theory is based on the polyhedral model, the *polyhedral extraction tool* (PET) [140] is used as a first step. PET is chosen above other polyhedral model extraction tools for its preservation of source line numbering, crucial for code annotations.

ASET is structured according to the algorithmic species theory. The different steps involved are illustrated in figure 3.9 and described as follows:

(a) PET is invoked to extract a polyhedral representation from static affine loop nests in C program code. In this step, artificial structure loops are added if necessary, as will be discussed in more detail.

(b) The base loops ($\vec{x}$) and structure loops ($\vec{y}$) are identified per loop nest, and the corresponding iteration domains ($\mathcal{D}_x$ and $\mathcal{D}_y$) are identified per statement. Furthermore, $\mathbf{A}$, $\mathbf{B}$ and $\vec{c}$ are computed to describe the array references.

(c) The dependence test is applied to determine whether the base loop iterations can safely be executed in any order. This is implemented as a combination of a GCD-test and a Banerjee-test, which will be described in detail.

(d) Algorithm 3.2 is applied to extract the array patterns.

(e) The array and structure access ranges are derived and everything is combined into an algorithmic species.

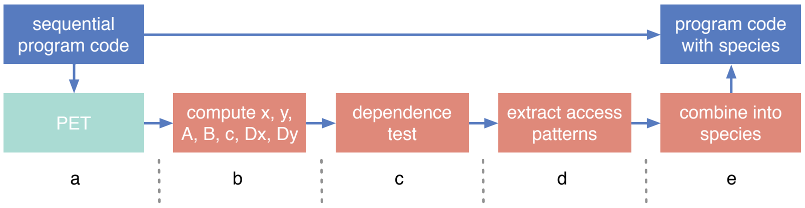


Figure 3.9: Overview of the algorithmic species extraction tool ASET.

With ASET, all loops in static affine program code are classified for which the dependence test of equation 3.19 holds. To limit the number of classifications, the polyhedral extraction tool PET allows the user to delimit source code targeted at parallelisation, thus omitting boundary or debug code such as the initialisation of variables or the printing of results. Because ASET can classify loop nests at each possible level, the resulting program code can have nested classifications. To limit the number of classifications, users can instruct ASET to not further consider loop nests within already classified code.

Creating artificial structure loops

To apply the (loop-based) species theory, program code is required to be structured in a pre-defined manner: using as many loops over array references as possible. This requirement is not imposed on the programmer, but ASET instead performs a temporary code transformation after obtaining the output of PET. For example, the *neighbourhood* references to array `a` in figure 3.10 (left) are explicit rather than embedded in a structure loop. As a result, algorithm 3.2 will not yield the intended results, since the code does not contain any structure loops. In order to overcome this problem, the program code needs to be transformed into the code shown on the right hand side of figure 3.10. Therefore, ASET identifies statements with multiple reads and derives address offsets with respect to the base loop variables (in the example the offsets are -1, 0 and 1). When offsets are consecutive and span a range of two or more, a temporary loop is introduced. In our example, this is achieved by introducing the artificial structure loop k.

Implementing the dependence test

To implement the dependence test (step (c) of figure 3.9) as described in equation 3.19, a comparison between array references of all loop iterations is required. Because this is not scalable, ASET uses a combination of two conservative but

```
for (i=1; i<128-1; i++) {
  m[i] = 0.33 * (a[i-1]+a[i]+a[i+1]);
}
```

```
for (i=1; i<128-1; i++) {
  temp = 0;
  for(k=-1; k<=1; k++) {
    temp += a[i+k];
  }
  m[i] = 0.33 * temp;
}
```

`a[1:126]|neighbourhood(-1:1)` $\rightarrow$ `m[1:126]|element`

Figure 3.10: The 1D Jacobi stencil of figure 3.6 (left), its functionally equivalent transformed version with a structure loop k (right), and the corresponding algorithmic species (bottom).

scalable tests: the GCD-test and the Banerjee-test [88]. The GCD-test (short for greatest common divisor) searches dependences among integer values of the loop variables, but does not consider the loop bounds. As a result, if dependences exist beyond the loop bounds, the GCD-test produces a false positive. In contrast, the Banerjee-test does consider loop bounds, but searches both integer and non-integer values. Again, this can result in a false positive. Both tests are combined within ASET: the GCD-test searches for integer values and the Banerjee-test searches for values within the loop bounds. This combination is sufficient to implement equation 3.19 for affine expressions: an integer value outside the loop bounds and a non-integer value within the loop bounds can only co-exist if multiple solutions exist. Because affine equalities are first-order expressions, they cannot produce multiple solutions. For example, the statement `A[2*i*i+2] = A[5*i]` yields an equality with two solutions: $2 \cdot i^2 + 2 = 5 \cdot i$, but is not affine. Other tests, such as the I-test [88] or the Omega test [118] provide more complex alternatives to our approach.

3.2.4 Evaluation and discussion

The previous sections have introduced the polyhedral model-based theory of algorithmic species. Now, this section evaluates the classification and reflects on the goals and requirements, as has been done for other classifications in section 3.1. The following is discussed in this section: 1) the classification of an entire benchmark suite, 2) the use of species for performance prediction, 3) the limitations of the theory, and 4) an evaluation of the set goals and requirements. For an evaluation of how algorithmic species can help improve the programmability of GPUs, we refer to chapter 4 which will discuss a compiler based on species.

Classification of a benchmark suite

The theory behind algorithmic species and ASET is evaluated by manually and automatically classifying program code from an entire benchmark suite. For this purpose, the PolyBench/C 3.2 benchmark suite [117] is selected, containing 30

Benchmark	Summary of functionality	#species	SLoC	
Linear algebra kernels				
2mm	Two matrix multiplications	2 (+4)	16/16	100%
3mm	Three matrix multiplications	3 (+6)	24/24	100%
atax	Matrix transpose and multiplication	2 (+2)	12/12	100%
bicg	Sub kernel of BiCGStab linear solver	2 (+2)	12/12	100%
cholesky	Cholesky decomposition kernel	2 (+1)	10/14	71%
doitgen	Multi-resolution analysis kernel	2 (+5)	17/17	100%
gemm	Matrix multiplication	1 (+2)	8/8	100%
gemver	Matrix-vector multiplication and addition	4 (+3)	18/18	100%
gesummv	Scalar, vector and matrix multiplication	1 (+1)	9/9	100%
mvt	Matrix-vector product and transpose	2 (+2)	10/10	100%
symm	Symmetric matrix multiplication	1 (+2)	11/13	85%
syr2k	Symmetric rank-2k operations	1 (+2)	9/9	100%
syrk	Symmetric rank-k operations	1 (+2)	8/8	100%
trisolv	Solver for linear triangular systems	1	4/7	57%
trmm	Triangular matrix multiplication kernel	1	3/7	43%
Linear algebra solvers				
gramschmidt	Algorithm for the Gram-Schmidt process	3 (+2)	16/21	76%
durbin	Levinson-Durbin recursion solver	2	6/18	33%
dynprog	2D dynamic programming algorithm	1 (+1)	5/17	29%
lu	LU decomposition kernel	2 (+1)	8/10	80%
ludcmp	LU decomposition kernel	4	12/33	36%
Data-mining				
correlation	Correlation computation kernel	4 (+4)	30/34	88%
covariance	Covariance computation kernel	3 (+3)	19/21	90%
Graph algorithms				
floyd-warshall	Find shortest path in weighted graph	0	0/7	0%
Image processing				
reg_detect	2D Regularity detection algorithm	3 (+2)	14/27	52%
Stencil operations				
adi	Alternating direction implicit solver	3	9/30	30%
fdtd-2d	2D finite different time domain kernel	4 (+3)	20/20	100%
fdtd-2d-apml	2D FDTD using an anisotropic layer	1 (+3)	28/28	100%
jacobi-1d-imper	1D Jacobi stencil computation	2	6/8	75%
jacobi-2d-imper	2D Jacobi stencil computation	2 (+2)	12/12	100%
seidel-2d	2D Seidel in-place stencil operation	0	0/7	0%

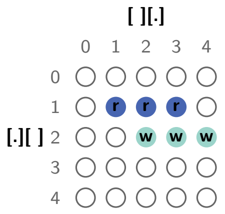
Table 3.2: Classification results for the PolyBench benchmark suite. The third column lists the number of non-nested species and between brackets the additional number of nested species. The last column gives the amount and percentage of classified source lines of code (SLoC).

benchmarks (157 loops in total) from 6 domains in scientific computing. PolyBench is chosen because it contains only program code satisfying the requirements to be representable in the polyhedral model: they contain only static affine loop nests. An overview of the benchmarks in the PolyBench suite is given on the left hand side of table 3.2.

The code in the PolyBench suite has been classified both manually (by hand) and automatically (using ASET), obtaining exactly the same results. However, we note that manual classification initially missed a few species, as it is not always straightforward to manually perform the dependence test of equation 3.19. The results are shown in the last two columns of table 3.2, and the amount of non-nested species in the third column. Additional nested classifications are shown between brackets, i.e. classifications of a loop that are already part of another species. From the total of 30 benchmarks, 28 benchmarks are identified to have loop-parallelism in their current form. All loops with parallelism in these 28 benchmarks are identified by a total of 115 species (not necessarily different from each other), of which 55 species are contained within others, i.e. they are nested species. All other loops do not meet our dependence test of equation 3.19.

From the total of 115 species identified, two examples are discussed in detail for which it is not trivial to see that certain loop iterations can be executed in any order. The left hand side of figure 3.11 first shows the result of the final loop nest of the `reg_detect` benchmark. Using the algorithmic species theory, a species for the i-loop is found as shown in the bottom of the figure: all iterations of the i-loop (lines 2–4) can be executed in any order. The execution of the i-loop is visualised in figure 3.11, showing reads to array `path` as **r** and writes as **w**.

```
1 for (j=1; j<=M-1; j++) {
2   for (i=j; i<=M-1; i++) {
3     path[j][i] = path[j-1][i-1] + mean[j][i];
4   }
5 }
```

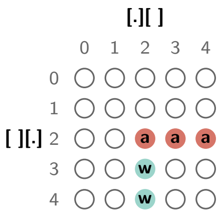


```
path[j-1:j-1,j-1:M-2]|element ∧ mean[j:j,j:M-1]|element →
                path[j:j,j:M-1]|element
```

Figure 3.11: A snippet from PolyBench's `reg_detect` (left), an illustration of references to `path` with `j=2` and `M=5` (right), and the species for lines 2–4 (bottom).

As a second example, the final loop nest of the `covariance` benchmark is discussed, as shown on the left hand side of figure 3.12. By applying the algorithmic species theory, we find that iterations of the $j2$-loop (lines 2–8) can be safely executed in any order. The classification for the $j2$ loop is shown in the bottom of the figure. The resulting species contains an overlapping *full* and *chunk* access: the input array `data` is accessed in two ways, as $(i, j1)$ and as $(i, j2)$. Writes to `symmat` are also performed in two different statements, as visualised on the right hand side of figure 3.12. Here, **a** is used for a write and read, and **w** is used for a write-only access. The visualisation shows why the iterations of the $j2$-loop can be safely executed in any order: every write to a `symmat` element depends only on `symmat` elements either read in the same $j2$ iteration or not read in this iteration of the $j1$-loop at all.

```
for (j1=0; j1<M; j1++) {
  for (j2=j1; j2<M; j2++) {
    symmat[j1][j2] = 0.0;
    for (i=0; i<N; i++) {
      symmat[j1][j2] += data[i][j1]*data[i][j2];
    }
    symmat[j2][j1] = symmat[j1][j2];
  }
}
```



$$\texttt{data[0:N-1,j1:j1]|full} \wedge \texttt{data[0:N-1,j1:M-1]|chunk(0:N-1,-)} \rightarrow$$
$$\texttt{symmat[j1:M-1,j1:j1]|element} \wedge \texttt{symmat[j1:j1,j1:M-1]|element}$$

Figure 3.12: A snippet from PolyBench's `covariance` (left), an illustration of references to `symmat` with `j1=2` and `N=M=5` (right), and the species for lines 2–8 (bottom).

Using species for performance prediction

Performance prediction is one of the possible goals we identified for an algorithm classification. We briefly mention the use of algorithmic species within the '*boat hull model*', a performance prediction technique based on the *roofline model* [144]. The boat hull model is a performance model that does not require program code to be available for a target (parallel) platform, but instead gives a rough estimate based on the algorithmic species identified in sequential code. The boat hull model is able to do this by generating a specific instance of the roofline model for a given algorithmic species: a species-specific roofline model. More details on the boat hull model can be found in [10] and [11].

Restrictions and abstractions

A number of restrictions equal to those of the polyhedral model are imposed upon the algorithmic species theory. First of all, for program code to be represented using our formulations, it must contain affine array accesses and conditions (i.e. expressible in the form of equation 3.6) and static affine loop control (i.e. loop bounds that can be expressed as a system of affine inequalities and do not change throughout the execution of the loop). Furthermore, arrays must be explicitly referenced (Fortran style), e.g. pointer arithmetic is not supported.

Furthermore, we make a remark on the fact that ranges and patterns in algorithmic species are upper-bounds. For example, if an *element* pattern ranges from 0 to 63, it might be possible that elements 10 and 40 are never referenced at all because of an if-condition. Similarly, ranges for *chunk* and *neighbourhood* are upper-bound: partial *chunks* or partial *neighbourhoods* with '*holes*' are still classified as *chunks* or *neighbourhoods*. In general, all accesses described by algorithmic species are '*may-accesses*': they can be predicated by some (loop) condition.

Finally, we note that some access patterns can be seen as special cases of others. For example, a *full* pattern is equal to a *chunk* pattern with a structure access range equal to the array access range. The inclusion of the *full* pattern can therefore be seen as *syntactic sugar*: it helps improve the compactness and readability of the species. Section 3.3 elaborates on such cases.

Evaluation of the goals and requirements

Earlier, section 3.1 has evaluated algorithm classifications against a number of requirements set in this chapter. Now, let us evaluate algorithmic species with respect to these requirements: algorithm classes are required to be 1) automatically extracted, 2) intuitive, 3) formally defined, 4) complete, and 5), fine-grained.

1. Species can be **extracted automatically** from C program code using ASET, a tool that follows directly from the algorithmic species theory. With ASET, a time consuming and error-prone manual classification is no longer required, making species suitable to be integrated into an automatic tool-chain.

2. Algorithmic species uses a very limited vocabulary: species are composed of a combination of only five different patterns. The names of these patterns are descriptive, creating an **intuitive** and easy to use classification. The complete species description forms a high level abstraction of the mathematical representation used in the polyhedral model. This is not only beneficial for manual uses, but can also benefit automatic uses: for example, this creates opportunities to use species for skeleton-based source-to-source compilation (as will be shown in chapter 4) or for performance prediction (e.g. the boat hull model). In both of these examples, prior to this work manual identification of some form of algorithm classes was required.

3. Notwithstanding intuitiveness, algorithmic species has a formal mathematical basis, based on mathematical representations in the polyhedral model. With this formal basis, every class is **formally defined** to correspond to its program code and vice versa. This can be helpful for correctness guarantees and to prevent ambiguity or lack of clarity. In contrast, many high abstraction-level classifications and skeleton classifications provide a textual description or merely a few examples to ‘define’ a particular class.

4. The classification is not **complete** in the sense that it will classify any piece of program code. Nevertheless, it is complete within certain pre-defined limits: any static affine loop nest with loop-parallelism can be classified under the algorithmic species theory. This is in contrast to classifications such as pattern languages or algorithmic skeletons, which require new class definitions to be added as different types of algorithms are evaluated.

5. Species are **fine-grained**. They capture information about the structure and amount of parallelism, memory requirements, atomicity, data-reuse and data-locality. Algorithmic species are finer-grained than related classifications such as skeletons and idioms. For example, in comparison with [40], more information is added, e.g. neighbourhood ranges and array ranges. On

the other hand, when compared to other program code classifications such as the polyhedral model, we find that algorithmic species are coarser-grained, abstracting away from the details.

Since the algorithmic species classification meets our set requirements, it is a candidate to meet our goals. Algorithmic species is a common microprocessor and programming language independent classification that can be applied for the various uses of a whole range of existing classifications. Algorithmic species is evaluated with respect to manual and automatic uses:

- Programmers will be able to use algorithmic species for various tasks, e.g. when developing, mapping, porting, optimising, profiling or debugging code. For example, programmers can use algorithmic species to relate to well known solutions for certain types of problems or to compare implementations with other programmers. If needed, they can omit details such as access ranges and refer to the combination of access patterns only. In this way, algorithmic species can replace existing high abstraction-level classifications such as Berkeley motifs [23] or parallel pattern languages [87]. In contrast to these existing classifications, species have the advantage of among others automatic extraction of classes and a finer-grained classification.

- As for automated uses, source-to-source compilation (see chapter 4) and the boat hull model [11] are identified as two techniques benefiting from the work on algorithmic species. Still, other tools and compilers could build upon our classification, forming algorithmic species into a common basis to capture properties of program code such as the structure of parallelism or the amount of data-reuse. For example, algorithmic species can improve the work build upon existing classifications such as skeletons (e.g [40, 41, 55]), idioms [42], or Æcute [77]. Advantages of species include a formal theory, automated extraction, and fine-grained information.

3.2.5 Conclusions

This section, has introduced 'algorithmic species', a classification of program code based on memory access patterns. The main goal of the classification is to apply structure to the types of program code that are targeted to be accelerated on specialised microprocessors such as GPUs. Algorithmic species provide a microprocessor-agnostic structured classification (or '*summary*') of program code. Programmers or compilers can use this for example to take parallelisation decisions, perform memory hierarchy optimisations, make performance predictions or compare solutions. We have also seen the classification's formal theory, which is based on the polyhedral model, and ASET, a tool to automatically extract species from program code.

Still, two main drawbacks of the presented algorithmic species theory are identified: 1) its applicability is limited to static affine loop nests, and 2) two equal species can still have significantly different memory access patterns (e.g. a tiled loop and its non-tiled counterpart). The first limitation is overcome in section 3.3, in which a new theory for algorithmic species is presented, allowing it to be applied beyond static affine loop nests. Furthermore, the second limitation is overcome in section 3.4, in which a finer-grained version of algorithmic species is presented, named SPECIES+.

3.3 Algorithmic species revisited

The previous section introduced algorithmic species. Because its underlying theory is based on the polyhedral model [59], two of our five requirements are only partially fulfilled: completeness and automatic extraction are only achieved for code that is represented as static affine loop nests. Therefore, this section presents a new theory behind algorithmic species to extend its applicability. Although the new theory is not based on the polyhedral model, it does still respect algorithmic species' vocabulary and intuitive meaning.

The new theory behind algorithmic species is based on characteristics of individual array references. This is motivated by an earlier observation: many of the original access patterns are special cases of others. For example, the *element* pattern can be seen as a *chunk* access pattern of size 1, and the *neighbourhood* pattern can be seen as an *chunk* access pattern with overlap. Therefore, the new theory is based on a single 'unified' access pattern, which can later be transformed into one of the original patterns that now serve the purpose of *syntactic sugar*.

This section first introduces the new unified access pattern: the array reference characterisation. Following, it is shown how these characterisations can be merged and transformed into algorithmic species. Next, a new automatic extraction tool based on the new theory is presented. Finally, the new theory and its extended applicability are evaluated.

3.3.1 Array reference characterisations

The new array reference-based theory for algorithmic species is based on characteristics of individual array references. This section defines this characterisation. For simplicity, only static affine loop nests are considered in the examples of this section. Later, section 3.3.4 will discuss other types of loop nests. In line with the earlier defined algorithmic species theory, our work is restricted to Fortran-style arrays: arrays do not alias and pointer arithmetic is not allowed.

Basic case: single loop and 1-dimensional arrays

For illustration purposes, only loop nests with a single loop and 1-dimensional array references are initially discussed. Later, the general case is considered: loop nests with one or more loops and references to one or multi dimensional arrays.

As an example of a single loop with 1-dimensional references, consider figure 3.13 and the reference to array `A` with respect to the i-loop. This reference can be characterised by its name (A), access type (r for read), domain with respect to the loop (lower-bound 2 and upper-bound 7), per-iteration accessed size (1 element), and its iteration step (1 element). This forms the 5-tuple characterisation $(A, r, [2..7], 1, 1)$. For the reference to `B`, $(B, w, [0..5], 1, 1)$ can be obtained in a similar way.

```
for (i=2; i<8; i++) {
  B[i-2] = A[i];
}
```

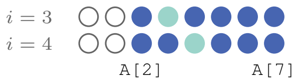


Figure 3.13: Example code with a static affine loop. The picture on the right illustrates with filled circles (each circle is an array element) the domain of `A`: left is `A[0]` and right `A[7]`. The red circles highlight references made in two different iterations of the i-loop (top and bottom).

In general, each array reference in a static affine loop nest can be characterised using such a 5-tuple $\mathcal{R} = (\mathcal{N}, \mathcal{A}, \mathcal{D}, \mathcal{E}, \mathcal{S})$. The tuple's elements are defined as:

- $\mathcal{N}$ is the array's name, given as a string.
- $\mathcal{A} \in (r, w)$ is the access type: r for read and w for write.
- $\mathcal{D} \in [\mathbb{Z}..\mathbb{Z}]$ gives the integer domain of array references with respect to the loop nest, represented as an interval with a lower-bound and upper-bound.
- $\mathcal{E} \in \mathbb{N}$ gives the number of elements accessed. Note that $\mathcal{E} = 1$ unless there is an additional loop inside the body of our reference loop nest (prior to merging as will be discussed in section 3.3.2).
- $\mathcal{S} \in \mathbb{Q}$ gives the step. Note that the step can be negative in case of a backwards counting loop, zero in case of a loop-independent reference, or a unit fraction in case of a non-monotonic step. For example, the fraction $\frac{1}{4}$ represents a step taken every 4 iterations.

To further illustrate the basic characterisation of array references, several other examples are discussed. First, consider the code snippet of figure 3.14. The first loop (lines 1–7) and the second loop (lines 8–10) are functionally equivalent. For both loops, the characterisation $(G, r, [0..4], 1, 2)$ is found for the reference to `G` and $(H, w, [0..2], 1, 1)$ is found for the reference to `H` with respect to the i-loop.

Next, consider the two functionally equivalent loops of figure 3.15. For these loops, the references are constructed with respect to the outer i-loop only. Although the two code snippets are functionally equivalent, the characterisation of the references to array `P` is different. For the first loop (lines 1–6), the reference to `P` is characterised as $(P, r, [0..7], 2, 2)$ and to `Q` as $(Q, w, [0..3], 1, 1)$. Note that a size and step $\mathcal{S} = \mathcal{E} = 2$ is obtained for `P` because of the j-loop. For the second

```
 1 for(i=0; i<8; i++) {
 2    if(i % 2 == 0) {
 3       if(i < 6) {
 4          H[i/2] = G[i];
 5       }
 6    }
 7 }
 8 for(i=0; i<3; i++) {
 9    H[i] = G[i*2];
10 }
```

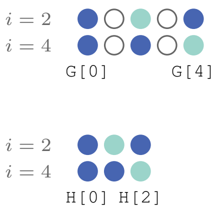


Figure 3.14: Example of two functionally equal loops (left). The reference to the two arrays is illustrated using filled circles, highlighting in red references for two iterations of the i-loop (right). The illustrations are valid for both loops.

```
1 for(i=0; i<4; i++) {
2    Q[i] = 0;
3    for(j=0; j<2; j++) {
4       Q[i] += P[2*i+j];
5    }
6 }
7 for(i=0; i<4; i++) {
8    Q[i] = P[2*i]+P[2*i+1];
9 }
```

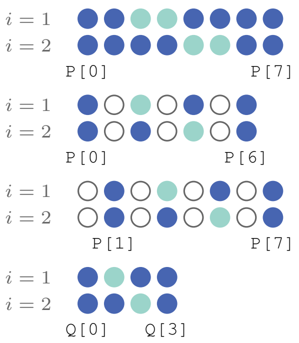


Figure 3.15: Example of two functionally equal loops (left). The right hand side illustrates (top to bottom): the reference to P for the first loop (lines 1–6), the first reference to P for the second loop (lines 7–9), the second reference to P for the second loop, and the reference to Q for both loops.

loop (lines 7–9), two references to P are found, described as $(P, r, [0..6], 1, 2)$ and $(P, r, [1..7], 1, 2)$. The characterisation of Q remains as before. To still obtain the same result, merging of two 5-tuples will be introduced in section 3.3.2.

General case: N loops and M-dimensional arrays

So far, all references have been to 1-dimensional arrays. To be able to characterise M-dimensional arrays, the 5-tuple $\mathcal{R}$ is modified by adding multiple dimensions to the domain $\mathcal{D}$, the number of elements $\mathcal{E}$, and the step $\mathcal{S}$. The different array dimensions are represented as a set with angular brackets, i.e. $\mathcal{D} = \langle \mathcal{D}_1, \mathcal{D}_2, ..., \mathcal{D}_M \rangle$, $\mathcal{E} = \langle \mathcal{E}_1, \mathcal{E}_2, ..., \mathcal{E}_M \rangle$ and $\mathcal{S} = \langle \mathcal{S}_1, \mathcal{S}_2, ..., \mathcal{S}_M \rangle$. The individual $\mathcal{D}_i$, $\mathcal{E}_i$ and $\mathcal{S}_i$ retain their definition as given for the 1-dimensional case. The total number of elements now becomes the product of the number of elements in each dimension. In case there is only a single dimension, the angular brackets are omitted from the notation to improve readability.

To illustrate the modified 5-tuple $\mathcal{R}$, array reference characterisations will be derived for the example code shown in lines 1–5 of figure 3.16. When the characterisation is applied with respect to both loops i and j, the tuples as shown in figure 3.16 are obtained (along with their corresponding algorithmic species). In this characterisation, $\frac{1}{8}$ as a step for `T` represents a unit step every 8 loop iterations, and $\langle\frac{1}{8},1\rangle$ as a step for `R` represents a unit step every 8 iterations in the first dimension, and a unit step each iteration in the second dimension (modulo the domain).

```
1 for(i=0; i<8; i++) {
2   for(j=0; j<8; j++) {
3     S[i*8+j] = R[i][j] + T[i];
4   }
5 }
```

```
6 for(k=0; k<64; i++) {
7   S[k] = R[k/8][k%8] + T[k/8];
8 }
```

$$
\begin{aligned}
(R, r, \langle[0..7],[0..7]\rangle, \langle 1,1\rangle, \langle\frac{1}{8},1\rangle) &\rightarrow \texttt{R[0:7,0:7]|element}\\
(S, w, [0..63], 1, 1) &\rightarrow \texttt{S[0:63]|element}\\
(T, r, [0..7], 1, \frac{1}{8}) &\rightarrow \texttt{T[0:7]|element}
\end{aligned}
$$

Figure 3.16: Example code with multiple loops and multi-dimensional arrays (left), functionally equivalent code with only a single loop (right), and the corresponding 5-tuple characterisations and species (bottom).

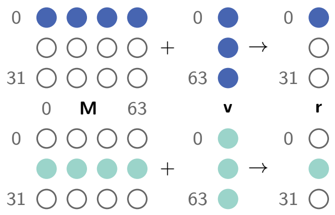
For the functionally equivalent code in lines 6–8 of figure 3.16, the same characterisations are obtained as for lines 1–5. The characterisations remain the same because of an abstraction made by the original theory: the amount nor order of loops are taken into account. This topic is further discussed in section 3.4, where a finer-grained classification is introduced that does not make this abstraction.

The matrix-vector multiplication of figure 3.17 is taken as a second example of multi-dimensional arrays. Characterising this code with respect to the i-loop, gives us the results as shown in the bottom of figure 3.17 (along with the partial species as defined by the polyhedral model-based theory). In the result, the step $\langle 1,0\rangle$ for `M` reflects the fact that references to the second dimension are independent of the i-loop: a whole row of the matrix is accessed at every iteration of the i-loop.

3.3.2 Array reference-based algorithmic species

Now that the array reference characterisations are illustrated (the new 'unified' access pattern), we present the array reference-based theory of algorithmic species. This section contains two main parts: 1) the merging of multiple array reference characterisations, and 2) the translation of merged characterisations into species.

```
for (i=0; i<32; i++) {
  r[i] = 0;
  for (j=0; j<64; j++) {
    r[i] += M[i][j] * v[j];
  }
}
```



$$
\begin{aligned}
(M, r, \langle[0..31],[0..63]\rangle, \langle 1,64\rangle, \langle 1,0\rangle) &\rightarrow \texttt{M[0:31,0:63]|chunk(-,0:63)} \\
(v, r, [0..63], 64, 0) &\rightarrow \texttt{v[0:63]|full} \\
(r, w, [0..31], 1, 1) &\rightarrow \texttt{r[0:31]|element}
\end{aligned}
$$

Figure 3.17: Matrix-vector multiplication, in essence equal to figure 3.5 (left), an illustration of the first two i-loop iterations (right), and its classification (bottom).

Merging array references

Before translating array reference characterisations into algorithmic species, a merging step is performed. This allows us to: 1) form compound access patterns such as *tile* and *neighbourhood*, and 2) abstract away implementation choices (e.g. consider the loop unrolling performed in figure 3.15). As the latter issue was not resolved in the polyhedral model-based theory, ASET was required to perform code transformations before classifying program code. The new theory addresses this issue by defining a merge operation for array reference characterisations.

For a pair of array references $\mathcal{R}_a$ and $\mathcal{R}_b$, merging is only considered when the name is equal ($\mathcal{N}_a = \mathcal{N}_b$), the access type is the same ($\mathcal{A}_a = \mathcal{A}_b$), and the step is equal ($\mathcal{S}_a = \mathcal{S}_b$). We have already seen an example that meets these conditions: the references to array `P` in the second loop (lines 7–9) of figure 3.15. For this example, we want to merge $(P, r, [0..6], 1, 2)$ and $(P, r, [1..7], 1, 2)$ into $(P, r, [0..7], 2, 2)$, such that we obtain the same result as for the characterisation of `P` in the first loop (lines 1–6).

Algorithm 3.3 describes the rules for merging. The algorithm is repeatedly applied until no changes to the set of array references R for a single loop nest are made. It considers each pair $\mathcal{R}_a, \mathcal{R}_b \in R$ and proceeds as follows:

- **[2]** Test the condition of matching name, access type, and step.
- **[3]** Test whether the lengths of the domains are equal and whether the domains intersect.
- **[4–5]** Calculate the new domain $\mathcal{D}_{new}$ as the union $\mathcal{D}_a \cup \mathcal{D}_b$ and the new number of elements $\mathcal{E}_{new}$ as the absolute difference between the bounds of the domains.

- **[6]** Continue only if the new number of elements $\mathcal{E}_{new}$ is not significantly different from the sum of the old number of elements. This is determined by a threshold t_{gap}, which will be discussed in more detail.
- **[7–8]** Replace the tuples $\mathcal{R}_a$ and $\mathcal{R}_b$ with $\mathcal{R}_{new}$.

ALGORITHM 3.3: Merging a pair of array reference characterisations.

```
   Input: array reference characterisations $R$ (w.r.t. a loop nest)
1  foreach $\{\mathcal{R}_a, \mathcal{R}_b\} \in R$ do
2      if $\mathcal{N}_a = \mathcal{N}_b$ and $\mathcal{A}_a = \mathcal{A}_b$ and $\mathcal{S}_a = \mathcal{S}_b$ then
3          if $|\mathcal{D}_a| = |\mathcal{D}_b|$ and $\mathcal{D}_a \cap \mathcal{D}_b \neq \emptyset$ then
4              $\mathcal{D}_{new} = \mathcal{D}_a \cup \mathcal{D}_b$
5              $\mathcal{E}_{new} = |min(\mathcal{D}_a) - min(\mathcal{D}_b)|$
6              if $\mathcal{E}_a + \mathcal{E}_b + t_{gap} > \mathcal{E}_{new}$ then
7                  $\mathcal{R}_{new} = (\mathcal{N}_a, \mathcal{A}_a, \mathcal{D}_{new}, \mathcal{E}_{new}, \mathcal{S}_a)$
8                  replace $\mathcal{R}_a$ and $\mathcal{R}_b$ with $\mathcal{R}_{new}$ in $R$
9              end
10         end
11     end
12 end
```

The merge operator in case of a *neighbourhood* access pattern is illustrated using the example shown in figure 3.18. In this example, 3 references to array V are found and characterised as $\mathcal{R}_{\texttt{V[i-1]}} = (V, r, [0..5], 1, 1)$, $\mathcal{R}_{\texttt{V[i]}} = (V, r, [1..6], 1, 1)$, and $\mathcal{R}_{\texttt{V[i+1]}} = (V, r, [2..7], 1, 1)$. When applying algorithm 3.3, the merged characterisation $(V, r, [0..7], 3, 1)$ is obtained, including a combined domain $\mathcal{D}$ and number of elements $\mathcal{E}$. This merged array reference characterisation captures the *neighbourhood* pattern's overlap between iterations as the number of elements $\mathcal{E}$ (3) is now larger than the step $\mathcal{S}$ (1).

```
for(i=1; i<7; i++) {
  W[i] = V[i-1] +
         V[i] +
         V[i+1];
}
```

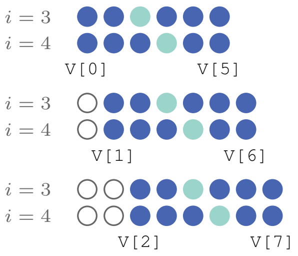


Figure 3.18: Example of an unrolled *neighbourhood* access pattern (left). The right hand side illustrates two iterations of the *i*-loop for the 3 references to V.

Next, the example of interpolation as shown in figure 3.19 is discussed. In this example, the set of read references per iteration is not convex: there is a gap between references `K[i-1]` and `K[i+1]`. For these references, $(K, r, [0..4], 1, 2)$ and $(K, r, [2..6], 1, 2)$ are found prior to merging. Now, these can be treated as two separate *element* accesses, or merged into a *neighbourhood* access. When performing the latter, the over-approximation $(K, r, [0..6], 3, 2)$ is obtained: 3 elements may be accessed each iteration, but only the 2 extreme elements are in fact accessed. Whether or not merging is performed in this case depends on the value of the t_{gap} variable in algorithm 3.3. The polyhedral model-based theory does not discuss the issue of non-convex sets, implying $t_{gap} = \infty$.

```
for(i=1; i<6; i+=2) {
  K[i] = K[i-1] + K[i+1];
}
```

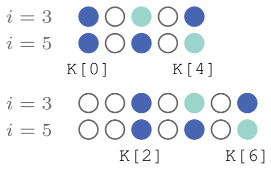


Figure 3.19: Example of an implementation of interpolation (left) and an illustration of two loop iterations for the read references to `K`.

Translating array reference characterisations into species

Once the found array references are merged (where possible), they can be translated into algorithmic species. Species can thus be seen as an abstract and more intuitive representation of a combination of array reference characterisations. For automated use (e.g. in compilers), it might be advantageous to use the characterisations directly, as they could provide additional information. However, for manual use, the intuitive algorithmic species can provide better understandability and usability.

Algorithm 3.4 extracts algorithmic species and their access patterns from array reference characterisations. The algorithm processes each characterisation $\mathcal{R}_a$ after merging as follows:

- [**3**–**6**] If $\mathcal{R}_a$ has a zero step, it belongs either to the *full* (for a read) or *shared* (for a write) pattern.
- [**7**–**8**] Else, if precisely a single element of $\mathcal{R}_a$ is accessed every iteration, it is classified as the *element* pattern.
- [**9**–**10**] Else, if the amount of elements accessed is larger than the step size, there is overlap between iterations. This is captured by the *neighbourhood* pattern.
- [**11**–**13**] If non of the above holds, $\mathcal{R}_a$ belongs to the *chunk* pattern. This is the case when multiple elements are accessed, but there is no overlap between successive iterations.

As shown in algorithm 3.4, the names ($\mathcal{N}_a$) and domains ($\mathcal{D}_a$) are prefixes to the access patterns. For the *neighbourhood* and *chunk* access patterns, the number of elements ($\mathcal{E}_a$) is a suffix. The final algorithmic species (combining all inputs and outputs) is obtained by taking the results of algorithm 3.4 (in X) and combining them as follows: $I_1 \wedge ... \wedge I_P \rightarrow O_1 \wedge ... \wedge O_Q$, in which I_p represents an input ($\mathcal{A}_p = r$) and O_q represents an output ($\mathcal{A}_q = w$).

ALGORITHM 3.4: Extracting species from array references.

```
Input: array reference characterisations $R$ after merging (w.r.t. a loop nest)
1  X = $\emptyset$
2  foreach $\mathcal{R}_a \in R$ do
3      if $\mathcal{S}_a = 0$ and $\mathcal{A}_a = r$ then
4          X ← $\mathcal{N}_a$ $\mathcal{D}_a$ full
5      else if $\mathcal{S}_a = 0$ and $\mathcal{A}_a = w$ then
6          X ← $\mathcal{N}_a$ $\mathcal{D}_a$ shared
7      else if $\mathcal{E}_a = 1$ then
8          X ← $\mathcal{N}_a$ $\mathcal{D}_a$ element
9      else if $\mathcal{S}_a < \mathcal{E}_a$ then
10         X ← $\mathcal{N}_a$ $\mathcal{D}_a$ neighbourhood ($\mathcal{E}_a$)
11     else
12         X ← $\mathcal{N}_a$ $\mathcal{D}_a$ chunk ($\mathcal{E}_a$)
13     end
14 end
```

3.3.3 Automatic extraction of species

Section 3.2.3 introduced ASET, the polyhedral model-based algorithmic species extraction tool. This section presents a new extraction tool that follows from the array reference-based theory of algorithmic species. The new tool is named A-DARWIN[7], short for automatic Darwin[8].

The new tool is largely equal to ASET in terms of functionality (apart from its extended applicability), but is different internally (see figure 3.20). A-DARWIN is based on a C parser, transforming C99 program code into an abstract syntax tree (AST). From the AST, A-DARWIN extracts the array references and their corresponding 5-tuple characterisations $\mathcal{R}$. Next, the tool follows the theory: it applies merging as described in algorithm 3.3 and extracts species as described by algorithm 3.4. Finally, the species are inserted as pragma's into the original program code.

Algorithmic species and A-DARWIN are not intended to be able to perform loop transformations, and can thus not create parallel loops if they are not already present. A parallelising tool (such as Pluto [34]) could be used as a pre-processor

[7]Source-code is available as part of BONES at `http://github.com/cnugteren/bones/`.
[8]A-DARWIN is named after Charles Darwin, the author of *'On the Origin of Species'*.

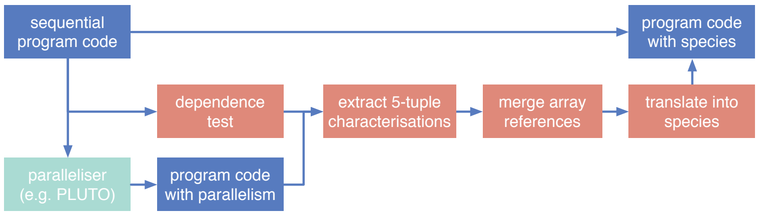

sequential program code
program code with species
dependence test
extract 5-tuple characterisations
merge array references
translate into species
paralleliser (e.g. PLUTO)
program code with parallelism

Figure 3.20: Overview of the algorithmic species extraction tool A-DARWIN.

to create (e.g. through loop skewing) and identify parallel loops (see figure 3.20). In case parallel loops are not identified a-priori, but are already present, A-DARWIN can also perform basic dependence analysis. This allows A-DARWIN to be usable as a stand-alone tool, similar to ASET. For the dependence analysis, Bernstein's conditions are applied first, yielding the pairs $\mathcal{R}_a$ and $\mathcal{R}_b$ for which $\mathcal{N}_a = \mathcal{N}_b$ and $\mathcal{A}_a \neq r \vee \mathcal{A}_b \neq r$. Following, a combination of the GCD and Banerjee tests [88] is applied on the remaining pairs. As already discussed in section 3.2.3, these tests are conservative for non affine equalities: this might not find all parallel loops. More advanced tests such as the I-test [88] or the Omega test [118] could improve the coverage of A-DARWIN.

Static analysis has a limited scope of applicability, as will be discussed in section 3.3.4 in more detail. Because some loop nests cannot be fully analysed, A-DARWIN will classify species in some cases as over-approximations. These over-approximations can currently only be tightened using the programmer's knowledge (manual). Alternatively, after modifications to A-DARWIN, run-time or profiling information could be used (dynamic).

3.3.4 Evaluation and discussion

The main reason to develop the array reference-based theory of algorithmic species was to extend its applicability. So far, all our examples have been of static affine loop nests, fitting the original polyhedral model-based theory as well. This section will therefore first evaluate the new theory on a number loop nests that do not fit the polyhedral model. Furthermore, the possibility of classifying individual statements with array reference characterisations will be shown. Finally, the granularity issue of species will be discussed by evaluating an example.

In line with the evaluation of the polyhedral model-based theory, A-DARWIN and the new theory have also been evaluated on the PolyBench benchmark suite. Since the same results as in section 3.2.4 and table 3.2 are obtained, these results are not further discussed.

Extended applicability

This section discusses a number of examples that violate one of the constraints of static affine loop nests: 1) non-static loop control, 2) non-affine loop bounds, 3) non-affine conditional statements, and 4) non-affine array references. The examples are given in figure 3.21.

```
 1 // Non-static control
 2 i = 0;
 3 while(i < 8) {
 4   B[i] = A[i];
 5   i = i + A[i];
 6 }
 7
 8 // Non-affine bound
 9 for(i=0; i < 8-i*i; i++) {
10   H[0] = G[i];
11 }
```

```
12 // Non-affine condition
13 for(i=0; i < 8; i++) {
14   if (P[i] > 12) {
15     P[i] = 0;
16   }
17 }
18
19 // Non-affine references
20 for(i=0; i < 8; i++) {
21   S[T[i]] = R[i*i];
22 }
```

Figure 3.21: Examples of non static affine loop nests: non-static control (top left), a non-affine bound (bottom left), a non-affine condition (top right), and non-affine references (bottom right).

First, consider the example with non-static control in lines 1–6 of figure 3.21. In every iteration of the loop, i is incremented by a value dependent on the memory state. This leads to a case where the iteration step $\mathcal{S}$ is unknown at compile-time and might not even be constant. In this case, the references are not characterised. This form of loop-carried dependence is also significantly limiting the possibilities of parallelisation, in particular for GPUs.

Next, consider the example with a non-affine loop bound (lines 8–11, figure 3.21). Although the loop bound is not affine, the upper-bound can still be found at compile-time for this example ($i \leq 2$). Doing so, $(G, r, [0..2], 1, 1)$ and $(H, w, [0..0], 1, 0)$ are obtained. However, deriving the domain $\mathcal{D}$ for a loop with non-affine loop bounds is not always possible. Consider the loop bound `i<G[i]` instead. In this case, an over-approximation of the domain can only be provided based on the programmer's knowledge or on the type of `G`. For example, the upper-bound could be 255 if `G` is a 1-byte `unsigned char` C data-type.

Lines 12–17 of figure 3.21 give an example of code with a non-affine conditional statement. In case the condition would not be present, the reference to `P` would be characterised as $(P, w, [0..7], 1, 1)$. Since it is not possible to know upfront whether or not the condition will evaluate to true or false, the same classification has to be used. This can be seen as an over-approximation: stepping through the domain with a unit step, but not necessarily performing a read access every time.

The final example in lines 19–22 of figure 3.21 shows an affine array reference `T` and two non-affine array references `R` and `S`. The affine reference is characterised as $(T, r, [0..7], 1, 1)$. The reference to `R` has a non-constant step. Therefore, $(R, r, [0..49], 1, 1)$ is used, giving an over-approximation of the domain and the step: not all elements are accessed. Type information of `T` could be included for the reference to `S`. Assuming a range of 0 to 255, $(S, w, [0..255], 256, 0)$ is obtained. This is classified as if all locations are written to in every loop iteration.

Overall, the new theory of algorithmic species allows for an extended applicability outside the abstractions of the polyhedral model. However, as has been shown, it is not always possible to give an exact classification or to extract an array reference characterisation automatically. In some cases, manual fine-tuning (as discussed for the examples) can help fine-tune the classification. Another option is to use run-time information (e.g. profiling) to fine-tune the array reference characterisations.

Per-statement classification

Next, another advantage of the array reference-based theory over the polyhedral model-based theory is illustrated: the possibility to classify individual statements. Figure 3.22 shows the matrix-vector multiplication of figure 3.17 again, but now with 5-tuple characterisations in-lined at different points. Note that the outermost classification is made with respect to the i-loop only. Furthermore, note that because the step $\mathcal{S}$ takes its dimensionality from the number of loops considered, characterisations can include the empty set ($\emptyset$). The example shows that the theory is not limited to loops, but can also be applied to individual statements. Furthermore, it illustrates the isolation of the loop body: although array `r` is read and written in the loop body, it is classified from the outer-loop perspective as write-only because all reads to individual elements occur after writes. The obtained representation shows similarities to the *convex array region* [46] representation, which is discussed in more detail in section 3.4.2.

```
$\mathcal{R}_{\mathtt{r}} = (r, w, [0..31], 1, 1)$
$\mathcal{R}_{\mathtt{M}} = (M, r, \langle[0..31][0..63]\rangle, \langle 1, 64\rangle, \langle 1, 0\rangle)$
$\mathcal{R}_{\mathtt{v}} = (v, r, [0..63], 1, 0)$
for (i=0; i<32; i++) {
   $\mathcal{R}_{\mathtt{r}} = (r, w, [i..i], 1, \emptyset)$
   r[i] = 0;
   $\mathcal{R}_{\mathtt{r}} = (r, w, [i..i], 1, 0)$
   $\mathcal{R}_{\mathtt{r}} = (r, r, [i..i], 1, 0)$
   $\mathcal{R}_{\mathtt{M}} = (M, r, \langle[i..i][0..63]\rangle, \langle 1, 1\rangle, \langle 0, 1\rangle)$
   $\mathcal{R}_{\mathtt{v}} = (v, r, [0..63], 1, 1)$
   for (j=0; j<64; j++) {
      $\mathcal{R}_{\mathtt{r}} = (r, w, [i..i], 1, \emptyset)$
      $\mathcal{R}_{\mathtt{r}} = (r, r, [i..i], 1, \emptyset)$
      $\mathcal{R}_{\mathtt{M}} = (M, r, \langle[i..i][j..j]\rangle, \langle 1, 1\rangle, \langle\emptyset, \emptyset\rangle)$
      $\mathcal{R}_{\mathtt{v}} = (v, r, [j..j], 1, \emptyset)$
      r[i] += M[i][j]*v[j];
   }
}
```

Figure 3.22: Matrix-vector multiplication with in-lined characterisations referring to the following loop or statement.

Limitations to the amount of detail

Finally, consider an example to which loop tiling is applied. Loop tiling is a loop transformation that can influence cache behaviour and thus significantly affects performance and energy efficiency. Figure 3.23 shows a basic example of non-tiled code (lines 1–3) and its tiled counterpart (lines 4–8). In this case, the tile size is 2 by 2, visible in the program code as the step-size of the *i* and *j* loops and the bounds of the *ii* and *jj* loops. When classifying this example using either of the algorithmic species theories, references to `E` would classify as the species '`E[0:7][0:7]element`' for both the original and tiled loops. In this case, species are not fine-grained enough to capture the difference.

```
1 for(i=0; i<8; i++)
2    for(j=0; j<8; j++)
3       E[i][j] = 0;
```

```
4 for(i=0; i<8; i=i+2)
5    for(j=0; j<8; j=j+2)
6       for(ii=0; ii<2; ii++)
7          for(jj=0; jj<2;jj++)
8             E[i+ii][j+jj] = 0;
```

Figure 3.23: Example code (left hand side) and a 2 by 2 tiled version (right hand side).

3.3.5 Conclusions

This section has introduced a new technique to classify array references in loop nests as 5-tuple array reference characterisations. It has been shown that these characterisations can be merged and transformed into the algorithmic species of section 3.2. The new theory is in contrast to the polyhedral model-based theory, not limited to static affine loop nests. However, we have also seen that in this case, automatic classification yields in many cases an over-approximation, requiring additional manual fine-tuning or dynamic analysis.

Furthermore, this section has also given an example of a limitation of algorithmic species: not all memory access pattern related aspects are captured. This is not limited to this particular example, there are other cases in which species are not able to distinguish efficient (in terms of performance or energy efficiency) from less efficient memory access patterns. Since memory access patterns are the primary feature on which species are based, and performance and energy efficiency are important aspects for the classification's goals, section 3.4 proposes to extend algorithmic species to capture more details.

3.4 Finer-grained species

Sections 3.2 and 3.3 have introduced the algorithmic species classification. However, section 3.3.4 has shown that species do not distinguish some memory access patterns that can influence performance and energy efficiency significantly. This section therefore introduce SPECIES+, a finer-grained classification based on the array reference theory of section 3.3.

3.4.1 Species+: a finer-grained classification

The new SPECIES+ classification is based on the array reference-based theory of algorithmic species. The new classification is therefore discussed as an extension to this theory: we refer to section 3.3 for background on the 5-tuple characterisations.

To motivate a finer-grained classification, consider the references to array X as shown in figure 3.24. All three references are characterised as $(X, r, [0..63], 1, 1)$ in the 5-tuple array reference characterisations with respect to the respective loop nest. However, cache behaviour can differ significantly, leading to changes in performance and energy efficiency. Examples are the differences between row-major and column-major accesses or between sequential and strided accesses, i.e. the difference between `X[i*8+j]` and `X[j*8+i]`. To be able to distinguish such cases, this section proposes to create a more detailed characterisation. In this way, the classification of accesses to X in figure 3.24 will be distinguishable.

```
for(i=0; i<8; i++) {
  for(j=0; j<8; j++) {
    Y[i][j] = X[i*8+j] + X[j*8+i];
  }
}
```

```
for(k=0; k<64; k++) {
  Z[k] = X[k];
}
```

Figure 3.24: Example showing three different ways to access the 64 first elements of array X.

The new classification SPECIES+ modifies the 5-tuple array reference characterisation by appending a repetition factor $\mathcal{X}$, creating a 6-tuple. The new repetition factor $\mathcal{X}$ reflects the iteration counts of loops: it is a set of N items (with N the number of loops in the nest) using the notation $\mathcal{X} = \mathcal{X}_1|\mathcal{X}_2|...|\mathcal{X}_N$. Additionally, $\mathcal{S}$ is modified. Section 3.3.1 already included the dimensions of the arrays into $\mathcal{S}$ to obtain $\mathcal{S} = \langle\mathcal{S}_1, \mathcal{S}_2, ..., \mathcal{S}_M\rangle$. Now, each step $\mathcal{S}_x$ is modified to become a set of size N, using the same notation as for $\mathcal{X}$: $\mathcal{S}_x = \mathcal{S}_{x,1}|\mathcal{S}_{x,2}|...|\mathcal{S}_{x,N}$. Note that a different notation is used for the dimensions of the arrays (angular brackets) and for the number of loops (pipes). The step $\mathcal{S}$ can also be represented as a matrix with N columns and M rows, as shown in equation 3.28.

$$\begin{bmatrix} \mathcal{S}_{1,1} & \mathcal{S}_{2,1} & ... & \mathcal{S}_{N,1} \\ \mathcal{S}_{1,2} & \mathcal{S}_{2,2} & ... & \mathcal{S}_{N,2} \\ ... & ... & & ... \\ \mathcal{S}_{1,M} & \mathcal{S}_{2,M} & ... & \mathcal{S}_{N,M} \end{bmatrix} \tag{3.28}$$

SPECIES+ is illustrated through the example of figure 3.25. For the access to array A in this example, $(A, r, \langle[0..7][0..7]\rangle, \langle 1, 1\rangle, \langle 1|0, 0|1\rangle, 8|8)$ is found as the new 6-tuple array reference characterisation. Here, the step $\langle 1|0, 0|1\rangle$ or $\left[\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right]$ represents a unit step in the first array dimension every iteration of the first loop i and a unit step in the second array dimension every iteration of the second loop j. The repetition factor 8|8 captures the iteration counts of the two loops. For the access to array `B` in figure 3.25, the 6-tuple remains the same except for the step, which becomes $\langle 0|1, 1|0\rangle$. Finally, for `C`, the step becomes $\langle 1, 1\rangle$ and the repetition factor becomes 8 (there is only a single loop), yielding $(C, r, \langle[0..7][0..7]\rangle, \langle 1, 1\rangle, \langle 1, 1\rangle, 8)$.

```
for(i=0; i<8; i++) {
   for(j=0; j<8; j++) {
      B[j][i] = A[i][j];
   }
}
for(i=0; i<8; i++) {
   C[i][i] = 0;
}
```

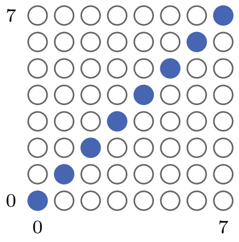


Figure 3.25: Additional examples of two-dimensional array references (left). The right hand side illustrates the accesses made to array `C`.

A second example is the motivating case with accesses to array `X` of figure 3.24. With the more detailed 6-tuple, the array reference characterisations become as shown in equation 3.29. The steps $\mathcal{S}$ now show the differences between the 3 access types: it is now possible to distinguish between a row-major and a column-major access for this example.

$$\begin{aligned}\mathcal{R}_{\texttt{X[i*8+j]}} &= (X, r, [0..63], 1, 8|1, 8|8)\\ \mathcal{R}_{\texttt{X[j*8+i]}} &= (X, r, [0..63], 1, 1|8, 8|8)\\ \mathcal{R}_{\texttt{X[k]}} &= (X, r, [0..63], 1, 1, 64)\end{aligned} \tag{3.29}$$

As a final illustrating example, the accesses made to arrays `R`, `S`, and `T` in example 3.16 are re-classified using SPECIES+. The 5-tuple array reference characterisation could not capture all details (as discussed in section 3.3.1). Now, with the more detailed 6-tuple, the differences are identified, as shown in equation 3.30.

$$\begin{aligned}\mathcal{R}_{\texttt{R[i][j]}} &= (R, r, \langle[0..7][0..7]\rangle, \langle 1, 1\rangle, \langle 1|0, 0|1\rangle, 8|8)\\ \mathcal{R}_{\texttt{R[k/8][k\%8]}} &= (R, r, \langle[0..7][0..7]\rangle, \langle 1, 1\rangle, \langle \frac{1}{8}, 1\rangle, 64)\\ \mathcal{R}_{\texttt{S[i*8+j]}} &= (S, w, [0..63], 1, 8|1, 8|8)\\ \mathcal{R}_{\texttt{S[k]}} &= (S, w, [0..63], 1, 1, 64)\\ \mathcal{R}_{\texttt{T[i]}} &= (T, r, [0..7], 1, 1|0, 8|8)\\ \mathcal{R}_{\texttt{T[k/8]}} &= (T, r, [0..7], 1, \frac{1}{8}, 64)\end{aligned} \tag{3.30}$$

3.4.2 Evaluation and discussion

The introduction of SPECIES+ has extended the 5-tuple array reference characterisations of section 3.3. This section discusses the following: 1) examples to illustrate the advantage of the classification's finer granularity, 2) limitations to the details captured by SPECIES+, and 3) related program code classifications. Note that A-DARWIN, as presented in section 3.3.3, is extended to automatically extract SPECIES+ descriptions as well.

Advantages of a finer granularity

To illustrate the advantages of SPECIES+'s increased detail, the `syrk` example from the PolyBench suite [117] as shown in figure 3.26 is discussed. When classifying this example as algorithmic species with respect to loops i and j, we find an *element* access pattern for array `B` both as input and as output. Furthermore, we find array `A` twice as input with the *chunk* pattern, in both cases accessing a full row. This classification, as shown in figure 3.26, does not make a distinction between the two accesses to array `A`. In contrast, when applying the 6-tuple classification SPECIES+ to array `A`, the results as shown in the bottom of figure 3.26 are obtained. In contrast to the algorithmic species description, the 6-tuples capture the difference between the two array accesses (a step for either the i or j-loop). Furthermore, SPECIES+ also identifies the data-reuse of the array `A` originating from the j-loop: every row is accessed N times for both `A[i][k]` and `A[j][k]`.

```
for (i=0; i<=N; i++) {
  for (j=0; j<=N; j++) {
    B[i][j] *= beta;
    for (k=0; k<=M; k++) {
      B[i][j] += alpha * A[i][k] * A[j][k];
    }
  }
}
```

```
B[0:N,0:N]|element ∧ A[0:N,0:M]|chunk(-,0:M) ∧
 A[0:N,0:M]|chunk(-,0:M) → B[0:N,0:N]|element
```

$$\mathcal{R}_{\texttt{A[i][k]}} = (A, r, \langle[0..N][0..M]\rangle, \langle 1, M\rangle, \langle 1|0, 0|0\rangle, N|N)$$
$$\mathcal{R}_{\texttt{A[j][k]}} = (A, r, \langle[0..N][0..M]\rangle, \langle 1, M\rangle, \langle 0|1, 0|0\rangle, N|N)$$

Figure 3.26: An example from PolyBench's `syrk` (top), its algorithmic species (middle), and its 6-tuple characterisation (bottom).

As another example, consider figure 3.27, showing a backwards counting loop with two reads to array `C`. Algorithmic species is unable to distinguish the two reads. However, when using the 6-tuple classification, the results are as shown in the right of figure 3.27. The difference between the two steps (-1 and 1) captures the order in which `C` is referred, relating the two references to each other.

```
for  (i=7; i>=0; i--) {
  D[i] = C[i] + C[7-i];
}
```

$$\mathcal{R}_{\texttt{C[i]}} = (C, r, [0..7], 1, -1, 8)$$
$$\mathcal{R}_{\texttt{C[7-i]}} = (C, r, [0..7], 1, 1, 8)$$

```
C[0:7]|element ∧ C[0:7]|element → C[0:7]|element
```

Figure 3.27: Example of a backwards counting loop with two reads to array `C` (left), its algorithmic species (bottom), and its 6-tuple characterisation (right).

Finally, the example of loop tiling as shown in figure 3.23 is re-evaluated. In the example, code (lines 1–3) is transformed using a 2 by 2 tile (lines 4–8). The tile-size is visible in the code through the step-sizes of the i and j loops and the bounds of the ii and jj loops. However, with algorithmic species, the reference to array `E` classify as '`E[0:7][0:7]element`' in both cases. Now, with SPECIES+, it is possible to capture the difference, as shown in equation 3.31.

$$\begin{aligned} \textit{original:}\quad & (E, w, \langle[0..7][0..7]\rangle, \langle 1,1\rangle, \langle 1|0, 0|1\rangle, 8|8) \\ \textit{tiled:}\quad & (E, w, \langle[0..7][0..7]\rangle, \langle 1,1\rangle, \langle 2|0|1|0, 0|2|0|1\rangle, 4|4|2|2) \end{aligned} \tag{3.31}$$

Limitations

Although the new 6-tuple characterisations are finer-grained than the 5-tuple characterisations, it is still an abstraction of program code and can thus not capture all relevant properties. As discussed in section 3.3.4, the array reference characterisations will become over-approximations in such cases. An example of this is the write access to the triangular matrix `F` as shown in lines 1–6 of figure 3.28. Because the bounds of the j-loop change every iteration of the i-loop, `F` has to be characterised as an over-approximation: $(F, w, \langle[0..7][0..7]\rangle, \langle 1,1\rangle, \langle 1|0, 0|1\rangle, 8|8)$. The exact iteration domain can in this case be described as a polyhedron, as is done in the work on array regions [46].

```
1 // Triangular matrix
2 for(i=0; i<8; i++) {
3   for(j=i; j<8; j++) {
4     F[i][j] = 0;
5   }
6 }
```

```
 7 // Random references
 8 for(i=0; i<8; i++) {
 9   G[rand()%8] = 0;
10 }
11
12 // Fractional indexing
13 for(i=0; i<8; i++) {
14   H[(i+1)/4] = 0;
15 }
```

Figure 3.28: Additional examples of loop nests to illustrate over-approximations.

As a second example, consider lines 7–10 of figure 3.28. Here, a random reference to array `G` is shown, independent of the loop iterator i. With the 6-tuple characterisation, $(G, w, [0..7], 8, 0, 8)$ is obtained, classifying the code as if referencing the entire array `G` every iteration. However, in reality, only one element is referenced per iteration.

As a final example, lines 12–15 of figure 3.28 show fractional indexing with an offset. Although it is possible to represent `H[i/4]` with a step of $\frac{1}{4}$, it is not possible to represent the offset in `H[(i+1)/4]`. As an extension, the domain could be allowed to include fractions for such non-affine array references. This would yield $(H, w, [\frac{1}{4}..2], 1, \frac{1}{4}, 8)$.

Related classifications

Earlier, section 3.1 gave an overview of a wide variety of algorithm classifications. Now, SPECIES+ will be compared in more detail with closely related classifications.

Related to the 6-tuple characterisation are *convex array regions* [46]. Array regions capture summaries of memory accesses (reads and writes) performed by a function, a loop, or one or more statements. For example, the elements that are read during a set of statements s with memory state σ for the basic example of figure 3.13 can be described as $\mathcal{R}(s,\sigma) = \{A[\phi_1] \mid 2 \leq \phi_1 \leq 7\}$. In this case, σ is not used. It is used for example when describing the reads when considering only the loop body of the example: $\mathcal{R}(s,\sigma) = \{A[\phi_1] \mid \phi_1 = \sigma(i)\}$. The $\sigma(i)$ notation indicates that i takes a particular value based on the memory state σ (the current loop iteration for this example). Array regions are described as a convex polyhedron and are thus abstractions of program code, similar to our domain description $\mathcal{D}$. In contrast to our work, array regions do not describe the complete access pattern, but merely capture our 6-tuple's name $\mathcal{N}$, direction $\mathcal{A}$, and domain $\mathcal{D}$. Array regions are used for example to perform loop fusion and fission, dependence analysis, and data transfer optimisations [72].

The Array-OL specification language [36] also shows similarities to our 6-tuple representation. Array-OL is different from our work and the polyhedral model in the fact that it models code at multiple levels. First of all, there is a task-level representing loop nests as communicating tasks. An individual task is then repeated according to Array-OL's $s_{repetition}$. At each repetition a task accesses data as *patterns*. Each pattern is characterised by a paving matrix (P), a fitting matrix (F) and a pattern shape ($s_{pattern}$). Furthermore, Array-OL provides an origin vector ($\mathbf{o}$) and the array size (s_{array}). Array-OL's $s_{repetition}$ and P are closely related to our repetition factor $\mathcal{X}$ and step $\mathcal{S}$ respectively. The patterns (convex polytopes) are abstracted in our representation as the number of elements $\mathcal{E}$. We illustrate the similarities with SPECIES+ by considering only a single Array-OL task and the `A[i][k]` access in figure 3.26 (let us assume that `A[j][k]` is no longer present and the whole k-loop is the elementary task). The domain of `A` is then captured as $\mathbf{o} = \begin{pmatrix}0 & 0\end{pmatrix}^T$ and $s_{array} = \begin{pmatrix}N & M\end{pmatrix}^T$, the inner-loop pattern as $F = (1)$ and $s_{pattern} = (M)$, and the repetition as $P = \begin{pmatrix}1 & 0\end{pmatrix}^T$ and $s_{repetition} = (N^2)$. In contrast to our work, Array-OL cannot be applied to non static affine loop nests and is not directly suitable for classification purposes.

Other related classifications work on a different abstraction level. This includes the representation of loop nests in the polyhedral model, such as the representation used in the integer set library *isl* [137]. In *isl*, iterations of a loop nest are represented as integer points in a polytope using first order logic. In other work, polyhedral process networks [28, 138] are introduced to provide a higher-level polyhedral-based classification of program code. As discussed in detail in section 3.1, the programming model Æcute [77] creates a decoupled access/execute specifications of program code. Finally, the language PENCIL [14] allows programmers or compilers to create summary functions to describe references.

3.4.3 Conclusions

This chapter has introduced SPECIES+, a more fine-grained version of array reference characterisations as used as basis for the algorithmic species theory. Because of the additional details captured by SPECIES+'s new 6-tuple characterisations, the classification is a step down from the descriptive names and intuitiveness of algorithmic species. However, it has been shown that SPECIES+ is able to capture details which the original species could not, including loop tiling and backwards counting loops. Such details can be relevant for performance and energy efficiency. For example, tiling a GPU implementation of a stencil computation can make a difference of a factor 3 in terms of performance [127].

With the original algorithmic species, SPECIES+, and A-DARWIN, a strong basis has been set to address the challenges of programming and code generation for parallel microprocessors. Because species and SPECIES+ are architecture-agnostic classifications, they can help to achieve (performance) portability as well.

"Any sufficiently advanced technology is indistinguishable from magic."

- Arthur C. Clarke (1973)

Chapter 4

Compilation using algorithmic skeletons

Throughout this thesis we have seen that programming has become a challenging task: programmers are faced with a variety of new parallel languages and are required to have detailed architectural knowledge to fully optimise their applications. In particular, chapter 2 discussed the current and future challenges of programmability and the related aspects performance, portability and productivity. To address these challenges, chapter 3 argued to start by adding structure to the problem. This has led to *algorithmic species*, a classification to help recognise, differentiate and understand different types of program code. This chapter will use this classification to perform source-to-source compilation.

The programmability of microprocessors such as GPUs can be improved by a *compiler*, a tool to automatically perform some of the manual programming steps involved. Traditional compiler tasks involve performing both labour-intensive tasks such as translation from a high-level programming language into machine code, and low-level optimisations such as register allocation and loop unrolling. Recently, compilers have included more advanced transformations (such as loop skewing in Pluto [34]) or have been designed specifically to perform separate (optimising) source-to-source compilation passes. An example of source-to-source compilation is the transformation of CPU code (e.g. C/C++) into GPU code (e.g. CUDA). This chapter focusses on such source-to-source compilers, because they can address programmability issues by potentially improving performance, portability and productivity. In particular, we discuss BONES, a new source-to-source compiler based on *algorithmic skeletons*.

Algorithmic skeletons [44] is a compiler technique motivated by the observation of similarities across efficient code for particular classes of program code. The technique is based on the use of parametrisable program code, known as *skeletons* or *skeleton implementations*. An individual skeleton can be seen as template code for a specific class of computations on a specific microprocessor. Users of skeleton-based compilers are required to identify a skeleton suitable for their program code and their target microprocessor, and can subsequently invoke the skeleton to obtain program code for the chosen target. If no skeleton implementation is suitable for the specific class or microprocessor, it can be added manually. Future program code of the same class can then benefit from reuse of the skeleton code. Benefits of skeleton-based compilation are among others the flexibility of extension to other targets and the performance potential: optimisations can be performed in the native language within the skeletons. Examples of recent skeleton-based source-to-source compilers are [18, 40, 55, 126].

In this chapter we present BONES, a new source-to-source compiler based on algorithmic skeletons. In contrast to other skeleton-based compilers, BONES uses a formally-defined algorithm classification and does not require programmers to manually identify classes nor to select a suitable skeleton. This is achieved by using the algorithmic species of chapter 3 as a basis. We give an overview of the relation of BONES and species in figure 4.1. This chapter gives a survey of existing source-to-source compilers targeting GPUs (section 4.1), a detailed description of BONES (section 4.2) and two of its main optimisations (sections 4.3 and 4.4), and a detailed evaluation and discussion (sections 4.5 and 4.6).

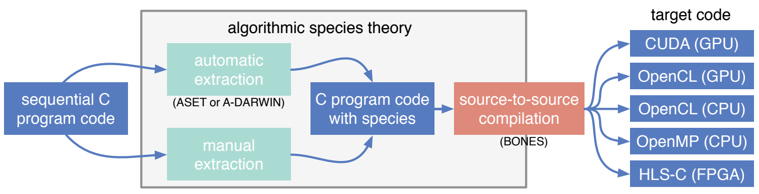


Figure 4.1: A graphical overview of the relation between the algorithmic species theory (chapter 3) and the BONES source-to-source compiler (this chapter).

4.1 A survey of source-to-source compilers

There are a large number of source-to-source compilers targeted at (partially) automating GPU programming. This section briefly introduces the most prominent compilers and discusses several of them in more detail. The focus lies on source-to-source compilers with a CUDA or OpenCL target. The following classes are distinguished:

- **Directives:** Several C-to-CUDA source-to-source compilers use directives in the form of pragma's. These user-supplied directives translate directly to compiler transformations, addressing the more labour intensive tasks such as host-accelerator memory transforms. Examples are *hi*CUDA [73] (discussed in section 4.1.1) and compilers using OpenMP-like directives [91, 106]. Also based on directives, but using CUDA both as input and output is the CUDA-Lite [135] source-to-source compiler.

- **Algorithmic skeletons:** Existing work also uses the algorithmic skeletons approach to target GPUs. The most prominent examples are SkePU [55] (discussed in section 4.1.2) and SkelCL [126]. These are not implemented as source-to-source compilers, but rather use C++ templates to invoke skeletons. Other skeleton approaches targeting CUDA include Muesli [56] and the work by Sato et al. [121].
- **Semi-automatic:** OpenACC (see also section 2.3) is a semi-automatic approach, as it performs code generation based on a combination of user-supplied directives and static analysis. Two C-to-CUDA compilers are based on OpenACC directives: PGI Accelerator [146] (discussed in section 4.1.3) and HMPP Workbench [52].
- **Fully automatic:** We identify two source-to-source compilers based on static analysis techniques: Par4All [22] and ppcg [139]. Both are discussed in more detail in section 4.1.4. Other automatic compilers are C-to-CUDA [25] and a CUDA version of Pluto [29], but they are either not publicly available or not fully functional.
- **Domain-specific languages:** Finally, several source-to-source compilers are introduced as part of new domain-specific languages. Examples include the Mint language [136], a medical DSL [100] and Chestnut [129].

To illustrate the programming style and the required programming effort for the source-to-source compilers, we take the 1D stencil computation of figure 4.2 as an example throughout this section.

```
int N = 512*512;
for(i=1; i<N-1; i++) {
  B[i] = 0.3*A[i-1] + 0.4*A[i] + 0.3*A[i+1];
}
```

Figure 4.2: A 1D stencil example used to illustrate existing source-to-source compilers.

4.1.1 Directives using *hi*CUDA

Source-to-source compilers based on directives rely on the help of programmers to generate code. A common case is the use of annotations (e.g. pragmas) in the source code that guide (or: direct) the compiler towards generating efficient target code. A good example is the C-to-CUDA compiler *hi*CUDA [73], which is discussed in this section.

Figure 4.3 applies *hi*CUDA directives to the example stencil code of figure 4.2. The original code is still present (lines 1, 7 and 10), but a number of directives have been added. First of all, array sizes have to be set in case of dynamically allocated memory (line 2). Second, memory has to be allocated on the GPU (lines 3–4), copied between CPU and GPU (lines 3 and 13) and freed (line 14). The kernel itself is defined in lines 5–12. Most of these directives translate directly into CUDA statements or CUDA library calls.

```
 1 int N = 512*512;
 2 #pragma hicuda shape A[N] B[N]
 3 #pragma hicuda global alloc A[*] copyin
 4 #pragma hicuda global alloc B[*]
 5 #pragma hicuda kernel conv tblock(N/256) thread(256)
 6 #pragma hicuda loop_partition over_tblock over_thread
 7 for(i=1; i<N-1; i++) {
 8   #pragma hicuda shared alloc A[i-1:i+1] copyin
 9   #pragma hicuda barrier
10   B[i] = 0.3*A[i-1] + 0.4*A[i] + 0.3*A[i+1];
11 }
12 #pragma hicuda kernel_end
13 #pragma hicuda global copyout B[*]
14 #pragma hicuda global free A B
```

Figure 4.3: The 1D stencil example annotated with *hi*CUDA directives.

Users of *hi*CUDA are still required to have GPU programming expertise. For example, the programmer has to specify the number of threads and threadblocks and has to specify which loops should be parallelised. Additionally, to use the on-chip scratchpad memory, the programmer has to supply directives defining which memory section to store locally and when to synchronize between threads (lines 8–9 in figure 4.3). An advantage of *hi*CUDA is its inter-procedural support. For example, 'kernel' directives can be placed around function calls, while other directives can be placed within these functions. Furthermore, the code generated by *hi*CUDA can be inspected and modified. However, the generated code is not easily readable and is thus not suitable for further fine-tuning.

4.1.2 Algorithmic skeletons through SkePU

Applying algorithmic skeletons to many-core architectures such as GPUs has been accomplished recently in several works, of which SkePU [55] is an example. SkePU is publicly available and supports both CUDA and OpenCL as targets.

Figure 4.4 shows the SkePU implementation of the stencil example of figure 4.2. Seven skeletons are supported (*map*, *mapArray*, *mapOverlap*, *mapReduce*, *reduce*, *scan* and *generate*), from which we select *mapOverlap* to match the stencil code (line 13). The figure also shows the copying and conversion of arrays into the `skepu::Vector` containers (lines 8–12 and 15–17). Furthermore, the functionality is defined (lines 2–4), and the kernel is invoked (line 14).

```
 1 // Global function definition
 2 OVERLAP_FUNC(stencil, float, 1, in,
 3   return 0.3*in[-1] + 0.4*in[0] + 0.3*in[1];
 4 )
 5
 6 // Main function
 7 int N = 512*512;
 8 skepu::Vector<float> A_v(N);
 9 skepu::Vector<float> B_v(N);
10 for (i=0; i<N; i++) {
11   A_v[i] = A[i];
12 }
13 skepu::MapOverlap<stencil> stencilKernel(new stencil);
14 stencilKernel(A_v,B_v);
15 for (i=0; i<N; i++) {
16   B[i] = B_v[i];
17 }
```

Figure 4.4: The 1D stencil example using SkePU's algorithmic skeletons.

SkePU implements skeletons as C++ libraries and therefore does not require separate compilation: including the supplied header files and compiling with the CUDA compiler is sufficient. This approach has also drawbacks: SkePU requires the original code to be rewritten, increasing the programming effort and limiting the portability. As shown in the example, SkePU also uses special data containers. This will not pose a problem for cases where the whole program can be designed using SkePU containers. However, for cases where kernels are considered as small parts of a larger application, a copy-in and a copy-out is required. SkePU furthermore supports lazy memory copying, can generate code for systems with multiple GPUs, and supports both CUDA and OpenCL.

4.1.3 OpenACC directives with PGI Accelerator

PGI Accelerator [146] is a commercial C and Fortran to CUDA source-to-source compiler. It performs extensive code analysis, but also relies on programmer input in the form of OpenACC directives. By combining compiler analysis with a varying degree of user-directives, the user remains in control of the effort versus performance trade-off.

An OpenACC version of the example of figure 4.2 is given in figure 4.5. In the ideal case, PGI Accelerator requires only a single directive to delimit the scope of acceleration (`acc region`, line 2) and will extract the remaining information using static analysis: which arrays to copy-in or copy-out, which loops to parallelise, which temporary results to store in on-chip memory, etc. If the compiler cannot find such information statically, the user will be asked to supply additional directives. Such a directive is provided for the stencil example (line 4) to inform the compiler that the iterations of the loop are independent of each other.

```c
int N = 512*512;
#pragma acc region
{
  #pragma acc for independent
  for(i=1; i<N-1; i++) {
    B[i] = 0.3*A[i-1] + 0.4*A[i] + 0.3*A[i+1];
  }
}
```

Figure 4.5: The 1D stencil example using OpenACC directives and PGI Accelerator.

The PGI Accelerator source-to-source compiler provides information to the user as to how the code is generated. The programmer can then supply additional directives to guide the compiler in a specific direction. The compiler furthermore analyses aspects such as thread occupancy, memory accesses, and register usage.

4.1.4 Automatic compilation with Par4All and PPCG

Several C-to-CUDA compilers are fully automatic: they are able to perform dependence analysis and loop transformations without user annotations. The most prominent examples are Par4All [22] and ppcg [139].

The code transformation and parallelisation framework PIPS is the main component of the Par4All source-to-source compiler, which generates CUDA code. The compiler takes unmodified C-code as input, such as the stencil code example of figure 4.2. However, as PIPS is based on convex array region analysis [46], restrictions apply to the input code, requiring loops to have static control and affine bounds and references [22, 46]. Par4All uses macros to hide CUDA statements from the generated code to improve readability.

The compiler ppcg is based on the polyhedral model and generates CUDA code. It has similar properties as Par4All: the compiler is fully automatic, but imposes restrictions on input code, only static affine loop nests are transformed. Both ppcg and Par4All are able to perform host-accelerator (e.g. CPU-GPU) transfer optimisations.

4.1.5 Evaluation and discussion

This section discussed several different source-to-source compilers, including compilers based on directives (*hi*CUDA and PGI Accelerator), compilers using algorithmic skeletons (SkePU), and compilers based on static analysis (PGI Accelerator, Par4All and ppcg). Table 4.1 gives a summary of their properties.

Par4All and ppcg are clear winners from a programmability, productivity, and portability perspective, as they require no modifications to the input program code (performance will be evaluated in section 4.5). When performance is not considered, the other three discussed source-to-source compilers fall short in the following two areas: 1) they are not fully automatic and require code restructuring

	*hi*CUDA	SkePU	PGI Acc.	Par4All	ppcg
technique	directives	skeletons	directives + analysis	static analysis	static analysis
freely available	✓	✓	×	✓	✓
CUDA target	✓	✓	✓	✓	✓
OpenCL target	×	✓	×	✓	✓
source-to-source	✓	×	✓	✓	✓
fully automatic	×	×	×	✓	✓
uses regular C data-types	✓	×	✓	✓	✓
data-sizes needed at run-time	✓	✓	×	✓	✓
generates readable code	×	N/A	×	✓	✓
generates multi-GPU code	×	✓	×	×	×
performs kernel fusion	×	×	×	✓	✓

Table 4.1: Overview of properties of the five discussed approaches.

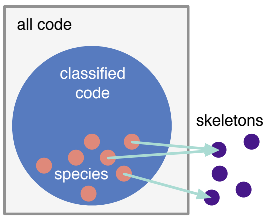
(SkePU) or annotations (*hi*CUDA and PGI Accelerator), and 2) they directly produce binaries (SkePU) or generate human unreadable code (*hi*CUDA and PGI Accelerator). The first shortcoming mostly affects application programmers who are unfamiliar with parallel architectures and concurrent programming, while the second shortcoming affects savvy programmers who leverage compilers to perform the initial parallelisation and are willing to further optimise the resulting code. This chapter addresses these shortcomings by using the algorithm classification of chapter 3 to drive a source-to-source compiler based on algorithmic skeletons.

4.2 A skeleton-based source-to-source compiler

This section introduces BONES, a source-to-source compiler based on algorithmic skeletons. The compiler takes C program code annotated with class information as an input (the algorithmic species from chapter 3). This ‘species information’ is used to determine which skeleton to use, and is additionally used to enable or disable additional transformations and optimisations that are not possible within the skeletons themselves. BONES[1] is written in Ruby and uses the C-to-AST module CAST[2] to be able to work on abstract syntax trees (ASTs). The compiler ships with a total of 15 pre-written skeletons for 5 different targets: CUDA for NVIDIA GPUs, OpenCL for AMD GPUs, OpenCL for CPUs (both for the AMD and the Intel SDK) and OpenMP for CPUs.

In contrast to existing skeleton-based compilers, BONES uses the algorithmic species information to select a suitable skeleton. This does not only enable automation of the tool-chain (by combining with ASET or A-DARWIN), but also overcomes common critique of skeleton-based compilers illustrated by questions such as ‘how difficult is it to select a suitable skeleton’ or ‘what if the user selects an incompatible skeleton’.

[1]Source-code is available at `http://github.com/cnugteren/bones/`.
[2]CAST can be found at `http://cast.rubyforge.org/`.



species (⋆ is a wildcard)	skeleton
`[⋆:⋆]|neighbourhood` → `[⋆:⋆]|element`	1D-caching
`[0:N,0:N]|chunk(-,0:N)` ⋆ → ⋆	shuffling x1
`[0:N,0:N]|chunk(-,0:N)` ∧ `[0:N,0:N]|chunk(-,0:N)` ⋆ → ⋆	shuffling x2
⋆`|element` → ⋆`|shared`	reduction
all other species	default

Figure 4.6: Illustration of the relation between algorithmic species and skeletons (left). Species are classes of code (small circles) within the region of interest (large circle) that map many-to-one to skeletons. The table on the right shows a practical case of a species to skeleton mapping for the GPU-CUDA target, discussed further in section 4.2.1.

Algorithmic species map in principle one-to-one to skeletons: the compiler will need to supply a skeleton for every species it wants to support. These skeletons must be constructed in such a way that they are correct (and preferably optimised) for all possible types of program code that belong to a particular species. However, for practical reasons (to save work and code duplication and to be able to accommodate an infinite amount of species), BONES provides a species-to-skeleton mapping such that multiple species can map to the same skeleton. For example, for the GPU-CUDA target, algorithmic species of the form '*neighbourhood* → *element*' and '*neighbourhood* ∧ *element* → *element*' both map to a single skeleton that performs explicit caching of the *neighbourhood* in the GPU's on-chip scratchpad memory. Figure 4.6 illustrates this many-to-one species-to-skeleton mapping: a limited amount of skeletons cover a larger amount of species. BONES provides optimised skeletons for the species encountered so far in all performed experiments and all used benchmarks. As shown in the figure, algorithmic species classify a subset of all possible code: it considers only parallel loop nests.

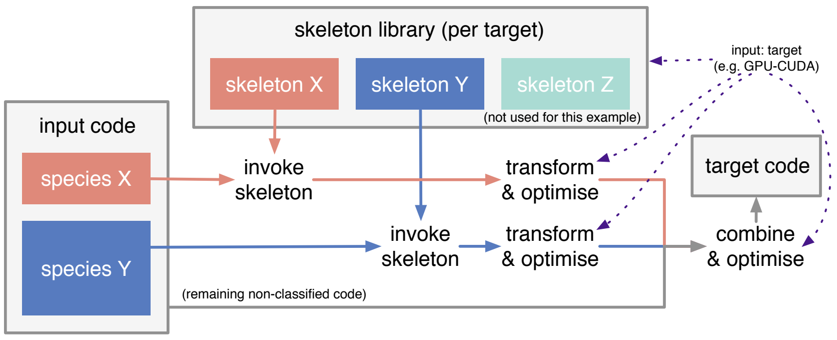


Figure 4.7: Illustration of the structure of the BONES compiler for an example piece of input code with two different species (left). The example shows the skeleton library (top) and the transformation and optimisation passes.

The workings of BONES are illustrated by figure 4.7. This figure shows an example input with two code segments identified as two different species ('species X' and 'species Y'). The compiler first loads and invokes the corresponding skeletons for a given target. The target is specified by the user, but could be selected dynamically based on the invoked skeleton and data-sizes as is done by SkePU [48]. Next, the skeleton-specific transformations and optimisations are performed. Finally, BONES combines the results to obtain target program code. The transformations and optimisations will be described in sections 4.2.2, 4.3 and 4.4.

Apart from using skeletons for the performance-critical parts of the code (the kernels), BONES also uses templates (or skeletons) to generate the remainder of the code. These skeletons are target specific, rather than target and species specific, but still have parameters (e.g. array name or array size). Examples of such skeletons include OpenCL/CUDA memory allocation, a host to accelerator data transfer, initialisation of a platform, and the inclusion of header files. Using skeletons for such tasks keeps the compiler lightweight and makes modification or extension to other targets straightforward.

4.2.1 Example skeletons

The use of skeletons within BONES is illustrated through an OpenMP skeleton. Figure 4.8 shows the skeleton itself and its instantiation for the matrix-vector multiplication example ($\vec{r} = \mathbf{M} \cdot \vec{v}$, figure 3.2). The skeleton (left) highlights the following keywords: `parallelism` represents the amount of potential parallelism found within the species and `code` fills in the transformed code. For illustration purposes, the example is heavily simplified, excluding comments, boundary and initialisation code, function calls and definitions, and makes several assumptions, such as the divisibility of the amount of parallelism by the thread count.

Additionally, figure 4.9 shows an example of a skeleton for the GPU-CUDA target. This skeleton is specific to species of the form '*0:N,0:N|chunk(-,0:N)* $\rightarrow$ *0:N,0:N|element*', similar to the matrix-vector multiplication example. A naive mapping to CUDA results in *uncoalesced* accesses to the *chunk* array: subsequent accesses will be made by the same thread. To re-enable coalescing in these cases (important for performance), a special skeleton with a pre-shuffling kernel is introduced. This skeleton (figure 4.9) has a kernel for the actual work (lines 1–8) and a kernel to reorder the input array by re-arranging data in the GPU's on-chip memory (lines 10–21). The use of this skeleton implies a transformation in the original code (e.g. from `M[i][j]` into `M[j][i]`), which is handled by the compiler. The skeleton highlights additional keywords (compared to Figure 4.8): `ids` computes the identifier corresponding to the current thread, `type` gives the data type, `dims` represents the array sizes, and `params` represents the chunk sizes. Again, this skeleton is simplified, e.g. not showing boundary checks or host code.

Currently, BONES provides 5 CUDA skeletons (see figure 4.6). This includes a 'default' skeleton for which the parallel loops are mapped to threads and the loop body is mapped to a CUDA kernel. A variation of this is a skeleton for *neighbour-*

```
int count;
count = omp_get_num_procs();
omp_set_num_threads(count);
#pragma omp parallel
{
  int tid;
  int work, start, end;
  tid = omp_get_thread_num();
  work = ⟨parallelism⟩/count;
  start = tid*work;
  end = (tid+1)*work;

  // Start the parallel work
  for(gid=start; gid<end; gid++){
    ⟨code⟩


  }
}
```

```
int count;
count = omp_get_num_procs();
omp_set_num_threads(count);
#pragma omp parallel
{
  int tid;
  int work, start, end;
  tid = omp_get_thread_num();
  work = 32/count;
  start = tid*work;
  end = (tid+1)*work;

  // Start the parallel work
  for(gid=start; gid<end; gid++) {
    r[gid] = 0;
    for (j=0; j<64; j++)
      r[gid] += M[gid][j]*v[j];
  }
}
```

Figure 4.8: An example simplified skeleton for OpenMP (left) and the same skeleton invoked for matrix-vector multiplication (right). Details and optimisations are omitted.

```
 1 // CUDA kernel for the actual work (simplified)
 2 __global__ void kernel0(...) {
 3   int gid = blockIdx.x*blockDim.x+threadIdx.x;
 4   if (gid < ⟨parallelism⟩) { // Incomplete blocks
 5     ⟨ids⟩
 6     ⟨code⟩
 7   }
 8 }
 9
10 // CUDA kernel for pre-shuffling (simplified)
11 __global__ void kernel1(...) {
12   int tx = threadIdx.x; int ty = threadIdx.y;
13   __shared__ ⟨type⟩ b[16][16];
14   int gid0 = blockIdx.x*blockDim.x + tx;
15   int gid1 = blockIdx.y*blockDim.y + ty;
16   int nid0 = blockIdx.y*blockDim.y + tx;
17   int nid1 = blockIdx.x*blockDim.x + ty;
18   b[ty][tx] = in[gid0 + gid1*⟨dims⟩/⟨params⟩];
19   __syncthreads();
20   out[nid0 + nid1*⟨params⟩] = b[tx][ty];
21 }
```

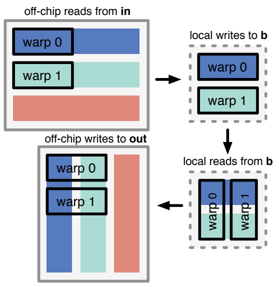


Figure 4.9: An example simplified skeleton (left) for the GPU-CUDA target with a pre-shuffling kernel (lines 10–21) to enable coalesced accesses. The shuffling is illustrated on the right.

hood type of computations, caching input data into the GPU's on-chip scratchpad memory. Furthermore, there are two skeletons with a pre-shuffling kernel (see figure 4.9), one for a single input and one for two inputs. Finally, there is a special associative reduction skeleton for species in the form of *'element* $\rightarrow$ *shared'*. In this case, an alternative is to use the default skeleton and ask the compiler to replace all operations involving *shared* variables by atomic counterparts.

4.2.2 Compiler optimisations

This section discusses basic optimisations: two more advanced optimisations are discussed separately in sections 4.3 (host-accelerator data transfers) and 4.4 (kernel fusion). BONES performs various basic transformations and optimisations, most of which are conditionally applied based on the algorithmic species. An example of a basic transformation is the replacement of array indices and names of input arrays in *neighbourhood*-based skeletons for GPUs by local indices and names. This transformation is a matter of name-changing, the actual definition of the local indices and the pre-fetching into local memories is performed within the corresponding skeletons.

Furthermore, BONES flattens data structures to a single dimension when generating GPU code (to satisfy CUDA/OpenCL requirements). Nested parallel loops are also flattened, decoupling the number of loops from the threads or workitems provided by CUDA and OpenCL. In contrast, many existing approaches (e.g. [22, 55, 139]) map multi-dimensional loops to multi-dimensional threads or workitems. Although this might be a straightforward solution, it limits the applicability of existing approaches to 2 or 3-dimensional loops and data structures. In contrast, BONES is able to handle arrays of any dimension and any degree of loop nesting.

Additionally, several performance-oriented transformations are made within BONES. This includes *register file caching* and *thread coarsening* [96]. Register file caching can replace array accesses (mapped to off-chip memories) with scalar accesses (mapped to registers) under certain conditions. For example, in matrix-vector multiplication (figure 3.2), accesses to vector $\vec{r}$ can be replaced by scalar accesses under the condition that a final store to $\vec{r}$ is added at the end of the loop body. In case there are insufficient registers available, spilling will ensure a neutral worst-case performance impact. Thread coarsening or thread merging is a technique to increase the workload per thread. This comes at the cost of parallelism, but could increase data reuse through locality or factor out common instructions [92]. In BONES, coarsening is implemented by replicating code line-by-line (and renaming variables where required) and is only considered if the species has data reuse. For example, in the case of *0:M,0:N|chunk(-,0:N)* $\rightarrow$ *0:M,0:N|element*, a total of $N \cdot M$ elements are produced, while only M chunks are available as input, resulting in the reuse of the input by a factor N. Coarsening is only enabled for kernels with sufficient parallelism (e.g. at least 2^{15} threads on a GPU) but without divergent control flow. The performance effects of register file caching and thread coarsening will be discussed in section 4.5.

4.3 Optimising host-accelerator data transfers

Several of today's parallel microprocessors are designed as an *accelerator*: they require a *host* processor to dispatch tasks. Furthermore, they might have a separate memory, requiring host-accelerator transfers of input and output arrays. Executing multiple kernels subsequently gives opportunities to optimise these transfers in several ways [72, 81]: 1) transfers can be omitted (e.g. subsequent kernels use the same data), 2) transfers can be done in parallel with host code (e.g. start the copy as soon as the data is ready), and 3) transfers related to a previous or upcoming kernel can run in parallel with kernel execution. In case the host and accelerator share a memory (e.g. CPU, fused CPU/GPU), transfers can be completely removed. This section discusses the optimisations for such cases first, followed by optimisations for cases when the host and accelerator have their own memory (e.g. GPU, Intel MIC).

To improve performance for the case in which the host and accelerator share the same memory, BONES enables *zero-copy* for the OpenCL targets using an aligned memory allocation scheme. In this case, a memory copy can be saved by performing a pointer-only copy (a zero-copy). BONES enables zero-copy in OpenCL for CPUs by fulfilling Intel's two requirements: 1) using specific OpenCL memory map and memory un-map functions, and 2), aligning all allocations to 128-byte boundaries [79]. To ensure aligned memory allocations in the original code, BONES provides an aligned dynamic memory allocation implementation and ensures that stack arrays are aligned, as has been done in prior work [81].

```
copy−in(A,1)
sync(1)
B ← A
copy−out(B,2)
sync(2)
copy−in(B,3)
copy−in(D,3)
sync(3)
C ← B + D
copy−out(C,4)
sync(4)
```

```
copy−in(A,1)
sync(1)
B ← A
copy−out(B,2)
sync(2)
copy−in(D,3)
sync(3)
C ← B + D
copy−out(C,4)
sync(4)
```

```
copy−in(A,1)
copy−in(D,3)
sync(1)
B ← A
copy−out(B,2)
sync(2)
sync(3)
C ← B + D
copy−out(C,4)
sync(4)
```

```
copyin(A,1)
copyin(D,3)
sync(1)
B ← A
copyout(B,4)
sync(3)
C ← B + D
copyout(C,4)
sync(4)
```

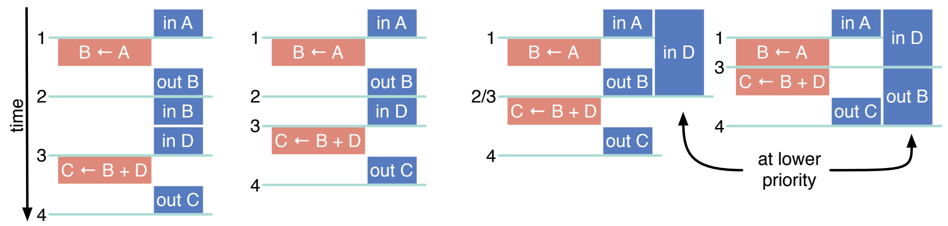


Figure 4.10: Pseudo-code illustrating several optimisation steps: original (left), partly optimised (mid-left and mid-right), fully optimised (right). The second argument to the copy-in and copy-out functions give the deadlines, corresponding to the numbers of the synchronisation barriers. The figures below show time progressing from top to bottom (not to scale).

Next, consider the case where the host and accelerator have their own memory. After marking the identified algorithmic species with pragmas in ASET (section 3.2.3) or A-DARWIN (section 3.3.3), the tools are instructed to: 1) mark inputs and outputs as copy-ins and copy-outs for the current kernel, and 2) add synchronisation pragmas after the transfers. BONES then generates a second asynchronous host thread, receiving transfer requests and performing synchronisations. In its most basic form, BONES performs a copy-in of all required data before starting a kernel, and performs a copy-out immediately afterwards. This is illustrated through an example with two kernels, shown in the left most side of figure 4.10.

After producing the initial non-optimised form, ASET or A-DARWIN perform different optimisation steps recursively: 1) copy-ins directly after copy-outs are removed (e.g. from left to mid-left in figure 4.10), 2) copy-ins are moved to the front if the data is not written by the previous species (e.g. from mid-left to mid-right in figure 4.10), 3) copy-outs can be delayed if if the data is not written by the next species (e.g. from mid-right to right in figure 4.10), 4) unused synchronisation barriers are omitted or merged, and 5) transfers are moved out of a loop body if possible (not shown in this example). The performance impacts of the transfer optimisations are discussed in section 4.5.

4.4 Kernel fusion

Loop fusion is a performance-oriented loop transformation that combines two loop nests into a single new loop nest [116]. Its dual is loop distribution, creating new loops by splitting a loop body. Because fusion has been extensively discussed in the literature (e.g. [40, 47, 86, 116, 139]), we focus on a special case of loop fusion: kernel fusion in the context of algorithmic species. Kernel fusion is specific in the sense that fusion must not introduce loop-carried dependences as they prevent parallel execution, and the loops that form the kernels are not considered interchangeable or transformable: they are already forming species.

Kernel fusion (and loop fusion in general) can be beneficial in terms of performance and energy efficiency for several reasons: 1) kernel start-up times can be reduced (e.g. thread-launch overheads), 2) optimisation opportunities might arise by merging the two loop bodies (e.g. common expression elimination), and 3) data locality can improve. On the other hand, kernel fusion can also degrade performance, e.g. by introducing additional control flow or cache contention.

Loop fusion can be decomposed into three main aspects that are strongly connected and often evaluated together [86]. Firstly, loop transformations such as interchange and shifting can be applied to make fusion possible and advantageous by creating compatible *loop headers* (step, bounds) [47]. Secondly, the safety of fusion needs to be evaluated: dependences need to be preserved when fusing two loops. Finally, the performance aspects need to be considered: fusion should only be applied if it is beneficial for performance or energy efficiency. Because loop headers can be made compatible before feeding C code into BONES and ASET/A-DARWIN (using tools such as Pluto or PoCC), this section discusses only the latter two aspects. Specifically, we re-formulate the problem of kernel fusion in the context of algorithmic species, discussing whether fusion is legal and when it is beneficial to be applied. Kernel fusion as discussed is implemented in A-DARWIN.

4.4.1 Legality of fusion

Consider two fusion candidate kernels X and Y with Y following X directly. Fusion of these kernels is legal in any of the following cases:

- **Independent case** Kernel Y does not read from or write to any output of kernel X and does not write to any input of kernel X. The only locality advantages might come from a shared input, of which an example is shown on the left hand side of figure 4.11.

- **Equal access pattern** Kernel Y reads from or writes to output(s) of kernel X or writes to input(s) of X as long as the access patterns match. An example is shown in the middle of figure 4.11, in which X writes to `B` and Y reads from `B` with the same pattern: *0:7|element*.

- **Compatible access pattern** Kernel Y reads from or writes to output(s) of kernel X and writes to input(s) of X for a combination of access patterns that preserves loop-carried dependences. The specific combinations are patterns with non-intersecting domains and patterns for which the intersecting (but unequal) domains are part of *chunk* accesses (formal definition in equation 4.1). Examples are shown in figure 4.12.

An example of a case where loop fusion is illegal is shown on the right hand side of figure 4.11. The access patterns to array `A` are incompatible (*0:7|element* and *1:8 |element*). Tools such as Pluto and PoCC can resolve this by shifting the second loop by one iteration, creating matching access patterns.

```
for(i=0; i<8; i++) {
  B[i] = A[i]+3;
}

for(i=0; i<8; i++) {
  C[i] = A[i*2]*4;
}
```

$\Downarrow$

```
for(i=0; i<8; i++) {
  B[i] = A[i]+3;
  C[i] = A[i*2]*4;
}
```

✓

```
for(i=0; i<8; i++) {
  B[i] = 0;
  for(j=0; j<4; j++)
    B[i] += A[i][j];
}

for(i=0; i<8; i++) {
  C[i] = 4*B[i];
}
```

$\Downarrow$

```
for(i=0; i<8; i++) {
  B[i] = 0;
  for(j=0; j<4; j++)
    B[i] += A[i][j];
  C[i] = 4*B[i];
}
```

✓

```
for(i=0; i<8; i++) {
  A[i] = 0;
}

for(i=0; i<8; i++) {
  B[i] = A[i+1];
}
```

$\Downarrow$

```
for(i=0; i<8; i++) {
  A[i] = 0;
  B[i] = A[i+1];
}
```

×

Figure 4.11: Three examples with two loop nests each (top) and the corresponding fused code (bottom). For the left and middle examples fusion is legal, for the right example it is not.

```
for(i=0; i<8; i++) {
  B[i] = 0;
   for(j=0; j<4; j++)
     B[i] += A[i][j];
}

for(i=0; i<8; i++) {
  A[i][2] = C[i];
}
```

```
for(i=0; i<8; i++) {
   val = A[i];
   for(j=0; j<4; j++)
     B[i][j] = val;
}
for(i=0; i<8; i++) {
  B[i][0] *= C[0];
  B[i][1] *= C[1];
}
```

```
for(i=0; i<8; i++) {
  A[i] = 0;
}

for(i=0; i<8; i++) {
  B[i] = A[i+400];
}
```

Figure 4.12: Examples of fusion candidates with compatible access patterns. Fusion is legal because the domain intersection is either part of the *chunk* accesses (left and middle) or because the domains are non-intersecting (right).

ALGORITHM 4.1: Legality of kernel fusion in the context of algorithmic species.

```
   Input: array reference characterisations $R_X$ and $R_Y$ for kernels X and Y
 1 foreach combination of $\mathcal{R}_x \in R_X$ and $\mathcal{R}_y \in R_Y$ do
 2     if ($\mathcal{N}_x = \mathcal{N}_y$) and ($\mathcal{A}_x = w$ or $\mathcal{A}_y = w$) then
 3         if $\mathcal{D}_x \neq \mathcal{D}_y$ or $\mathcal{E}_x \neq \mathcal{E}_y$ or $\mathcal{S}_x \neq \mathcal{S}_y$ then // equal access patterns
 4             if $\mathcal{D}_x \cap \mathcal{D}_y \neq \emptyset$ then           // non-intersecting domains
 5                 if $\mathcal{R}_x$ and $\mathcal{R}_y$ are not compatible then   // equation 4.1
 6                     return false
 7                 end
 8             end
 9         end
10     end
11 end
12 return true
```

Legality of kernel fusion can be translated into the 5-tuple notation as used for the array reference-based theory of algorithmic species (section 3.3.1). For kernels X and Y, this results in equation 4.1 (assuming a 1D array for simplicity), to be applied to all array reference combinations $(\mathcal{R}_x, \mathcal{R}_y)$ for which the names match $(\mathcal{N}_x = \mathcal{N}_y)$ and at least one of the references is a write. This reduces to a domain intersection test for two *element*-type accesses, because in that case the step and the number of elements are of unit size: $\mathcal{S}_x = \mathcal{S}_y = \mathcal{E}_x = \mathcal{E}_y = 1$.

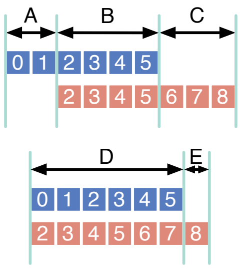
$$\begin{aligned} &\nexists(i_x \cdot \mathcal{S}_x + e_x = i_y \cdot \mathcal{S}_y + e_y) \\ \textbf{with} \quad & i_x \neq i_y \quad \textbf{and} \quad 0 \leq e_x < \mathcal{E}_x \quad \textbf{and} \quad 0 \leq e_y < \mathcal{E}_y \\ \textbf{and} \quad & \mathcal{D}_x^{lower} \leq i_x \leq \mathcal{D}_x^{upper} \quad \textbf{and} \quad \mathcal{D}_y^{lower} \leq i_y \leq \mathcal{D}_y^{upper} \end{aligned} \tag{4.1}$$

A-DARWIN tests for the legality of fusion according to algorithm 4.1. The algorithm iterates over all combinations of inputs and outputs (line 1), verifying the three earlier discussed cases: independence (line 2), an equal access pattern (line 3) and a compatible access pattern (lines 4 and 5). If the test fails for one of the combinations, kernel fusion is not legal (line 6). Only if all combinations pass, kernel fusion can be applied (line 12). The test for a compatible access pattern uses the dependence test of equation 4.1, and is implemented as a combination of the GCD-test and the Banerjee-test (see also section 3.2.3).

4.4.2 Performance considerations

After testing for legality of kernel fusion, a decision needs to be made whether and how to apply kernel fusion. Deciding whether to apply kernel fusion is a problem dependent on many factors, including architectural properties. Furthermore, loop fusion in general (with respect to optimising performance) has been theoretically proven to be a difficult problem, even for specific objectives such as maximising data reuse [47]. Because of the complexity and the number of variables involved, BONES asks the user to decide whether or not to perform kernel fusion. Per default, kernel fusion is performed when legal. Section 4.5.1 will evaluate the benefit of fusion for a number of experiments.

There are different ways kernel fusion can be implemented when loop headers are not perfectly matching. An example of non-matching loop bounds is given on the left hand side of figure 4.13: the first loop runs from 0 to 5 (blue) and the second from 2 to 8 (red). This results in tree code sections: A) only the first loop is active, B) both loops are active, and C) only the second loop is active. If there are no restricting dependences, the second loop can be shifted, obtaining two sections: D) both loops are active, and E) only the second loop is active. In the latter case, fusion can be applied in two different ways as shown in figure 4.13: 1) a conditional statement in the body bounds the execution of the D section, or 2) a second loop nest is created to complete the E section separately from the main body. The first technique is applied in BONES. Although this will create a number of idle threads (depending on the length of the E section) and introduces additional control flow, it does save the overhead of launching a second (typically much smaller) kernel. In particular for a GPU, the overhead of idle threads is negligible and the conditional statement will result in at most one warp with branch divergence.



```
for(i=0; i<7; i++) {
  if (i<6) {
    f(first[i]);
  }
  f(second[i+2]);
}
```

```
for(i=0; i<6; i++) {
  f(first[i]);
  f(second[i+2]);
}
for(i=6; i<7; i++) {
  f(second[i+2]);
}
```

Figure 4.13: Kernel fusion for non-matching loop bounds. Shifting the second loop creates two sections (D and E), which can be implemented either using a conditional statement (middle) or a second loop nest (right).

4.5 Experimental results

This section evaluates BONES experimentally in three steps: 1) by measuring the benefits of the presented compiler optimisations, 2) by comparing the different targets of BONES to each other, and 3) by comparing the GPU-CUDA target of BONES to state-of-the-art C-to-CUDA compilers.

All experiments are based on the PolyBench/C 3.2 benchmark suite [117], the same suite that was used in section 3.2.4 to evaluate algorithmic species. This suite allows a head-to-head comparison against polyhedral-based compilers [22, 139] and enables automatic extraction of algorithmic species using ASET or A-DARWIN. From PolyBench's 30 benchmarks, 28 contain parallelism in their current form. In these benchmarks, a total of 60 species are identified (not including nested species). However, several species are inner-loops with a small amount of iterations and little work, resulting in execution times of less than millisecond, dominated by start-up and measurement overheads. We therefore exclude: `adi`, `cholesky`, `dynprog`, `durbin`, `fdtd2d-apml`, `gramschmidt`, `lu`, `ludcmp`, `reg_detect`, `symm`, `trmm` and `trisolv`. The exclusion of these benchmarks can be automated by integrating a roofline-like performance model (e.g. [11]). All in all, 34 not necessarily unique species are included, spread across 16 benchmarks. Within a benchmark, multiple species are numbered sequentially.

The experiments in this section include all 5 targets supported by BONES. An overview of the targets and the corresponding experimental set-up is given in table 4.2. The OpenCL and C compilers are instructed to auto-vectorise code in order to use the CPU's AVX vector instructions. The CPU's Turbo Boost technology is disabled. PolyBench is configured to use 'large datasets' and single-precision floating point numbers. Results of multiple runs are averaged, and each run starts with a warm-up dummy computation followed by a cache flush.

target	language	compiler	hardware	core count
GPU-CUDA	CUDA	NVCC 5.0	NVIDIA GTX 470	448 CUDA
GPU-OpenCL-AMD	OpenCL	AMD APP 2.7	AMD HD7950	1792 stream
CPU-OpenCL-AMD	OpenCL	AMD APP 2.7	Intel Core i7-3770	4 (8 threads)
CPU-OpenCL-Intel	OpenCL	Intel OpenCL '12	Intel Core i7-3770	4 (8 threads)
CPU-OpenMP	OpenMP	GCC 4.6.3	Intel Core i7-3770	4 (8 threads)
CPU-reference	C	GCC 4.6.3	Intel Core i7-3770	4 (8 threads)

Table 4.2: Configuration set-up for the 5 parallel targets and the reference target.

The latest versions of two state-of-the-art polyhedral-based compilers are included for comparison: PAR4ALL (version 1.4.1, May 2012) [22] and PPCG (version 0.01, July 2013) [139]. Apart from C-to-CUDA [29] (limited to kernel generation only) and Pluto [34] (evolved into PPCG), these are the only available fully-automatic compilers able to generate CUDA code directly from C. The tiling options for PPCG are set to default to keep the results fully automatic: manual tuning of the tiling parameter is not performed.

4.5.1 Evaluating compiler optimisations

Section 4.2.2 introduced the optimisations thread coarsening, register caching and zero-copy host-accelerator transfers. Furthermore, sections 4.3 and 4.4 introduced host-accelerator transfer optimisations and kernel fusion. This section evaluates these optimisations separately.

Thread coarsening is evaluated for the GPU-CUDA target. Figure 4.14 shows the speed-ups when applying coarsening (2x, 4x and 8x) over non-coarsened code. Several benchmarks (`2mm`, `3mm`, `gemm`, `syr2k` and `syrk`) benefit significantly from coarsening. In fact, these benchmarks are the cases for which coarsening is enabled by default based on the species (see section 4.2.2 for details) with a factor 4 (based on experimental observations). Several other benchmarks that use a coarsening factor of 4 per default benefit only slightly: `doitgen`, `fdtd2d`, `jacobi1d` and `jacobi2d`. Other benchmarks (not using coarsening per default) are either not affected or suffer from performance loss because of a reduced amount of parallelism and/or degraded cache performance. A detailed study of the effects of coarsening is beyond the scope of this work. Other work (e.g. [96]) discusses the effects of coarsening on divergent control flow.

Figure 4.15 shows the impact of register caching for the GPU-CUDA target. The results are either positive or neutral, and show a geometric mean speed-up of 1.8x. Register caching is in particular beneficial for the PolyBench suite, because its benchmarks favour the use of array references above the introduction of new temporary variables. The introduction of such variables (register caching) can therefore greatly reduce the number of memory accesses made.

Zero-copy is a technique to perform a pointer-only copy for OpenCL programs with a shared memory (e.g. CPUs). The left hand side of figure 4.16 shows the benefits of omitting the data transfers for the Intel OpenCL CPU target. The benefit is minimal for cases where the kernel execution time dominates the total execution time (e.g. `syr2k` and `syrk`). In other cases (e.g. `jacobi1d` and `jacobi2d`), the pointer-copies save significant time. In a few cases (e.g. `3mm`), the elimination of the data transfers can even improve cache behaviour.

The right hand side of figure 4.16 shows the speed-ups obtained by performing the host-accelerator data transfer optimisations as described in section 4.3. The speed-up is significant in cases where data transfers can be moved outside of loops that contain species (e.g. `jacobi1d` and `jacobi2d`). Overall, this results in a geometric mean speed-up of 1.4x.

Kernel fusion is evaluated in two parts. First, three combinations of two artificial loop nests are evaluated, which are shown in the bottom half of figure 4.17. The first two loops contain unrelated computations (`B` $\leftarrow$ `A` and `D` $\leftarrow$ `C`), the second set shares a read (`B` $\leftarrow$ `A` and `D` $\leftarrow$ `A`), and the last two share a read/write access (`B` $\leftarrow$ `A` and `D` $\leftarrow$ `B`). The dimension `N` is set to 2048^2, and the `work` function performs a configurable amount of self-multiplications (e.g. `t1*t1*t1`). The top half of figure 4.17 shows the speed-ups of fused code over non-fused code for three targets, timing kernel execution only. The following observations are made:

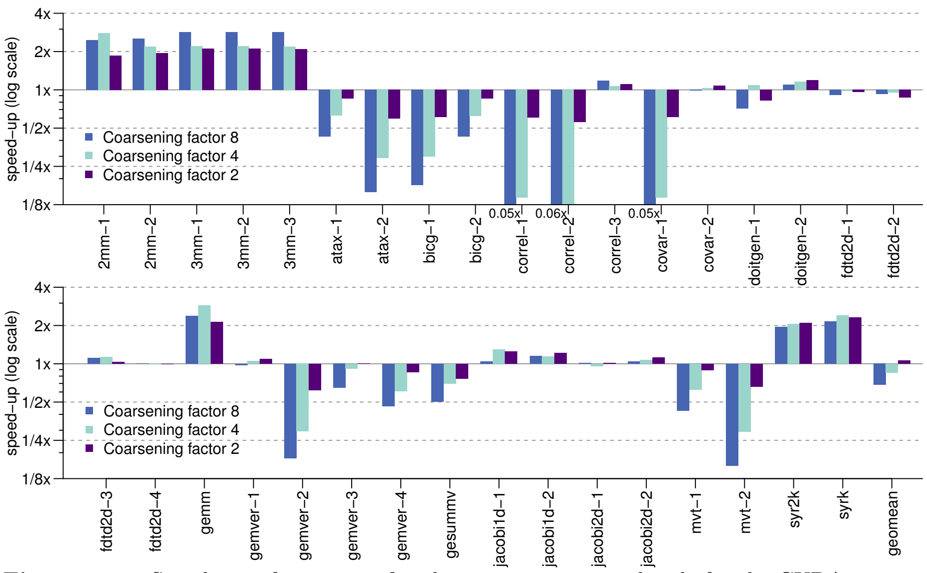


Figure 4.14: Speed-ups of **coarsened** code over non-coarsened code for the CUDA target.

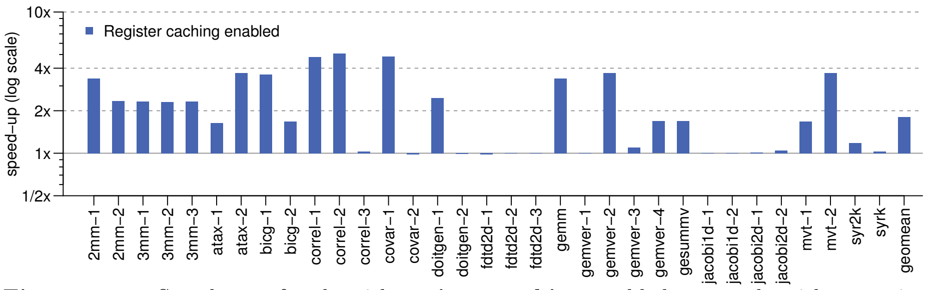


Figure 4.15: Speed-ups of code with **register caching** enabled over code without register caching for the GPU-CUDA target.

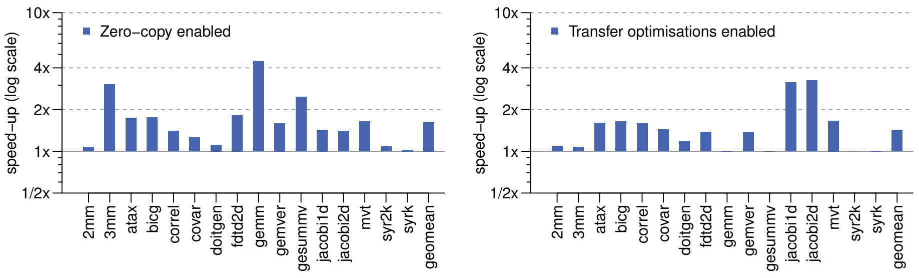
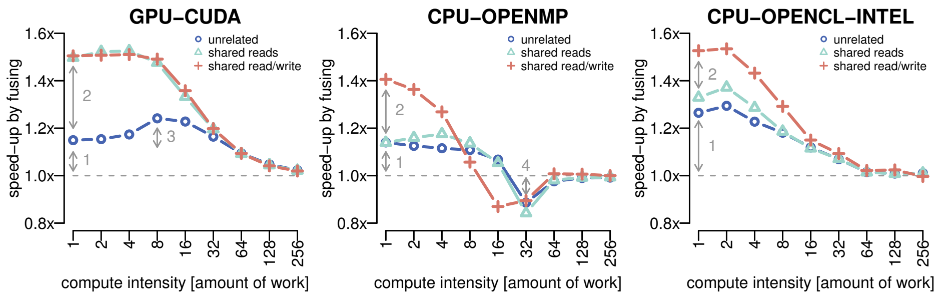


Figure 4.16: Speed-ups of code with **zero-copy** enabled over code with data transfers for the Intel OpenCL CPU target (left) and speed-ups of **host-accelerator transfer optimisations** over naive data transfers for the GPU-CUDA target (right). Execution time of the entire benchmark is considered rather than performance of the individual kernels.



```
for(i=0; i<N; i++) {
  t1 = A[i];
  B[i] = work(t1);
}
for(i=0; i<N; i++) {
  t2 = C[i];
  D[i] = work(t2);
}
```

```
for(i=0; i<N; i++) {
  t1 = A[i];
  B[i] = work(t1);
}
for(i=0; i<N; i++) {
  t2 = A[i];
  D[i] = work(t2);
}
```

```
for(i=0; i<N; i++) {
  t1 = A[i];
  B[i] = work(t1);
}
for(i=0; i<N; i++) {
  t2 = B[i];
  D[i] = work(t2);
}
```

Figure 4.17: Speed-up results when applying **kernel fusion** for three different targets (top) for artificially created code (bottom): two unrelated loop nests (bottom left), two loop nests sharing reads from A (bottom middle), and two loop nests sharing accesses to B (bottom right). The numbered arrows in the graphs refer to phenomena discussed in section 4.5.1.

1. There is a measurable gain when fusing two kernels, even when they are unrelated. This is mainly due to the increased amount of work per thread, allowing for a more efficient execution. For example, common instructions can be shared (e.g. the calculation of the thread index), the compiler has a larger scope for optimisations, and latencies can be hidden more effectively. Kernel launch overhead is also reduced, in particular beneficial for OpenCL.

2. Fusing two kernels can be beneficial for data locality when they access the same data. For the GPU-CUDA target, this saves expensive off-chip accesses for both sharing cases. For the CPU cases however, the only observed performance increase comes from a shared read/write: OpenMP and OpenCL load the value of the shared read twice.

3. The benefit of fusion for two unrelated CUDA kernels actually increases when reaching a moderate amount of work (8–16). This is due to the fact that the (memory) latencies can be hidden better. The relative benefit of fusion decreases as work increases further, as the computations themselves become the bottleneck.

4. Fusion can also decrease performance, as is observed in the OpenMP case. Performance of the fused code is slightly worse (around 10%) at the transition between memory-bandwidth limited execution and compute limited execution (amount of work ±32). This could be due to the increased register usage of the fused version.

Second, kernel fusion is evaluated on the PolyBench suite. Based on algorithm 4.1, four cases of fusion are found: `3mm` (first two kernels), `bicg` (both kernels), `correl` (first two kernels) and `mvt` (both kernels). In other cases, loop fusion in general is possible (as available in PPCG), but kernel fusion in the context of species is not. An example is `2mm`, which allows the outer loops of the two kernels to be fused at the cost of parallelism: the inner loops now become part of the kernel body. Another example is `atax`, which allows fusion after loop interchange of the second loop nest, again at the cost of parallelism. Table 4.3 shows the results of the cases where kernel fusion is legal. In three cases, kernel fusion results in a significant slow-down due to worsened cache performance. In other cases, minor speed-ups are obtained (up to 8%).

target	`3mm`	`bicg`	`correl`	`mvt`
GPU-CUDA	1.00x	0.61x	1.01x	0.52x
CPU-OPENMP	0.96x	1.08x	1.00x	1.00x
CPU-OPENCL-INTEL	0.38x	1.07x	1.08x	1.03x

Table 4.3: Speed-ups when applying kernel fusion to PolyBench for three different targets.

4.5.2 Comparison of multiple targets

BONES has three targets that execute on the same hardware: two OpenCL targets and an OpenMP target. Figure 4.18 shows a comparison of these targets for individual kernels, compared to the reference C code (the input to BONES). On average, this results in speed-ups of 2.4x (Intel SDK OpenCL), 2.8x (AMD SDK OpenCL) and 3.1x (OpenMP). Despite the use of the same hardware, there are still significant performance variations. Differences include the lower thread creation cost for OpenMP, different thread grouping and scheduling policies, and different compilers (and thus also different auto-vectorisers). A detailed comparison of the different targets is beyond the scope of this work, but can be found for other benchmarks in the work by Shen et al. [124]. We furthermore note that performance can be improved by first applying a parallelising and optimising compiler, such as Pluto [34] or the work by Park et al. [109].

The PolyBench/C reference code used to produce the results of figure 4.18 is naive and unoptimised. In fact, simply changing the default compiler (GCC) to ICC (version 14.0.0) yields speed-ups in most cases (geometric mean 1.5x). These speed-ups become even more significant when using ICC instead of GCC for the generated OpenMP code (geometric mean 1.7x).

Next, figure 4.19 shows a comparison of performance across different hardware: two GPUs and a CPU. The two orders of magnitude speed-up observed for some of the GPU targets should be taken with a grain of salt: the comparison is against naive single-threaded CPU code. The two GPUs show comparable performance, a result of the higher theoretical performance of the AMD GPU on one hand (2.8GFLOPS and 240GB/s versus 1.1GFLOPS and 133GB/s), and less optimised

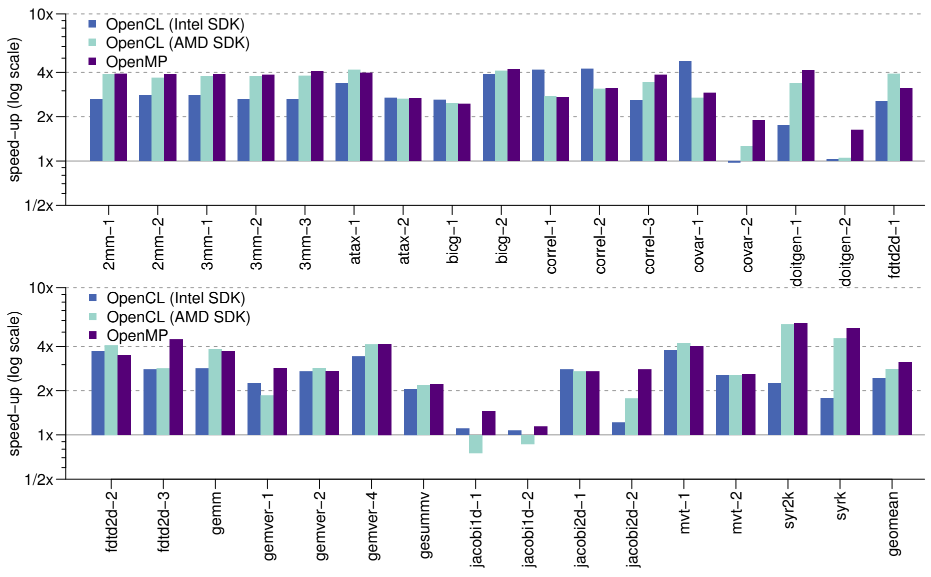


Figure 4.18: Comparison of 3 multi-threaded CPU targets against single-threaded naive CPU code (the reference input code to BONES).

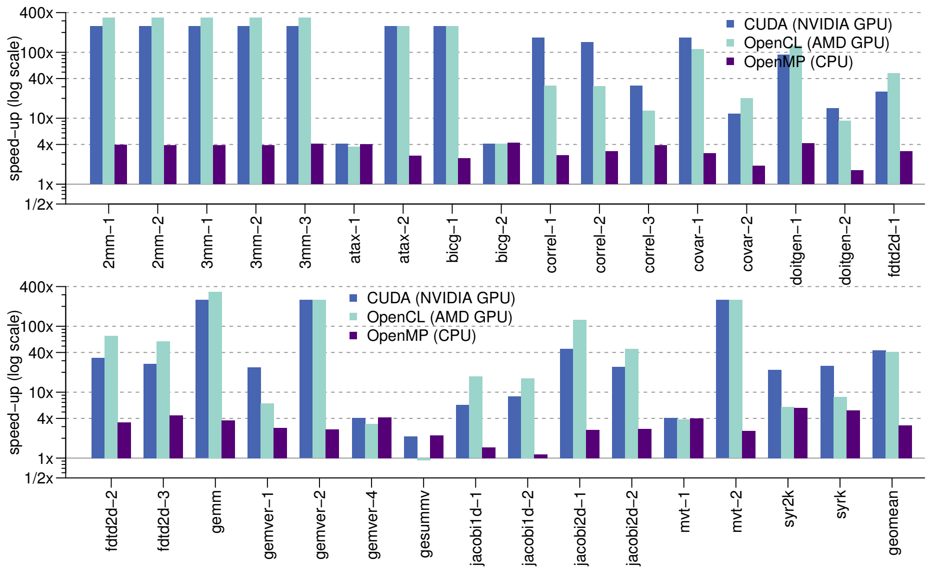


Figure 4.19: Comparison of performance across different hardware: an NVIDIA GPU running CUDA code, an AMD GPU running OpenCL code, and a CPU running OpenMP code. The reference is the single-threaded naive CPU code that is used as input to BONES.

OpenCL GPU skeletons on the other hand (compared to CUDA skeletons). The GPU targets achieve a geometric mean speed-up of an order of magnitude over OpenMP code, limited by several kernels that do not benefit as much as others from GPU acceleration. Examples are the limited parallelism of `correl-3` and the strided (uncoalesced) memory accesses of `mvt-1`.

4.5.3 Comparison against the state-of-the-art

Two experiments are performed to compare BONES with the state-of-the-art C-to-CUDA compilers PAR4ALL and PPCG. The first experiment evaluates the individual kernels only: kernels are generated for each of the algorithmic species in isolation using BONES, PAR4ALL, and PPCG. Figure 4.20 shows the results in terms of speed-up of BONES compared to PAR4ALL and PPCG. The following observations are made:

- BONES shows significantly better performance (2x or more) for 21 kernels compared to PAR4ALL and for 7 kernels compared to PPCG.
- PPCG is measurably faster only for `covar-2` (1.4x). This is a result of parallelisation of only the outer-loop (in contrast to both loops for BONES) which allows tiling of the inner-loop (to improve data locality).
- Several kernels use a skeleton in the form of figure 4.9 to ensure coalesced memory accesses, yielding significant speed-ups over both PAR4ALL and PPCG. The advantages over the default skeleton are 2.4x (`atax-1`), 1.8x (`bicg-2`), 2.3x (`gemver-4`), 1.8x (`mvt-1`) and 7.8x (`syrk`). A related skeleton with two pre-shuffling kernels is used for two other kernels, yielding speed-ups over the default skeleton of 3.2x (`gesummv`) and 15.8x (`syr2k`). The other PolyBench benchmarks use the default skeleton.
- In the cases of matrix-multiplication variants `2mm` and `3mm`, BONES is on-par with PPCG. In these cases, BONES relies on caches and thread coarsening, while PPCG performs loop tiling and uses the local memory.
- In many benchmarks (`2mm`, `3mm`, `doitgen`, `fdtd2d`, `gemm`, `jacobi1d`, `jacobi2d`, `syrk`, `syr2k`), BONES performs thread coarsening, outperforming PAR4ALL. PPCG also performs thread coarsening in most of these cases, but by a factor 2 instead of 4.
- The geometric mean speed-up of kernels generated by BONES is 2.4x compared to PAR4ALL and 1.4x compared to PPCG.

The second set of experiments considers the entire benchmark, or 'scop' in PolyBench terminology. Figure 4.21 shows the speed-up of BONES compared to PAR4ALL and PPCG. Additionally, PolyBench/GPU [70] is included as a reference, which provides hand written non-optimised CUDA code (optimised hand

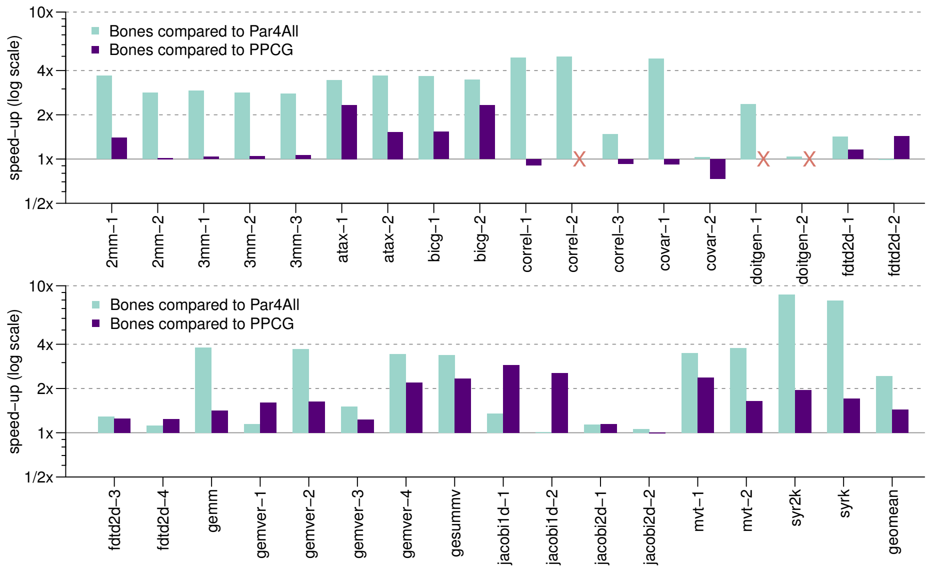


Figure 4.20: Performance of BONES compared to PAR4ALL and PPCG for the GPU-CUDA target (higher is in favour of BONES). PPCG was unable to generate code for `correl-2`, `doitgen-1` and `doitgen-2` (marked by a red cross).

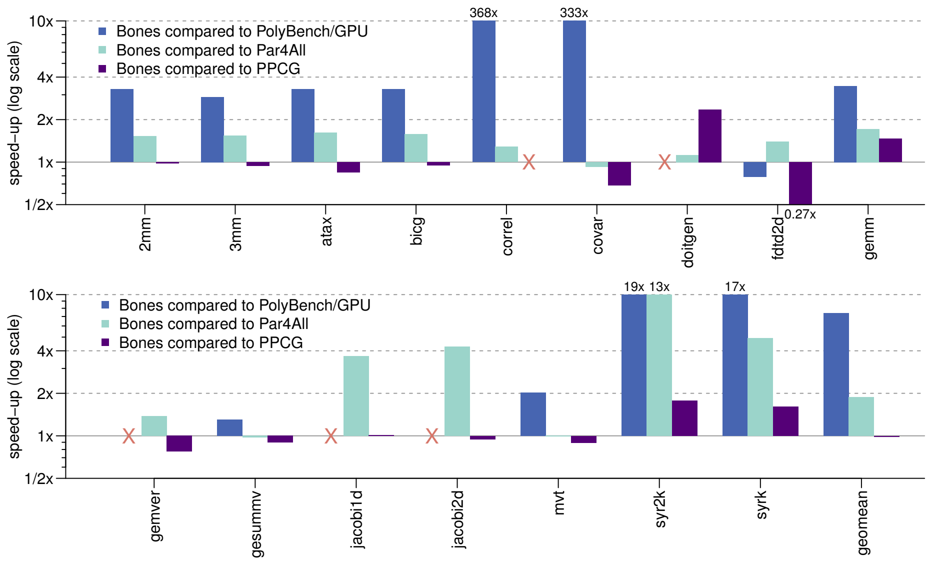


Figure 4.21: Comparison of BONES with PAR4ALL, PPCG and hand-written PolyBench/GPU code for the complete benchmarks. PolyBench/GPU is not available for `doitgen`, `gemver`, `jacobi1d`, and `jacobi2d` (marked by a red cross).

written code is not available). We observe that this hand written code is in many cases significantly slower when compared to compiler generated code: on average 7.4x compared to BONES. We furthermore note that the (experimental) option to optimise CPU-GPU transfers for PAR4ALL (`com-optimisation`) only supports static arrays and does not work for these benchmarks. The right hand side of figure 4.16 gives an idea of the performance gains future versions of PAR4ALL might achieve when such optimisations are fully implemented. Figure 4.21 shows that, in almost all cases, performance of BONES is either on-par or better when compared to the state-of-the-art. The geometric mean speed-ups are 1.9x and 1.0x when compared to PAR4ALL and PPCG respectively. These results are not as good as the results obtained for kernel execution only (figure 4.20). There are two reasons for this:

1. **Inter-kernel optimisations** Kernel fusion is the only inter-kernel optimisation performed by BONES. In contrast, the other compilers are able to distribute loops (fission) and to interchange nested loops (whereas BONES considers the kernels to be fixed). The polyhedral model is well-suited for such optimisations, allowing PPCG for example to do well on `atax` despite the fact that both of its kernels individually perform worse compared to BONES. For this particular example, PPCG creates four kernels out of the original two, enabling loop interchange and resulting in coalesced accesses. Such optimisations (e.g. distribution) are currently not possible within BONES, but could be enabled by using Pluto as a front-end.

2. **Host-accelerator data transfers** Although BONES already performs host-accelerator transfer optimisations, PPCG is able to find additional optimisations. An example is `fdtd2d`, for which PPCG moves several CPU-GPU transfers to the outer 'time' loop. This demonstrates that there is still room for further host-accelerator data transfer optimisations in BONES.

4.6 Discussion

The integration of algorithmic species with a skeleton-based compiler has resulted in a unique source-to-source compilation approach. This novel combination can be seen as a way to profit from the benefits of skeleton-based compilation without its main drawbacks.

Skeleton-based compilation has several benefits. Firstly, compilation requires only basic transformations that can be performed at abstract syntax tree level, omitting the need for intermediate representations that often lose code structure and variable naming. This allows the compiler to generate readable code, enabling opportunities for further fine-tuning and manual optimisation. Furthermore, the skeletons themselves can be formatted to include structure and code comments to improve readability. Secondly, skeleton-based compilation benefits from the flexibility of being able to improve the compiler (by modifying the skeletons) or extend to other targets (by writing new skeletons). An example is the recent addition of an FPGA high-level synthesis (HLS) target. Finally, several 'optimisations' within skeletons are not permutations of the original code. An example is the additional pre-processing kernel of figure 4.9 (lines 10–21), which cannot be described as a transformation of the original code.

Compared to other skeleton-based compilers, BONES is the first that can be used in an automatic tool-chain because of the integration of algorithmic species. This removes the requirements of existing skeleton-based approaches (e.g. SkePU and SkeCL) to manually select skeletons and perform code modifications. On top of this, the integration with algorithmic species provides a clear, structured, and formally defined way of using skeletons, which can be beneficial in cases where manual classification is unavoidable.

However, the possibility of generating efficient code using skeletons and species is limited. This is illustrated by figure 4.22, which numbers the two main reasons: 1) not all optimisations can be expressed in the form of a skeleton, and 2) algorithmic species do not contain all performance relevant details. An example of the first is thread coarsening, which can be applied (yes or no) based on the species, but cannot be implemented in the form of a skeleton. An example of the second is register caching, which is a traditional compiler optimisation independent of the code's algorithmic species. BONES is therefore designed as a combination between a skeleton-based compiler (with the previously discussed benefits) and a traditional compiler (allowing for competitive performance).

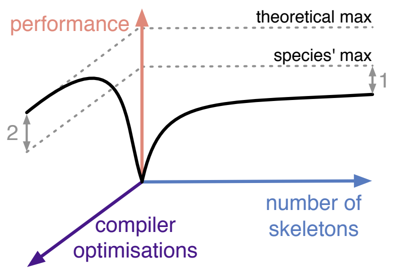


Figure 4.22: Comparison of the theoretical achievable performance of traditional compiler optimisations with skeleton-based optimisations. The numbers refer to performance limitations discussed in this section.

Another aspect of skeleton-based compilation shown in figure 4.22 is the performance with respect to the number of (optimised) skeletons implemented. The more skeletons are provided for the different species, the higher the aggregated theoretical performance. However, performance is limited by the earlier mentioned aspects (the two numbers in figure 4.22), but with a good reason: if an increasing amount of detail is captured by the algorithm classification, implementing skeletons converges towards implementing a library. In practise, a small amount of skeletons can already deliver good performance (in combination with compiler optimisations). An example is a 'default' skeleton for the CUDA target, which maps the species' parallel loop iterations onto GPU threads.

Apart from being a combination of a skeleton-based compiler and a traditional compiler, BONES is also unique in the sense that it explicitly uses species as an intermediate step. In this way, code analysis (ASET or A-DARWIN) is separated from the actual compilation (BONES). This has two main advantages: 1) either of the two steps can be replaced or reused in other work, and 2) the programmer can help the compiler where needed by modifying species or manually adding species to unclassified code.

We make a final note that BONES is a proof-of-concept: many of the optimisations discussed in this chapter are in reality complex problems that require thorough analysis. Examples are thread coarsening for divergent code [96], host-accelerator transfer optimisations [72, 81] and kernel fusion [47, 86, 116]. Nonetheless, the experimental results for the PolyBench benchmark suite have shown that the combination of skeletons and compiler optimisations within BONES can deliver competitive performance. In comparison to two state-of-the-art automatic compilers (PAR4ALL and PPCG), BONES obtains geometric mean speed-ups of 2.4x and 1.4x for CUDA kernels. Future work will be required to verify these numbers for case-studies and other benchmarks. The targets other than GPU-CUDA have not been compared against existing compilers, but do show significant speed-ups compared to the single-threaded input code to BONES.

"Arthur blinked at the screens and felt he was missing something important. Suddendly he realised what it was. 'Is there any tea on this spaceship?' he asked."

- Douglas Adams (The hitchhikers guide to the galaxy, 1979)

Chapter 5
Towards a programmable GPU architecture

So far, this thesis has improved the programmability of GPUs by presenting new program code classification techniques and a new source-to-source compiler. Apart from this programming language and compiler-based approach, the programmability of GPUs can also be improved from a processor architecture point of view. Although this chapter mainly considers modelling and performance insight, its end goal is to propose architectural changes to improve programmability.

As discussed in chapter 2, GPUs use on-chip programmer-managed memory (*scratchpad*) and hardware-managed memory (*cache*) to counter the memory wall [61]. In particular for integrated solutions with general-purpose memories (e.g. ARM Mali, AMD Fusion), off-chip bandwidth is scarce: the GPU's full potential can only be exploited when using the on-chip memories efficiently.

The goal of a GPU's on-chip memory is not to reduce latencies (as is the case for CPUs), because GPUs are designed to hide their memory latencies through fine-grained multi-threading. Instead, the GPU's on-chip memories serve the purpose of reducing the off-chip memory traffic. An increased cache hit rate will translate to performance improvements for memory-intensive programs, as off-chip memory traffic (the performance limiting factor) is decreased proportionally. In fact, many GPU programs are memory bandwidth intensive: for an example set of benchmarks, this is as much as 18 out of 31 [84]. Specific examples of cache optimisations include cache blocking for sparse matrix vector multiplication (5x speed-up) [149] and tiling for a stencil computation (3x speed-up).

With programming models such as CUDA and OpenCL, programmers create a large number of independent[1] threads that execute a single piece of program code (a *kernel*). Still, microprocessors such as the GPU do not exploit the potential of spatial and temporal data-locality enabled by this independence. Therefore, section 5.2 proposes *locality-aware thread scheduling*: changing the schedule of threads, warps and threadblocks to match a kernel's memory access patterns.

To get insight into locality and cache behaviour, section 5.1 first presents a detailed model of the GPU's caches. This model is created to evaluate the quality of thread schedules in the context of locality-aware thread scheduling, but can also be used to perform *design space exploration* of the cache parameters.

NVIDIA's Fermi architecture is used as an example throughout this chapter. Fermi has up to 16 cores, each containing 32 processing elements and a 64KB on-chip data memory, configurable as a combination of a scratchpad and a L1 cache (16/48KB or 48/16KB). All cores share an L2 cache of up to 768KB. Threadblocks are mapped in their entirety onto a core. Together, threads from one or more threadblocks can form a set of *active* threads on a single core. Such a set of active threads executes concurrently in a multi-threaded fashion as *warps*: groups of 32 threads executing in lock-step in an SIMD-like fashion on a single core, dividing the workload over the core's processing elements [107].

Figure 5.1 shows an example of a 16KB 4-way associative L1 cache, as is available in Fermi GPUs [147]. The cache can store 128 *cache-lines* of 128 bytes each [147]. The 128 lines are divided over 32 sets, each containing 4 lines: every memory address is mapped onto one of the sets of the cache using a *mapping function*. Reads to the off-chip memory are cached in the L1 cache, writes are not. The GPU's cache replacement policy is unknown, however, GPGPU-Sim [27] assumes a least-recently-used (LRU) policy, although no proof is given.

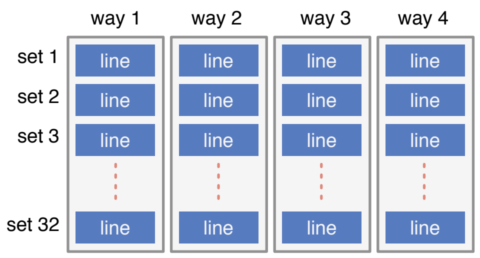


Figure 5.1: A 16KB 4-way associative L1 cache with 128 byte lines, as found in Fermi GPUs.

5.1 A detailed GPU cache model

Since GPU's rely on their on-chip memories to reduce off-chip memory traffic, optimising GPU programs for cache locality has become important for perfor-

[1]Independent apart from explicit per-block synchronisation barriers.

mance and energy. However, to be able to perform cache locality optimisations efficiently, *insight* into the types of cache misses and a *prediction* of the amount and type of cache misses is essential, as shown for example in [24, 32, 83, 119]. A cache model can also be used to guide compilers to select their optimisation parameters, e.g. a loop-tiling factor and a thread coarsening factor. An example is the polyhedral model based C-to-CUDA compiler PPCG [139], which leaves the problem of tile-size selection to the programmer because of a lack of insight into cache behaviour. Additionally, a model can accelerate design space exploration, i.e. finding cost-efficient values for cache parameters such as associativity or the cache-line size. An analytical cache model can thus help to obtain insight into cache usage, to guide programmers and compilers, and to evaluate the effects of cache parameters on cache miss rates.

A well-known cache model is the 3C model [75], distinguishing three types of cache misses: 1) compulsory (or cold): misses because of a first time access, 2) capacity: misses because of a limited cache size, and 3) conflict: misses due to a limited cache associativity or a non-ideal replacement policy. To estimate the amount of cache misses based on the 3C model, a *reuse distance* profile (or 'stack') can be constructed from a memory access trace [32]. The reuse distance theory keeps track of memory requests, moving recently used addresses to the top of an address stack. Addresses not yet present in the stack are the compulsory misses, and addresses with a stack depth larger than the cache size are the capacity misses. Although this model does not take conflict misses into account, it gives a good lower bound for the total miss rate on sequential architectures [32] and even on multi-core CPUs [122].

Existing performance and power models for GPUs (e.g. [26, 76]) have not included a cache model up to now: they are only valid for (older) GPUs without data caches. However, understanding cache behaviour is important as off-chip memory bandwidth is becoming increasingly scarce relative to compute power [61]. The main challenges of creating a cache model for GPUs lie in the execution model: as we will see in this section, fine-grained multi-threading and parallelism make it non-trivial to find the order in which memory requests appear to the cache. Therefore, this work focuses on the GPU's L1 data caches: after it is known in what order memory accesses appear in the L1 cache and which of those miss, existing multi-core CPU models can be applied to model the GPU's L2 cache. Because reuse distance theory can only be applied to an ordered memory access trace, it is not directly suited for GPUs. This section therefore extends the reuse distance theory to model GPU caches. The following extensions are presented:

1. The reuse distance theory is adjusted to match the GPU's **parallel execution** model (section 5.1.3). This includes modelling threads, warps, threadblocks, cores, and sets of active threadblocks.

2. The GPU's **memory latency** is modelled by keeping track of in-flight accesses and their arrival times, introducing a new type of misses: *latency*

misses (section 5.1.4). Furthermore, the memory's non-uniformity is modelled by sampling from a half-normal distribution.

3. Limited **associativity** is modelled by creating a private reuse distance stack per cache set (section 5.1.5). The GPU's mapping of addresses to sets is identified through a micro-benchmark.
4. The effects of **miss-status holding-registers** (MSHRs) are modelled, storing in-flight memory request information (section 5.1.6).
5. Threads within a warp are executed in lock-step, but individual warps can make different progress. This **warp divergence** is modelled by simulating a thread-pool from which warps can be selected for execution (section 5.1.7).

The model is implemented in C++ (optimised for performance) and the source-code is available on-line[2], including a custom CUDA memory access tracer for the Ocelot emulator [50] (section 5.1.8). Apart from proposing the 5 extensions to the reuse distance theory, this section also: 1) finds two architectural details through micro-benchmarking (section 5.1.9), 2) validates the model for two cache configurations and for two benchmark suites (section 5.1.10), and 3) demonstrates the usability of the model for design space exploration by showing an example cache parameter sweep (section 5.1.11).

The GPU's L1 cache handles only off-chip loads: stores are handled by the L2 cache only, not by the L1 cache [107]. Therefore, only loads are considered, although the cache model can be applied to stores as well.

5.1.1 Related work

There is only a single other complete GPU cache model presented in the literature (to the best of our knowledge). This model by Tang et al. [132] is also based on reuse distance theory. However, there are a number of reasons why we propose a new cache model. First, in contrast to our work, Tang et al. model only a single threadblock, assume warps to execute in lock-step, do not model MSHRs and the mapping of addresses to sets, and do not give any details on the used memory latency model. Second, their validation is very limited: 1) they validate against a GPU simulator, not against real hardware, and 2) they include only basic, hand-picked kernels with non-representative input data-sizes. Third, their model is limited to kernels that can be statically analysed. In contrast, we support any GPU kernel: we use an emulator to generate traces. Our final reason is practical: their model is not available in the form of source-code or binary.

Another cache model [108] is part of a performance model of a complete GPU, but assumes hit and miss rates to be given. Furthermore, other work has used reuse distance to analyse non-GPU multi-core and many-core workloads [43, 120, 122]. In contrast to our work, they investigate cache contention caused by running

[2]Source-code is available at '`http://github.com/cnugteren/gpu-cache-model/`'.

multiple programs on different cores. Because they do not target GPUs, many of their assumptions (e.g. no data reuse among threads, execution order known) are not valid for our work (and vice versa).

5.1.2 Background: reuse distance theory

This section briefly introduces reuse distance theory. The used cache-related terminology is as follows [38]. 'Cache-line' describes a location in the cache, while 'cache-block' refers to the data that goes into a cache-line. Furthermore, **S** represents the number of sets in a cache.

Given an ordered memory access trace, a reuse distance profile (or stack) [32] can be computed as follows. For each access, the reuse distance is the number of unique addresses accessed between this access and the most recent previous access to the same address. When there is no previous access, the distance is set to infinity (∞). Constructing a reuse distance profile can be done at for example address granularity or at cache-line granularity. An example of both is given in table 5.1, assuming cache-lines of 4 elements (time progresses from left to right).

access	x[0]	x[5]	x[3]	x[9]	x[3]	x[3]	x[5]
address	0	5	3	9	3	3	5
distance	∞	∞	∞	∞	1	0	2
cache-line	0	1	0	2	0	0	1
distance	∞	∞	1	∞	1	0	2

Table 5.1: Example of reuse distance computation, assuming a cache-line size of 4 elements.

A reuse distance profile can be used directly to obtain cache hit/miss rates. Given a fully-associative cache of n lines with a least recently used (LRU) replacement policy, any access with a reuse distance d larger than or equal to n ($d \geq n$) will miss. Vice versa, when $d < n$, the access will hit in the cache. In this way, the reuse distance profile at cache-line granularity gives the compulsory miss rate ($d = \infty$) and the capacity miss rate ($d \geq n$ and $d \neq \infty$). For the example in table 5.1, given a cache size of 2 lines, we find 3 compulsory misses (42%), and 1 capacity miss (14%). This can also be visualised by constructing a histogram, containing all necessary data to compute compulsory and capacity miss rates. A histogram for this example is given in figure 5.2 (at cache-line granularity).

distance	0	1	2	∞
frequency	1	2	1	3

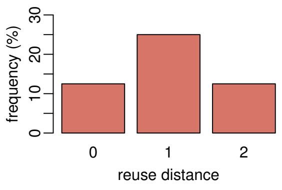


Figure 5.2: Table 5.1's reuse distance profile as a table (left) and as a histogram (right).

5.1.3 Parallel execution model

A GPU typically executes thousands of small, light-weight threads. Because of the parallelism expressed in the execution model, these threads can to some extent be executed independently on different cores. Furthermore, due to limited resources (e.g. register file, scratchpad memory), not all threads can be *active* at the same time, i.e. eligible for execution. Tang et al. [132] argue that there is limited reuse across different threadblocks on the same core: they model only a single block of threads on a single core. To create a more realistic model, we do model complete sets of active threads (one or more threadblocks). Furthermore, we model multiple of such sets and multiple cores (because the workload can vary for different cores).

To determine which threads execute together as a set of active threads on a single core, we follow Fermi's execution model (an example is shown in figure 5.3). First, threadblocks are divided round-robin over the cores until they are full. Then, a new threadblock is scheduled when another threadblock is done (first-done, first-serve). For each core, threadblocks are grouped in sets of active threads according to the block-size and the resource limitations as listed in section G.1 of the CUDA programming guide [107]. Furthermore, threads in a warp are scheduled simultaneously. Determining the scheduling *order* among warps in a set of active threads is not straightforward (e.g. dependent on thread workload and cache contents): this is approximated step-by-step in the following sections.

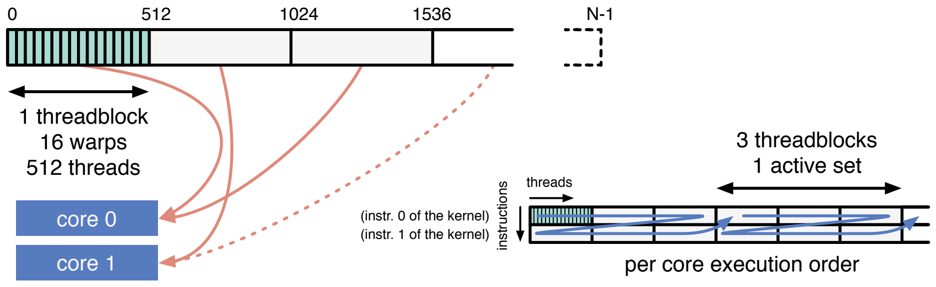


Figure 5.3: Visualisation of the GPU's execution model for an example with blocks of 512 threads and 2 cores (left), and the execution order (blue arrows) for a kernel with 2 load instructions per thread, round-robin scheduling over warps, and 3 active threadblocks (right).

Transforming the parallel execution model into an ordered memory access trace[3] can be done by: 1) applying the GPU's thread-scheduling policy, and 2) by taking into account pipeline and memory latencies. For now, we assume a basic round-robin scheduling policy among warps in a set of active threads (divergence is discussed in sections 5.1.6 and 5.1.7), and zero-latency hardware (latencies are discussed in section 5.1.4). Now, for a given kernel, its execution can be

[3]The traces used are not ordered since they are not obtained from simulation: they are rather unordered lists of memory accesses and only contain ordering information with respect to a single thread. Simple trace example: thread 0 addr 109 782 110, thread 1 addr 209 882 210.

sequentialised to obtain an instruction trace. We illustrate this with an educational example: a kernel with 4 threads, each performing 2 loads (`x[2*tid]` and `x[2*tid+1]`, for which '`tid`' denotes a thread's unique identifier). Given the round-robin scheduling of threads and no latencies, we obtain the reuse distances (for lines) as shown in table 5.2, assuming a cache-line size of 4 elements, a single thread per warp, and only a single set of threads on a single core.

instruction	0	0	0	0	1	1	1	1
thread ID	0	1	2	3	0	1	2	3
address	0	2	4	6	1	3	5	7
cache-line	0	0	1	1	0	0	1	1
distance	∞	0	∞	0	1	0	1	0

Table 5.2: Reuse distance computation for a GPU for an example with 4 threads, each performing 2 loads. A cache-line size of 4 elements is assumed.

However, before a reuse distance profile can be constructed from a given thread order, memory requests need to be combined according to the GPU's *memory coalescing* capabilities. Coalescing is applied in specific cases, for example when threads from a single warp access the same cache-line. Coalescing is implemented according to the specifications of the GPU architecture, as described in section G.4.2 of the CUDA programming guide [107].

5.1.4 Memory latencies

In reuse distance theory for sequential processors it is assumed that either: 1) memory latencies are non-existent, or 2) memory accesses cannot overtake each other. Although individual threads on a GPU do execute kernel code in-order, above assumptions are not valid when considering multiple (concurrent) threads. Moreover, the GPU's memory latencies are typically high compared to CPU latencies. Therefore, reuse distance theory is extended to model the GPU's latencies.

We first introduce the notion of time. Every column in the reuse distance theory is now assigned with a monotonously increasing time-stamp[4]. Now, each access is assigned a specific latency to delay its effect in the reuse distance theory. We illustrate this based on the same example as shown in table 5.2 with 2 accesses and 4 threads. Table 5.3 shows the updated results: every memory request occurs at a fixed time (0–7) and is assigned a latency (a fixed value of 2 in this example). Accumulation of an access's latency with its issued time-stamp determines when the request will have effect in the cache. This is shown in the '*effect at*' row of table 5.3. Now, computation of the reuse distances is no longer based on the '*cache-line*' row, but on the new '*cache effect*' row, as shown in the table. In this particular example, the cache-line data is shifted by 2 time-stamps (highlighted).

Adding the notion of time and latency to the reuse distance theory changes the reuse distances obtained. This can be seen for example by comparing the

[4]These time-stamps are not meant to reflect actual processor clock-cycles or time.

time	0	1	2	3	4	5	6	7	8	9
instruction	0	0	0	0	1	1	1	1	-	-
thread ID	0	1	2	3	0	1	2	3	-	-
address	0	2	4	6	1	3	5	7	-	-
cache-line	0	0	1	1	0	0	1	1	-	-
cache effect	-	-	0	0	1	1	0	0	1	1
distance	∞	∞	∞	∞	0	1	0	1	-	-
hit/miss	m	m	m	m	h	h	h	h	-	-
latency	2	2	2	2	2	2	2	2	-	-
effect at	2	3	4	5	6	7	8	9	-	-

Table 5.3: Extended reuse distance theory with a fixed latency of 2. The same example with 2 loads per thread and 4 threads in total is considered.

distances found in tables 5.2 and 5.3. The addition of latencies can thus transform capacity misses in hits and vice-versa. However, this approach can also introduce additional infinite distances (∞) that are not compulsory misses. To repair this, the notion of *latency misses* is introduced: requests that miss in the cache because an earlier request to the same cache-line is still in-flight.

So far, we have modelled a fixed latency. To better reflect reality, two additional aspects are also modelled: 1) conditional latencies applied depending on the reuse distance, and 2) non-uniform memory latencies. This requires a distinction between cache hits (to model the pipeline latency) and cache misses (to model the memory latency). This requires us to embed information about the cache size in the model, making the reuse distance profile no longer cache-size independent.

The example of table 5.3 is extended to include a hit latency of 0 and a miss latency of 2. Assuming a cache-size of 2 lines, the results as shown in table 5.4 are obtained. We observe that the reuse distances change again, influenced by the reduced latency of the last 4 memory accesses. Furthermore, we note that multiple 'cache effects' can now occur simultaneously at a single time-stamp in the model (highlighted in the table). Such simultaneous accesses are handled in the order in which the memory accesses were issued.

time	0	1	2	3	4	5	6	7
instruction	0	0	0	0	1	1	1	1
thread ID	0	1	2	3	0	1	2	3
address	0	2	4	6	1	3	5	7
cache-line	0	0	1	1	0	0	1	1
cache effect	-	-	0	0	1 0	1 0	1	1
distance	∞	∞	∞	∞	0	0	1	0
hit/miss	m	m	m	m	h	h	h	h
latency	2	2	2	2	0	0	0	0
effect at	2	3	4	5	4	5	6	7

Table 5.4: Reuse distance theory with conditional latencies: 2 for a cache miss and 0 for a hit.

This theory is extended to a more realistic model by clipping the ‘effect at’ time to the time of a still in-flight request for the same cache-line (if present). This will for example change the ‘effect at’ time of the request at time-stamp 1 in table 5.4 from 3 to 2, as the request for line 0 was already made at time-stamp 0.

Finally, the non-uniform latency of accessing the GPU’s off-chip memory is modelled. Because a detailed model of the memory latency is beyond the scope of this work (it requires a full GPU model or simulator, including e.g. the pipeline and interconnect), a probabilistic approach is taken. The memory latency is modelled as $\lambda_{min} + |\mathcal{N}(0, \sigma^2)|$: a fixed minimum latency λ_{min} offset by the absolute value of a normal distribution $\mathcal{N}(\mu, \sigma^2)$ with zero mean, i.e. a half-normal distribution. Parameters are the memory’s best-case latency λ_{min} and a measure for the its non-uniformity: the standard deviation σ of the half-normal distribution.

The ‘latencies’ discussed in this section are not real latencies: the cache model is not a complete GPU model and does not have a notion of actual clock cycles. For example, there can be a varying number of non-memory operations between two memory accesses, affecting the latency between subsequent memory operations. To model the effects of non-memory operations would require integration with a complete GPU model, which is beyond the scope of this work. Therefore, the latencies in the model should be seen as a way to capture the global ordering roughly rather than as a way to obtain an exact reuse distance profile.

5.1.5 Cache associativity

The reuse distance theory models the compulsory and capacity misses, but does not take into account misses caused by the limited associativity of a cache (part of the conflict misses). It has been shown that such misses form a relatively small percentage of the total amount of misses for sequential processors, even in the case of a direct mapped cache [32]. However, typical GPU programs are more sensitive to associativity, because they often show regular memory access patterns on large data structures (e.g. matrix or image operations). To improve the accuracy of our model, we extend the reuse distance theory to model cache associativity.

The reuse distance theory can be extended to model associativity as follows. Instead of keeping track of a single reuse stack, a private stack is created for each set in the cache. In that way, a set becomes a small cache with a size in lines equal to the number of ways, i.e. the associativity. For a fully-associative cache, this reduces again to a single stack because it has only a single set.

Along with the introduction of multiple sets (and their corresponding reuse stacks), we need to define a mapping of memory addresses to sets. Such a mapping can be either obtained directly by taking the last $log_2(\mathbf{S})$ bits from the line address, or by a more advanced *hashing function*, creating a hash-associative cache [38]. The simulator GPGPU-Sim [27] uses a direct mapping for Fermi GPUs, but does not claim that this is realistic. Therefore, because Fermi’s mapping function is not public knowledge, a micro-benchmark was constructed (see section 5.1.9), finding a hashing function with a 5-bits XOR operation for a Fermi GPU.

5.1.6 Miss-status holding-registers

A GPU can have only a finite number of memory requests pending: pending requests are stored in miss-status holding registers (MSHRs), per-core registers that keep track of *in-flight* (in progress) memory requests. The reuse distance theory is extended to model such registers to improve the accuracy of the cache model. MSHRs are organised in such a way that each entry can service a unique cache-line request: requests to the same cache-line are merged into a single entry (up to a certain limit). A limited amount of registers limits the number of outstanding memory requests: either all MSHR entries are occupied when a new cache-line is requested, or an MSHR entry corresponding to a specific cache-line is full. In either case, the active warp will be stalled because it cannot perform any more memory requests. While waiting for an entry to become free, the GPU will process other warps that do not require MSHRs.

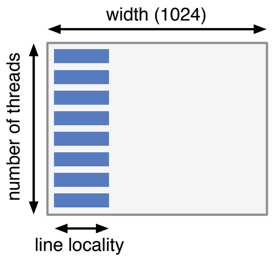
We model the limited amount of MSHR entries, but assume that requests to the same cache-line are merged into a single entry. Our model keeps track of the number of unique outstanding memory requests. Before a warp modifies the reuse stack, it is ensured that it is either a hit or that the number of outstanding requests is not exceeding the number of MSHRs. If the warp cannot continue, it is put on-hold and issued again at a later time. This is illustrated in table 5.5, in which the same example as in table 5.4 is shown, but now with the assumption that there is only a single MSHR available. Only threads 0 and 2 are shown to make the example illustrative and concise. From table 5.5, we see that instruction 0 of thread 2 is cancelled and re-issued at a later time. Also, we see that instruction 1 of thread 2 (issued at time 4) does not have to be postponed: it uses the already occupied MSHR for cache-line 1.

time	0	1	2	3	4	5	6
instruction	0	0	1	0	1	-	-
thread ID	0	2	0	2	2	-	-
address	0	4	1	4	5	-	-
cache-line	0	1	0	1	1	-	-
cache effect	-	-	0 0	-	-	1	1
distance	∞	∞	0	∞	∞	-	-
MSHRs used	0	1	0	0	1	-	-
status	miss	cancel	hit	miss	miss	-	-
MSHRs used	1	-	0	1	1	-	-
latency	2	-	0	2	2	-	-
effect at	2	-	2	5	6	-	-

Table 5.5: Example of reuse distance theory with MSHR modelling, assuming a single MSHR available. Only threads 0 and 2 of the example are considered for illustration purposes.

Similar to the case of the hash function of the cache, it is not publicly known how many MSHRs a GPU core has. The GPU simulator GPGPU-Sim [27] uses a default of 32 MSHRs per core for a Fermi GPU, but does not claim that this value is realistic. Through micro-benchmarking (see section 5.1.9), we find that a Fermi GPU core has 64 MSHRs and a single warp can use only up to 6 MSHRs.

The relevance of modelling MSHRs in our cache model is demonstrated with a simple experiment for the GPU's 16KB (128 lines) cache. The experiment consists of a kernel that performs a copy of a 2D matrix in a column-major fashion: each thread copies an entire row. Cache-line locality is not among threads (accesses are uncoalesced) but within each thread. Figure 5.4 illustrates the experiment and shows the results, varying the height of the 2D matrix (the number of threads: one threadblock only). A constant width of 1024 is set and a data-size of 4 bytes is used. The results show that the measured cache miss rates do not correspond to the miss rates when assuming an ordered round-robin schedule. We conclude from the table that, because of the limited number of MSHRs, certain threads run ahead of others. This can yield to performance improvements (256–1024 threads) or losses (64–128) compared to a fair round-robin schedule. In other words: for a GPU cache model to be accurate, MSHRs need to be modelled.



threads	measured misses	expected (round-robin)
32	3.13%	3.13% $= \frac{1}{32}$
64	3.77%	3.13%
128	32.71%	3.13%
256	42.05%	100.00%
512	67.20%	100.00%
1024	82.28%	100.00%

Figure 5.4: Toy experiment demonstrating the relevance of MSHR modelling (left) and the results for a 2D matrix with a height of 32–1024 and a width of 1024 (right).

For example, when running 128 threads in round-robin, each cache-line can store a single cache-block (for a cache of 128 cache-lines). However, when warps diverge, threads can run ahead and request new cache-blocks while other threads are still using their previous cache-block. Due to a non-oracle replacement policy, this can result in additional misses. On the other hand, when running 256 threads in round-robin, threads 128–255 overwrite the cache-blocks required by threads 0–127. In this case, divergence can only ameliorate cache behaviour: when threads run faster than others, they can benefit from their intra-thread cache-line locality.

5.1.7 Warp divergence

As a final extension to the reuse distance theory, a warp divergence model is introduced. *Warp divergence* is defined as the process that causes program counters of warps to differ from each other as execution progresses. This is not to be confused with the non-cache related concept of *thread divergence*, which describes divergence within warps caused by branch instructions taken by a subset of threads in a warp. Instead, we discuss warp divergence: divergence among warps as a result of aspects such as on-chip local memory bank conflicts, non-uniform memory access latencies, (instruction) cache misses, and per-warp branches in program code.

Because the model does not cover the entire GPU and only uses memory reference traces, not all possible sources of warp divergence are modelled. Instead, the focus lies on the memory-related sources: 1) data-cache hit and miss latencies, 2) non-uniform off-chip memory latencies, and 3) the limited size of the MSHR table. The first two sources are introduced in section 5.1.4 and the third in section 5.1.6. This section models how these sources affect the execution order.

To model warp divergence, the concept of a warp queue is introduced. Initially, the queue is filled with all active warps (from one or more threadblocks) ordered by warp identifier (thread identifier divided by the warp size). As long as the queue is non-empty, a warp is selected based on a first-in first-out (FIFO) policy and a single memory request is processed for each thread in the reuse distance model. After a warp finishes a memory request, it is not directly pushed to the back of the warp queue. Instead, it is delayed proportionally to the corresponding memory request's latency. Furthermore, if a warp does not succeed because all MSHRs are in use, it is immediately sent to the back of the warp queue.

5.1.8 Implementation of the model

This section gives an overview of the model's implementation and infrastructure. The components as shown in figure 5.5 are discussed: A the Ocelot tracer, B the allocation of warps and blocks to cores, C a memory coalescing model, D the reuse distance theory plus extensions, and E verification with hardware counters.

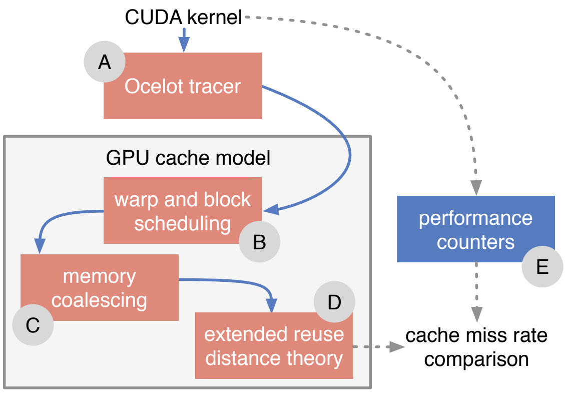

CUDA kernel
A
Ocelot tracer
GPU cache model
warp and block scheduling
B
memory coalescing
C
D
extended reuse distance theory
performance counters
E
cache miss rate comparison

Figure 5.5: The infrastructure of the cache model, including a tracer and hardware verification.

The Ocelot GPU emulator [50] is used to produce (unordered) memory access traces for CUDA kernels. A custom tracer (A) is implemented on top of Ocelot, creating a trace containing for each access: 1) the thread ID, 2) whether it is a read or a write, 3) the memory address, and 4) the size of the memory access.

Because Ocelot does not simulate the GPU, the 'traces' are actually unordered lists of memory accesses rather than ordered traces that can be obtained from simulators. The only ordering in the traces is with respect to the instruction stream within an individual thread.

Before the reuse distance theory can be applied, the memory accesses have to be ordered. Therefore, we first perform the allocation of threads to warps, warps to threadblocks, and threadblocks to cores (B). We follow the GPU's execution model as discussed in section 5.1.3 and in section 4.1 of the CUDA programming guide [107]. This thread to warp allocation can be modified when running the cache model for architecture exploration purposes, e.g. to implement a warp scheduler performing thread block compaction [62] or two-level warp scheduling [105][5].

Next, a memory coalescing model (C) is implemented according to the behaviour as defined in section G.4.2 of the CUDA programming guide [107]. Coalescing is modelled before applying the reuse distance theory, as this can give a significant reduction in computational and memory complexity of the cache model: coalescing can compact the memory trace significantly.

All extensions to the reuse distance theory are implemented on top of the original theory (D). This allows reuse of the already available efficient implementations for sequential processors [20]. A naïve implementation of a reuse distance stack has a computational complexity of $\mathcal{O}(NM)$, in which N is the trace length (the total number of memory accesses) and M the number of unique accesses. To handle the GPU's large number of threads and accesses, a more computationally efficient version is used: a binary-tree C++ implementation of Bennett and Kruskal's algorithm [20]. This implementation has a computational complexity of $\mathcal{O}(N\log(N))$, is independent of M, and gives a better scaling for traces where M is proportional to N. When modelling associativity, we increase the complexity by creating a binary-tree for each set in the cache. However, because the number of accesses per set is pre-computed, the size of each tree is reduced accordingly, achieving an overall comparable complexity. Further optimisations could be made to reduce the memory footprint (currently around 2GB for benchmarks from section 5.1.10), for example using a splay tree [20, 51].

To reduce the overall complexity and computational requirements, the number of threads can be limited in two ways: 1) a limited number of cores can be modelled, generalising results across all cores, and 2) a limited number of threads can be modelled. These core and thread counts are configurable parameters, set to a single core with up to 8192 threads for our experiments.

Finally, a verification method based on hardware counters (E) is included in our infrastructure. NVIDIA's profiler NVPROF is used to output the measured number of cache-line hits and misses in the L1 data cache. The comparison of these numbers with the cache model's result is automated.

[5]Note that implementing such techniques will require additional information from an external source, e.g. from the Ocelot emulator.

5.1.9 Micro-benchmarks

To complete the models of sections 5.1.5 (associativity) and 5.1.6 (MSHRs), additional information was required. This information was obtained through micro-benchmarking: carefully designing a benchmark to extract details on the GPU architecture. This section describes the micro-benchmarks and their results.

Associativity micro-benchmark

The first micro-benchmark is designed to find the mapping of addresses to cache sets, crucial information to model associativity. Our micro-benchmark (shown in figure 5.6) launches a single block of 128 threads (4 warps), each performing 3 stages. In the first stage, each thread performs 32 coalesced loads designed to fill the entire 16KB of the L1 cache with subsequent addresses (an assumption at this point). This access pattern is repeated in the third stage while measuring the latencies of the individual loads. If we do not perform anything in the second stage, all loads show a low latency and are thus cache hits. This verifies our assumption. Now, performing a single load in the second stage will give increased memory latencies for some of the loads[6] in the third stage, as they become cache misses. By performing a sweep over different loads for the second stage, a mapping of addresses that belong to the same set is obtained. We find only up to 4 cache misses each time in the third stage as long as line-aligned accesses are performed: this is because Fermi's 16KB L1 cache is 4-way associative [147].

```
__global__ void mb1(int* mem, int* time, int sweepval) {

  // Stage 1
  for (i=0; i<32; i++)
    temp = mem[tid + i*128];

  // Stage 2
  if (tid == 0)
    temp = mem[sweepval];

  // Stage 3
  for (i=0; i<32; i++) {
    start = clock();
    temp = mem[tid + i*128];
    time[tid + i*128] = clock() - start;
  }
}
```

Figure 5.6: Micro-benchmark to find the mapping of memory addresses to sets. The code is heavily simplified, omitting for example synchronisation barriers.

From the obtained mapping, the hashing function used to map addresses to sets is reverse-engineered. For the 16KB cache with 32 sets, we find that the 5 bits

[6]The number of misses is dependent on the order of accesses by the 128 threads and the cache replacement policy.

7–11 and the 5 bits 13, 14, 15, 17, 19 of the byte-address are input to an XOR port to obtain a $log_2(\mathbf{S}_{16KB}) = 5$ bits set index, as shown in figure 5.7. The first gap in the address (the 12th bit) is a consequence of the cache configuration possibilities: Fermi's cache can also be configured as a 48KB 6-way associative cache with 64 sets. If the micro-benchmark is repeated, we find that the $log_2(\mathbf{S}_{48KB}) = 6$ set index bits are constructed by taking the 16KB's 5 bits (after the XOR operation) and prefixing bit 12.

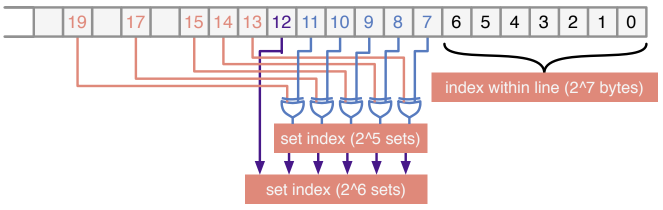


Figure 5.7: Details of the mapping of the byte-address bits within a cache-line and for the set index, shown for both 16KB (32 sets) and 48KB (64 sets) L1 caches.

To verify the found hashing function, an experiment with strided accesses is performed for the 16KB case. We construct a kernel with two identical loops, each time performing a number of non-overlapping 128-byte coalesced loads. The kernel is configured with a single warp only. The miss rate is measured at cache-line granularity using NVIDIA's profiler NVPROF. A sweep is performed over the number of loop iterations and the stride of the memory accesses. The results are shown in figure 5.8, showing either a cache miss rate of 100% (misses in both loops) or 50% (only misses in the first loop). The final row counts the number of set index bits varied across the loads, derived as the number of 50% miss rates in the row minus 1 (4 loop iterations always fit in a single set). The figure confirms our hypothesis, as the number of varied set bits corresponds to the number of bits included in the hashing function counting from the log_2 of the stride. For example, with a stride of 2^{12} and 128 loads, bits 12–18 are included, of which only 4 bits (13, 14, 15, 17) are used in the computation of the set index.

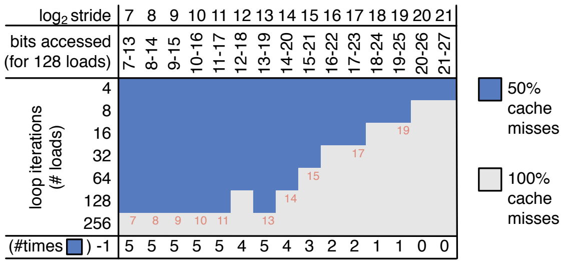


Figure 5.8: Results of the hash function verification experiment, with the number of varied set index bits given in the final row. In red are the additional set bits used (from right to left).

MSHR micro-benchmark

Similar to the case of the hash function for associativity, it is not known how many MSHRs are available in the GPU. Therefore, we constructed the following micro-benchmark to find the number of MSHRs per core. Initially, a CUDA kernel with only a single thread is launched. The kernel, as shown in figure 5.9, performs a configurable number of non-overlapping loads without dependences, which are timed in their entirety. The idea is that the GPU will issue multiple loads at a time, limited by the number of MSHRs. The results are shown in figure 5.10 for a varying number of warps and a varying number of loads per warp[7].

```
__global__ void mb2(int* mem, int* time) {
  if (tid % 32 == 0) {
    start = clock();

    // Loop of independent loads (unrolled)
    for (i=0; i<NUM_LOADS; i++)
      temp = mem[32*(tid + i*NUM_WARPS*32)];

    time[tid/32] = clock() - start;
  }
}
```

Figure 5.9: Micro-benchmark to find the number of MSHRs (simplified for readability).

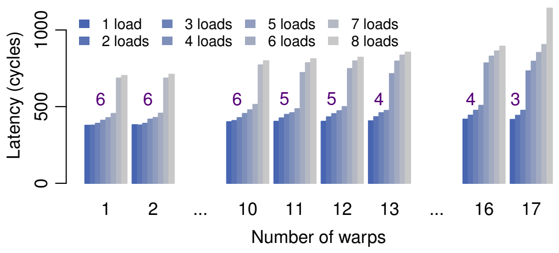


Figure 5.10: Selection of micro-benchmark results, showing the latency of 1–8 loads per warp for a single threadblock with 1–17 warps. The floating numbers above the bars indicate the number of low latency loads.

When evaluating the results of figure 5.10 for a single warp (leftmost bars), we see that performing up to 6 loads yields a similar latency. When performing an additional 7th load, we observe a sudden increase in latency. From this data, we conclude that there are only 6 MSHRs available in this case: performing a 7th (or 13th, 19th, etc.) request increases the latency significantly. However, when evaluating the results of launching multiple warps, we observe that additional

[7]We iterate over the number of warps and run with a single thread because threads cannot be timed individually: threads in a warp run in lock-step.

accesses can be performed without increasing the latency significantly[8]. In fact, this is true for up to 10 warps, allowing a total of $10 \cdot 6 = 60$ simultaneous requests. The figure shows a decrease to 5 loads per warp for 11 warps, 4 for 13 warps, and 3 for 17 warps. From this, we conclude that there are 64 MSHRs (e.g. 16 warps with 4 simultaneous requests each). We also conclude that a single warp is only allowed to use up to 6 entries, although this could be unrelated to MSHRs, e.g. there could be a limit on the number of outstanding incomplete *instructions*.

5.1.10 Verification of the model

To demonstrate the usefulness and accuracy of the cache model, the modelled cache miss rates are compared against miss rates using hardware counters on a Fermi GPU and against a simulator. The verification is performed for both the 16KB 32-set 4-way and 48KB 64-set 6-way cache configuration on a GeForce GTX470[9]. To ensure a wide variety of kernels, two complete benchmark suites[10] are included: PolyBench/GPU [117] and Parboil [128]. The only exclusions made are the `mb_sad_calc` kernel from Parboil's `sad` benchmark, because it relies on the GPU's texture memory and texture cache, and the `histo_main` kernel from Parboil's `histo` benchmark, as it only uses atomic memory accesses. PolyBench/GPU is configured to use default data-sizes, and Parboil to use the 'medium' inputs (or 'large' where unavailable). The iteration limit is set to a maximum of 2 for Parboil benchmarks with multiple iterations. The two benchmark suites differ significantly: PolyBench/GPU contains mostly naive implementations of matrix-multiplication variants (e.g. no on-chip memory usage, limited parallelism), whereas Parboil contains heavily optimised kernels of all sorts. This difference also becomes apparent when measuring the impact of disabling the GPU's L1 data-cache (the L2 is still enabled): the geometric mean performance drops by 5% (Parboil) and 15% (PolyBench/GPU).

Comparison against hardware counters

Using the infrastructure described in section 5.1.8, modelled and measured cache miss rates are collected for the two cache configurations. The results are shown in figure 5.11 for the 16KB configuration, in which the kernel invocations are mapped to the x-axis. Bracketed letters are used in case a kernel is invoked multiple times. For each kernel, we show on the left hand side the modelled L1 data cache miss rate, and on the right hand side the measured miss rate using the profiler. The following types of modelled misses are distinguished: 1) compulsory misses, 2) capacity misses, 3) associativity misses, 4) MSHR misses, and 5) latency misses. We make the following observations from the results:

[8]As the number of instructions increases when performing more loads or running more warps, the measured latency increases a bit as well.

[9]16KB 4-way and 48KB 6-way are the only cache configurations for Fermi GPUs. Newer Kepler GPUs also support a 32KB configuration [107].

[10]This also includes cases where 'caching' is performed manually using the scratchpad memory.

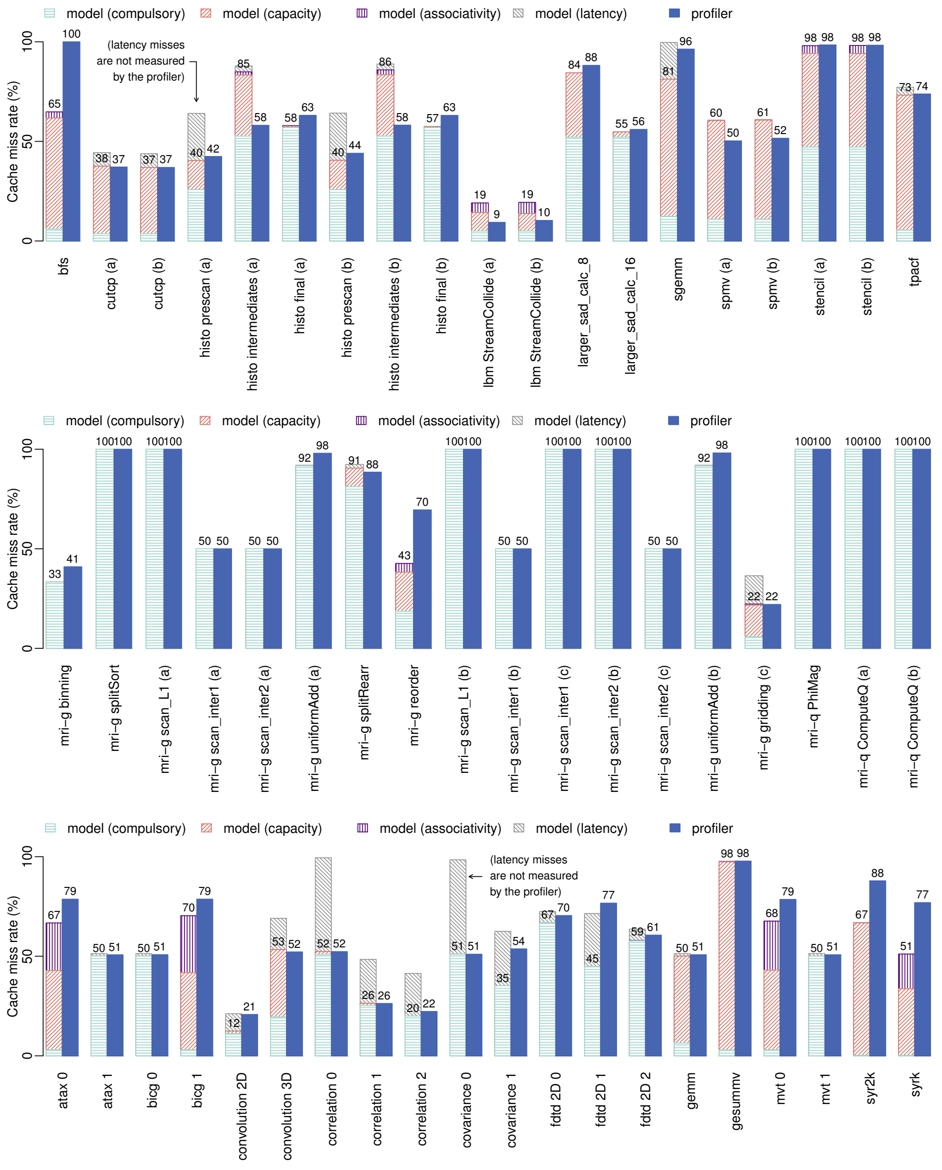


Figure 5.11: Verification results for the Parboil (top and middle) and PolyBench/GPU (bottom) benchmark suites, showing the modelled (left) and measured (right) 16KB L1 cache miss rates for individual kernels. Matching values represent a high accuracy, significantly distinct values represent a low accuracy. In this comparison, we exclude the latency misses (in grey) in the cache miss rate number: the profiler does not include these types of misses as they don't cause additional memory requests.

- We observe that the modelled compulsory misses are lower or equal to the measured misses for all kernels. This is important because the amount of compulsory misses is cache parameter independent. Furthermore, we note that this results in a perfect model for cases where the only type of misses are compulsory misses, e.g. in many of the `mri-g` and `mri-q` kernels.
- Overall, most kernels show almost no associativity misses. However, there are still cases where associativity misses account for a significant fraction of the total amount of misses, in particular for PolyBench/GPU.
- These benchmarks show no additional misses caused by the limited number of MSHRs. In contrary, limiting the size of the MSHR table reduces the cache miss rate in many cases, as will also be shown in section 5.1.11.
- The kernels that show the largest difference between measured and modelled misses (e.g. `bfs`, `atax 0`, `histo intermediates`) are very sensitive to the memory latency parameter. To improve the accuracy for these benchmarks, the model needs to be extended beyond caches only, allowing more realistic latency values to be used.

The results of figure 5.11 are summarised in the top half of figure 5.12, augmented with the results for the 48KB configuration (not shown in detail). The arithmetic mean in absolute error[11] for our model is 6.4% for the 16KB configuration and 8.3% for the 48KB configuration. Finally, three scenarios are tested where a single component of the model is disabled each time, showing how much the introduced extensions to the reuse distance theory contribute to the precision of the model. The 6.4% arithmetic mean in absolute error changes as follows for the 16KB cache configuration: 1) a 9.6% error when associativity is not modelled, 2) a 12.1% error when latencies are not modelled, and 3) a 7.1% error when the number of MSHRs is unlimited.

Comparison against simulation

As a secondary verification, the cache model is compared against the GPGPU-Sim simulator (version 3.2.0) [27]. The simulator is configured with the specifications of the GTX470 GPU (both 16KB and 48KB caches) and runs the two benchmark suites. The results are reported in the bottom half of figure 5.12, in which we show the absolute difference in cache miss rate compared to the results of the profiler. The simulator shows on average a larger error compared to our model: it produces a mean absolute error of 18.1% for the 16KB configuration and 21.4% for the 48KB configuration. Additionally, the run-time of the simulator is on average a factor 268x higher than our model. For example, GPGPU-Sim completes `cutcp` in 10 hours, whereas our model completes the same benchmark in 10 seconds (excluding a one-time 4 minutes emulation in Ocelot).

[11]Note: the absolute error of a metric measured in percentages (miss rate) is also given in percentages.

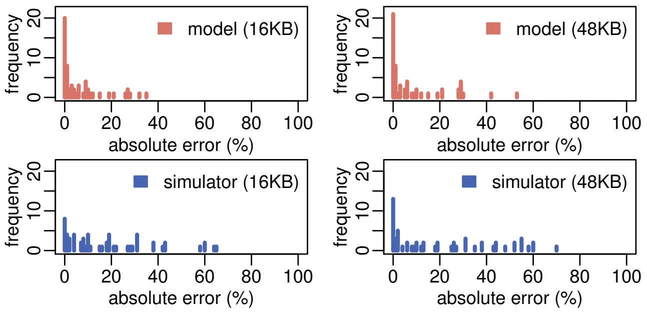


Figure 5.12: Histograms of the absolute error in cache miss rate for the model (top) and for GPGPU-Sim (bottom).

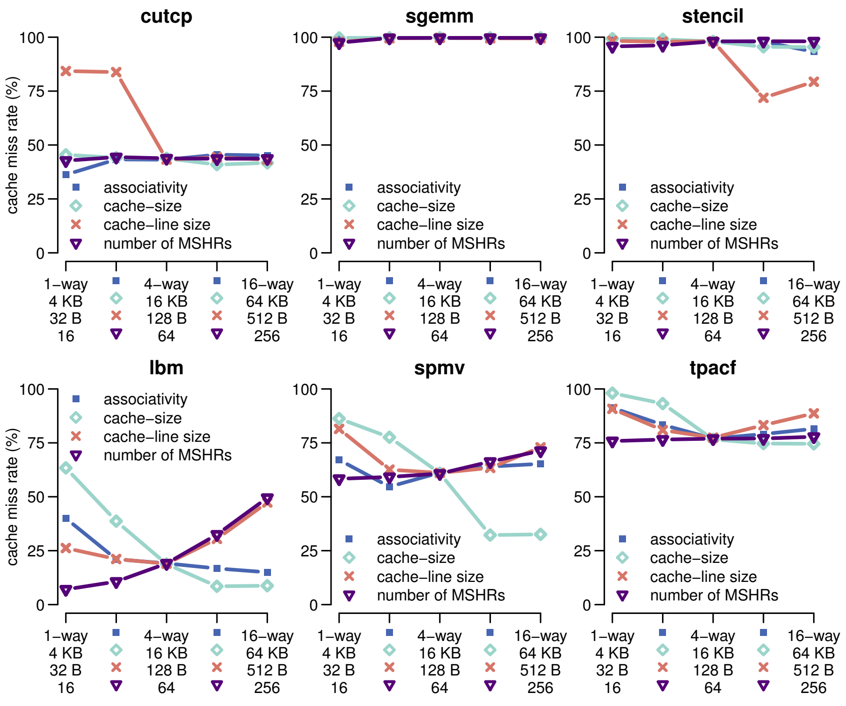


Figure 5.13: Evaluation of different values for four parameters. Different series within a graph represent different parameters and use their own x-axis. All other parameters are set to the defaults, as shown in the middle of each graph.

5.1.11 Example use: evaluating cache parameters

To illustrate one of the uses of our cache model, a sweep over cache parameters is performed. Evaluating all design points or finding optimal design points is beyond the scope of this work: we merely illustrate the possibilities of the cache model.

Four different values are chosen to evaluate four of the main cache parameters: 1) the associativity, 2) the cache-size, 3) the cache-line size, and 4) the number of MSHRs. The values evaluated are 0.25x, 0.5x, 2x, and 4x the GPU's original value for the 16KB configuration. The first (or only) kernel of six of Parboil's benchmarks are included in the results of figure 5.13: `cutcp`, `lbm`, `sgemm`, `spmv`, `stencil` and `tpacf`. These benchmarks are chosen because of their mix of different types of misses. We make the following observations:

- Associativity is a parameter of little importance for the evaluated benchmarks. Small benefits of a high associativity are only visible for `stencil` and `lbm`, benchmarks originally showing 2–3% associativity misses. Because hits and misses influence the thread order, a lower associativity can sometimes give a lower miss rate, as is the case for `spmv` and `cutcp`.

- The cache-size is the most important parameter for `lbm` and `spmv`, showing significant miss rate reductions.

- Cache-line size can have both a positive and a negative influence on the miss rate. For our benchmarks, a cache-line size of either 128 or 256 bytes gives the best results.

- Using only 16 or 32 MSHRs yields better cache behaviour for `lbm` and `spmv`: a low number of MSHRs allows inter-thread locality to be better exploited (see section 5.1.6). The other benchmarks are not significantly influenced by the MSHR parameter.

5.1.12 Summary and future work

This work has shown that reuse distance theory can be used to model GPU caches in detail by extending it with: 1) the mapping and scheduling of the GPU's threads, warps, threadblocks, cores and sets of active threads, 2) the notion of in-flight memory requests and conditional and non-uniform latencies, 3) cache associativity, 4) the effects of miss-status holding-registers (MSHRs), 5) and warp scheduling and divergence. Additionally, we showed through micro-benchmarks how a Fermi GPU maps addresses to sets (for its hash-associative caches) and how many MSHRs are available per core.

The new cache model has been evaluated against two benchmark suites, comparing modelled miss rates for the GPU's L1 data caches against measured miss rates using hardware counters. The results distinguish different types of cache misses. An example are latency misses, a type not even measured by hardware counters. On average, our model predicts cache miss rates with an absolute error of 6.4% (16KB 4-way) and 8.3% (48KB 6-way). From the 57 tested kernel invocations, 47 lie within a 10% absolute error margin. Compared to the GPU simulator GPGPU-Sim, our cache model shows a better accuracy (6–8% versus 18–21%) and a lower run-time (267x on average). The importance of the discussed extensions becomes clear when evaluating them separately, showing a reduction in average absolute error when modelling: cache associativity (9.6% → 6.4%), latencies (12.1% → 6.4%), and a limited amount of MSHRs (7.1% → 6.4%).

A more accurate memory latency and warp divergence model can help improve the cache model further, but would require integration with a full GPU execution model. Additionally, the model can be extended to include other GPU caches, such as the L2, the texture caches, or Kepler's new read-only L1 cache. Future work includes the verification of the model on AMD and ARM GPUs.

The cache model by itself does not improve the programmability of GPUs. Still, it can be a valuable tool for programmers. When optimising their code, they can benefit from insight in the behaviour of their code given by the cache model's breakdown of types of misses. The cache model can also be used alongside compilers (e.g. the BONES compiler from section 4) by guiding optimisation decisions. For example, a cache model can be used to decide whether or not to perform thread coarsening. Nevertheless, the cache model's main use in this thesis is to investigate how the GPU's thread scheduler can be modified to improve cache behaviour.

5.2 A case for locality-aware thread scheduling

Programming models such as CUDA and OpenCL allow the programmer to specify the independence of threads, removing ordering constraints. Still, parallel architectures such as the GPU do not exploit the potential of data-locality enabled by this independence. Therefore, programmers are currently still required to manually perform data-locality optimisations such as memory coalescing or loop tiling.

This section makes a case for *locality-aware thread scheduling*: re-ordering threads to increase data-locality. We demonstrate that locality-aware thread scheduling can considerably benefit a multi-threaded architecture (such as the GPU) in terms of performance. In particular, we perform an in-depth analysis of the potential of multi-level (threads, warps, blocks) locality-aware thread scheduling for GPUs, considering among others cache performance, memory coalescing and bank locality. This section does not aim to improve performance for already optimised code, but is instead motivated by non-optimised program code and the performance potential of locality-aware thread scheduling.

This section discusses: 1) the potential of locality-aware thread scheduling is identified and quantified (section 5.2.3), and 2) an in-depth evaluation of two example kernels (section 5.2.4).

5.2.1 Related work

Locality-aware thread scheduling has been investigated for non-GPU microprocessors in earlier work. For instance, Philbin et al. [113] formalise the problem of locality-aware thread scheduling for a single-core processor. In other work [131], threads are grouped based on data-locality for multi-threaded multi-core processors, introducing a metric of *thread similarity*. Corvino et al. [45] express similarity between threads using a tiled representation of data accesses and find a thread schedule by solving the travelling salesman problem. OpenMP 4.0 allows locality information annotations to be added by programmers [104]. Furthermore, Ding and Zhong [51] propose a model to estimate locality based on reuse distances. These approaches cannot be applied directly to GPUs, as they do not take into account aspects such as: scalability to many threads, cache sizes, the thread-warp-block hierarchy, nor the active thread count.

Recent work on GPUs has investigated the potential of scheduling less active threads to improve cache behaviour. Kayiran et al. [84] propose a compute and memory-intensity heuristic to select the active thread count. Furthermore, Rogers et al. [119] propose a hardware approach: the number of active threads is adapted at run-time based on *lost locality* counters. However, these works only consider active thread count reduction: they do not investigate thread scheduling.

Warp scheduling for GPUs is another active research topic. However, this is mainly in the context of divergent control flow rather than data-locality. By dynamically regrouping threads into warps, those following the same execution path can be scheduled together. Dynamic warp formation in the context of memory access coalescing is discussed in several works [90, 102]. Recent work has focussed on two-level warp scheduling, a technique to reduce the impact of long latency memory operations [67, 82, 105]. This is orthogonal to our work and can be applied on top of locality-aware thread scheduling.

5.2.2 Experimental setup

The experiments are performed on the simulator GPGPU-Sim (version 3.2.1) [27] using a GeForce GTX580 configuration (Fermi) with a 16KB L1 cache (128 byte cache-lines) and a 768KB L2 cache. The GTX580 has 16 SIMT cores (or SMs) for a total of 512 '*CUDA cores*'. From the simulation results we report IPC (instructions per cycle counted as the throughput of all CUDA cores and load/store units), cache miss rates, and load balancing amongst off-chip memory banks.

Implementation in GPGPU-Sim

GPGPU-Sim was modified to perform thread scheduling experiments. The scheduling mechanism of a GPU (and of the simulator) is non-trivial, including multiple hierarchies and dynamic aspects (e.g. influenced by memory latencies). Therefore, the scheduling mechanism in GPGPU-Sim was left intact and instead a preprocessing 'mapping' step was applied to the thread and block identifiers. This mapping step takes thread identifiers t_i and block identifiers b_i and calculates new identifiers as $t'_i = f(t_i)$ and $b'_i = g(b_i)$. The functions $f()$ and $g()$ implement alternative thread schedules as will be discussed in section 5.2.3. Because the mapping is applied before the hardware thread scheduling is applied, the effect is equivalent to applying the $f()$ and $g()$ to the software thread and block identifiers.

Benchmark selection

This chapter includes results for 6 non-optimised CUDA benchmarks, i.e. sub-optimal implementations rather than fine-tuned benchmarks (e.g. Parboil or Rodinia). The main reason for this choice is that this work aims to improve the programmability of the GPU rather than the maximum performance. In other words, if performance of these naive non-optimised benchmarks can be improved without having to change the program code, GPU acceleration is made available to a wider audience. Even expert programmers can benefit from increased flexibility and require fewer optimisations to achieve the full potential of the GPU.

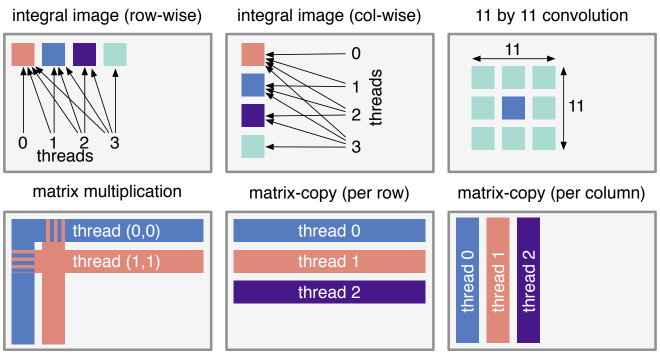


Figure 5.14: Illustrations of the memory access patterns of the 6 benchmarks.

The benchmarks are: the computation of an integral image (both row-major and column-major), a 2D convolution (11 by 11), a 2D matrix copy (each thread copies either a row or a column), and a naive matrix-multiplication. Image and matrix sizes are 512 by 512. Figure 5.14 illustrates their memory access patterns:

1. **Integral image (row-wise):** Every thread at coordinates (x, y) in a 2D image produces a single output pixel at (x, y) by consuming all input pixels (x', y) for which $x' \leq x$. In the example, thread 0 consumes input 0 (red), thread 1 consumes inputs 0 and 1 (red and blue), and so on.

2. **Integral image (column-wise):** Equal to the row-wise version, but each thread instead consumes all input pixels (x, y') for which $y' \leq y$.

3. **11 by 11 convolution:** Each thread produces a pixel in a 2D image by consuming a pixel at the same coordinates and its neighbourhood (green).

4. **Matrix-multiplication:** Each thread with (x, y)-coordinates consumes a row $(*, y)$ of an input matrix and a column $(x, *)$ of another input matrix to produce a single element in an output matrix at (x, y).

5. **Matrix copy (per row):** Each thread consumes a row of an input matrix to produce the corresponding row in an output matrix.

6. **Matrix copy (per column):** As before, but now consuming and producing entire columns instead of rows.

5.2.3 The potential of thread scheduling

Many GPU programs contain a large number of independent threads that can be freely re-ordered. This re-ordering (changing the thread schedule) is motivated by the following data-locality performance optimisations: 1) multiple threads accessing a single cache-line must be grouped in a warp (memory coalescing), 2) threads having strong inter-thread locality must be grouped within a single threadblock (sharing a L1 cache), 3) threadblocks with data-locality must be executed either on a single core in temporal vicinity or simultaneously on different cores (sharing a L2 cache), 4) threads executing simultaneously must minimise pollution of the shared caches, and 5) threads executing simultaneously must spread their accesses as evenly as possible across the memory banks.

Consider an SPMD (single-program multiple-data) kernel with n independent threads $t_1, t_2, ..., t_n$, each referencing a number of data elements. This work assumes that all n threads are independent[12] and can be reordered as $r = n!$ distinct sequences $s_1, s_2, ..., s_r$. The problem of locality-aware thread scheduling is to find a sequence s_i of n threads such that execution time is minimal. On a GPU, thread scheduling influences execution time in terms of efficient use of the caches, memory coalescing, memory bank locality, and careful selection of the number of active threads.

Candidate thread schedules

Various thread schedules are tested in GPGPU-Sim to quantify the potential of locality-aware thread scheduling. Because the number of threads n is typically large (e.g. 2^{18}), it is impractical to test all r orderings. Therefore, only a limited set of schedules is considered: regular and structured schedules, matching the target regular and structured programs. The selected schedules are illustrated in figure 5.15 and discussed below. Note that these schedules represent the mapping step discussed in section 5.2.2 and are still subject to the GPU's multi-level scheduling mechanism. The schedules are:

[12]Dependences in SPMD programs (e.g. synchronisation barriers within threadblocks) can be added as constraints on the thread ordering.

1. **Sequential:** The unmodified original ordering, i.e. $f(x) = x$ and $g(x) = x$. Note that, although it is a sequential ordering from a pre-processing perspective, the actual ordering is still subject to the GPU's thread, warp, and block scheduling policies.
2. **Stride(a, b):** An ordering with a configurable stride (a) and granularity (b) (e.g. warp or threadblock granularity) with respect to the original ordering. Strided schedules have the potential to e.g. ameliorate bad choices of a 2D-coordinate to thread mapping [127].
3. **Zigzag(a, b):** An ordering assuming a 2D grid of threads, reversing the ordering of odd rows. The parameters are the row-length (a) and the granularity (b). Zigzag can exploit 2D locality, but might degrade coalescing for small granularities.
4. **Tile(a, b, c):** 2D tiling in a 2D grid. Tiling takes as parameter the length of a row (a) and the dimensions of the tile (b x c). Tiling has been shown to have potential to exploit locality on GPUs [127].
5. **Hilbert(a):** A space filling fractal for squared grids of size a exploiting 2D locality [89].

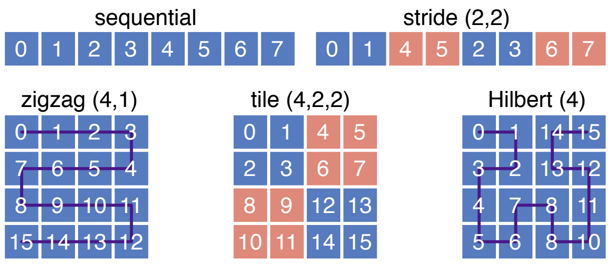


Figure 5.15: Examples with 8 or 16 threads. The numbering shows the new sequence and the layout the original sequence (left-to-right, top-to-bottom).

Two different threadblock-schedulers are implemented on top of the candidate schedules: either schedule threadblocks over cores in a round-robin fashion (left hand side of figure 5.16) or allocate subsequent threadblocks to subsequent cores (right hand side of figure 5.16). In case threadblocks with locality are grouped close to each other, the first threadblock-scheduler can benefit from locality in the L2 cache (in space among different cores), while the second can benefit from locality in the L1 cache (in time among different threadblocks).

A total of 2170 candidate schedules is considered. This includes a sweep over the 5 orderings, several parameter values (e.g. stride-size, granularity), and the two threadblock-schedulers. Furthermore, five different active thread counts (64, 128, 256, 512 and 1024) are included to identify the trade-offs between cache contention and parallelism [84, 119].

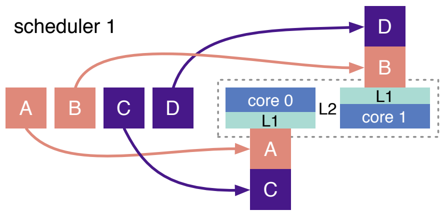


Figure 5.16: Illustrating scheduling techniques for threadblocks *A*–*D*, assuming locality between blocks *A* and *B* (red) and between *C* and *D* (purple). This results either in L2 locality (left) or L1 locality (right).

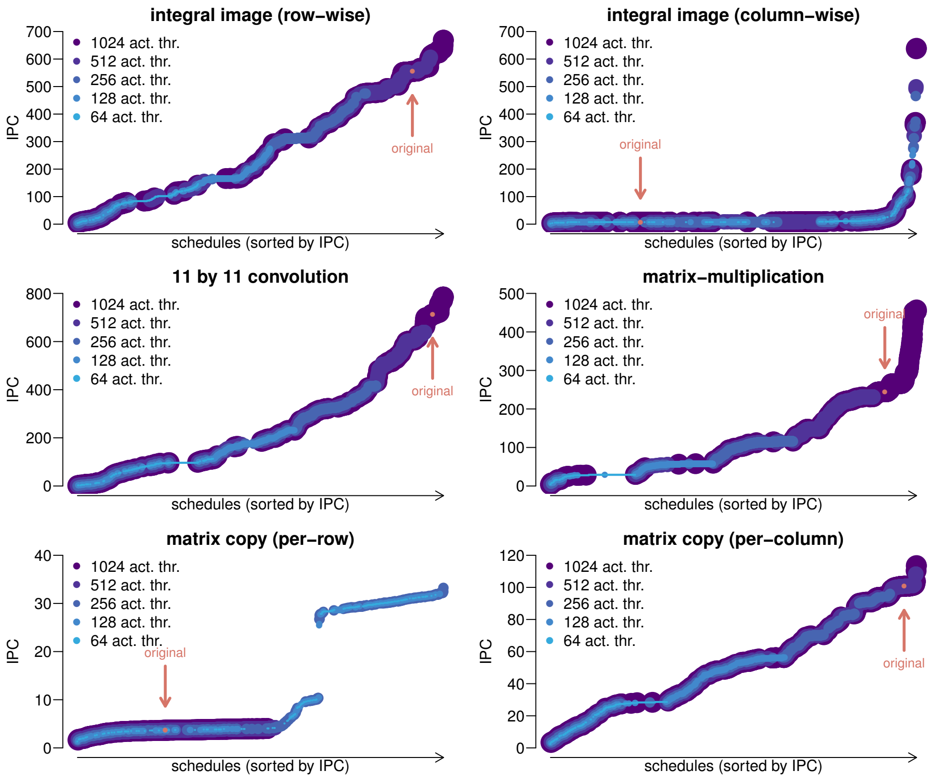


Figure 5.17: Sorted IPC (higher is better) for 2170 schedules per benchmark. The vertical red arrow identifies the original schedule (no changes applied to GPGPU-Sim). Darker and larger glyphs represent more active threads, lighter and smaller glyphs represent fewer active threads.

Experimental results

Figure 5.17 shows the IPC results when simulating all candidate schedules for the benchmarks with GPGPU-Sim. Each set of 2170 results is sorted by their achieved IPC (from low to high). The original (unmodified) schedule is highlighted, its horizontal position indicating the performance potential. Note that these graphs merely identify the '*landscape*', detailed results are presented and discussed in section 5.2.4. We make the following observations with respect to figure 5.17:

1. **Integral image (row-wise):** There is a wide performance variation among the different schedules: IPC ranges from 2 to 700. The default schedule is already performing well: it has coalesced memory accesses and uses the caches efficiently. Still, there is opportunity for a 20% performance improvement, achieved for example by using a 8 by 16 tiled schedule. The active thread count is not strongly correlated to performance. Even so, the best 5% schedules all use 1024 active threads.

2. **Integral image (column-wise):** The default schedule at an IPC of 7 is suffering from uncoalesced memory accesses and bad cache locality for this purposely poorly design kernel. Using a schedule with a stride equal to the width of the image resolves these problems, bringing performance back to the level of the row-wise integral image computation.

3. **11 by 11 convolution:** The overall results look similar to the row-wise integral image case at first glance. However, inspection of the results shows that the best candidates are zigzag as opposed to tiled schedules, achieving up to 10% improvement over the default.

4. **Matrix-multiplication:** The results show that there is up to 87% performance to gain over the default schedule. Section 5.2.4 will analyse this.

5. **Matrix copy (per row):** The active thread count is of significant importance in this naive implementation of matrix copy, although the performance is in general low. Schedules with 512 or 1024 active threads (including the default) yield an IPC of 5 at best, while schedules with 64, 128, or 256 active threads show a potential for an IPC of up to 34. This is the only tested case where more active threads does not yield better performance.

6. **Matrix copy (per column):** Better overall performance compared to per-row copy. Section 5.2.4 analyses the results and the 12% potential.

Note that in contrast to the two integral image cases, it is not possible to achieve equal performance for the two matrix copy cases. The reason for this is the integral image's flexibility: each thread computes a single result. In contrast, the matrix copy processes (in the current implementation) an entire row or column per thread, limiting the scheduling freedom.

Additional benchmarks

The same testing methodology was applied to several other naive benchmarks. An example is the computation of an 8 by 8 discrete cosine transform (DCT) on a 2048 by 2048 input using a nested for-loop in the kernel body with 64 iterations. A sweep through the different thread schedules led to a schedule with a performance of 3.2x the original (an increase from an IPC of 175 to 570). This 'best' schedule applies a stride of 512 at a granularity of 8, moving multiple groups of threads belonging to one 8 by 8 transform (64 threads) together into a single threadblock.

Similarly, the symmetric rank-k kernel (`syrk`) from PolyBench shows a 3 times speed-up. Several other tested benchmarks have not shown significant changes at all. This includes matrix-vector summation (`gesummv`) from the PolyBench benchmark and the breath-first-search (`bfs`) and `srad` kernels from Rodinia. These results were expected, as these benchmarks contain already optimised code.

5.2.4 Detailed case studies

Section 5.2.3 illustrated the performance impacts for a subset of all possible thread schedules for 6 benchmarks. We showed that the performance potential varies from limited (e.g. 10% for the convolution benchmark) to significant (e.g. 87% for matrix-multiplication). We also saw different best performing schedules for different benchmarks and a varying correlation between performance and active thread count. To get additional insight from the results, this section discusses two of the benchmarks in more detail. However, because of the large amount of data (schedules, benchmarks, metrics), only a subset of the data is presented.

Matrix-multiplication

Matrix-multiplication is one of the examples that shows a significant performance potential (up to 87%) from its default IPC of 245. To identify the reason why certain schedules perform better than others, we take a detailed look at the simulation results for the strided schedules. Because the stride ordering has two parameters (`P1` for the stride and `P2` for the granularity), the data can be visualised as a 2D heatmap. Figure 5.18 shows the heatmaps for the IPC and the L1 and L2 cache miss rates[13], as well as the correlation between the them.

Figure 5.18 shows a high inverse correlation (-0.8) between the IPC and the L1 miss rate: the 4 best candidates (with IPC $>$ 300) all have the lowest L1 miss rate (16%). Although a low L2 miss rate also contributes to a high IPC, figure 5.18 (bottom right) shows a lower correlation. The results of figure 5.18 can be explained after detailed investigation. First of all, schedules with a small granularity (`P2` $<$ 32) can reduce the amount of coalescing significantly, leading to a low IPC and high cache miss rates. Second, schedules with a large stride

[13]The L2 miss rates are w.r.t the requests made to the L2 (the local miss-rate). Multiplying the L1 and L2 miss rates yields the miss rate w.r.t the total number of requests.

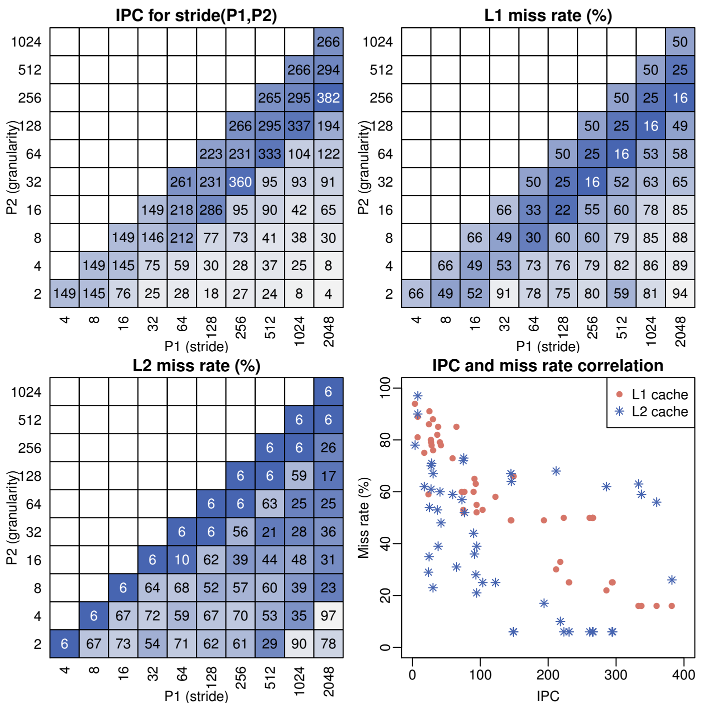


Figure 5.18: Simulation results for the matrix-multiplication example for strided schedules. Shown are IPC (higher is better), L1 and L2 miss rates (lower is better) and their correlation.

and a large granularity form small 'tiles' in the 2D space of the matrix, improving locality. Finding the best tile dimensions is non-trivial and dependent on among others matrix dimensions and cache configuration. In this case, a ratio of 8:1 for `P1` and `P2` yields the best results for L1 and 2:1 for the L2 cache.

The left hand side of figure 5.19 shows the correlation plots of all the 2170 schedules for the matrix-multiplication example for 3 metrics: the top graph shows the correlation between IPC (y-axis) and L1 miss rate (x-axis), the bottom between IPC and L2 miss rate. From these results, we observe that the strided and tiled schedules have similar behaviour: they both cover the entire IPC and miss rate spectrum and show a high correlation between the IPC and L1 miss rate. We also observe a large amount of schedules with a L1 cache miss rate of around 50%, including the default and zigzag schedules.

Per-column matrix copy

The correlation plots for the per-column matrix copy are shown in the right half of figure 5.19. From the correlation plots, we immediately observe that the IPC and cache miss rates are not as correlated as for the matrix-multiplication. In

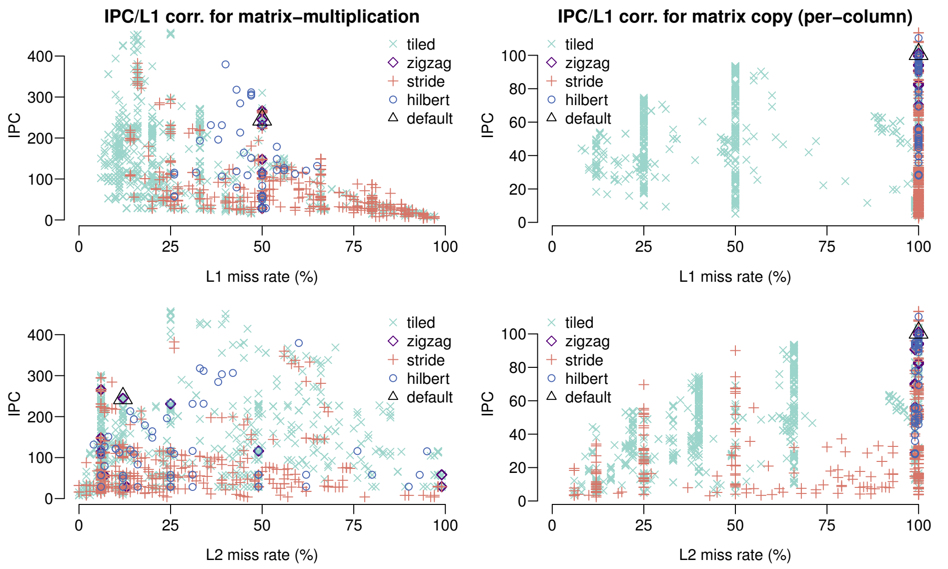


Figure 5.19: Correlation plots for IPC (higher is better) and cache miss rates (lower is better) for the matrix-multiplication example (left) and the per-column matrix copy (right). Different colours/shapes represent different schedule types.

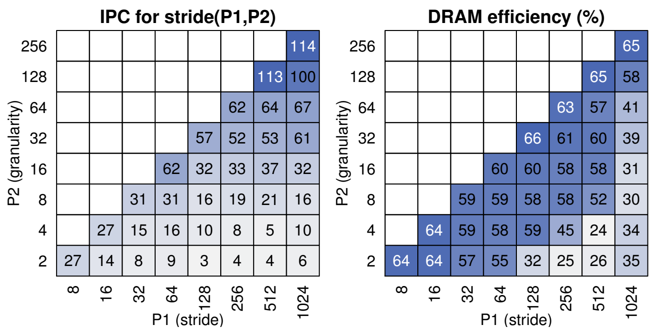


Figure 5.20: Simulation results for the per-column matrix copy using strided schedules. Shown are the IPC and the DRAM bank efficiency (higher is better).

fact, the best performing schedules have L1 and L2 cache miss rates of 100%. We furthermore observe that L1 cache miss rates only vary for tiled schedules and that most of them are distributed in a log_2 fashion: they have values of 100%, 50%, 25% and 12.5%. These 'improved' miss rates are cases where a lowered ($\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$) memory coalescing rate[14] results in additional cache hits.

Unlike the matrix-multiplication example, cache miss rates are not correlated with the IPC. Therefore, figure 5.20 focuses on other aspects: it shows the IPC and

[14]Coalescing is not visualised in this work because GPGPU-Sim lacks the corresponding counters.

DRAM bank efficiency (still for strided schedules). A low DRAM bank efficiency is the cause of an uneven distribution of accesses over the banks (6 for the simulated GPU): certain phases of the benchmark access only a subset of banks, limiting throughput. Although DRAM efficiency is correlated to the IPC, figure 5.20 also shows that it is not the only contributing effect. As with matrix-multiplication, coalescing also plays a role, explaining the low IPC for `P2` < 32.

5.2.5 Summary and future work

This section identified the potential for locality-aware thread scheduling on GPUs: re-ordering threads to increase data-locality and subsequently performance and energy efficiency. A set of 2170 candidate thread schedules were simulated for 6 non-optimised CUDA benchmarks, showing a performance potential varying from 10% to multiple orders of magnitude. The benchmarks were explicitly chosen to be non-optimised: enabling competitive performance for such benchmarks will greatly improve the programmability of the GPU. A detailed study of two of these benchmarks has identified various aspects to consider when performing locality-aware thread scheduling, including cache miss rates, coalescing, bank locality, and the number of active threads. An example benchmark is a straightforward implementation of matrix multiplication, which achieved a 87% performance increase by modifying the thread schedule.

Although we have shown that locality-aware thread scheduling has potential to improve the programmability and can considerably improve performance, we have also shown that it is non-trivial to find the best thread schedule. Future work will need to investigate how to find such a good (or the best) thread schedule among all possibilities. A solution could be found by evaluating schedules using a complete or partial performance model. This is motived by the detailed studies of section 5.2.4, which have shown that performance is correlated to one or more metrics such as memory access coalescing or cache miss rate. An example of this is the use of the L1 cache model presented in section 5.1, which will be able to find the best schedules in the case of the matrix-multiplication example, but will fail to do so in the case of the per-column matrix copy. Investigating such an approach in more detail is left for future work.

"Measuring programming progress by lines of code is like measuring aircraft building progress by weight."

- Unknown source

Chapter 6

Conclusions and future work

This section summarises the conclusions of the individual chapters and concludes the thesis. Furthermore, it suggests directions for future work.

6.1 Conclusions

GPUs have become increasingly popular as accelerators and are here to stay: with energy efficiency as a main performance metric and dark silicon on the horizon, computer architecture will need to rely on specialised microprocessors in the future even more than it already does now. Unfortunately, this decreases programmability: to exploit the GPU's full potential, programmers are required to understand the working of the architecture, have to manage parallelism and synchronisation, deal with complex memory hierarchies and so on. For this reason, industry and academia have started investigating into different techniques to improve programmability and the related metrics of performance, portability and productivity. This has led amongst others to work on high-level programming languages, intermediate representations, architectural support for programmability, iterative compilation and source-to-source compilers.

This work addressed the programmability issues of GPUs with a source-to-source compiler based on a structured program code classification. Such a program code classification can be beneficial for both programmers (manual approach) and compilers (automatic approach). Because none of the explored existing classifications provide a good fit for this goal, new memory access pattern-based classifications were introduced. The first is 'algorithmic species', a formalised algorithm classification based on the polyhedral model. However, two main drawbacks of this theory were observed: 1) it is limited to static affine loop nests, and 2) the classes fail to capture some important aspects for performance and energy efficiency (e.g. tiled loops). The first drawback is resolved by the introduction of a new formal theory for the original algorithmic species, and the second by increasing the amount of detail in classes, yielding the finer-grained SPECIES+ classification. This creates a trade-off, in which algorithmic species can provide a readable and intuitive classification (suitable for automatic and manual uses), while SPECIES+ is less readable but captures additional details.

BONES is a new source-to-source compiler based upon the algorithmic species classification. The compiler transforms sequential C code into CUDA, OpenCL or OpenMP by providing a pre-optimised template implementation (an 'algorithmic skeleton') for each algorithmic species. Previous to this work, compilers based on algorithmic skeletons used 'classifications' without formal definitions which were meant to be extended as new classes were encountered. This is no longer needed as a result of using the algorithmic species classification within BONES. Furthermore, by using tools to automatically extract species from source code, BONES is the first skeleton-based compiler integrated in a fully automatic flow. Although most of the code generation is performed based on skeletons within BONES, the compiler also includes traditional optimisation passes such as register caching, thread coarsening, zero-copy, host-accelerator transfer optimisations, and kernel fusion. Chapter 4.5 showed the importance of these optimisations: a compiler relying on skeletons alone will not be able to deliver competitive performance. Because BONES is based on the more intuitive algorithmic species and not on the more detailed SPECIES+ classification, there will still be room for optimisations after compilation. Therefore, BONES generates editable and readable code, allowing further fine-tuning by expert programmers or auto-tuners.

Algorithmic species and BONES improve the programmability of GPUs from a programming language and compiler perspective, however, it can also be improved from an architectural point of view. In particular, this thesis identified the potential of re-ordering the GPU's threads in a locality-aware manner: scheduling work such that threads accessing the same data execute close to each other in time or space. Although a locality-aware thread scheduling technique has not yet been found, the impact on performance and programmability is shown to be significant. The first steps to find a solution have been discussed in the form of a detailed cache model. This GPU cache model, based on reuse distance theory, can be a tool for programmers and processor architects to obtain insight into cache behaviour and could potentially be used to find a locality-aware thread schedule.

This thesis has shown that the programmability of GPUs can be improved from both a compiler and an architectural perspective. In particular, the compiler approach has improved portability and productivity (two of the three metrics illustrated in figure 1.2) by abstracting away from target specific languages such as OpenCL and CUDA. Performance (the third metric) is achieved through the compiler's skeletons and optimisations. This work also identified that programmability can be improved from an architectural perspective, in particular by intelligent thread scheduling. This can reduce the programmer's effort to fine-tune code, improving on portability and productivity while maintaining performance. Some of the techniques discussed to address the programmability issues can be addressed from either of the two perspectives, but others are more naturally addressed by either a compiler (e.g. kernel fusion) or by modifying the architecture (e.g. zig-zag thread scheduling).

Although this thesis has discussed some techniques specific for GPUs, it has not lost its generality. The program code classification, the source-to-source compiler, and even the proposed locality-aware thread scheduling are not limited to GPUs only: the techniques can be applied to other (parallel) microprocessors such as multi-core CPUs. Moreover, the algorithmic species classification can be used outside the scope of 'improving programmability', for example for performance prediction.

6.2 Future work

This thesis has made a case for locality-aware thread scheduling, identifying the potential to improve programmability. Future work must investigate how to find such a locality-aware schedule in order to create the proposed architecture in which the scheduler optimises for thread locality.

This work also briefly mentioned the 'boat hull model' [11], a species-specific roofline model. This performance prediction technique was based on an earlier pre-species algorithm classification. Updating the boat hull model with support for algorithmic species or even SPECIES+ can improve the model in various ways, including automation and a more detailed prediction. Integrating such a model into the BONES source-to-source compiler can help to automate the selection of a target (e.g. CPU versus GPU) or can be used to estimate how far performance is from the theoretical achievable maximum.

The compiler BONES can furthermore be extended to give feedback to the user, possibly in combination with a performance model. For example, a user can be notified when a particular GPU kernel has little parallelism, or the user can receive a detailed parallelisation report. This enables a user to learn from the compiler, but it can also result in better code, as the user might implement a different algorithm that suits the target processor better next time.

The first steps of using BONES and algorithmic species for hardware design have already been made in the form of high-level synthesis skeletons. Related to this, species could also be used in the future to design an application specific programmable accelerator. For example, given details about the type of species occurring in a target domain, a hardware designer can decide whether or not to include (and how to dimension) a cache for *neighbourhood* accesses or a read-only memory for *full* input patterns.

The idea of algorithmic species can also be extended into a programming model, creating something along the lines of existing domain specific languages for GPUs (e.g. [100]). Such a species-based programming model can motivate programmers to directly think in terms of memory access patterns, but cannot benefit from the automatic identification of species currently provided by A-DARWIN.

Bibliography

Refereed papers and patent covered in this thesis

[1] C. Nugteren, G.-J. v. d. Braak, and H. Corporaal. Future of GPGPU Micro-Architectural Parameters. In *DATE: Design Automation and Test in Europe*, 2013.

[2] C. Nugteren, G.-J. v. d. Braak, H. Corporaal, and H. Bal. A Detailed GPU Cache Model Based on Reuse Distance Theory. In *HPCA-20: International Symposium on High Performance Computer Architecture*. IEEE, 2014.

[3] C. Nugteren and H. Corporaal. Introducing 'Bones': A Parallelizing Source-to-Source Compiler Based on Algorithmic Skeletons. In *GPGPU-5: Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2012.

[4] C. Nugteren, R. Corvino, and H. Corporaal. Algorithmic Species Revisited: A Program Code Classification Based on Array References. In *MuCoCoS-6: International Workshop on Multi-/Many-core Computing Systems*. IEEE, 2013.

[5] C. Nugteren, P. Custers, and H. Corporaal. Algorithmic Species: An Algorithm Classification of Affine Loop Nests for Parallel Programming. *ACM Transactions on Architecture and Code Optimisations*, 9(4):Article 40, 2013.

[6] C. Nugteren, P. Custers, and H. Corporaal. Automatic Skeleton-Based Compilation through Integration with an Algorithm Classification. In *APPT: International Conference on Advanced Parallel Processing Technology*. Springer, 2013.

[7] UK Patent Application GB1321841.7. Configuring Thread Scheduling on a Multi-Threaded Data Processing Apparatus, 2013.

Other first-author refereed papers

[8] C. Nugteren, G.-J. v. d. Braak, and H. Corporaal. Roofline-aware DVFS for GPUs. In *ADAPT-4: International Workshop on Adaptive Self-Tuning Computing Systems*. ACM, 2014.

[9] C. Nugteren, G.-J. v. d. Braak, H. Corporaal, and B. Mesman. High Performance Predictable Histogramming on GPUs: Exploring and Evaluating Algorithm Trade-offs. In *GPGPU-4: Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2011.

[10] C. Nugteren and H. Corporaal. The Boat Hull Model: Adapting the Roofline Model to Enable Performance Prediction for Parallel Computing. In *PPoPP-17: Symposium on Principles and Practice of Parallel Programming (2-page poster article)*. ACM, 2012.

[11] C. Nugteren and H. Corporaal. The Boat Hull Model: Enabling Performance Prediction for Parallel Computing Prior to Code Development. In *CF-9: International Conference on Computing Frontiers*. ACM, 2012.

[12] C. Nugteren, H. Corporaal, and B. Mesman. Skeleton-based Automatic Parallelization of Image Processing Algorithms for GPUs. In *SAMOS-XI: International Conference on Embedded Computer Systems*. IEEE, 2011.

[13] C. Nugteren, B. Mesman, and H. Corporaal. Analyzing CUDA's Compiler through the Visualization of Decoded GPU Binaries. In *ODES-8: Workshop on Optimizations for DSP and Embedded Systems*, 2010.

Other (co-)authored papers

[14] R. Baghdadi, A. Cohen, S. Guelton, S. Verdoolaege, J. Inoue, T. Grosser, G. Kouveli, A. Kravets, A. Lokhmotov, C. Nugteren, F. Waters, and A. F. Donaldson. PENCIL: Towards a Platform-Neutral Compute Intermediate Language for DSLs. In *WOLFHPC-2: International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, 2012.

[15] G.-J. v. d. Braak, C. Nugteren, B. Mesman, and H. Corporaal. Fast Hough Transform on GPUs: Exploration of Algorithm Trade-Offs. In *ACIVS: Advanced Concepts for Intelligent Vision Systems*. Springer, 2011.

[16] G.-J. v. d. Braak, C. Nugteren, B. Mesman, and H. Corporaal. GPU-Vote: A Framework for Accelerating Voting Algorithms on GPU. In *Euro-Par: European Conference on Parallel and Distributed Computing*, 2012.

[17] C. Nugteren and H. Corporaal. A Modular and Parameterisable Classification of Algorithms. Technical Report ESR-2011-02, Eindhoven University of Technology, 2011.

Main bibliography

[18] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. FastFlow: High-Level and Efficient Streaming on Multi-Core. *Programming Multi-core and Many-core Computing Systems*, 13, Jan. 2013.

[19] R. Allen and K. Kennedy. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, Oct. 1987.

[20] G. Almási, C. Caşcaval, and D. Padua. Calculating Stack Distances Efficiently. In *MSP: Workshop on Memory System Performance*. ACM, 2002.

[21] A. v. Amesfoort, A. L. Varbanescu, H. Sips, and R. v. Nieuwpoort. Evaluating Multi-Core Platforms for HPC Data-Intensive Kernels. In *CF-6: International Conference on Computing Frontiers*. ACM, 2009.

[22] M. Amini, B. Creusillet, S. Even, R. Keryell, O. Goubier, S. Guelton, J. Mcmahon, F.-X. Pasquier, G. Péan, and P. Villalon. Par4All: From Convex Array Regions to Heterogeneous Computing. In *IMPACT-2 : International Workshop on Polyhedral Compilation Techniques*, 2012.

[23] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A View of the Parallel Computing Landscape. *Communications of the ACM*, 52:56–67, October 2009.

[24] D. Bacon, S. Graham, and O. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26, 1994.

[25] S. Baghdadi, A. Größlinger, and A. Cohen. Putting Automatic Polyhedral Compilation for GPGPU to Work. In *CPC-15: Workshop on Compilers for Parallel Computers*, 2010.

[26] S. Baghsorkhi, M. Delahaye, S. Patel, W. Gropp, and W.-M. Hwu. An Adaptive Performance Modeling Tool for GPU Architectures. In *PPoPP-15: Symposium on Principles and Practice of Parallel Programming*. ACM, 2010.

[27] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing CUDA Workloads using a Detailed GPU Simulator. In *ISPASS: International Symposium on Performance Analysis of Systems and Software*. IEEE, 2009.

[28] A. Balevic and B. Kienhuis. KPN2GPU: An Approach for Discovery and Exploitation of Fine-Grain Data Parallelism in Process Networks. *ACM SIGARCH Computer Architure News*, 39(4):66–71, 2011.

[29] M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA Code Generation for Affine Programs. In *CC-19: International Conference on Compiler Construction*. Springer, 2010.

[30] C. Bertolli, A. Betts, G. Mudalige, M. Giles, and P. Kelly. Design and Performance of the OP2 Library for Unstructured Mesh Applications. In *CGWS-1: Workshop on Grids, Clouds and P2P Computing*, 2011.

[31] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson. GPUVerify: A Verifier for GPU Kernels. In *OOPSLA: International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, 2012.

[32] K. Beyls and E. D'Hollander. Reuse Distance as a Metric for Cache Behavior. In *IASTED: Conference on Parallel and Distributed Computing and Systems*, 2001.

[33] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. Loh, D. McCauley, P. Morrow, D. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. Shen, and C. Webb. Die Stacking (3D) Microarchitecture. In *MICRO-39: International Symposium on Microarchitecture*. IEEE, 2006.

[34] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *PLDI-29: Conference on Programming Language Design and Implementation*. ACM, 2008.

[35] S. Borkar. Thousand Core Chips: A Technology Perspective. In *DAC-44: Design Automation Conference*. ACM, 2007.

[36] P. Boulet. Array-OL Revisited, Multidimensional Intensive Signal Processing Specification. Technical Report RR-6113, INRIA, 2007.

[37] G.-J. v. d. Braak and H. Corporaal. GPU-CC: A Reconfigurable GPU Architecture with Communicating Cores. In *M-SCOPES-16: International Workshop on Software and Compilers for Embedded Systems*. ACM, 2013.

[38] M. Brehob. *On the Mathematics of Caching*. PhD thesis, Michigan State University, 2003.

[39] W. Caarls. *Automated Design of Application-Specific Smart Camera Architectures*. PhD thesis, Delft University of Technology, 2008.

[40] W. Caarls, P. Jonker, and H. Corporaal. Algorithmic Skeletons for Stream Programming in Embedded Heterogeneous Parallel Image Processing Applications. In *IPDPS-20: International Parallel and Distributed Processing Symposium*. IEEE, 2006.

[41] D. Campbell. Towards the Classification of Algorithmic Skeletons. Technical Report YCS 276, University of York, 1996.

[42] L. Carrington, M. Tikir, C. Olschanowsky, M. Laurenzano, J. Peraza, A. Snavely, and S. Poole. An Idiom-finding Tool for Increasing Productivity of Accelerators. In *ICS-25: International Conference on Supercomputing.* ACM, 2011.

[43] X. Chen and T. Aamodt. A First-order Fine-grained Multithreaded Throughput Model. In *HPCA-15: International Symposium on High Performance Computer Architecture.* IEEE, 2009.

[44] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, 1991.

[45] R. Corvino, S. Mancini, and R. Guizzetti. Automatic Generation of a Parallel Tile Processing Unit for Algorithms with Non-Affine Array References. In *IFMT-1: International Forum on Next-Generation Multicore/Manycore Technologies.* ACM, 2008.

[46] B. Creusillet and F. Irigoin. Exact versus Approximate Array Region Analyses. In *LCPC-10: International Workshop on Languages and Compilers for Parallel Computing.* Springer, 1997.

[47] A. Darte. On the Complexity of Loop Fusion. *Parallel Computing*, 26(9):1175 – 1193, 2000.

[48] U. Dastgeer, L. Li, and C. Kessler. Adaptive Implementation Selection in the SkePU Skeleton Programming Library. In *APPT: International Conference on Advanced Parallel Processing Technology.* Springer, 2013.

[49] R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. LeBlanc. Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[50] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A Dynamic Optimization Framework for Bulk-Synchronous Applications in Heterogeneous Systems. In *PACT-19: International Conference on Parallel Architectures and Compilation Techniques.* ACM, 2010.

[51] C. Ding and Y. Zhong. Predicting Whole-Program Locality through Reuse Distance Analysis. In *PLDI-24: Conference on Programming Language Design and Implementation.* ACM, 2003.

[52] R. Dolbeau, S. Bihan, and F. Bodin. HMPP: A Hybrid Multi-core Parallel Programming Environment. In *GPGPU-1: Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[53] C. Dubach, T. Jones, E. Bonilla, G. Fursin, and M. O'Boyle. Portable Compiler Optimisation across Embedded Programs and Microarchitectures using Machine Learning. In *MICRO-42: International Symposium on Microarchitecture*. IEEE, 2009.

[54] A. Duran, E. Ayguade, R. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: A Proposal For Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters*, 21(02), 2011.

[55] J. Enmyren and C. Kessler. SkePU: A Multi-backend Skeleton Programming Library for Multi-GPU Systems. In *HLPP-4: International Workshop on High-level Parallel Programming and Applications*. ACM, 2010.

[56] S. Ernsting and H. Kuchen. Data Parallel Skeletons for GPU Clusters and Multi-GPU Systems. *Advances in Parallel Computing*, 22:509–518, 2011.

[57] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *ISCA-38: International Symposium on Computer Architecture*. ACM, 2011.

[58] J. Fang, A. Varbanescu, and H. Sips. A Comprehensive Performance Comparison of CUDA and OpenCL. In *ICPP-42: International Conference on Parallel Processing*. IEEE, 2011.

[59] P. Feautrier. Dataflow Analysis of Array and Scalar References. *Springer International Journal of Parallel Programming*, 20:23–53, 1991.

[60] W.-C. Feng and K. Cameron. The Green500 List: Encouraging Sustainable Supercomputing. *IEEE Computer*, 40(12):50–55, 2007.

[61] S. Fuller and L. Millett. Computing Performance: Game Over or Next Level? *IEEE Computer*, 44, 2011.

[62] W. Fung and T. Aamodt. Thread Block Compaction for Efficient SIMT Control Flow. In *HPCA-17: International Symposium on High Performance Computer Architecture*. IEEE, 2011.

[63] W. Fung, I. Sham, G. Yuan, and T. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *MICRO-40: International Symposium on Microarchitecture*. IEEE, 2007.

[64] G. Fursin, Y. Kashnikov, A. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, F. Bodin, P. Barnard, E. Ashton, E. Bonilla, J. Thomson, C. Williams, and M. O'Boyle. Milepost GCC: Machine Learning Enabled Self-tuning Compiler. *Springer International Journal of Parallel Programming*, 39(3):296–327, 2011.

[65] M. Garland and D. Kirk. Understanding Throughput-Oriented Architectures. *Communications of the ACM*, 53:58–66, November 2010.

[66] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. Parallel Computing Experiences with CUDA. *IEEE Micro*, 28(4):13–27, 2008.

[67] M. Gebhart, D. Johnson, D. Tarjan, S. Keckler, W. Dally, E. Lindholm, and K. Skadron. A Hierarchical Thread Scheduler and Register File for Energy-Efficient Throughput Processors. *ACM Transactions on Computer Systems*, 30:8:1–8:38, 2012.

[68] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor. The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future. *IEEE Micro*, 31(2):86–95, 2011.

[69] V. Govindaraju, C.-H. Ho, and K. Sankaralingam. Dynamically Specialized Datapaths for Energy Efficient Computing. In *HPCA-17: International Symposium on High Performance Computer Architecture*. IEEE, 2011.

[70] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-tuning a High-Level Language Targeted to GPU Codes. In *INPAR: Workshop on Innovative Parallel Computing*, 2012.

[71] D. Grewe and A. Lokhmotov. Automatically Generating and Tuning GPU Code for Sparse Matrix-Vector Multiplication from a High-Level Representation. In *GPGPU-4: Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2011.

[72] S. Guelton, M. Amini, and B. Creusillet. Beyond Do Loops: Data Transfer Generation with Convex Array Regions. In *LCPC-25: International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2012.

[73] T. Han and T. Abdelrahman. hiCUDA: High-Level GPGPU Programming. *IEEE Transactions on Parallel and Distributed Systems*, 22:78 –90, Jan 2011.

[74] Havli. GPU Hardware Museum. On-line: http://www.hw-museum.cz/.

[75] M. Hill and A. Smith. Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers*, 38, 1989.

[76] S. Hong and H. Kim. An Integrated GPU Power and Performance Model. In *ISCA-37: International Symposium on Computer Architecture*. ACM, 2010.

[77] L. Howes, A. Lokhmotov, A. Donaldson, and P. Kelly. Deriving Efficient Data Movement from Decoupled Access/Execute Specifications. In *HiPEAC-4: International Conference on High Performance Embedded Architectures and Compilers*. Springer, 2009.

[78] Intel. ARK Processor Database. On-line: http://ark.intel.com/.

[79] Intel. *Writing Optimal OpenCL Code with Intel OpenCL SDK*, 2013.

[80] E. Ipek, M. Kirman, N. Kirman, and J. Martinez. Core fusion: Accommodating software diversity in chip multiprocessors. In *ISCA-34: International Symposium on Computer Architecture*. ACM, 2007.

[81] T. Jablin, J. Jablin, P. Prabhu, F. Liu, and D. August. Dynamically Managed Data for CPU-GPU Architectures. In *CGO: International Symposium on Code Generation and Optimization*. ACM, 2012.

[82] A. Jog, O. Kayiran, N. Nachiappan, A. Mishra, M. Kandemir, O. Mutlu, R. Iyer, and C. Das. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. In *ASPLOS-18: International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2013.

[83] M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam. A Hyperplane Based Approach for Optimizing Spatial Locality in Loop Nests. In *ICS-12: International Conference on Supercomputing*. ACM, 1998.

[84] O. Kayiran, A. Jog, M. Kandemir, and C. Das. Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs. In *PACT-22: International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2013.

[85] S. Keckler, W. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, 31:7–17, 2011.

[86] K. Kennedy and K. McKinley. Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution. In *LCPC-7: International Workshop on Languages and Compilers for Parallel Computing*. Springer, 1994.

[87] K. Keutzer and T. Mattson. A Design Pattern Language for Engineering (Parallel) Software. In *Intel Technology Journal*, 2010.

[88] X. Kong, D. Klappholz, and K. Psarris. The I Test: An Improved Dependence Test for Automatic Parallelization and Vectorization. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):342–349, 1991.

[89] I. N. Kontaxakis. *Contribution to Image Segmentation and Integral Image Coding*. PhD thesis, University of Athens, 2008.

[90] A. Lashgar, A. Baniasadi, and A. Khonsari. Dynamic Warp Resizing: Analysis and Benefits in High-Performance SIMT. In *ICCD-30: International Conference on Computer Design*. IEEE, 2012.

[91] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *SC: International Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2010.

[92] Y. Lee, R. Krashinsky, V. Grover, S. Keckler, and K. Asanovic. Convergence and Scalarization for Data-Parallel Architectures. In *CGO: International Symposium on Code Generation and Optimization*. IEEE, 2013.

[93] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. Kim, T. Aamodt, and V. Reddi. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *ISCA-40: International Symposium on Computer Architecture*. ACM, 2013.

[94] Y. Li, J. Dongarra, and S. Tomov. A Note on Auto-tuning GEMM for GPUs. In *ICCS: International Conference on Computational Science*. Springer, 2009.

[95] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architectures. *IEEE Micro*, 28:39–55, 2008.

[96] A. Magni, C. Dubach, and M. O'Boyle. A Large-Scale Cross-Architecture Evaluation of Thread-Coarsening. In *SC: International Conference on High Performance Computing Networking, Storage and Analysis*, 2013.

[97] B. Massingill, T. Mattson, and B. Sanders. A Pattern Language for Parallel Application Programming. In *PLoP-6: Pattern Languages of Programs Workshop*, 1999.

[98] T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2004.

[99] B. Mei, A. Lambrechts, J.-Y. Mignolet, D. Verkest, and R. Lauwereins. Architecture exploration for a reconfigurable architecture template. *IEEE Design Test of Computers*, 22(2):90–101, 2005.

[100] R. Membarth, F. Hannig, J. Teich, M. Körner, and W. Eckert. Generating Device-specific GPU Code for Local Operators in Medical Imaging. In *IPDPS-26: International Parallel and Distributed Processing Symposium*. IEEE, 2012.

[101] R. Membarth, A. Lokhmotov, and J. Teich. Generating GPU Code from a High-level Representation for Image Processing Kernels. In *HPPC: Workshop on Highly Parallel Processing on a Chip*. Springer, 2011.

[102] J. Meng, D. Tarjan, and K. Skadron. Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance. In *ISCA-37: International Symposium on Computer Architecture*. ACM, 2010.

[103] G. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8), April 1965.

[104] A. Muddukrishna, P. Jonsson, V. Vlassov, and M. Brorsson. Locality-Aware Task Scheduling and Data Distribution on NUMA Systems. In *IWOMP-9: International Workshop on OpenMP*. Springer, 2013.

[105] V. Narasiman, M. Shebanow, C. Lee, R. Miftakhutdinov, O. Mutlu, and Y. Patt. Improving GPU Performance via Large Warps and Two-level Warp Scheduling. In *MICRO-44: International Symposium on Microarchitecture*. ACM, 2011.

[106] G. Noaje, C. Jaillet, and M. Krajecki. Source-to- Source Code Translator: OpenMP C to CUDA. In *HPCC-13: International Conference on High Performance Computing and Communications*. IEEE, 2011.

[107] NVIDIA. *CUDA C Programming Guide 5.5*, 2013.

[108] A. Parakh, M. Balakrishnan, and K. Paul. Performance Estimation of GPUs with Cache. In *IPDPSW-26: International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*. IEEE, 2012.

[109] E. Park, L.-N. Pouchet, J. Cavazos, A. Cohen, and P. Sadayappan. Predictive Modeling in a Polyhedral Optimization Space. In *CGO: International Symposium on Code Generation and Optimization*. IEEE, 2011.

[110] D. Patterson. The Top 10 Innovations in the New Fermi Architecture, and the Top 3 Next Challenges. NVIDIA Whitepaper, 2009.

[111] D. Patterson. The Trouble with Multi-Core. *IEEE Spectrum*, 47(7):28–32, 53, 2010.

[112] S. Pennycook, S. Hammond, S. Wright, J. Herdman, I. Miller, and S. Jarvis. An Investigation of the Performance Portability of OpenCL. *Elsevier Journal of Parallel and Distributed Computing*, 2012.

[113] J. Philbin, J. Edler, O. Anshus, C. Douglas, and K. Li. Thread Scheduling for Cache Locality. In *ASPLOS-7: International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 1996.

[114] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The Tao of Parallelism in Algorithms. In *PLDI-32: Conference on Programming Language Design and Implementation.* ACM, 2011.

[115] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative Optimization in the Polyhedral Model: Part I, One-Dimensional Time. In *CGO: International Symposium on Code Generation and Optimization.* IEEE, 2007.

[116] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Combined Iterative and Model-driven Optimization in an Automatic Parallelization Framework. In *SC: International Conference on High Performance Computing Networking, Storage and Analysis.* IEEE, 2010.

[117] L.-N. Pouchet and S. Grauer-Gray. PolyBench: The Polyhedral Benchmark Suite. On-line: http://www.cse.ohio-state.edu/∼pouchet/software/polybench/.

[118] W. Pugh. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. In *SC: International Conference on High Performance Computing Networking, Storage and Analysis.* ACM, 1991.

[119] T. Rogers, M. O'Connor, and T. Aamodt. Cache-Conscious Wavefront Scheduling. In *MICRO-45: International Symposium on Microarchitecture.* IEEE, 2012.

[120] A. Sandberg, A. Sembrant, E. Hagersten, and D. Black-Schaffer. Modeling Performance Variation due to Cache Sharing. In *HPCA-19: International Symposium on High Performance Computer Architecture.* IEEE, 2013.

[121] S. Sato and H. Iwasaki. A Skeletal Parallel Framework with Fusion Optimizer for GPGPU Programming. In *APLAS-7: Asian Symposium on Programming Languages and Systems.* Springer, 2009.

[122] D. Schuff, B. Parsons, and V. Pai. Multicore-Aware Reuse Distance Analysis. In *IPDPSW-24: International Symposium on Parallel and Distributed Processing Workshops and PhD Forum.* IEEE, 2010.

[123] D. She, Y. He, L. Waeijen, and H. Corporaal. OpenCL Code Generation for Low Energy Wide SIMD Architectures with Explicit Datapath. In *SAMOS-XIII: International Conference on Embedded Computer Systems.* IEEE, 2013.

[124] J. Shen, J. Fang, H. Sips, and A. Varbanescu. Performance Gaps between OpenMP and OpenCL for Multi-core CPUs. In *ICPPW: International Conference on Parallel Processing Workshops.* IEEE, 2012.

[125] K. Skadron. The Rodinia Benchmark Suite. On-line: http://lava.cs.virginia.edu/Rodinia/.

[126] M. Steuwer, P. Kegel, and S. Gorlatch. SkelCL - A Portable Skeleton Library for High-Level GPU Programming. In *IPDPSW-25: International Symposium on Parallel and Distributed Processing Workshops and PhD Forum.* IEEE, 2011.

[127] J. Stratton, N. Anssari, C. Rodrigues, I.-J. Sung, N. Obeid, L. Chang, G. Liu, and W. Hwu. Optimization and Architecture Effects on GPU Computing Workload Performance. In *INPAR: Workshop on Innovative Parallel Computing.* IEEE, 2012.

[128] J. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. Liu, and W.-M. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical Report IMPACT-12-01, University of Illinois, 2012.

[129] A. Stromme, R. Carlson, and T. Newhall. Chestnut: A GPU Programming Language for Non-Experts. In *PMAM: Programming Models and Applications for Multicores and Manycores.* ACM, 2012.

[130] H. Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), March 2005.

[131] D. Tam, R. Azimi, and M. Stumm. Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors. In *EuroSys-2: European Conference on Computer Systems.* ACM, 2007.

[132] T. Tang, X. Yang, and Y. Lin. Cache Miss Analysis for GPU Programs Based on Stack Distance Profile. In *ICDCS-31: International Conference on Distributed Computing Systems.* IEEE, 2011.

[133] M. Taylor. Is Dark Silicon Useful?: Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse. In *DAC-49: Design Automation Conference.* ACM, 2012.

[134] TechPowerUp. GPU Database. On-line: http://www.techpowerup.com/gpudb/.

[135] S.-Z. Ueng, M. Lathara, S. Baghsorkhi, and W.-M. Hwu. CUDA-Lite: Reducing GPU Programming Complexity. In *LCPC-21: International Workshop on Languages and Compilers for Parallel Computing.* Springer, 2008.

[136] D. Unat, X. Cai, and S. Baden. Mint: Realizing CUDA Performance in 3D Stencil Methods with Annotated C. In *ICS-25: International Conference on Supercomputing.* ACM, 2011.

[137] S. Verdoolaege. isl: An Integer Set Library for the Polyhedral Model. In *ICMS-3: International Conference on Mathematical Software*. Springer, 2010.

[138] S. Verdoolaege. Polyhedral Process Networks. *Handbook of Signal Processing Systems*, pages 931–965, 2010.

[139] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. Polyhedral Parallel Code Generation for CUDA. *ACM Transactions on Architecture and Code Optimisations*, 9(4):Article 54, Jan. 2013.

[140] S. Verdoolaege and T. Grosser. Polyhedral Extraction Tool. In *IMPACT-2: International Workshop on Polyhedral Compilation Techniques*, 2012.

[141] S. Wienke, P. Springer, C. Terboven, and D. Mey. OpenACC - First Experiences with Real-World Applications. In *Euro-Par: European Conference on Parallel and Distributed Computing*. Springer, 2012.

[142] A. Wijs and D. Bosnacki. Improving GPU Sparse Matrix-Vector Multiplication for Probabilistic Model Checking. In *SPIN-19: Workshop on Model Checking Software*. Springer, 2012.

[143] Wikipedia. List of Intel Microprocessors. On-line: http://wikipedia.org/wiki/List_of_Intel_microprocessors/.

[144] S. Williams, A. Waterman, and D. Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52:65–76, Apr 2009.

[145] M. Wolf and M. Lam. A Data Locality Optimizing Algorithm. In *PLDI-12: Conference on Programming Language Design and Implementation*. ACM, 1991.

[146] M. Wolfe. Implementing the PGI Accelerator Model. In *GPGPU-3: Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2010.

[147] H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU Microarchitecture through Microbenchmarking. In *ISPASS: International Symposium on Performance Analysis of Systems and Software*. IEEE, 2010.

[148] W. Wulf and S. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Computer Arch. News*, 23(1):20–24, 1995.

[149] W. Xu, H. Zhang, S. Jiao, D. Wang, F. Song, and Z. Liu. Optimizing Sparse Matrix Vector Multiplication Using Cache Blocking Method on Fermi GPU. In *SNPD-13: International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel Distributed Computing*. IEEE, 2012.

Summary

Improving the Programmability of GPU Architectures

Throughout the past decades, the tremendous growth of single-core performance has been the key-enabler for digital technology to become ubiquitous in our society. Recently, diminishing returns on Dennard scaling resulted in power dissipation issues, leading to reduced performance growth. Performance growth has since been re-enabled by multi-core processors as well as by exploiting the energy efficiency of specialised accelerators such as graphics processing units (GPUs). This has led to a heterogeneous and parallel computing environment, making programming a challenging task. Programmers are faced with a variety of new languages and are required to deal with the architecture's parallelism and memory hierarchy. This has become increasingly important, especially considering the *memory wall* and the prospect of *dark silicon*. Apart from programming, issues such as code maintainability and portability have become of major importance.

To address these issues, this thesis first introduces *algorithmic species*: a classification of program code based on memory access patterns. Algorithmic species is a structured classification that programmers and compilers can use for example to take parallelisation decisions or to perform memory access optimisations. The algorithmic species classification is used in a skeleton-based compiler to automatically generate efficient and readable code for GPUs and other parallel processors. To do so, C code is first automatically annotated with species information. The annotated code is subsequently fed into BONES, a source-to-source compiler that provides pre-optimised code templates ('skeletons') for specific algorithmic species. By applying traditional and species-based optimisations such as thread coarsening and kernel fusion on top of this, BONES is able to generate competitive code. Combining skeletons with a program code classification (the species) creates a unique code generation approach, integrating a skeleton-based compiler into an automated compilation flow for the first time.

Furthermore, this thesis proposes to change the GPU's thread scheduling mechanism to improve its programmability. Programming models for GPUs allow programmers to specify the independence of threads, removing ordering constraints. Still, GPUs do not exploit the potential for locality (e.g. improving cache performance) enabled by this independence: threads are scheduled in a fixed order. This thesis quantifies the potential of scheduling in a 'locality-aware' manner. A detailed reuse-distance based cache model for GPUs is introduced to provide better insight into locality and cache behaviour.

Acknowledgements

First of all, I thank my advisor and first promotor Henk Corporaal. His supervision style encouraged independent work, greatly stimulating creativity. Although his feedback on my written work (including this thesis) was often last-minute, it was well worth the wait: his comments were always constructive and useful. I am thankful to have been able to work under his supervision. I also thank Bart Mesman for being my daily supervisor in the first year. I very much appreciated his weekly brainstorm sessions, the so-called 'architecture meetings'. I also thank Henri Bal, my second promotor. Even though he was only involved in the last year, the end result benefited considerably from his feedback and advice.

Second, I thank Paul Kelly, Albert Cohen, Ana-Lucia Varbanescu, Gerard de Haan and Johan Lukkien for taking place in the doctorate committee. I also thank them for their guidance and valuable feedback.

I thank all others that directly contributed to this thesis. In particular, I thank Gert-Jan for being co-author on many articles, for our numerous scientific discussions, and for guiding me through (GPU) programming problems. I also thank him for all his practical help, including our 'road-trips' to buy new GPUs from the local computer stores. My thanks also go to the reviews and discussions of co-authors Rosilde and Shakith. The hard work and valuable contributions of the master-students Pieter and Colin was also highly appreciated.

I also thank the people at ARM for their discussions, support and friendship. First and foremost I thank Anton Lokhmotov, who has introduced me to ARM and shepherded me afterwards, and Elena Stöhr, my manager. I thank my friends and office-mates Alexey, Georgia, Joey, Murat, and everyone else on the Mali OpenCL compiler team. In particular I thank my friend and colleague Alberto, with whom I had many fruitful discussions, some of which even led to scientific publications.

My stay at ARM was sponsored by HiPEAC, a network encouraging collaboration among researchers. I met many fellow students through this network and by visiting international conferences. Our discussions (and social events) have contributed indirectly to this thesis. Thanks go to José, Ananya, Carlo, Foivos, Andrew, Nuno, and Aleksandar, just to name a few. I also thank Dominik, Christophe, Chris, and all the other sociable members of Edinburgh's Informatics Forum.

I also thank the members of the PARsE team at the TU/e for their constructive feedback during any one of my many presentations. Thanks go to Zhenyu, Maurice, Dongrui, Yifan, Luc, Erkan and all the others. Your feedback and remarks have contributed to this result.

Many thanks to everyone at the Electronic Systems group at the TU/e. Thanks go to my office-mates Davit, Manil, Firew, Roel, Luc, Wei, Chris, and Shiqi, who were always very helpful and kind. Many thanks go as well to Raymond, Marcel, Sven, Yang, Marc, Sander, Martijn and everyone else at the coffee table for the daily (non-)scientific discussions. My thanks also go to Benny for being everything a teaching assistant can wish for. I also value the work, support, and care of the group leaders Ralph Otten and Twan Basten, the project manager Jan, and the secretaries Marja and Rian.

I also thank my family and friends for being supportive. At age 6, my parents learned me how to program in Logo (🐢) on an Atari ST. I thank Mark and Michiel for making me a GPU (or 3D-accelerator) enthousiast as early as 1998. I also thank my father and sister for being exemplars of how to become a doctor. Finally, I thank my girlfriend Simone for her love, advice, cooking and support.

About the author

Cedric was born in the Netherlands, but grew up in Lusaka (Zambia) where he went to l'École Française. At age of 7, he moved back to his country of birth and attended primary and secondary school in Dordrecht. He came to Eindhoven as a student at the age of 18, where he successfully completed a bachelor in Electrical Engineering (in 3 years) and a master in Embedded Systems (in 2 years) at Eindhoven University of Technology (TU/e). He spent half a year of his master program abroad in Valencia, Spain. In November 2009, he signed a 1-year contract at the TU/e, which was later succeeded by a 3-year PhD contract. Cedric spent four months of his time as a PhD-student at ARM in Cambridge (UK), where he conducted research as part of ARM's Mali OpenCL compiler team.

His research interests include GPUs, (parallel) programming, code generation, high-performance computing, multi/many-core processor architecture, and performance modelling. Apart from his research, Cedric is interested in entrepreneurship and web-design: examples are his own company Kinento and the website of the PARsE research team.

Cedric's PhD at the TU/e has led to a journal article and several publications in international conferences and workshops (see page 141). After finishing writing this thesis, he worked for 4 months as an intern at NVIDIA in Santa Clara, California.