Imperial College London
Department of Computing

Indexed dependence metadata and its applications in software performance optimisation

Lee William Howes

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London, February 2010

Abstract

To achieve continued performance improvements, modern microprocessor design is tending to concentrate an increasing proportion of hardware on computation units with less automatic management of data movement and extraction of parallelism. As a result, architectures increasingly include multiple computation cores and complicated, software-managed memory hierarchies. Compilers have difficulty characterizing the behaviour of a kernel in a general enough manner to enable automatic generation of efficient code in any but the most straightforward of cases.

We propose the concept of indexed dependence metadata to improve application development and mapping onto such architectures. The metadata represent both the iteration space of a kernel and the mapping of that iteration space from a given index to the set of data elements that iteration might use: thus the dependence metadata is indexed by the kernel's iteration space. This explicit mapping allows the compiler or runtime to optimise the program more efficiently, and improves the program structure for the developer. We argue that this form of explicit interface specification reduces the need for premature, architecture-specific optimisation. It improves program portability, supports inter-component optimisation and enables generation of efficient data movement code.

We offer the following contributions: an introduction to the concept of indexed dependence metadata as a generalisation of stream programming, a demonstration of its advantages in a component programming system, the decoupled access/execute model for C++ programs, and how indexed dependence metadata might be used to improve the programming model for GPU-based designs. Our experimental results with prototype implementations show that indexed dependence metadata supports automatic synthesis of double-buffered data movement for the Cell processor and enables aggressive loop fusion optimisations in image processing, linear algebra and multigrid application case studies.

Declaration of originality

This document consists of research work conducted in the Department of Computing at Imperial College London between 2005 and 2009. I declare that the work presented is my own, except where specifically acknowledged in the text.

Acknowledgements

I gratefully acknowledge in particular the support of my supervisors Paul Kelly and Tony Field. Paul for providing regular and thorough discussions of topics that were in no way limited to the work, but were at all times interesting and enjoyable. Tony for regular chats over coffee and during meetings but mostly for his very effective help in improving my writing.

I'd like to thank Jay Cornwall for his contributions through frequent discussion on programming models for visual effects; and the members of both the Custom Computing and Software Performance Optimisation research groups for general help towards completing a PhD.

David Thomas, Anton Lokhmotov and Alastair Donaldson have been particularly helpful through their collaborations on the work in this thesis and the related papers. David for his contributions and expertise in random number generation, Anton for wide ranging support and discussion of the Æcute work, and Alastair for his help in producing the comparison benchmarks for Chapter 7.

Finally I'd like to thank my family for their encouragement, the EPSRC for funding this work through a PhD studentship and my examiners for their diligent reading of the text and suggested improvements.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The "memory wall" is a concept that has been discussed frequently over the years. Wulf and McKee noted back in 1994 [WM95] that "we are going to hit a wall in the improvement of system performance unless something *basic* changes." They suggested that based on certain assumptions, performance of applications would be memory limited within a decade.

A decade later McKee published an update [McK04]. She states that MFLOPS have increased by 50% per year and sustained memory bandwidth at only 35% per year. Latency increased 80x measured in "equivalent FLOPS" between 1990 and 2003. Transaction processing workloads see 65% node idle times and, more worryingly still, scientific computations see 95% idle times. She asserts that for many types of application the memory wall is being reached.

> If we want performance rates to continue to increase, whether processor performance resumes its prior growth, or stalls near current levels, we need a paradigm shift in our programming models to enable the application of significant parallelism to real applications without heroic programming efforts. John D. McCalpin[1]

Caches and banked memory aim to reduce this problem. Heavily banked memory as seen in vector supercomputers suffers from hardware overhead in crossbars. To maintain high cache hit rates a large amount of CPU area must be dedicated to cache control logic. Unfortunately, memory access patterns can easily conflict with the pre-fetching logic built into caches. This problem has led to processors like the IBM/Sony/Toshiba Cell processor being designed without automatic caching; rather the programmer is left to handle the problem himself. In the words of Peter Hofstee, chief architect of the Cell processor:

> . . . managing memory locality becomes the main factor determining software performance, and writers of compilers and high-performance software alike spend much of

[1] In a presentation given at the University of California at San Diego/San Diego Supercomputing Center in February 2004.

> their time **reverse-engineering** and **defeating** the sophisticated mechanisms that **automatically** bring data on to and off the chip. Given the large number of transistors devoted to these mechanisms this is an unsatisfactory situation. [Hof05]

Unfortunately, implementing data movement correctly is a tedious and error-prone process, leading to premature optimisation for a given architecture. Such optimisation is not portable, and disrupts the code base.

Given a series of computational kernels that relate to each other and each compute over the entire data set, data communicated between kernels will be copied into the cache and repeatedly removed by later data. Maintaining locality adds additional workload on the programmer. New programming models are developed to counter this additional complexity: stream programming, memory-hierarchy programming and hierarchically-tiled arrays attempt to solve the problem. In this thesis we present another approach to the problem: the use of metadata to describe data access relationships and allowing the compiler to infer the appropriate use of the memory hierarchy.

1.2 Thesis and contributions

We propose *indexed dependence metadata*, and the Ækute programming model as a specific instantiation, as methods both to reduce the memory wall problem in software to a manageable level and to localise the changes needed in a code base to support future architectural changes to the memory hierarchies used by hardware manufacturers to reduce its effects. *Indexed dependence metadata* separates the execution plan of a computation from the description of its data access pattern. We aim to show that this separation, combined with compiler and run-time techniques acting on the information it provides, enables performance improvements that would be difficult or impossible to attain previously.

To support this claim we provide the following set of contributions:

- We demonstrate in Chapter 4 the GPU variant of a naive unified programming description for GPUs, FPGAs and the small vector units on the Sony PlayStation 2. We also show how this programming model only has limited scope and that for more complicated problems it is too restrictive.
- In Chapter 5 we introduce the concept of *indexed dependence metadata*. This metadata can be seen as a generalisation of stream programming that enables a wider range of optimisations and code generation opportunities than we could achieve with the model in Chapter 4.
- We show how indexed dependence metadata can be used in a component programming system. We demonstrate in Chapter 6 the use of indexed dependence metadata on component interfaces, and implement a framework that uses the metadata constructs to perform fusion and array contraction of connected components. We then show how these optimisations give performance benefits.
- Chapter 7 introduces the concept of decoupled access/execute metadata and the Ækute programming model. This variant of indexed dependence metadata is used in a programming system for the Cell processor, automatically executing DMA operations for kernels written to execute only on local memory regions.

- In the final contribution chapter, Chapter 8, we show the performance variation possible depending on the implementation of even simple kernels on GPU architectures. We show how code generation, using the Æcute model and indexed dependence metadata, can be used to ease the development process and automate the production of high-performance GPU code.

1.3 Structure of this document

Chapter 2 discusses the concept of decoupling in terms of both architectural decoupling and software decoupling. Decoupling is an important concept vital to the later discussions on indexed dependence metadata.

Chapter 3 summarises the state of the art in high-performance hardware, software and compilation methods. In this chapter we can see metadata concepts in the wider context, before we later discuss these constructs and how they apply to our work.

Chapter 4 discusses early work on stream programming of GPUs using a simple and literal stream pipeline originally developed for FPGA development. The work in this chapter resulted in the publications at FPL 2006 [HBM+06] and EDGE 2006 [HPMB06] in combination with the work of Paul Price on targeting the PlayStation 2.

Chapter 5 introduces the idea of Indexed Dependence Metadata as a method of representing a more flexible mapping of computation to data than is available from stream programming. The metadata ideas introduced in this chapter form the basis of Chapter 6, Chapter 7 and Chapter 8. These later chapters place the metadata in specific implementations and show how it can be applied.

Chapter 6 discusses how Indexed Dependence Metadata can be used to optimise sets of components. Using metadata of the form described in Chapter 5 we develop a component programming framework. The information provided by metadata on the component interfaces describes adequately the interactions between components and data, and hence components and components. Cross-component optimisations and parallelisation decisions are possible using this metadata without requiring composition-time analysis of the component code. The work in this chapter is published [HLKF08a] in the proceedings of the HipHaC workshop at MICRO 2008.

In Chapter 7 we present the Æcute programming model that demonstrates how the principles of Indexed Dependence Metadata can be used in a programming model to ease data movement on novel computer architectures. We construct a C++ framework that wraps metadata in data types. These data types wrap all data accesses within a kernel and allow the runtime to manage data movement. This chapter is published [HLDK09c] at HiPEAC 2009.

Chapter 8 discusses the difficulties inherent in programming for GPUs and the wide range of code variations necessary to obtain good performance, in particular from NVIDIA's CUDA targeted hardware. We propose indexed dependence metadata, and the related Æcute programming model, as a solution to this problem and give some idea of how the code for such kernels could be efficiently generated. This work is published in SAAHPC 2009 [HLDK09a] and HPPC 2009 [HLDK09b].

We summarise the thesis in Chapter 9 and comment on how well the work achieved its goals, finishing with a discussion of possible future directions that this work might take.

1.4 Publications resulting from work to date

Comparing FPGAs to Graphics Accelerators and the PlayStation 2 Using a Unified Source Description [HBM+06]

This paper describes early work on stream programming for the GPU and PS2 published at the IEEE Conference on Field Programmable Logic and Applications, 2006 [HBM+06]. In this work we took data-flow stream descriptions of kernels and generated GPU and PS2 code from them to compare with FPGA implementations. The advantages and limits of this approach guided research towards the later approaches. We discuss this work further in Chapter 4.

Accelerating the Development of Hardware Accelerators [HPMB06]

Poster on FFT performance and stream programming of GPUs, published at the EDGE workshop at the University of North Carolina, Chapel Hill. This unrefereed publication also presents Chapter 4, with further work on high performance FFT implementations.

Automating generation of data movement code for processors with distributed memories [HLKD08]

Short paper presented at the 5th HiPEAC industrial workshop. HP Labs, Barcelona, Spain, June 2008.

Optimising component composition using indexed dependence metadata [HLKF08a]

Presented at the HipHaC workshop at MICRO 2008 to discuss how metadata can be used on the interfaces of components to allow optimisations across component boundaries. By providing dependency information on the component interfaces internal analysis is no longer necessary and a wider range of optimisations can be performed. This work is discussed in Chapter 6.

Deriving efficient data movement from decoupled Access/Execute specifications [HLDK09c]

Indexed dependence metadata can be used in a subtly different form to provide data movement information on esoteric architectures. In this work, published at HiPEAC 2009, we present the decoupled access/execute (Æcute) programming model and a library implementation that provides a convenient programming model for the Cell processor. The work is described in Chapter 7.

Decoupled Access/Execute meta-programming for GPU-accelerated systems [HLDK09a]

We suggest that indexed dependence metadata can also be used to support the data movement and iteration space optimisations necessary to obtain good performance from GPU-based architectures. This work, published at SAAHPC 2009 discusses the difficulties of programming for such architectures and proposes the Æcute programming model as a solution. The work is described in Chapter 8.

Towards Metaprogramming for Parallel Systems on a Chip [HLDK09b]

An extended discussion of the Æcute model as a programming methodology for GPU systems. This paper discusses the tradeoffs necessary when producing high performance code and the requirement to fully understand the tradeoffs and produce code that can be ported easily to new architectures. The work was presented at the HPPC 2009 workshop.

1.5 Other publications

Efficient random number generation and application using CUDA [HT07]

Chapter in GPU Gems 3 discussing Gaussian random number generation on NVIDIA GPUs using the new CUDA technology and the use of these random numbers in Monte Carlo simulations for finance.

A Comparison of CPUs, GPUs, FPGAs, and Massively Parallel Processor Arrays for Random Number Generation [THL09]

An extension of the GPU Gems work to consider random number generation on a wider set of architectures published at FPGA 2009.

High-performance SIMT code generation in an active visual effects library [CHK+09]

Closely related to the work on indexed dependence metadata this paper discusses the application of domain-specific metadata to a visual effects library. Generation of efficient CPU and GPU-code from visual effects kernels reduces the development and maintenance overhead. The use of dependence metadata in the form of skeletal representations of basic kernels allows the visual effects framework to generate efficient code for a wide range of effects.

Performance Comparison of Graphics Processors to Reconfigurable Logic: A Case Study [CCLHa]

Accepted for publication in IEEE Transactions on Computers, September 2009. Takes a systematic approach to the comparison of the GPU and reconfigurable logic. Uses five case study algorithms and two target devices and summarises the results.

A Systematic Design Space Exploration Approach to Customising Multi-processor Architectures: Exemplified using Graphics Processors [CCLHb]

Accepted for publication in SAMOS Journal, November 2009. Describes a systematic approach to customising homogeneous multi-processor architectures using a novel design space exploration tool and parameterisable system model.

Chapter 2

Background: decoupling

2.1 Introduction

Applications can generally be divided into computational elements or *kernels* that perform different tasks towards the overall goal. At the same time, the work performed by a given application can often be divided orthogonally into a set of concepts that weave through the various kernels. By acknowledging these concepts both when developing and executing the application we can gain advantage by utilising the various hardware features to best effect, for example in using hardware DMA engines to move data around, or simply by cleanly pipelining the execution to allow out-of-order execution logic to work efficiently. We can correctly target each feature at development time without excessive overlap in the process of this *decoupling*, removing the need to extract the separate streams later.

In this chapter we discuss the concepts of decoupled hardware and software, as they stand today. Later we will see how these concepts relate to the core chapters of this thesis.

2.2 Decoupling in hardware

Decoupling in hardware generally separates the memory access structures from computation structures such that they maintain a degree of independence. This may be through separate processors, co-processors or through the use of programmer controlled data regions.

2.2.1 The decoupled access/execute architecture

Smith [Smi84] discusses the idea of a *decoupled access/execute architecture* that separates the system into two distinct functional units, with separate instruction streams. These units can be termed the *access processor* and the *execute processor*. Each unit has a distinct set of registers and execute separate programs serving different functions. The *access processor* performs all data movement operations between main memory and a FIFO queue which feeds the *execute processor*, including all address computation operations. The *execute processor* takes operands off the queue, performs computations and places results on a return queue. The separation and resulting drift in alignment

between the two executing instruction streams acts like a sophisticated superset of standard cache prefetching and hence reduces memory access latency and improves throughput by removing memory stalls.

Complexities in this architecture arise from the need to synchronise the two processors. Computed memory addresses must be passed from the *execute processor* to the *access processor*. Loads and stores that are reordered relative to the computation stream must correctly handle address reuse: programmed interlocks or address comparison tables are proposed for this purpose. Conditional branches change control flow and must be correctly coordinated between the two processors.

Smith argues that this dual instruction stream architecture suffers from limitations. It is simple, with relatively straightforward implementations (though this might not carry over into modern massively out-of-order architectures) but the complexity of having the programmer or compiler writer deal with multiple instruction streams is a severe overhead. The hardware is also wasteful, given the need for multiple fetch/decode units. Smith suggests single instruction stream decoupled access/execute architectures, using techniques such a stream interleaving. This means sharing the fetch/decode logic, but being separable in terms of scheduling. Large scale out-of-order processors go some way to remove the same restrictions that the *decoupled access/execute architecture* aims to solve. By shifting the instruction streams relative to each other when dozens of instructions are in flight, more sophisticated arrangements and fine grained dependencies can be utilised. This complexity comes with the cost of a severe computation overhead.

Topham et al. [TRM+95] discuss the sources of loss-of-decoupling events. These are defined as a flows of information in the wrong direction through a decoupled pipeline. That is to say that data flows from the execution stage to the access stage. Such events are the major loss of performance for decoupled architectures. They list a number of sources and propose solutions:

Indirect accesses
: Where accesses require an additional memory lookup to be determined. Pre-queuing is proposed as a solution to this problem.

Computed array indices
: Where the address to read is computed during the execution stage. Loop distribution is a possible solution.

I-cache disruption
: Any stall of the instruction stream causes a problem, this is not specific to decoupled architectures.

Control transfer
: Where control flow for the access is dependent on the execution stage. Sometimes solved through if-conversion.

While loops
: Any loop that terminates at an iteration that cannot be pre-computed. One approach to dealing with this is speculative execution.

Talla et al. [TJ01] suggest that while some of the features made their way into commercial processors, decoupled access/execute architectures did not catch on in general because general purpose applications do not possess sufficient regularity. Multimedia applications possess greater regularity, and

represent a dominant workload on desktops and workstations. They apply decoupled techniques to media extensions to a general purpose processor, extracting not only the memory access instructions, but also SIMD-specific permute operations as "overhead" that can be decoupled from computation.

Watson et al. [WR95] discuss decoupling for parallel virtual-shared-memory architectures. Where memory is distributed across many processors, but should be viewed virtually as a shared space, prefetching of data is particularly complicated. Not only must the system decide what to prefetch locally, but in addition data must often be prefetched by copying from a remote machine with a very high latency. The high latency means that decoupled execution can offer substantial benefits.

Bird et al. [BRT93] introduce *control decoupling*. Control decoupling adds a third processor, called the *control processor* to the *address processor* and *data processor* they separately describe. Like the address processor, the control processor needs no floating point logic. Unlike the address processor it includes a full set of logical, integer arithmetic and branch operations. The result is that the control processor is executing one part of the program, the address processor an earlier part and the data processor a section that is earlier still. They conclude that decoupling is a very powerful technique for minimising the impact of memory latency and that it has wide application in real applications.

2.2.2 Run-ahead threads and the Rock processor

Similar in principle to Smith's *decoupled access/execute architecture* is the concept of *run-ahead* or *scout* threads. In architectures with such support a stall in the memory system will activate a checkpoint of the processor state. The processor will then run through the checkpoint, executing the instruction stream and performing address computations and load operations. When the memory fault is serviced the processor will be interrupted, the state recovered from checkpoint data, and execution continued from the point at which it left off. This can be seen as the *access* instruction stream, and while not executing truly in parallel with the *execute* stream, the data is prefetched in a shifted instruction schedule, much as in Smith's single instruction stream decoupled access/execute proposal.

Run-ahead execution can be seen as a sophisticated form of prefetcher, that has relatively low hardware overhead because it makes use of the otherwise idle execution logic of the processor. It can be seen as an extension of an in-order [DM97] processor that reduces the need for the complexity of out-of-order logic. In removing the need for out-of-order logic, run-ahead execution effectively increases the set of instructions that can be processed and allows the processor to step over a stall.

Run-ahead need not be an alternative to out-of-order execution. Mutlu et al. [MSWP03] propose a run-ahead system for out-of-order processor designs whereby the processor begins to run-ahead when an unserviceable instruction reaches the head of the scheduling window. The run-ahead system attempts to increase memory latency tolerance with relatively little additional hardware requirement. They show through simulation that a 128-entry instruction window with run-ahead execution gains 22% in instructions-per-cycle and performs within 1% of a machine without the run-ahead capability but with a 384-entry instruction window.

In an SMT processor resource sharing between threads can lead to problems with memory-bound threads occupying shared resources such as the re-order buffer. Ramírez et al. [RPSV08] propose a run-ahead mode for threads on an SMT processor that gains the throughput advantages of the sophisticated prefetching, but reduces the effect on other running threads that a stalling thread would have.

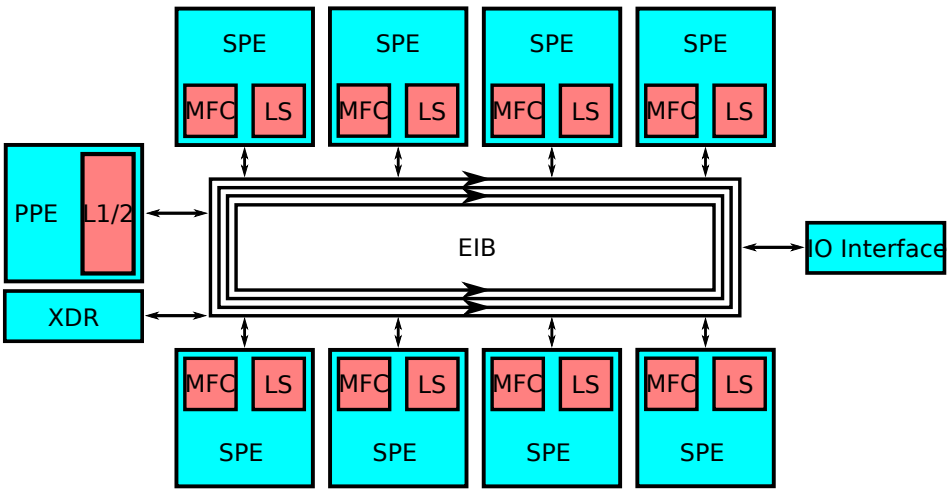


Figure 2.1: The IBM/Sony/Toshiba Cell Broadband Engine processor.

The Rock [TC08] processor from Sun Microsystems, if it reaches the market, is expected to support SMT and transactional memory as well as scout threads. An interesting aspect of Sun's design is that the scout threads are capable of retiring instructions. Rock's run-ahead support is hence not entirely speculative, as in the earlier cases. Instructions that cannot be retired are placed in a queue to be dealt with by the main thread when it exits the stall. In this fashion hardware is used more efficiently than a standard large out-of-order instruction window. An additional execution mode, *simultaneous speculative threading* allows the two SMT hardware threads on the core to execute the same code, but where one will behave speculatively and the other execute only from the deferred instruction queue, stalling where necessary.

2.2.3 The Cell BE processor

The Cell Broadband Engine processor from Sony, Toshiba and IBM is a form of decoupled architecture. As can be seen in Figure 2.1 it comprises:

Power processing element
A main, fully programmable core with vector extensions. It has 2-way simultaneous multithreading and is based on the Power ISA. The computational performance of this core is modest compared with a modern stand-alone general purpose core.

Synergistic processing elements
Up to 8 co-processors. Each co-processor is an in-order core with 4-way vector support and 256KiB of local storage. The SPEs lack any form of branch prediction or out-of-order execution and rely on compiler support to schedule instructions to allow computation to proceed efficiently. For yield reasons the number of SPEs can be varied. In the PlayStation 3 games console there are 7 active SPEs, 1 of which is used solely by the operating system.

Element interconnect bus
A high bandwidth ring network connecting the 9 cores and the main memory interface. It consists of 4 concentric 16B wide channels, each supporting up to 3 concurrent transactions.

The local store of each SPEs is software controlled, unlike a hardware cache. The only way to move data in and out of local store is through DMA operations. In most cell programming, particularly where performance is required, these DMA operations are coded manually and carefully. Software-managed caching support is available, but performance is relatively poor.

When writing DMA operations there is a clear separation of code between memory access and computation. The main computation performed on the SPE will occur in local store. The movement of data into local store is a separate operation, that has to be treated as a pure data movement, whereas on most general purpose architectures the computation could be performed on global memory locations directly, and the movement into the cache would be implicit.

In this fashion we can see the Cell SPE architecture as an example of decoupling. The PPE does not fit into this category, but the computational throughput of the PPE is relatively low and only limited computation and management is performed there.

Alignment constraints for the DMA transfers lead to a high complexity when dealing with DMA transfers. The complexity arising from these alignment issues and from the simple necessity to use DMA transfers leads to design decisions for a basic Cell implementation that can be seen to represent premature optimisation of an application. These decisions might be better left for later in the optimisation process.

2.2.4 GPUs as decoupled processors

The modern GPU design differs substantially from earlier designs. While earlier GPUs were highly graphics dedicated, NVIDIA's CUDA [NVI] programming model, and the more advanced hardware associated with it, changed this. As a dedicated graphics processor, the GPU was originally designed to process thousands of independent pixels with no communication between computational elements.

The CUDA model introduced shared memory regions on sets of SIMD processors, as seen in Figure 2.2. In the CUDA model an individual SIMD element is treated as, and programmed as, a separate thread. The threads are executed in blocks that contain many SIMD sets which can be scheduled. The shared memory regions are read/writable by any of the threads in a given block, and hence independent executing threads in the block can communicate. To ensure good performance, memory reads and writes must be arranged correctly such that sets of 16 threads, executing as a single SIMD instruction, access an efficient pattern of memory addresses. In cases where a serialisation might run horizontally across a data set, and therefore the parallelism might run vertically, SIMD elements would need to be arranged vertically. This is not efficient, and therefore staging the memory access via shared memory gives better performance. Once staging is implemented the implementation uses different read, compute and write code and hence we see decoupling of accesses from execution.

Unlike the Cell, the GPU has access to caches if data are treated as textures. In current hardware textures are read only and hence hardware caching only works on read-only data. As a result, code with simple data sharing can be optimised using the shared memory as a cache, as with Cell. Unlike Cell, this is not always necessary and so can be seen as a performance optimisation with less of a problem with premature optimisation.

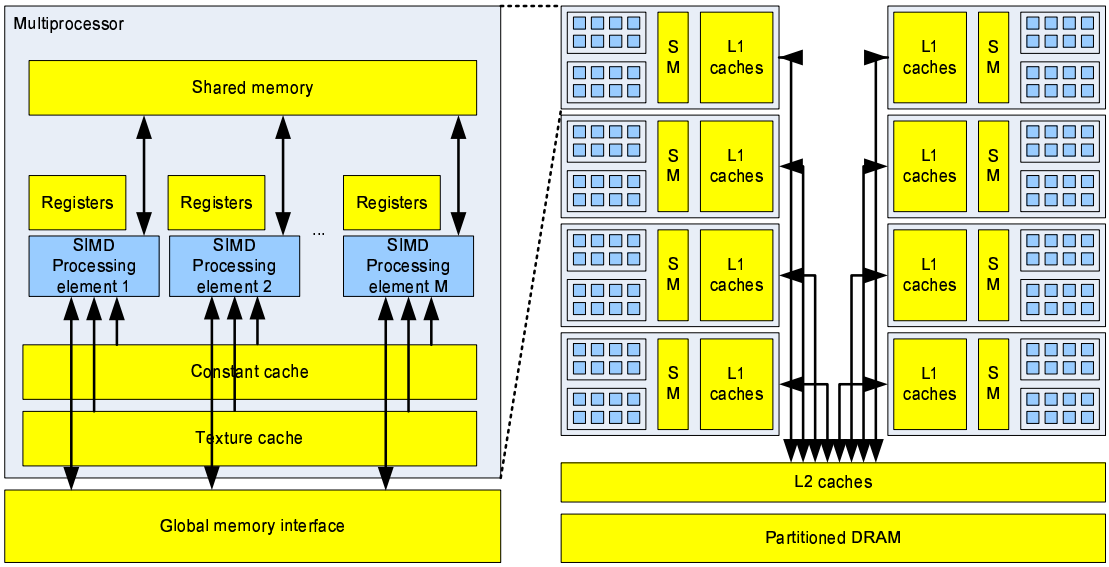


Figure 2.2: The G80 architecture as implemented in the NVIDIA 8800 GTX GPU with 16 multiprocessors. 8 scalar processors clocked at double core frequency make up a multiprocessor. Each scalar processor has access to shared memory, some set of registers and access to the constant cache, texture cache and global memory. Access to global memory (off-chip DRAM) is via a crossbar interface and set of L2 caches into 6 separate RAM partitions.

2.2.5 Anton

The Anton [SDD+07] machine from D.E.Shaw Research is a computer designed specifically for performing molecular dynamics (MD) simulations. Current equipment has been limited in the length of simulation that is feasible to perform in a reasonable time scale. While massive parallelisation can allow the simulation of a large number of small simulations, completing a single "deep" simulation in a reasonable amount of time requires more localised computing power than is has previously been available. Anton targets this problem with a dedicated architecture.

The *flexible subsystem* of the Anton architecture uses scratch pad memories in each core. These memories are used by workloads such as the integration stage of the MD computation, during which a large amount of data must be moved in and out of the scratch pad. To maintain efficiency each *processing slice* of the architecture contains a *remote access unit* (*RAU*) which provides an array of transfer descriptors and state machines. Each transfer descriptor describes a data transfer between scratch pad memory and the system. The *RAU* operates asynchronously on multiple concurrent transfers, freeing the core to perform other tasks. According to the authors of [SDD+07], "This background data transfer capability is crucial for performance, as it enables overlapped communication and computation."

2.3 Decoupling in software

Decoupling in software supports the conceptual pipelining of different concepts, again usually memory accesses and computation, while not necessarily depending on hardware support for those features.

2.3.1 Inspector executor

Saltz et al. [SMC91] discuss the Inspector/Executor paradigm. *Inspectors* perform execution time preprocessing, *executors* are the transformed versions of loop structures that carry out calculations planned by the inspector. A compiler transformation takes a loop annotated as a *doconsider* loop and generates separate inspector and executor code fragments. The inspector loop generates a wavefront number for each loop index, which are then sorted based on the wavefront numbers. The computation is performed by each processor on a subset of indexes, based on this wavefront information.

Rangan et al. [RVOA08] look at pipelined-multithreading (PMT) as a solution to parallelism. Unlike languages like StreamIT, which support PMT at the source-code level, they discuss decoupled software pipelining (DSWP) as a non-speculative automatic PMT transformation of existing code. Series of threads are extracted from code and communicate via FIFOs, such that the execution is decoupled and can better handle memory latency. Improvements in performance range to as much as 1.75 times for 8 threads. However, some benchmarks see reduced performance.

2.3.2 The polyhedral model

Traditional compiler loop transformations have tended to be relatively ad-hoc. The polyhedral model formalises the representation of and reasoning about loop nests using systems of affine recurrence equations defined over polyhedral shaped domains [QRW00]. The model uses algebraic representations of loop domains, iteration variables, dependences and relationships to allow complicated transformations to be performed in a consistent and convenient form.

We can see this as a form of decoupling. Whereas a traditional compiler representation maintains a full representation of the code including its memory accesses, the polyhedral model separates representations of the iteration space of the loop nest from each individual memory mapping in that nest. In this way mathematically grounded transformations can be applied to individual components and sequences of transformations can be applied while maintaining correctness. The *schedule* of an iteration domain can be changed without changing how the domain maps to the memory accesses.

The polyhedral model can be used for parallelisation of complex codes [Len93, Fea96]. Polyhedral parallelisation has been applied to *while* loops [Gri96] and non-affine loop nests, as well as the more natural *for* loops. Griebl later extends this work to distributed memory architectures in *LooPO* [Gri04]. Ellmenreich et al. apply the model to functional programs [EGL98] and Bastoul et al. [BCG+03] show how the model can be used for a wider range of transformations than just parallelisation. The authors note in particular that "Although classical transformations are hampered from the lack of information about loop bounds, they may be feasible in a polyhedral representation separating domains from affine schedules and authorizing per-statement operations." Recent improvements have allowed work on the composition [CGT04] of, and iterative search [BCG+03, CSG+05] for, appropriate loop transformations.

Griebl [GLW98] and Bastoul [Bas04] have both worked in this area. *CLooG* [Bas04, CLO] is probably the best known library for this purpose, and the one used for code generation in this thesis. *CLooG* is a polyhedral code generator that acts as a plugin for various polyhedral libraries – initially *Polylib* [Wil97][pol], a library designed to operate on unions of polyhedra of any dimension with support for various operations and constraints, although later versions of CLooG support other backends. PLUTO [BHRS08] is a polyhedral parallelising framework and code generator, which expresses data

dependence as a set of polyhedra derived from data flow analysis. PLUTO is a source to source transformation framework that takes C programs and outputs OpenMP parallel code, and uses CLooG as the code generation part of its tool chain.

Recent work by Pop et al. [PSC+06] looks at integrating polyhedron-based analysis into GCC with the *GRAPHITE* branch now integrated into the GCC mainline and available for public download. Girbal et al. [GVB+06] suggest that transformations are eased by utilising a clear separation between matrix representations of: the iteration domain, statement schedule, data access functions and data layout. The internal complexity of program transformations should not change significantly, unlike in standard compiler frameworks where each transformation will substantially change the internal representation and might then increase complexity. To allow reasoning about the transformation, complexity increases should not be visible until after code generation.

Größlinger [Grö09] and Baskaran et al. [BBK+08] investigate using the polyhedral model for automating data movement in memory hierarchies. Größlinger in particular takes care to precisely manage the movement of individual data elements where possible to make optimum use of the local memory regions available.

What is the polyhedral model?

Given the two loop nests in Figure 2.3d we can project the iteration spaces of the two nests onto a grid as we see in Figure 2.3c. Note that each two-dimensional loop is represented by a convex polyhedron in two-dimensional space. We can take the polyhedron and represent it as a set of inequalities defining its boundaries. For the two statements $S1$ and $S2$ in Figure 2.3 , we see the following set of inequalities:

S1			S2		
i	$\geq$	0	i	$\geq$	2
i	$\leq$	10	i	$\leq$	7
j	$\geq$	3	j	$\geq$	i
j	$\leq$	10	j	$\leq$	11

As we can see from the inequalities and the related polyhedra, the iteration spaces of the two statements overlap. Depending on the dependence information between the two statements, this may offer scope for a loop fusion optimisation. We shall see this again when we discuss scheduling later.

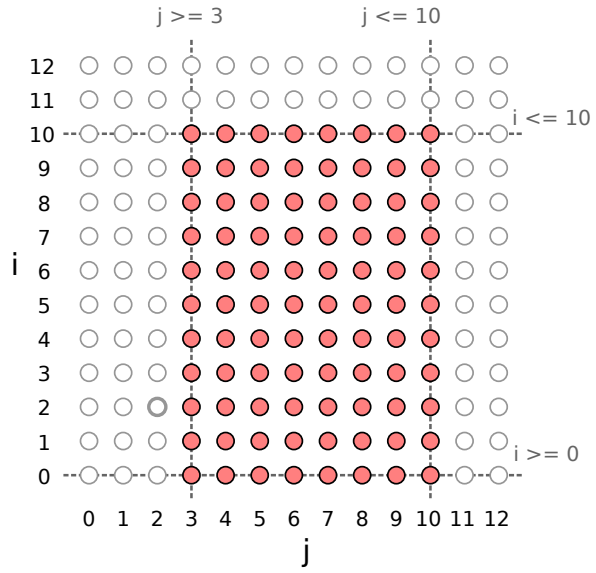
If we take the above sets of inequalities and represent them as a matrix, we have a clear mathematical structure to manipulate using standard linear algebra. For example, assuming we represent the inequalities all in $\geq$ form, and take the first column of the matrix to represent the binary status of whether the line represents an equality or an inequality, we could generate the following matrices for the inequality sets for statements *S1* and *S2*:

Matrix for S1

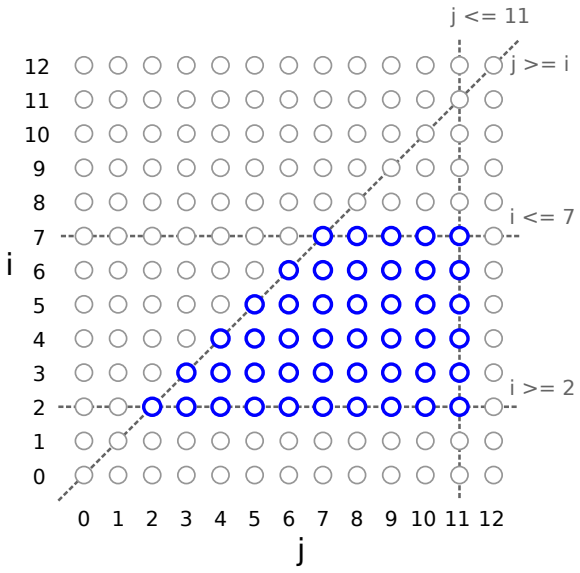
$= 0/\geq 0$	i	j	1	Definition
1	1	0	0	$i \geq 0$
1	-1	0	10	$10 - i \geq 0$
1	0	1	-3	$j - 3 \geq 0$
1	0	-1	10	$10 - j \geq 0$

Matrix for S2

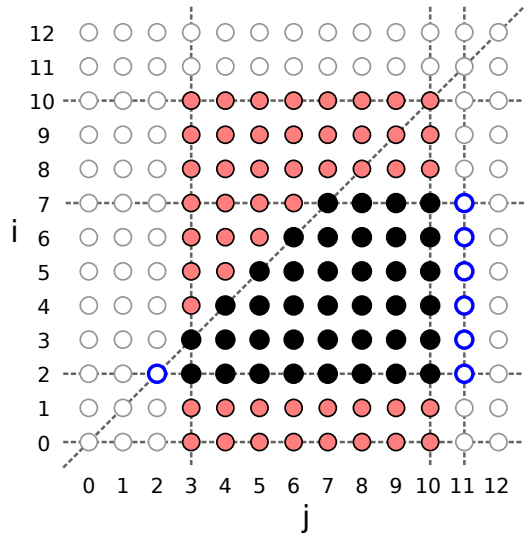
$= 0/\geq 0$	i	j	1	Definition
1	1	0	-2	$i - 2 \geq 0$
1	-1	0	7	$7 - i \geq 0$
1	-1	1	0	$j - i \geq 0$
1	0	-1	11	$11 - j \geq 0$



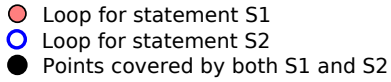
(a) Loop S1 represented polyhedrally.



(b) Loop S2 represent polyhedrally.



(c) Loops S1 and S2 overlaid on the same polyhedral space. The black region shows where both loop bodies are present in the iteration domain. Note, in particular, the set of regions where only one loop is present. These regions must be correctly covered by the loop nest if the loops are to be executed together.




```
for( int i = 0; i < 10; ++i )
  for( int j = 3; j < 11; ++j )
    S1;

for( int i = 2; i < 8; ++i )
  for( int j = i; j < 12; ++j )
    S2;
```

(d) C code for a pair of loops containing S1 and S2.

Figure 2.3: The code for a pair of loops and a polyhedral representation of the execution domains of the same loops. Each point in the diagram is an execution instance of the appropriate statement at a given point in the execution domain and hence with a particular set of loop variable assignments. The dotted lines denote the inequalities that define the loop bounds and hence divide the iteration space into half-spaces and eventually into a fully bounded polyhedron.

This simple mathematical representation is extremely useful. We can consider transformations of loops as sets of affine scheduling functions. Each statement has a scheduling function that maps from a point in its iteration space to a logical execution time – by default this might be the identity transformation or a trivial 2D to 1D address flattening. If we wish to perform a transformation on the loop nest we can apply a transformation matrix to the iteration domains as seen in Figure 2.4.

Any affine transformation of loop nests is possible, and can be chained with little increase in computational complexity. Non-affine transformations such as strip mining can be achieved using a slightly more complicated process. A strip mined loop gains additional dimensions. To perform such a transformation using a polyhedral framework, first additional dimensions must be added to the matrices. Further matrix operations convert from the lower dimensional matrices into the correct result matrix. Such transformations are still clearly defined operations on matrices. Given a set of matrices and transformations the result is a set of matrices representing statement execution schedules and domains. These transformations are performed free of syntax, in a predictable mathematical environment.

Given a transformed polyhedral representation, the task is to generate a loop nest that visits each integer point in the polyhedron of each statement in the set, such that the execution order of statements, and hence dependences between those statements, is correctly preserved. CLooG [Bas04, CLO] is an example of such a generator. It accepts a matrix defined similarly to those above as input, and assumes that no dependencies exist that would make the generated code incorrect. Any dependence information should have been represented in matrix form and used to transform the schedule earlier in the process. CLooG will generate loops recursively from the outer dimensions in, breaking the polyhedra into regions and generating loops to visit each region containing the appropriate set of statements. Even large fused sets of loops can be easily generated because the fused set is simply a set of overlaid polyhedra. Application of the one generation algorithm will correctly generate a fused loop nest. Depending on the options supplied, CLooG can also generate parts of the loop nest as conditionals rather than separate loops, helping to reduce the code explosion that can easily occur from the presence of multiple overlayed polyhedra.

Unfortunately the worst case code expansion is bad. With a substantial set of fused loops there will be many small regions with fewer than the full set of fused loops present. We can see this in Figure 2.3c where there is a single element in the bottom left corner that must be executed independently of other loop constructs. If unrolling is used to reduce the control overhead, such regions will be represented by their own distinct code blocks. If the loop structure is large enough, thousands of lines of code may be generated. Alternatively we can see this as an advantage because it is possible to generate such complicated code that by hand would be impossible to maintain. In reality a balance would have to be achieved to limit the generated code size, either by limiting the unrolling of loops or by attempting to strictly bound the iteration spaces.

In addition to domains and schedules as discussed above, the polyhedral model generalises to include dependence information and representations of access functions. This extra information enables solutions to optimising loop schedules to be found mathematically in the presence of a full description of affine dependence information. By calculating the area covered by slices through polyhedra, or by the sizes of faces of partitions it is possible to compute memory and communication volume requirements of different schedules, further supporting optimisation decisions.

Matrix representation of S1

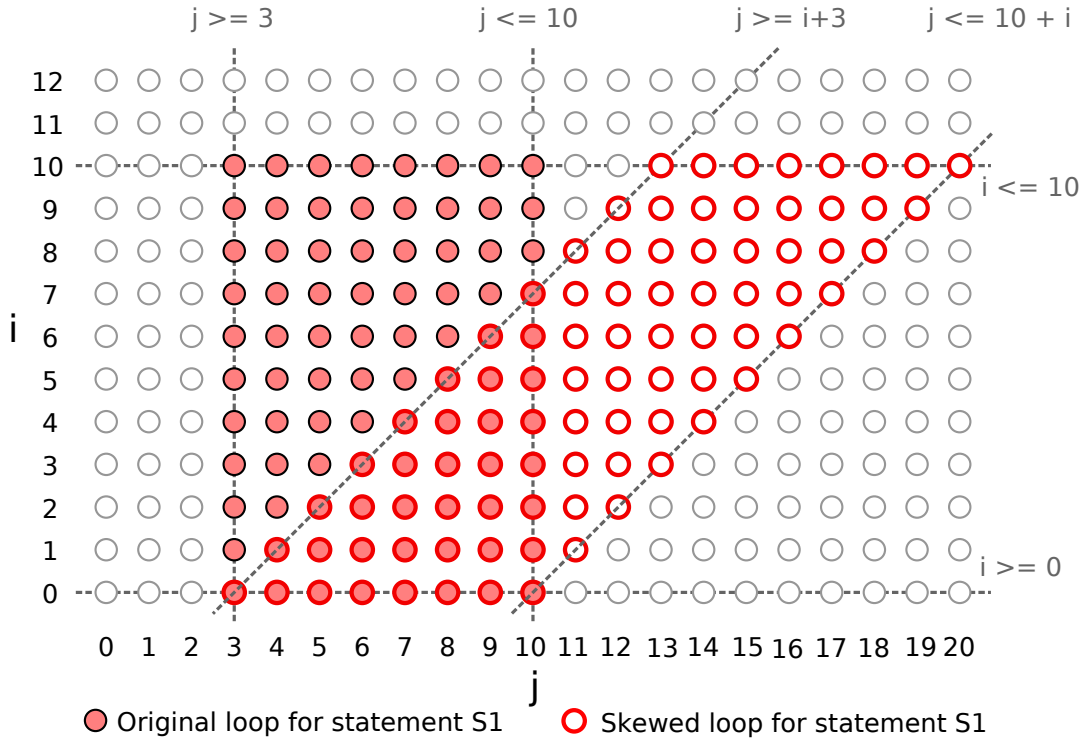
Domain			
1	1	0	0
1	-1	0	10
1	0	1	-3
1	0	-1	10

Schedule			
1	0	0	0
0	1	0	0
0	-2	1	0
0	0	0	1

Result			
1	1	0	0
1	-1	0	10
1	-2	1	-3
1	2	-1	10

Inequalities		
i	$\geq$	0
$10 - i$	$\geq$	0
$j - 2i - 3$	$\geq$	0
$2i + 10 - j$	$\geq$	0

(a) Given the original domain matrix for S1 we can apply a scheduling tranformation that, when multiplied with the matrix, produces a new schedule. In this case the new schedule is a skewing of the iteration space which can be represented by the set of inequalities shown. $domainmatrix \times schedulematrix = resultmatrix$



(b) Unskewed and skewed loops represented as polyhedra.

Figure 2.4: Given a matrix representation of loop S1 from Figure 2.3 we wish to schedule the loop to execute on some hardware. The simplest way to achieve this is to define a schedule that execute the iteration space in lexicographic order: top left to top right and so on such that iteration $2, 2$ would execute at time $2 \times width + 2$ ($(0, 0)$, $(0, 1) \ldots (1, 0)$ etc.). This would be represented by the identity transformation matrix, which is the default in a code generator such as CLooG. Instead in this case we wish to schedule the execution of the loop such that dependencies with other loop or the memory system can be corrected. We apply a transformation matrix to the iteration space to achieve this skewing.

2.4 Summary

In this chapter we have discussed decoupling in terms of both decoupled computer architectures and decoupled programming models. In the next chapter we will go through a review of literature in high performance software and its applications to various hardware. In Chapter 4 we will see a stream programming model for GPUs which introduces a simple form of decoupled programming and then introduced indexed dependence metadata, which enables a flexible form of decoupled programming, in Chapter 5.

Chapter 3

Recent developments in high performance software

3.1 Introduction

In addition to research on decoupling, there is a wide body of work on creating high performance software both from the point of view of hardware-independent optimisation, and for optimising software for specific hardware.

Examples exist of highly optimised task-dedicated libraries such as the Math Kernel Library (MKL) and Performance Primitives (IPP) from Intel which have various high performance goals. Other work has been carried out on self-optimising libraries or library generators such as *FFTW* [FJ98], *SPIRAL* [PMJ+05] and *ATLAS* [WP05][atl09]; these are intended for offering high performance on FFTs, general signal processing and the basic linear algebra subroutines (BLAS) such as matrix multiplication. We can view such library calls as components in a larger software implementation and, indeed, as a form of decoupling of the task from the wider application.

Other efforts have been made to ease the programming of various high performance architectures. Examples of this include the *Brook for GPUs* [BFH+04] and *Sequoia* [FHK+06] projects from Stanford, which aim to assist development for Graphics Processing Units (GPUs) and the Cell processor respectively. Library and development environments are one side of the story, another side offers us methods for improving the performance of current software. For example we have multi threading frameworks such as *OpenMP* [DM98] and loop transformation engines such as the *X-language* [DBR+05] and Imperial's own *Taskgraph library* [BHKM03].

The aim of this literature review is to investigate the various methods available for developing and using software and esoteric architectures for high performance, and then to draw conclusions based on this about how our work should interact with this wider state of the world.

3.2 Stream programming

It is well known that the performance improvements in CPUs over the years have not been matched by improvements to memory bandwidth or latency. *Stream programming* attempts to reduce that

problem by defining a clear model for the action of computation on data and is an important concept in many areas of both high performance and of embedded computing. The concept of a stream contains inherent data and/or task level parallelism, and hence architectures designed to work with this model can be made highly efficient. Streaming also enhances locality. Data that is used only by a single self-contained block of execution, known as a *kernel*, is described as strictly local to the kernel. Flowing data is defined as local to the communication between two kernels. In this way efficiency can be gained from the way localised data is treated.

As a basic concept stream programming involves applying a specific computation kernel to each individual element in a stream. However, given its use in literature, streaming can divide into two concepts, both applications of the above definition, but with subtly different implications for software and hardware architectures that claim to utilise a streaming model:

Task parallelism
: This is arguably a more traditional, and certainly to my mind more obvious, way of interpreting streaming. In this way of looking streaming it is a pipelining concept. Data flows from one computational element to another. There is explicit task parallelism, where a task operating at one point in the stream pushes data towards a task at another point. Individual kernels at nodes in the stream graph can possess state. Such state is capable of creating an ordering constraint on the processing of kernel elements.

Data parallelism
: In the marketing literature of companies such as AMD and NVIDIA streams are treated as fully parallel constructs where the kernel operating at on one data element in the input stream is independent of the same kernel running at other points in the stream. This is a convenient model for thinking about parallel programming, but claiming that it is the definition of streaming risks confusion as we aim to clarify later. Clearly in such cases kernels cannot maintain persistent state.

In reality there is continuum between the two, where models might be implemented in terms of data flow but not allow state to be maintained on nodes.

Streams as flowing interconnected units of computation arise on various architectures. At a fine-grained level, with their fully programmable routing and interconnection structures, FPGAs are well suited to the implementation of data-flow architectures. Mencer's *A Stream Compiler* (ASC) [Men06] supports streaming architectures for FPGAs, offering flexibility in the control of area and throughput constraints on computation elements. An *ASC* program consists of a set of entities connected in a structure that defines the data flow between them and offers efficient computation in a large pipeline. *ASC* also allows high flexibility in component implementation and contains a build system allowing design space exploration of a wide range of designs with a high degree of efficiency [MPHL03].

The GPU design, where a computation is applied in parallel to a vast number of pixels, is inherently suited to the data-parallel view of stream programming [Ven03]. The parallel and non-communicating nature of pixel computations on the GPU leads us to view this as a purely parallel streaming model, where stream state is not easy to support. Indeed, in current GPUs, running multiple communicating kernels on the hardware is difficult or impossible[1], and so the hardware is not suited to task parallelism at all. The hardware is still treated as a stream processor because the complexity of the kernels

[1] DX11 class hardware is capable of scheduling multiple kernels in parallel but programming models do not directly support this functionality and device-wide synchronisation is difficult to implement.

that can be applied at each data element is high. Stream programming environments such as *Brook for GPUs* [BFH+04] and *Sh* [MQP02], the latter stemming from a graphics shader programming system, have opened up the field for development and have lead, in the case of Sh fairly directly, to commercial products from PeakStream Inc. (now part of Google) and RapidMind. Clearspeed's CSX architecture [MGS+01][TP05] is based around GPU concepts to design a high performance floating-point processor with a stream-style programming model.

From Stanford emerged the Merrimac project [DLD+03], leading to the Brook language and hence *Brook for GPUs* [BFH+04]. *Merrimac* implements a stream architecture and related interconnection network to enable high performance stream computation with the aim of "1 petaFLOP on only[2] 8192 nodes". The *Merrimac* design uses combinations of local and stream register files to maintain data locality and hence performance. The design is merely a sketch and has not been implemented except in single node simulations. Also at Stanford, Owens et al. looked at polygon rendering on the Imagine stream architecture [ODK+00], making use of the parallelism and latency tolerance inherent in polygon rendering to obtain high performance on the architecture. Imagine is designed to allow data to efficiently flow from one ALU to another in stream fashion, but simultaneously exploits data-level parallelism by performing computations on streams in SIMD fashion, but with support for cases where the stream is not purely data-parallel. Brook is a data-parallel language where each kernel invocation is treated as independent and localised, allowing it to fit conveniently onto a stream architecture.

StreamIt [TKA02] use the concept of sequences of data items, called *streams*, which are operated on by pure functions, called *filters*. Clear (and often static) data-flow relationships between filters enable cross-component optimisations. The purity of the filters means that StreamIT applications are both task and data-parallel.

As part of the ACOTES [CRM+08] project, Carpenter et al. [CRM+07] present a machine description and streaming model as an extension of the C language. Tasks are independent with their data private, and communicate via point-to-point streams. Hence here we see another example of task-parallel streaming.

3.3 Programming parallel architectures

Various approaches have been explored for programming architectures with explicit hardware parallelism. We see the largest microprocessor companies moving increasingly in the direction of explicit parallelism while at the same time still having the range of massively parallel architectures that have always been present. In addition, we see movement from reconfigurable hardware companies using FPGA components to move into the high performance computing arena with devices such as hypertransport pluggable FPGAs from DRC [Coma] and XtremeData [Xtr]; FPGA extended supercomputers from SRC [KP06][Comb], Cray [Inc] and SGI [SGI]; also from GPU manufacturers NVIDIA [NVI] and AMD (formerly ATI) [PSG06], expanding the uses to which their GPU technology can be put.

[2] "only" in 2003 terms

3.3.1 Programming Graphics Processing Units

Graphics Processing Units (GPUs) have changed radically over the last few years[Cor04]. The technology has developed from simple graphics dedicated pipelines through utilising simply programmable pipeline stages where setting simple registers could change the way colours were combined, to its current state whereby this architecture is fully programmable, the graphics pipeline logical, and the amount of dedicated graphics logic reducing with time. As programmability has increased, use has been made of the high floating point throughput of the architectures to perform computations that are not related to graphics [gpg]. To do this directly has historically required knowledge of the graphics programming APIs, largely OpenGL and Direct3D, and a reasonably good understanding of graphics programming to deal with polygon setup, projection modes and so on. The non-graphics computation was treated as a rendering into a rectangular array represented as the image plane. Languages such as *Brook for GPUs* [BFH+04], the GPU-related version of the Brook [DLD+03] stream programming language from Stanford, streaming extensions to the Sh [MQP02] shader language and our work on compiling the *ASC* [Men06] language to GPUs [How05] (known as ASC2GPU and discussed in Chapter 4) attempt to make the programming process easier by masking the graphics API with a more-easily-dealt-with stream- or array-based API. Brook uses a number of extensions to the C language to define streams and kernels to operate on these streams, generating code for OpenGL or Direct3D based backends and in various shader languages [MGAK03] [GLS]. Sh and ASC2GPU use collections of C++ classes and operator overloading to make programming the GPU easier directly through the standard compiler infrastructure. Sh originated as a language for programming graphics shaders, and hence started off very graphics-oriented, later being adapted to the stream model. *ASC2GPU* originated from A Stream Compiler for FPGAs, implementing an algorithm as a data-flow structure streaming the dataset through the pipeline as necessary.

Recent advances to GPUs have allowed them to be viewed as general-purpose multi-core architectures [OLG+07]. Before merging with AMD, ATI announced a stream-programming initiative centred around their *Close to the Metal* (CTM) [PSG06] programming methodology. *CTM* allows low level programming of the graphics processor, making use of the *scatter* and *gather* memory access abilities of the latest ATI hardware (something that was missing in the previous generation) and allowing development for the GPU without using the OpenGL or DirectX, graphics-oriented, approach that is noticeable in earlier programming methodologies. The Folding@Home [LSSP02] project from Stanford now uses this technology to accelerate protein folding. NVIDIA's latest architecture supports a similar approach that they call the Compute Unified Device Architecture (CUDA) [NVI] which also gives access to many features of the very latest generation of hardware such as integer arithmetic and inter-thread communication through shared memory using a C based programming model. The third edition of the "GPU Gems" series of books from NVIDIA contains a number of chapters on using CUDA, including our own work on random number generation [HT07]. These advances make the hardware developed by these manufacturers worth investigating for a wide range of applications.

A further development of a shader-like programming model, similar to and arguably based on NVIDIA's CUDA, is the OpenCL [The09] standard that has come out of the Khronos standardisation group. OpenCL is designed to support a wide range of hardware and is in turn supported by a varied set of manufacturers and release quality implementations are becoming available at the time of publication of this document.

Lee et al. [LME09] discuss a system that converts OpenMP code to CUDA. They convert OpenMP work-sharing constructs to CUDA threads and perform a set of data-layout optimisations to improve

locality and performance. The results show unexpectedly large improvements over the studied CPU, which suggests the possibility that the CPU code they use is not very well optimised.

Volkov et al. [VD08] investigate the NVIDIA GPU as an architecture for accelerating linear algebra, with particularly good performance results. The most interesting contributions of their work are the summary of the architecture in realistic terms, leaving to some extend the marketing description and fairly comparing the design with other architectures, and the discussion of cache sizes which are not fully documented by NVIDIA. In particular they note that the GPU is similar in design and optimisation style to earlier vector machines.

3.3.2 Programming the Cell

The Cell Broadband Engine [Hof05][PAB+05] from IBM, Sony and Toshiba is a high performance multi-core processor that is currently most commonly found in the Sony PlayStation 3 games console and discussed in Chapter 2.

Sequoia [FHK+06] approaches development for Cell by treating the processor as a memory=hierarchy configuration. CellSs [BPBL06] is a programming model for the Cell architecture from Barcelona Supercomputing Centre. Similar to Sequoia, CellSs annotations to C programs specify a task and its arguments intended for execution on the SPEs. In Sequoia and CellSs, the working set of a task is specified by intent qualifiers (such as *in*, *out* and *inout*) and dimensions for each input and output dataset. Explicitly defining working sets allows for optimisation (e.g. overlapping transfers between distinct memory modules with computation or eliminating transfers within the same memory module). However, the programmer has to tailor a task using parameters (specified with the **tunable** keyword in Sequoia), so that its working set fits into local memory. Using special blocking primitives on data can clutter code and obscure relationships between the working sets of sibling tasks.

RapidMind [MD06] and PeakStream both enable development for Cell using a stream programming methodology. Cell shows promise for scientific computing [WSO+06] and offers good performance on ray tracing [BWSF06], FFT [GC05] and various other algorithms demonstrated by IBM and Mercury Computer Systems.

In Sieve C++ [LMR07][Cod], a C++ extension from Codeplay Software, the programmer can place a code fragment inside a *sieve scope*—a new lexical scope prefixed with the **sieve** keyword—thereby instructing the compiler to *delay* writes to memory locations defined outside of the scope (global memory), and apply them *in order* on exit from the scope. The semantics of sieve scopes can be considered as generalising to composite statements the semantics of the Fortran 90 single-statement vector assignments [AK02]. This semantics, named call-by-value-delay-result (CBVDR) [LMR07], disallows flow dependences and preserves name dependences on data in global memory, and by reducing the need for dependence analysis to data in local memory makes C++ code more amenable to automatic parallelisation.

3.4 Memory hierarchy programming

To complement stream programming, another approach to hide the discrepancies in performance throughout a memory hierarchy is to concentrate on programming in such a way as to match the

layout of the hierarchy. This approach involves blocking data to fit into small, high-speed levels of the hierarchy and developing prefetching techniques to hide memory system latency.

3.4.1 Explicit memory hierarchy programming

The *Sequoia* [FHK+06] project from Stanford University enables explicit programming of a system's memory hierarchy whereby the application is developed in a generalised fashion and adaptation to a specific hierarchy is performed by the runtime system using associated metadata that describes the architecture to be mapped to. A program can be presented as a task hierarchy where the number of nodes in the hierarchy depends on the size of the problem. Final *leaf* tasks perform low level compilation and *inner* tasks control the execution of leaves. Data motion is explicit and the only way to pass data between nodes is through parameters to tasks. Copy elimination [KPR+07] methods are applied in the compiler to reduce overhead where possible. The explicit copies allow for isolation of execution such that synchronisation primitives in a given task are not necessary. Mappings of tasks onto architectures specify the number of layers in the hierarchy (inner tasks can be repeated to fill these layers) and the sizes at each layer. For best performance, leaf nodes can be optimised using architecture specific (e.g. SSE) code.

The work on *Sequoia* builds on earlier work by Alpern et al. on *parallel memory hierarchies* [AC93], *space-limited procedures* [ACF95] and the later work by Gatlin and Carter on the *Architecture Cognizant Divide and Conquer* (ACDC) compiler [GC99]. The *space limited procedures* work involves writing a generic program with different variants having differing performance characteristics, and specialising the generic program to an idealised model of the host computer. The model used is based on the parallel memory hierarchy work which is based around procedures that call other procedures in a tree structure, where each lower level of the tree must occupy a smaller memory footprint than its parent. *ACDC* attempts implementing divide-and-conquer algorithms with a variant policy that defines, at each level of the division, which variant to choose. *ACDC* generalises some of the techniques used in *FFTW*, which were in that case specific to FFT. The authors show that algorithms such as FFT, which has strides that can easily cause cache thrashing, benefit greatly from an architecture-aware approach, where algorithms like merge sort where the data is contiguous work fine with an architecture-oblivious approach. The work uses the notion of *isolator variants*, which guarantee that variants chosen below the isolator do not affect those chosen above, and use a dynamic programming algorithm around these.

Chapel [CCZ04, CCZ07] is a language designed to ease the burden of parallel programming. It is designed to be very general, not restricting programmers in the way that its predecessor ZPL [CCDS04, CLLS98] might and while maintaining the flexibility of MPI. Chapel supports data parallelism, task parallelism and concurrent programming; or combinations of the three. It is also designed to include many of the features of modern languages such as Java, that are lacking for a developer working on high performance software in C or Fortran. Data locality can be specified using the concept of a *locale* in much the way it is in programmed-global-address-space languages such as co-array Fortran [NR05]. A unit of a parallel architecture capable of performing computation and with uniform access to a machine's memory is the basic requirement. Such locales can be structured to better support the algorithm.

Cilk [Joe96, BJK+96] approaches the problem by separating a program into C-like tasks that spawn and synchronize threads representing sub-tasks. The runtime system's job is to schedule the computation for efficient execution. KeLP [FBK98] (which led to Chombo [CBK+]) supported runtime

parallelism and had explicit regions, in fact a "region calculus". However their regions represent partitions of iteration and data spaces, whereas in this work we represent the mapping between points in the iteration space and memory locations.

3.4.2 Optimising for memory hierarchies

An alternative approach to directly programming memory hierarchies is to transform the code from a naive form to a hierarchy-matching execution pattern. There are many different optimising transformations available for compilers [BGS94], but few are used effectively in standard compilers, leading to the development of languages and methods intended to assist this process.

The *Alpha* language [VMQ91] is intended for synthesis of systolic algorithms, using recurrence relations as introduced by Karp, Miller and Winograd [KMW67]. Alpha is a functional single-assignment language where execution order is inferred from dependencies. Alpha works on variables indexed on points of a convex set. Transformations into a space-time iteration space can be performed using unimodular affine mappings.

X [DBR+05] is a language that provides the facility to annotate constructs with transformations to apply. These parameters can specify loop unrolling, tile size and similar features to be applied by a translator into a high level language to be compiled. In this way, X incorporates some of the features of library generators and indeed is intended to be used as an intermediate language by such systems. The language uses *#pragma* constructs to notate transformations in C code which can be applied both to loops (interchange, strip mining etc) and statements (split- and shift-enabling notation of software pipelining). Performance results obtained suggest that *X* can generate better-performing code than *ATLAS*, assuming efficient copy routines are used where necessary.

Fursin's work on iterative compilation in compilers attempts to extend techniques used in projects like *ATLAS* to more general computation [FC07] while also increasing the interactivity of the process using a set of transformation files to allow communication between the compiler and external tools.

CUDA-lite [UBLmH08] uses annotations to generate data movement code for CUDA. The aim in their work is to assume the programmer has decided upon a parallelism and correct GPU-blocking of the code for reasonable performance, but wishes the CUDA-lite system to generate appropriate data movement for the various input and output arrays used by the kernel. The simple annotations ease part of the programming burden, but still leave CUDA as a low-level parallel-programming model.

Taskgraph [BHKM03] is a library designed to allow specialisation of components according to parameters or runtime information. *Taskgraph* represents code as structures within the C++ program and makes use of the *SUIF* compiler framework to perform transformations on an internal representation of this code. The work shows how large performance improvements can be obtained through runtime specialisation of code (such as specialising to a given convolution filter) even when the compilation overhead is taken into account, assuming the dataset is large enough. Taskgraph supports backends for Cell, CUDA and OpenMP.

While memory-hierarchy optimisation can be performed on a level-by-level basis using cost functions, Mitchell et al. [MCFH97] point out that various levels of the memory hierarchy can interact. Cost functions that minimise a variable for a single level of the hierarchy can lead to a globally sub-optimal solution. For example, optimal cache tiling can lead to low ILP and vice versa. They propose cost functions that deal with multiple levels of the hierarchy in concert.

3.4.3 Matrix multiplication

A common example for memory-hierarchy optimisation is matrix multiplication and there is a vast amount of literature available. Many vendor libraries are available that highly optimise the computation for a given architecture such as Intel's Math Kernel Library. These libraries obtain very high performance by utilising the SIMD extensions of the architecture and using very fine-grained knowledge of the pros and cons of various instruction scheduling approaches. Without fine tuning kernels by hand, libraries like *ATLAS* [WP05] [atl09] aim to achieve high performance through tuning of a set of parameters including tile sizes, unroll factors and instruction scheduling latency. Yotov et. al [YLR+05] discuss these parameters and propose that *ATLAS*'s empirical search is in fact unnecessary. They argue that similar levels of performance cam be obtained by using model-based solutions based on parameters such as the cache size to generate high performance loop nests for a given architecture. Further work in this area by Epshteyn et al. [EGD+05] proposes that a hybrid of the empirical and model based approaches can achieve better performance than either alone. Where Yotov's model considers the optimal size for blocking the multiplication loop nest to be based on an inequality in terms of the level 1 cache size and line size, Epshteyn's model extends this to compute the optimum level 2 blocking size, as this is more efficient on many architectures. To extend the model effectively for the level 2 cache a more conservative approach must be taken that takes into account the effect of conflict misses. They show that in some cases the model alone is not accurate, particularly in the case of the L2 blocking factor, and that the addition of a level of empirical search, directed by the model, can achieve higher performance overall.

Goto et al. [GvdG02] discuss how translation look-aside buffer (TLB) misses can become the most important performance differentiator during computation of a matrix multiplication. The argument is that much of the overhead of processing comes from initiating a stream of data from the level 2 cache, and that a large part of this overhead is due to TLB misses. Misses in the TLB differ from cache misses in that it is possible to prefetch to avoid stalling on cache misses which can then be covered by executing other instructions, but the processor will stall on TLB misses and hence prefetching cannot mask the miss. One solution to the problem is the packing of data into contiguous memory, a technique that is often used in matrix multiplication implementations, where the overhead from copying is outweighed by the performance advantages of reducing TLB misses. In [GG07] this work is extended to describe, in a more general sense, how high performance matrix multiplication can be achieved. These results are more through theory than experimentation, which is a good fit with Yotov's conclusions. In addition the paper presents Goto's work on the Goto BLAS implementation, which claims to be the fastest available [Got].

Li et al. [LG05] note that ATLAS' performance lags behind vendor libraries which are carefully hand optimised. The goal of their work is to attempt to bridge the gap by using recursive matrix layouts to place blocks in consecutive memory locations and to focus the search on levels of tiling and tile sizes. Referencing Yotov et al. [YLR+05] they note that with a single tiling level a model can predict the best tile size and almost match ATLAS for performance, however for multiple levels of tiling the size of the matrices themselves is also important. This approach builds on work on recursive approaches by Chatterjee et al. [CLPT02] who demonstrated that recursive layouts until the data fits in the cache (but no lower) can beat normal layouts of data. Their results suggest that on many systems they can achieve higher performance than ATLAS by using a classifier-learning system based around a set of condition/action rules to choose the optimal matrix partition. One important point of their work is that they compare to ATLAS as a code generator only with no hand coded kernels to make the test fair.

3.5 Skeletons

Related to techniques for programming for memory hierarchies and parallel architectures is the idea of skeletons [DFH+93]. A skeleton captures a basic algorithm structure that is common to a set of different applications. The skeleton can be efficiently mapped to a specific machine and then we can program the algorithms to the skeleton, and take advantage of the performance tuning performed for the given skeleton on the machine in question. Program transformations are also made more flexible by specifying transformations from one skeleton into another. If we have mapped an algorithm onto skeleton *A*, and have a mapping from skeleton *A* to skeleton *B*, then we can convert the algorithm to work in the skeleton *B* form, offering wider scope for optimisations such as fusion.

The skeleton approach offers the additional advantage of easing the modelling of performance by associating each skeleton/machine pair with a performance model allowing accurate estimation of the performance of the skeleton on that machine. This then allows accurate modelling of performance of the algorithm on the machine, via the chosen skeleton.

Following a similar loose principle, recent work by Solar-Lezama et al. on sketching [SLAT+07] aims to automate the optimisation of simple computation kernels. Rather than only declaratively specifying details of the current implementation sketching supports a rough definition of an optimised implementation and attempts to search for a series of transformations to convert one to the other.

3.6 Libraries and library generators

When a particular computation has to be performed regularly in a wide range of applications, concentration of the computation in a single library makes more sense than having each application writer redo the work. For this reason, hardware vendors often write libraries to support high-performance execution of these common kernels on their hardware. In addition to these vendor libraries and to reduce the time spent updating such libraries various high performance library projects have been developed.

FFTW [FJ05][FJ98] aims to perform fast Fourier transforms as efficiently as possible on a given architecture. It is an adaptive library and uses a combination of code generation, for optimal small transforms, and runtime composition of transform approaches to divide large transforms efficiently at runtime on a given architecture. The combination of these features allows FFTW to rival vendor libraries with a high degree of success.

ATLAS [WPD01][WP05][atl09], and earlier work on *PHiPAC* [BACD97] aim to optimise matrix-matrix multiplication for arbitrary architectures. The approach taken in both cases is one of empirical search, although the two projects are slightly different in that *PHiPAC* takes a more holistic approach than *ATLAS*, which highly optimises small kernels and maintains a more general system for executing those kernels on blocks of the larger problem. An *ATLAS* kernel is optimized around a set of parameters configuring block sizes, loop unrolling, arithmetic and load scheduling and generates general square matrix multiplication kernels, as well as cleanup code for dealing with edge cases. In addition to the generated code, *ATLAS* supports submitted kernels that can be performance-tested by the installer on the architecture and, if satisfactory, can be selected and used.

The *Broadway* [GL05] compiler uses a set of annotations, much like pre- and post conditions used on methods, to define side effects of library calls externally to the compiled code. While not a library

itself, this allows the compiler to perform optimisations such as code hoisting based on dependence-analysis information about a library. *Broadway* is implemented as a source-to-source translator, that performs data-flow analysis and code transformations based on the analysis. The annotation language also allows specification of rules such as *if matrix A is 0, reduce multiplication to set-zero operation* which can remove operations if high level operations are inlined and contain lower-level operations. The work is applied to a PLAPACK library averaging around 40 annotations per routine (ranging as high as 200).

SPIRAL [PMJ+05] generates highly-optimised library routines for performing digital signal processing (DSP) algorithms, or more generally linear transforms. *SPIRAL* is designed to work on algorithms that can be mathematically decomposed into a set of expressions written internally in a *signal processing language*. The most obvious work on the *SPIRAL* project is devoted to FFT computation, but there is more recent work on the discrete wavelet transform (DWT) [GPM04], useful for compression algorithms like JPEG2000 and the *SPIRAL* team has recently been investigating non-linear transforms such as matrix multiplication. *SPIRAL* has multiple output forms allowing it to generate efficient CPU code as well as Verilog for FPGAs [FHPM02].

STAPL [AJR+03], the Standard Template Active Parallel Library, is an attempt to ease the transition to parallel programming through the use of interfaces, containers and algorithms that are similar to, and supersets of, versions in the C++ Standard Template Library (STL). STAPL supports the standard STL functionality, guaranteeing serial consistency on compatible functions, but in addition supports the *pAlgorithm*, *pContainer* and *pRange*. *pAlgorithm* is a superset of STL algorithms with enhanced semantics or that do not exist in the STL at all. The *pContainer* contains data in the same way as an STL container, but allows semi-random access to its elements, as necessary for parallel programming. In addition, standard STL iterators are supported, maintaining backward compatibility with the STL. The third feature of STAPL is the *pRange*, which allows random access to its subranges, but in a final subrange iteration must occur in iterator order. Through pRanges STAPL supports hierarchical parallelism, claimed to be an advantage over models such as NESL [Ble93]. Like STAPL, Intel's Thread Building Blocks [CM08] is a template based library that supports parallel algorithms and containers designed to ease the development of multi-threaded applications.

Also from Intel, the Ct [Ghu07] system is another C++ library approach to supporting parallel programming. Ct uses operator overloading on collection classes to build complicated parallel execution plans in a functional style, with support for aggressive reordering of operations. Code is generated and dynamically compiled from the execution plans to improve platform support and customisation.

3.7 Runtime code generation and compilation

Runtime code compilation aims to compile code in such a way that it is optimised for a specific runtime instance of an architecture, be that a specific CPU model, a specific variant of a GPU or similar variations. Runtime compilation is commonly used by GPU vendors to allow compilation for a specific GPU architecture [MGAK03] [GLS] without fixing the low-level binary format across the entire range of devices. In the GPU case it is common that runtime compilation only occurs from a low level assembly-like intermediate representation having had the high-level parsing and loop-structure optimisation performed at compile time, however it is often also the case that the full high-level language is contained as a string within the controlling source language and is directly passed to the compiler at runtime. This latter case offers a chance to improve readability of the code,

having all the executable structures in one place, however the downside is that type checking is hard to perform on this code at compile time and much safety checking must then be moved to runtime.

Taskgraph [BHKM03] provides facilities to perform loop transformations. In addition, as the data structures are generated at runtime, *Taskgraph* facilitates runtime code generation and tuning. The results show that in certain cases, such as specialising code to a specific convolution filter, even with the compilation overhead being taken into account with a large enough dataset large performance improvements can be made.

Multi-stage programming (MSP) [Tah04] [TS97] takes the notion of runtime code generation and integrates it fully into a language. Part of the aim of the integration is that ensuring type safety of the generated code becomes easier. The type system can be fully integrated into the language, and can cover both the high level code, and the code that is generated by the high level code. *MetaOCaml*[CTHL03] is an example of a language supporting multi-stage programming. MetaOCaml is limited to OCaml code within OCaml code, and hence the techniques used do not necessarily generalise to other languages. However, work has also been performed on implicitly heterogeneous versions of the language, such that a simple subset of OCaml is generated and then translated into various different low-level languages. This process is termed "Offshoring".

MetaOCaml's offshoring process and the approach used for GPU languages such as GLSL [GLS] are very different. Whereas *MetaOCaml* insists you program your multi-stage program in *OCaml*, and is then capable of generating code in a lower level language for high-performance compilation, the GPU approach offers the chance to program a specific language to generate code for the GPU, and compile this at runtime from any source language that has OpenGL support. The flexibility offered by this heterogeneous approach has advantages, however the clear disadvantage is that the generator does not guarantee type correctness as full type checking of the GPU code can only be performed at run time. As a result, the developer has less confidence of its correctness than if the generator itself is checked.

3.8 Combinations of components

Combonent-based programming is related to execution kernels but with more emphasis on programming to black-box interfaces. Combining sets of components offers scope for cross-component optimisations. Modern compilers are capable of inter-procedural optimisation, which can provide performance benefits, at the expense of increased compile time. Optimisation across components is arguably a step up from that. Cross-component optimisation comes in two main variations, data placement and computation optimisation.

The *Sequoia* [KPR+07] compiler attempts to perform copy elimination between components to reduce the amount of time spent copying data, but does not perform a great amount of inter-component optimisation in terms of computation. Budd et al. [Bud88] discussed methods for compiling languages based on composition of high level operators. The context in this case was functional languages and used an intermediate representation allowing a set of basic transformations to be performed. Their optimisations reduced the composition to being smaller than the sum total of the components. Their approach is supposed to be applicable to any language in which the "primary method of program construction is composition". Beckmann et al. [BJK02] look at data placement strategies when dealing with component compositions in parallel algorithms They note that inter-procedural analysis can

attempt to solve the same problem, but at the expense of breaking optimisations of highly tuned components, and that a balance should be found.

Ashby et. al. [AKO] discuss cross-component optimisation in terms of Aldor, an ML-like language and the *FOAM* intermediate representation. Their results include fusing of basic BLAS level 1 routines, written in Aldor, and compare with *ATLAS* (localised tuning as distinct from cross-component tuning) and against coding the entire problem directly in Fortran with encouraging results.

Knit [RFS+00] is a component definition and linking language for systems code. It aims to improve performance and reduce errors in combinations of components, primarily from standard component kits. The authors argue that traditional compilers and linkers provide poor support for components because they are designed for library-based software. Compiled objects then declare services within a global space of names, requiring a specific name use in the client for linking to operate. Knit supports programmer-directed linking of components with lower overhead than would be required by a programmer using COM interfaces.

ICENI [FMM+02] is a middle-ware layer for grid systems that supports components with metadata enabling intelligent scheduling of execution. The system supports performance models based on the content of the metadata to guide the choice of implementations and to map an efficient execution to a set of grid resources given execution time or cost minimisation requirements.

Component models can support adaptivity and adaptive components show benefits in both embedded systems [MYBC03] and more generally in distributed systems (see for example [BGT08]).

Dowling and Cahill [DC01] offer a useful framework, emphasising the importance of separating adaptational from computational code. Recent work on the Common Component Architecture (CCA) looks at composing, substituting and reconfiguring components during application execution [MRA+06].

3.9 Predictability of execution time

One of the aspects of *ACDC* [GC99] is the concept of *isolator variants*. The concept of the isolator variant is to separate the hierarchy such that optimisations above a certain point do not affect the changes that can be made below that point. This is one aspect of predictability. In general, optimisations can cause dependencies such that other changes that might work given one situation, work considerably less well in others. This can be clearly seen in *ATLAS*, where the code structure chosen for one architecture can be radically different from that chosen for another. For real time systems, particularly, worst-case execution time is important to consider. Zhang et al. [ZBN93] note that estimations can be pessimistic if features such as cache and pipelining are not taken into account and Nilsen et al. [NR95] extend modelling to more modern processors, attempting to avoid the absolute worst case of having every instruction miss the cache and making a more realistic worst case estimate. Puschner's editorial [PB00] details a large selection of work on this subject. Gustafsson [Gus02] discusses the application of worst case execution time prediction techniques to object-oriented programs. He points out that while garbage-collected languages like Smalltalk and Java cause problems due to the lack of predictability of the garbage collection, non-garbage-collected languages such as C++ cause similar predictability issues when including the possibility of memory leaks. In real-time systems the contract between components in a component-based system includes timing information, and that components have to satisfy these requirements to fulfill the contract [GM02]. Changing combinations of these components changes the contractual relationships.

Synchronous programming languages offer one approach to predicting execution time. *Esterel* [Ber00] and Lustre [CPHP87] are two examples of synchronous data-flow programming languages. Edwards et al. show that *Esterel* can be compiled to sequential code [Edw00] allowing static scheduling of concurrency for predictable performance, aimed at embedded systems.

Virtualisation technology, such as Xen [BDF+03], increases the flexibility with which hardware can be used and thus adds another layer of complexity to predicting performance. This is particularly true where applications can be dynamically migrated from one server to another over the virtualisation system [BD02]. This ability to migrate also motivates a self-tuning approach: components that are moved to a different virtual server could easily benefit from re-optimisation to context.

3.10 Conclusions resulting from the review

Given the performance advantages from carefully tuning matrix multiplication to the memory hierarchy, it is clear that programming for the hierarchy is advantageous. There are three main approaches to doing this: explicit programming, as in *Sequoia*; explicit transformations, as in *X*; or library generators such as *ATLAS*. It could be argued that the last two options are the same but in the library generator a wrapper is applied to find the optimum structure and can only operate on the pre-arranged code; *X*-style transformations can be applied to arbitrary code. The Sequoia approach would need a similar structure to fine tune its tile sizes and so on. Sequoia's hierarchical structuring allows it to manage the recursion of computation into large data sets effectively, but it is not clear that its applicability is very wide-ranging.

Code transformation approaches such as *X* appear to perform comparably to code generators such as *ATLAS*, assuming we exclude hand-coded kernels. As a result it seems reasonable to apply code-transformation approaches to components in our work and expect to achieve a respectable level of performance. Care must of course be taken to ensure that any time dependent aspects of the component set are dealt with correctly when performing cross-component optimisations. Work on optimisation of code for GPUs and the Cell shows that these platforms are amenable to such transformations as well as more commonplace CPUs with caching, programmable memory and TLB issues coming into play. There is then no reason why, using a code-transformation approach, code cannot be transformed and optimised onto these architectures as well as any other, although vectorisation in these cases tends to become more important and so the challenge of integrating fine-grained vector-style parallelism into the transformation framework is an important challenge.

Component metadata, such as that used for architectural mapping in Sequoia, can contain information from a wide range of possibilities concerning resource requirements and demands for fulfillment. This offers wide scope for annotating a component post-optimisation to enable the use of variously optimised components in different situations. Clearly there is scope for research in the generation of that metadata. Runtime code generation as offered by Taskgraph and the multi-level programming work offers the possibility of adapting components as changes occur to resource availability to flexibly generate new, and newly annotated components to fit the situation the system now finds itself in. Components can not only have metadata generated based on new mappings, but also re-map the component based on changes to the metadata, which can be updated to request tighter or looser time demands and so on.

3.11 Summary

In this review we have discussed many aspects of high performance software and targeting esoteric hardware. Many of the concepts we have seen allow the targeting of decoupled hardware, or small decoupled aspects of hardware, more easily. Throughout the rest of this thesis we will see how stream programming and metadata can take us closer towards the goal of efficient development for esoteric architectures.

Chapter 4

Stream programming of GPUs

4.1 Introduction

Stream programming (see Section 3.2) is a solution to development for parallel architectures that offers benefits in terms of ease of extraction of parallelism. When writing a stream application we think in terms of the computation applied to a single element, and the computation will be applied repeatedly to each element in the stream. In this chapter we propose a GPU programming adaptation of a stream compiler originally designed for developing for FPGAs. In addition we briefly discuss an application to programming for the PS2. The goal of this chapter is to give some idea of the limitations of such a low-level stream programming methodology. We use this stream programming approach to lead on to a more developed programming methodology in the following chapter.

4.2 A Stream Compiler

A Stream Compiler (ASC) is a compiler that generates stream architectures for FPGAs. ASC is a library based on C++ meta-programming [Men02]. Variables of various types can be created and connected efficiently using standard C++ operators. The connected nodes represented by variables form a data-flow graph. FIFOs of the correct lengths are inserted automatically to balance the data-flow pipeline. Behavioral synthesis, as used in tools like the work by Venkataramani et al. [VBCG04], is prone to difficulty. ASC allows a direct approach to hardware design letting the programmer directly control the implementation. Optimisations can be specified by controlling bit-widths, amount of pipelining on individual operations and allowing for area/throughput tradeoffs. The programming burden of adding these optimisations is intended to be small [MPHL03]. We can see a simple example of ASC code in Figure 4.1.

4.3 The GPU as an general purpose accelerator

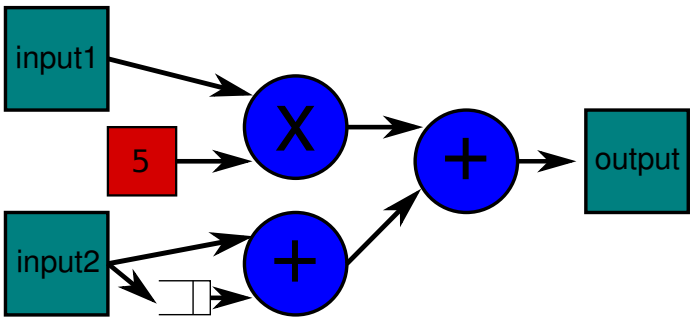
The Graphics Processing Unit is a vital component of a modern computer system. This acceleration hardware is dedicated to offloading the computation necessary to render and display complicated

```
int input1[ SIZE ];
int input2[ SIZE ];
int temp[ SIZE ];
int output[ SIZE ];

for( int i = 0; i < SIZE; ++i )
{
  temp[ i ] = input2[ i ] + input2[ i - 1 ];
  output[ i ] = input1[ i ] +
    5*temp[i];
}
```

(a) C code - ignoring the necessary range check.

```
STREAM_START;
// variables and bitwidths
HWint input1( IN, 32 );
HWint input2( IN, 32 );
HWint temp( TMP, 32 );
HWint output( OUT, 32 );

STREAM_LOOP( SIZE );
STREAM_OPTIMIZE = THROUGHPUT;

temp = input2 + prev( input2, 1 );
output = input1 + 5*temp;

STREAM_END;
```

(b) ASC code.



(c) Data flow representation.

Figure 4.1: C code with a simple loop compared with ASC code representing a hardware stream version of the same loop. Optimization mode setting is for maximum throughput. The *prev* operator looks back in the stream - in hardware using a FIFO buffer.

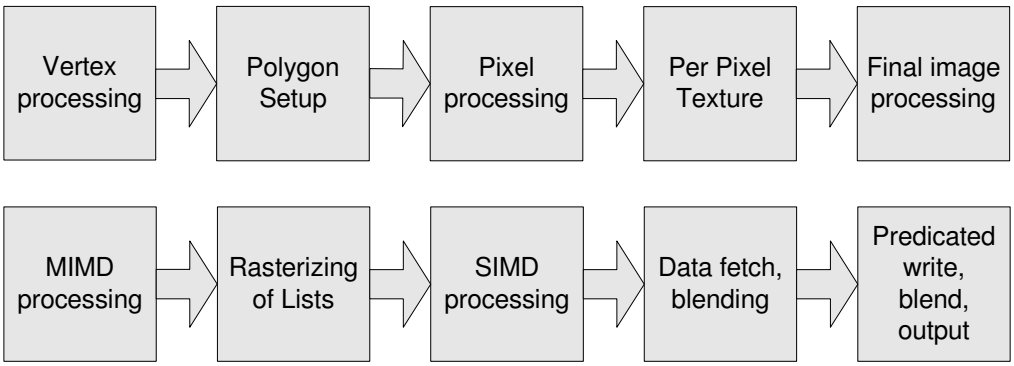


Figure 4.2: The stages of the 2004-era GPU pipeline as used for graphics (above) and general purpose processing (below).

3D scenes comprising thousands of polygons. This chapter considers the GPU as the 2004 vintage technology in use when the work was performed and predating CUDA. This graphics technology is adapted for general purpose computation through the graphics API and is still highly restricted in its graphics-oriented design.

The GPU in this context can be seen in the top half of Figure 4.2. The hardware:

- Processes a series of vertices.
- Sets up polygons made out of these vertices.
- Generates pixel sets from the polygons through a process of rasterisation.
- Applies colours from stored images known as textures to pixels in a fixed-function texturing system.
- Performs final processing over the image data.

We can see this computation in a general purpose sense in the bottom of Figure 4.2. The part of the pipeline primarily useful for generation purpose computation on GPUs is the pixel/fragment processing unit. The fragment processor is a powerful SIMD engine designed to process thousands of pixels efficiently. We can view the arrays of pixels as rectangular single-precision floating-point data buffers for general computation [Ven03].

4.4 Targeting the PS2

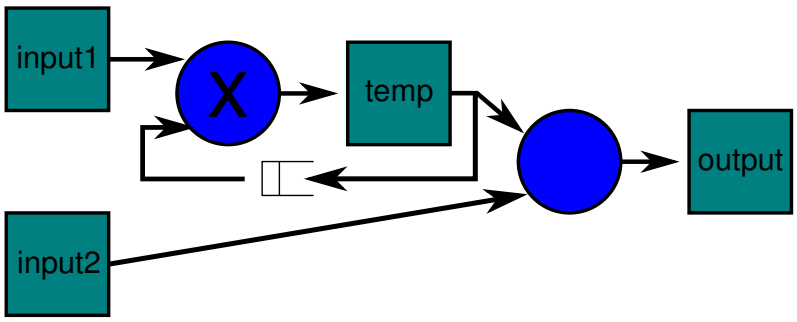
A portion of the work leading to results in this chapter, implemented by Paul Price, targeted Sony's PS2 console. The PS2's *Emotion Engine* processor combines a main processing core with a pair of vector processing units. This is a design that shows similarity to the later Cell processor that Sony part developed. The PS2 backend for ASC generates a single kernel using low-level vector assembly instructions to compute the entire data-flow from end to end on a single computation element. This solution is appropriate for the vector units due to their single-threaded nature.

```
int input1[ SIZE ];
int input2[ SIZE ];
int temp[ SIZE ];
int output[ SIZE ];

temp[0] = 1;

for( int i = 0; i < SIZE; ++i )
{
  temp[ i ] = input1[ i ] * temp[ i - 1 ];
  output[ i ] = temp[ i ] + input2[ i ];
  output[ i ] = input1[ i ] +
    5*input2[i];
}
```

(a) C code.

```
STREAM_START;
// variables and bitwidths
HWint input1( IN, 32 );
HWint input2( IN, 32 );
HWint temp( REGISTER, 32 );
HWint output( OUT, 32 );
STREAM_LOOP( SIZE );

temp = 1;
output = temp + input2;
temp = input1 * temp;

STREAM_END;
```

(b) ASC code.



(c) Data flow representation.

Figure 4.3: Feedback in an ASC design. The register variable will be written once per stream iteration, but can be read any number of times offering a feedback structure. Constant initialisation ensures that the register has a starting value.

From the ASC data-flow representation the system generates an *abstract syntax tree* (AST) supporting both vector and scalar instructions that performs computations across short vectors of stream elements where possible. The streams are blocked into small chunks that will fit into local memory and results for the chunks generated appropriately.

4.5 Targeting the GPU

ASC is designed for FPGAs. In this technology a stream maps directly, with wires and true FIFOs representing connections between variable nodes in the data-flow graph, which are implemented as registers. In addition, an ASC data flow graph is capable of representing feedback using *REGISTER* variables (see Figure 4.1c “temp”. In a serial C implementation such a construction is not a problem. Equally, in an FPGA circuit where we are primarily concerned with pipeline parallelism, as implemented by ASC, implementation is straightforward. We see how this works in Figure 4.3.

The traditional design of the GPU, as standard at the time of this work, is a heavily graphics-oriented architecture. The hardware is designed to efficiently execute a large number of independent operations, using cached input data. This design maps efficiently to rendering a scene of polygons, each

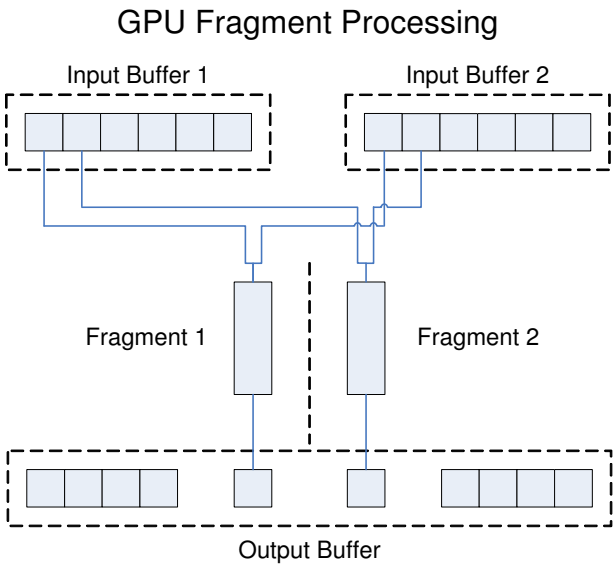


Figure 4.4: GPU fragments are processed independently, sharing randomly addressable inputs and with fixed location outputs.

one made of a large number of pixels. The pixels in a given polygon will generally share a set of input values in the form of textures, and generally the texture addressing will not vary in complicated ways such that the efficiency of cache use is high. Pixel, or fragment, computations cannot communicate, and at this stage of the technology also could not provide intermediate output values. Each executing fragment has a small set of output values that map directly into an output buffer, with no ability to perform scattered writes. We see this diagrammatically in Figure 4.4.

The implementation of registers, memories and data-flow cycles, which are common in finite-state-machine and data-flow style streaming computations are limited on GPUs by the lack of inter-fragment communication. The GPU must be treated as a pure stream architecture, where each output can be computed based on input values alone and where processors cannot maintain state between stream items. Given this, feedback cycles must be removed entirely.

4.5.1 Repeated computation

We saw ASC's *prev* operator in Figure 4.1 representing a conceptual FIFO buffer in the stream. In the example in the figure we see this implemented fairly simply in C code. Indeed, the GPU could implement this similarly with a gather operation on its input data structures. However, in the case that the *prev* operator is applied to an intermediate value with more than one index, as seen in Figure 4.5.1 then we have a more complicated situation. As fragments cannot communicate, we cannot reuse the computed data. Instead, we are faced with two options:

Splitting the stream graph

In this case we can take all the nodes before the *prev* operator, that is before the point in the graph where stream indexing is performed, and treat that subgraph as a separate kernel as we can see in in Figure 4.5.1. The sub-graph's kernel generates an intermediate data set that will be used as input to the rest of the kernel, arbitrarily indexed as the indexing operators require.

The overhead in this case arises from the higher memory bandwidth and storage requirements needed to store and access the intermediate data set.

Duplicating computation

The alternative solution is to repeat the computation performed in the graph nodes preceding the indexing operator, such that the kernel's inputs are indexed multiple times. We can see this approach in Figure 4.5.1. This might improve access locality, depending on how the hardware schedules the operations and the length of the kernel, but leads to a larger amount of computation being performed overall, and as a result can perform less well under certain circumstances.

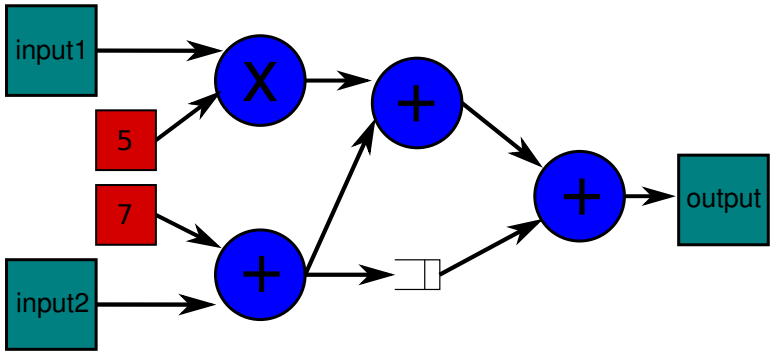
The decision to use one approach or another depends on the particular graph being considered. If the graph section reached through the indexing operator is large or many different indexes into the intermediate value are required then computing an intermediate buffer will perform better as the code size is reduced and the arithmetic overhead of recomputing becomes high. If the graph partition is small, with a small number of arithmetic operations, then recomputing offers a more efficient use of resources, performing better overall. In the results discussed in this chapter the better performing of the two options is used.

4.5.2 Compilation to the GPU

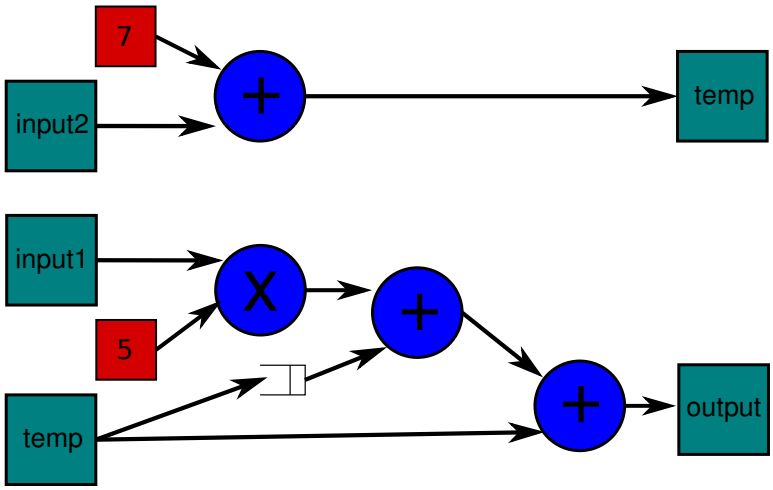
Compilation of ASC code to the GPU goes through a series of stages as in Figure 4.6. ASC programs represent the flow of data in an abstract stream processor. Execution of an ASC program such as that in Figure 4.7 generates a data flow graph. The ASC data flow graph is processed internally to generate a set of abstract syntax trees (ASTs) representing fragment programs necessary to represent the algorithm. Each fragment program is output as C-like code as in Figure 4.8 that is represented as a string in the generated application. This string is passed at runtime to the GLSL [GLS] compiler, which takes the C-like code of the GLSL kernel and compiles it for a specific GPU. Kernel-calling functionality is generated to use the string to correctly call a kernel, as well as generating the appropriate graphics geometry to ensure that the GPU executes the computation for the correct set of addresses.

Most intermediate nodes of the data flow graph, representing temporary stream variables, appear as registers in a fragment program. In each executing instance of a fragment program, generating a single output stream element, these registers take intermediate values of the ASC temporary variables. Each fragment program execution works from a given set of input buffers to an output buffer. A buffer of n elements represents a variable over n stream iterations.

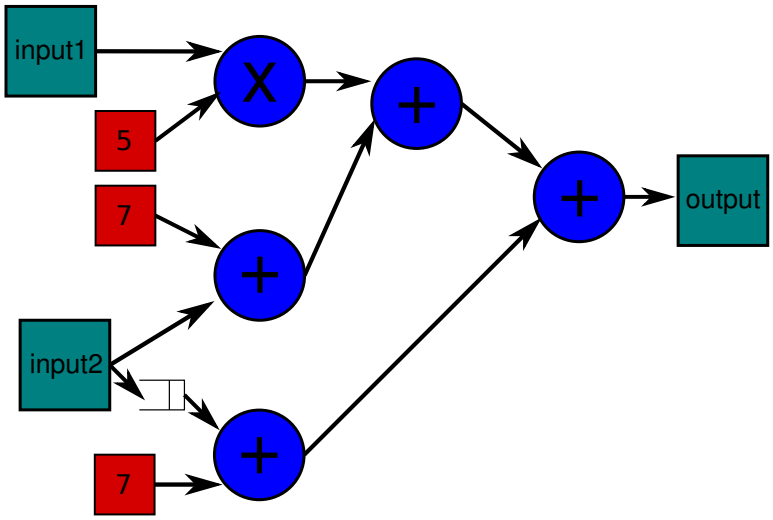
The implementation of registers, memories and data-flow cycles common in finite-state-machine descriptions is limited on the GPU by the lack of inter-fragment communication. The GPU must be treated as a pure stream architecture, where each output can be calculated based on input values alone. Due to the stream limitation, registers and memories are only converted from the ASC implementation if they are read only. Given the read-only restriction on registers and memories, feedback cycles are removed entirely. Delays or FIFOs, common in digital circuits and also in stream applications must be recreated either through additional code that reads from a different stream address, logically a different temporal offset, or through the use of intermediate buffers storing the same data item at various time points into which a subsequent kernel can address.



(a) A data flow graph with stream look-back.



(b) The split version of the graph. The temporary stream is generated in full, and can then be indexed arbitrarily straight from memory in the second part of the graph. The two graphs generate separate GPU kernels with a communicating array.



(c) Repeating the computation using earlier indexing into the input stream is an alternative. The down side is that the computation becomes more complicated.

Figure 4.5: A slightly more complicated data flow with a decision to be made about how to deal with the look back.

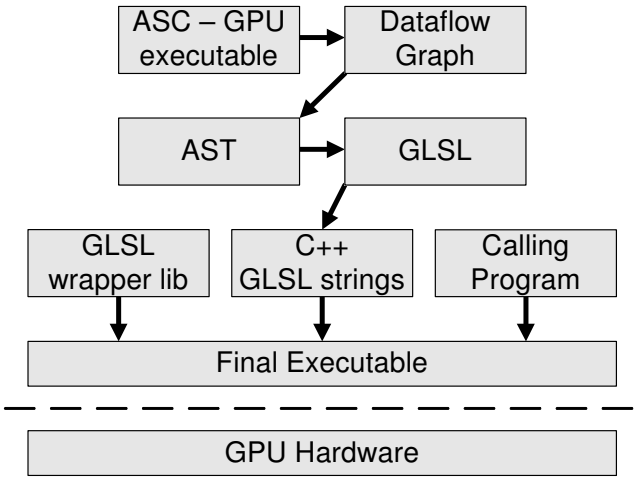


Figure 4.6: Stages of compilation from ASC to the GPU executable.

```
STREAM_START;
HWfloat inputvar(IN, FLOATFORMAT, "in");
HWfloat temporary(TMP, FLOATFORMAT, "tmp");
HWfloat intermediate(TMP, FLOATFORMAT, "tmp2");
HWfloat outputvar(OUT, FLOATFORMAT, "out");

STREAM_LOOP(40);

temporary = inputvar + prev(inputvar,2);
intermediate = temporary + prev(temporary,2);
outputvar = inputvar + prev(intermediate,3)
        + prev(temporary,4);
STREAM_END_GLSL;
```

Figure 4.7: An ASC program.

4.6 Applications

To demonstrate our approach and to investigate performance comparisons we look at three applications to accelerate: a Monte Carlo simulation, a Fast Fourier Transform and a simple weighted sum calculation. Since current GPUs are limited to single precision floating point we restrict ourselves to single precision to make the comparison fair, although the FPGA supports adaptable precision. In each case performance results of the raw implementations are be compared against each other, and also against an implementation of the same algorithm running on a fast Pentium 4 processor.

Monte Carlo simulation: Monte Carlo methods are algorithms employing random (or pseudo-random) numbers to solve computational problems. One example of the use is in area sampling: rather than sampling every point, random points spread evenly through the region are used. As a result information is provided that can be generalized across the region. In this case we implement a slight simplification of a Monte Carlo simulation originally intended for simulating the value of European call options: a typical financial application based on given parameters. The original asset price is specified and a set of randomly generated sequences of subsequent asset prices is generated over a given time-frame. The Monte Carlo simulation contains a static loop which maps well onto the GPU architecture.

```
string ks__temporary10 =
  "void main(uniform samplerRect in__inputvar1)\n"
  "{\n"
  "  vec4 _temporary10;\n"
  "  vec4 in__inputvar1_var_P0;\n"
  "  in__inputvar1_var_P0 =
     textureRect( in__inputvar1, vec2(gl_TexCoord[0].s, 0)).rgba;\n"
  "  vec4 in__inputvar1_var_P2;\n"
  "  in__inputvar1_var_P2.r = in__inputvar1_var_P0.b;\n"
  "  in__inputvar1_var_P2.g = in__inputvar1_var_P0.a;\n"
  "  in__inputvar1_var_P2.b =
     textureRect( in__inputvar1, vec2(gl_TexCoord[0].s + 1, 0)).r;\n"
  "  in__inputvar1_var_P2.a =
     textureRect( in__inputvar1, vec2(gl_TexCoord[0].s + 1, 0)).g;\n"
  "  _temporary10.rgba = (
     in__inputvar1_var_P2.rgba + in__inputvar1_var_P0.rgba);\n"
  "  gl_FragColor.rgba = _temporary10;\n"
  "}\n"; /* End kernel k__temporary10*/
```

Figure 4.8: Figure 4.7 output as GLSL.

FFT: Fourier transforms have many uses, particularly in audio and visual applications. In this case we look at an implementation of a radix-2 butterfly. This was originally an ASC FPGA example and has been implemented for both the GPU and PlayStation 2 largely to show that it is possible to do so with little or no work. It should be noted, however, that a full FFT pipeline is not implemented. Instead only the butterfly is accelerated and hence a high degree of inefficiency is present due to the data movement and rearrangement necessary to perform a full Fourier transform.

Weighted sum: The weighted-sum algorithm multiplies the last four values in the stream by constants, totals those values, and then totals the last four of those sums. The weighted sum is a simple calculation of a form seen in filtering algorithms. The algorithm makes use of the *prev* function that inserts a delay in hardware and uses intermediate buffers on the GPU as a result.

In the case of the Monte Carlo and FFT computations we can use ASC abstractions to easily optimize the FPGA implementation utilizing BlockRAMs to keep data on the chip. The Monte Carlo static loop prohibits the use of registers between logic elements, computing the inner loop between buffers repeatedly offers full scope for pipelining. The FFT requires multiple passes, in the naive implementation data reordering between passes is performed in software on the host computer. In the buffered version, intermediate data is stored and reordered on the accelerator.

4.7 Results

We show results of the Monte Carlo, FFT and weighted sum algorithms on our three target architectures. In addition we show execution times for the same algorithms written in C and running on a Pentium 4. GPU timings are performed on a Athlon 64 2000+ machine, FPGA tests on a 2 GHz Pentium 4 and Pentium results on a 3.2 GHz Pentium 4.

We use a Xilinx Virtex-II 6000 FPGA running on an ADM-XRCII PCI card and an NVIDIA 6800Ultra GPU in an AGP slot. We compile the ASC code for the FPGA and GPU using GCC 3.3, on the PS2 using GCC 2.95 (the maximum available for the architecture) and the Pentium code using Intel

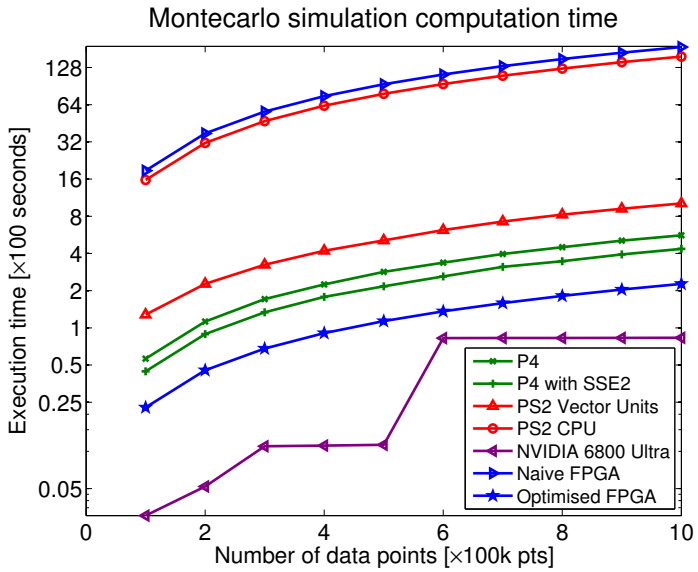


Figure 4.9: Monte Carlo simulation using a Pentium 4 3.2 GHz CPU using Intel's C compiler with -O3 optimization and full optimization including vectorization. We compare with the same execution running on an NVIDIA 6800 Ultra, the PS2's vector units, the PS2 CPU and a Xilinx Virtex2-6000-6 FPGA with and without on-chip buffering.

C++ version 9.0. The Intel compiler is used on the Pentium due to its support for vectorisation and SSE-2 extensions, and we can see a performance advantage from vectorisation in the graphs. Timing results use real transfer-to-hardware times measured by calls to the *gettimeofday* function, except for the optimized FFT and Monte Carlo which are based on cycle-accurate estimates. FPGA computations are optimized for throughput in all cases except the naive Monte Carlo. The GPU results use the preferred method of stream indexing operation, as discussed in Section 4.5.1.

Figure 4.9 and Figure 4.11 show the performance comparison between the architectures for the Monte Carlo and weighted-sum algorithms working on dataset sizes ranging from 100,000 data points to one million data points. Figure 4.10 uses a range of powers of two from 16 to 524288 for the FFT, however the optimized implementation cannot currently support more than 8192 points due to memory limitations. This small FFT could be used to calculate results for larger transforms.

The Monte Carlo simulation shows how well the GPU can perform when the algorithm is well matched to its architecture. The GPU execution is 3 times faster than the nearest competitor. Each executing fragment program of the GPU maps onto a single Monte Carlo simulation leading to as much parallelism as the GPU can offer. The FPGA Monte Carlo simulation shows how the static LOOP construct makes the FPGA implementation inefficient and some idea of how this can be rectified is shown by the buffered Monte Carlo (which is implemented in 24 bit precision due to area limitations which can be corrected with a larger FPGA). The circuit is unpipelined and clocks at 0.4 MHz against the pipelined version clocking at 34 MHz. The optimized FFT circuit only runs at around 50 MHz indicating that there is still room for improvement in ASC's circuit generation.

Results from the FFT show a wide performance range. The inefficient memory rearranging and high transfer time percentage on the GPU (shown in Figure 4.12) lead to low efficiency on block transfer architectures compared with the Pentium 4. The Pentium 4 performs the simulation 13 times faster than the GPU and 57 times faster than the unoptimized FPGA. FPGA performance is improved vastly by on-chip buffering. Current progress in this area limits us to 8192 points but this will improve with

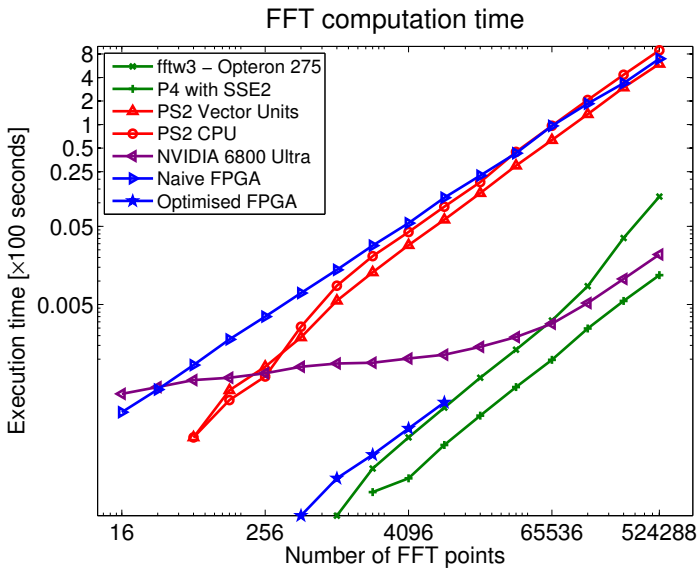


Figure 4.10: Radix-2 FFT using a Pentium 4 3.2 GHz CPU using Intel's C compiler version 9.0 with -O3 optimization and full optimization including vectorization. We compare with the same execution running on an NVIDIA 6800 Ultra, the PS2's vector units, the PS2 CPU and a Xilinx Virtex2-6000-6 FPGA with and without on-chip buffering.

time.

The weighted sum results show high performance for the SSE-2 optimized Pentium 4. The algorithm is largely register bound and hence highly efficient in a general purpose processor. The block transfers harm the performance of the FPGA and GPU greatly. PS2 vector units are closer to the CPU and are therefore less affected by data transfer times, and hence we achieve higher performance. We see that the Pentium 4 with SSE only performs 28% faster than the non-SSE Pentium, but 58 times faster than the FPGA.

4.8 Limitations in the ASC approach to programming GPUs

The work discussed in this section has described an approach to developing for both the GPU and for the PS2, using a data-flow-based stream programming paradigm. This model is easy to develop for, and shows clear benefits when creating data flow architectures in a similar fashion to implementing directly as digital circuits. One area where this style of stream programming can show benefits is in an architecture such as the Cell, where data can be streamed from one SPE to another. Many streaming programming models have been developed for this purpose, StreamIT [TKA02] and IBM's System/S are examples of such languages. In these models, stream nodes execute are more sophisticated programs with possibly complicated loop structures. The ASC model is rather more low level. The stream node is at the point of a single arithmetic operation, and chains of these operations are connected in a data flow graph. This low-level model makes kernel-level loops harder to deal with in a clean, readable way.

In addition, ASC's main FPGA backend is generative in nature. Low level FPGA circuits are instantiated directly from the C++ code, without much flexibility for working on an intermediate representation. The GPU and PS2 backends in this study inherit a certain amount of this inflexibility from the

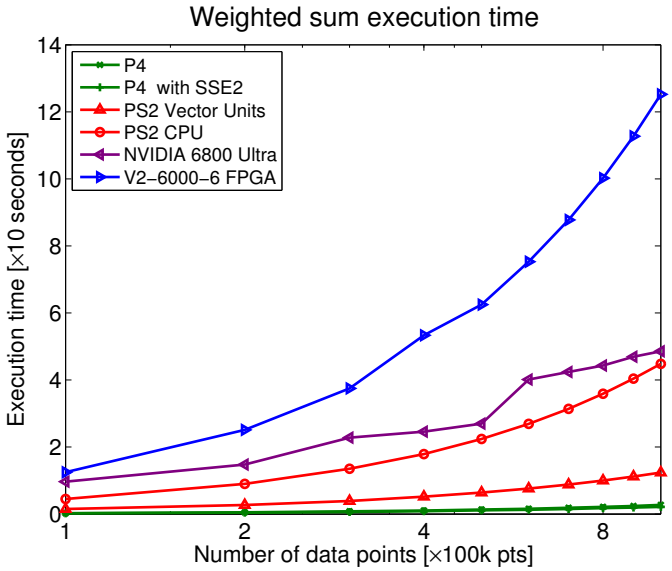


Figure 4.11: Weighted sum calculation using a Pentium 4 3.2 GHz CPU using Intel's C compiler with -O3 optimization and full optimization including vectorization. We compare with the same execution running on an NVIDIA 6800 Ultra, the PS2's vector units, the PS2 CPU and a Xilinx Virtex2-6000-6 FPGA.

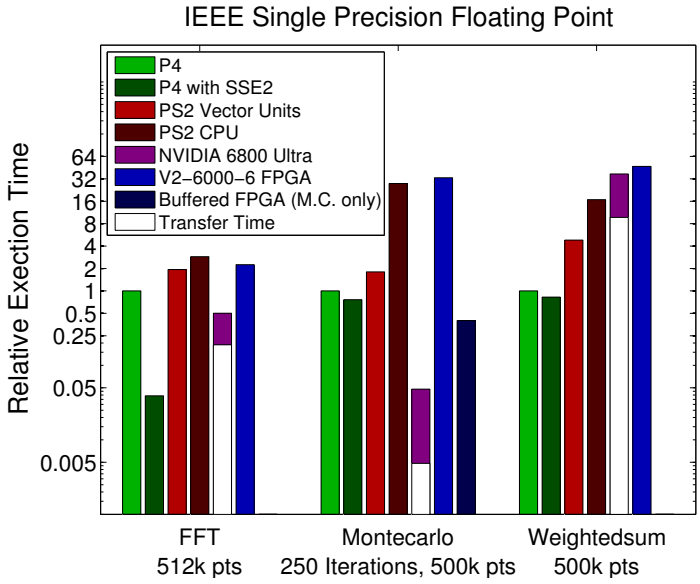


Figure 4.12: Direct comparison of the three examples including, where relevant, a breakdown of execution time and transfer overhead. The results are normalized for each application relative to the respective Pentium 4 non-SSE execution time. Buffered FPGA results are currently only available for the Monte Carlo at these dataset sizes.

original development of the meta-programming environment.

4.9 Summary

In this chapter we have discussed a method for programming GPU architectures and PlayStation 2 vector units using a single-source stream programming model. We have aimed to demonstrate that using a single source description, even if it does not achieve peak performance on any single architecture, improves productivity. Tuning a partial implementation is often easier than starting from scratch, but maintaining the tuning optimisations for multiple architectures can be a challenge in itself. We also saw that the ASC model suffers from particular general limitations as a programming model.

We can treat the stream programming model as a form of decoupling, but a very limited form. It is decoupling where:

- the iteration space is simple, and one-dimensional
- the iteration space is closely integrated with the layout and dimensions of the data
- the mapping to data is either the identity or the identity combined with simple and fixed FIFO-buffer-style mappings

In the next chapter we will develop the idea of indexed dependence metadata, which generalises the concept of stream programming as a parallel programming model. Later we will see how this metadata allows us to automate a wide range of optimisations.

Chapter 5

Indexed dependence metadata

5.1 Introduction

As we saw in Chapter 4, stream programming can ease the development problem for GPUs, as well as for other architectures such as FPGAs. In particular, the stream programming models give the compiler enough information to cleanly parallelise and generate data-movement operations, where this information might be hard to extract automatically from C-like code. As with any programming model, this is only true if the model maps cleanly to the algorithm in question. If the architecture and algorithm design both fit an element-wise data flow model, then the element-by-element approach seen in stream programming models is perfectly suited to this situation.

As we discussed in Section 3.2, we can see stream programming as:

Data-flow models
: Data-flow streaming models are centred around point-to-point communication and work very well on data that flows through a system.

Parallel models
: Parallel models treat streams as large vectors with complicated operations applied to each element in the vector.

The two concepts are related, but it is the parallel model that we wish to consider here. In terms of parallel programming, the data is "streamed" through a single task. Data that flows between tasks is an optional extra. Data in a stream can be operated on in parallel. However, by its nature the concept of a flow is a serial one, so to support parallel programming the stream model must be restricted slightly. In particular, a flowing stream has no problem with the concept of computational feedback: for example a streaming summation with an accumulation register.

We can remove the concept of feedback to break the serialisation requirement and ensure that streams can be executed with full parallelism. Once in this state, a stream provides an iteration space that:

- offers an element-wise mapping of the iteration space to data.

- removes the need to analyse the system for parallelism because communication between iteration space points is nonexistent by definition.
- allows partitioning of the stream because the parallelism structure is explicit.

The above are valuable provisions and there is no doubt that stream programming has been a useful programming model. Unfortunately, it suffers from specific shortcomings:

- Multi-dimensional iteration spaces do not cleanly fit the stream model. Other than when using simple reduction operations, it is difficult to map streams between data structures of different dimensionality. For example, mapping a one-dimensional stream to a two-dimensional data set.
- Multiple inputs and outputs must be related closely with matching access patterns. The iteration space is defined by the data structures and hence if there are multiple shapes of structure the iteration space is less clearly defined.
- Conceptually, each step in the stream's iteration space processes an input element and generates an output element. Reuse of data relies on sliding windows and logical FIFO buffers. More varied patterns of data use are harder to incorporate.

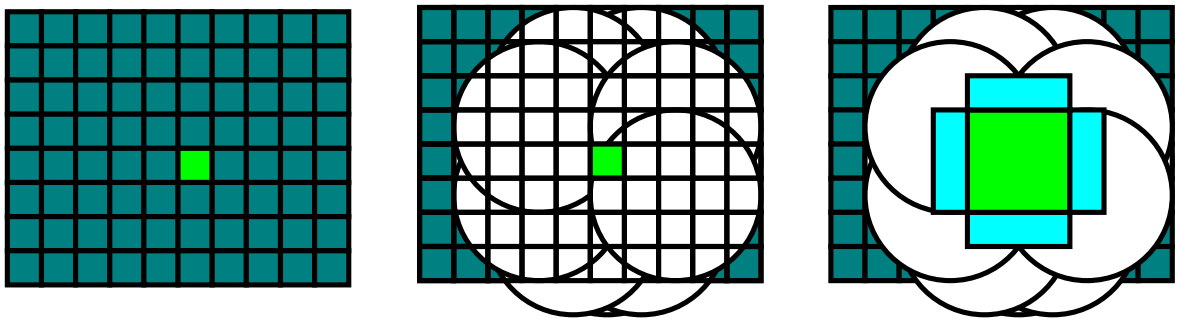
The essence of stream programming is decoupling. While there are limitations in the flexibility of iteration spaces and the mappings to data, the core idea is that the data elements available for a given iteration of a computation kernel are independent of the kernel's code. The kernel itself operates on the data elements provided to it, and is decoupled from access to the larger data structure. Streams that allow random access on read (*gather*) and, less often, streams that allow random access on write (*scatter*) are available in stream-based programming languages such as Brook [BFH+04]. However, these gather and scatter streams break the decoupling of data access patterns from execution and make data-movement optimisation more difficult.

5.1.1 A motivating example

While we can see that stream programming has limits on its flexibility of notation, there are cases where clearly defined declarative mappings from computation to data offer benefits. This can be the case if the mapping from computation to data is impossible to infer from the source code. Often this is true for plain C code.

Even when mappings from execution to data can be inferred from computation kernels, a lack of robustness in the inference process supports the aim of providing an explicit mapping. While one particular implementation may allow a compiler to infer parallelism and data mappings, a small change in the code can generate a situation in which the compiler can no longer infer this information or the compiler might change and no longer be capable of inferring the information from a given situation. Performance might drop considerably before and after the change with little consistency in methods to recover the former behaviour.

The *circular max filter* is an example where parallelism is difficult or impossible to infer from the original, C-level, kernel code. Conceptually, the filter expands a given pixel into a circular surrounding region and takes the maximum value of all the overlaid circles.



(a) Given a target pixel.

(b) A circular region of source pixels could expand to cover it.

(c) The resulting possible set of source pixels as a simple rectangle and additional regions.

Figure 5.1: The expanded pixel circles from which an output will be generated that might hit the target pixel fall into a circular region. The range of pixel values in this region is computable, but complicated enough to be difficult to infer from code. The more likely CPU algorithm would use sparse lists for the edge values rather than trigonometric calculations. As a result inferring information from the code structure is almost almost impossible.

While expanding circles represent the natural visualisation of the algorithm, parallel code implemented in that fashion would require a large number of atomic comparison/assignment operations incurring substantial overhead. For a realistic implementation on a parallel architecture we reverse the process, generating a single output pixel from a circular input region comprising all the possible pixels whose circles overlay the target pixel. This region is also a circle. The source regison is searched for a maximum value scanned for the maximum value that can be found, which is then output to the output pixel as appropriate. We can see the algorithm described in Figure 5.1.

When using the reverse process, searching a circular input region for the maximum value, we can optimise the algorithm as we see in Figure 5.1c. To read the input values efficiently the loop structure will use two passes: an inner rectangle with a simple loop nest supplemented by a list of edge pixels. The edge pixel set can either be obtained from a sparse pixel list, or accessed using a complicated nest of loops with trigonometric loop bounds. The set of edge pixels creates an indirection in the algorithm that makes analysis of the kernel code highly challenging, if not impossible.

As the developer of the kernel, we know the bounding region that we need to process. We can compute the x and y distances of input pixels from the centre point or output pixel using the circle radius. We can either define a circle mathematically or define a rectangle representing the maximum possible read distance. If we can represent this information to the compiler or run-time system, then accesses can be optimised without the need for analysis. The computation and data movement can be optimised as a result.

Stream programming goes some way towards this, but the data-driven stream programming approach makes representations of this kind of algorithm inflexible. While the maximum read distance could be represented as the stream access windows in the Brook language, the mathematical definition of the circular mapping would not be possible. Given a large enough circle the difference in efficiency would be substantial. When given access only to simple mappings such as these, the flexibility of the programmer is limited. He is unable to fully use his knowledge of the problem domain to describe the problem to a high level of efficiency. On some architectures this might not be an issue: on a cache-based architecture, for example, however large the region described in the stream only data

that is actually accessed will be moved into the cache. However, on other architectures the inefficiency of such an approximate mapping can become noticeable: on the Cell processor, for example, movement of data is a separate stage and for a large circle the amount of data copied unnecessarily would be significant. In addition, if we define the circle mathematically with an additional bounding region the runtime system and compiler have the flexibility to move only the data within the circle, or the entire bounding region, depending on how large the difference in efficiency will be on a given target architecture. Stream models do not offer representations with such fine grained flexibility of representation.

5.2 Our suggested solution: indexed dependence metadata

We propose *indexed dependence metadata* as a generalisation of stream programming. Indexed dependence metadata defines two features:

- An iteration space of some number of dimensions.
- A mapping from each point in that iteration space to a set of input and output data structures.

A given element in the iteration space is mapped to the set of input and output data elements the computation may require at that point in its execution. This mapping alone is enough to describe the dependence information for the iteration space, given a shared read/write data structure. In real implementations of indexed dependence metadata, as we shall see in Chapter 6 and Chapter 7, the addition of an execution ordering guarantee allows for a more robust and more efficiently analysed description.

Formally we see the metadata as a combination of execution and mapping information. Given a set of data elements D in some abstract input and output data structures the metadata in its simplest case satisfies the following definitions:

Definition 5.1 *Execute metadata $I \subset \mathbb{Z}^n$ is a finite, n-dimensional* iteration space*, for some $n > 0$;*

Definition 5.2 *Data mapping, or access, metadata for a kernel is a tuple $A = (D_r, D_w)$, where:*

- $M_r : I \to \mathcal{P}(D)$ *specifies the set of abstract data elements $D_r(i)$ that may be* read *on iteration $i \in I$;*
- $M_w : I \to \mathcal{P}(D)$ *specifies the set of abstract data elements $D_w(i)$ that may be* written *on iteration $i \in I$.*

Thus we can see that the a given point in an iteration space *indexes* its input and output data structures, or *dependencies*.

We can see a simple example in Figure 5.2. In this fashion we extend the decoupling ideal partially seen in stream programming. We fully decouple a computation defined as a kernel that executes over some iteration space from description of the data elements that that kernel will read. Memory lookups

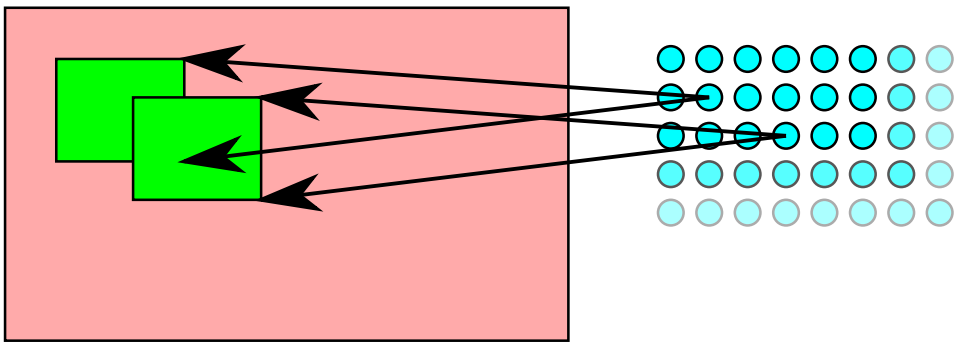


Figure 5.2: Indexed dependence metadata relating the elements of a simple iteration space to a rectangular data structure. Each point in the iteration has access to a rectangular set of data elements.

can optionally be performed in place or decoupled depending on the implementation, but the robustness that is necessary in ensuring that this decision can be made is provided by the programmer's provision of metadata for the kernel.

The limitations with stream programming's mappings are naturally solved in a dependence metadata model. The n-dimensional iteration space is mapped explicitly to m-dimensional data.

Indexed dependence metadata offers a flexible representation with the following properties:

- An iteration space of arbitrary dimensionality that maps arbitrarily to data: mapping from one number of dimensions to another is trivial.
- Iteration-space-driven, and hence computation-driven, execution plans. This is in contrast to the largely data-driven execution plans of stream programming or hierarchically tiled arrays, and removes the requirement for the inputs and outputs to relate closely to each other.
- Allows the programmer to pick algorithms and map the appropriate data structures to them, offering an efficient decoupling of kernel development from architectural mapping.

The metadata can be interpreted as guidance such that it is added to a computation kernel, but the kernel would execute perfectly well as code without the metadata. Alternatively, the kernel can be defined such that it only has access to the limited set of data elements that it might use, with no global data access, and hence relies on the metadata to execute correctly. While only the former might be seen as true metadata, the concept is the same in either approach.

Chapter 6 discusses components that can fit into either category, but where we have chosen to use the former method. The programmer is offered more flexibility at the risk that the kernel definition might not satisfy the metadata, with undefined consequences. The second option is used in Chapter 7 to enforce kernel behaviour that matches the input metadata.

In addition to specifying the execution and access plan for a given kernel, the metadata fulfills the function of describing a very specific interface to the kernel. Other kernels that pass data in to or out of the kernel in question know not only which data-sets are to be passed but also which iterations will be using the data. We will see where this is useful in Chapter 6.

Even in cases where analysis of code *might* be possible, indexed dependence metadata serves a purpose by robustly embedding information to remove the luck of analysis. It maintains the simple

element-wise kernels that are the primary benefit of parallel stream programming models, and yet increases the flexibility of mapping representation to improve the usefulness of the model.

We can represent certain forms of Skeleton, as discussed in Section 3.5, in metadata terms. The skeleton-inspired methods described by Cornwall et al. [CHK+09] use skeletal kernel descriptors to limit the set of memory access patterns to optimise to a small set of image processing benchmarks used in film post-production. The defined set is flexible enough to allow an artist wide scope for creating interesting effects and yet simple enough to allow automated optimisation of these effect kernels. These kernels can also be seen as a form of indexed dependence metadata and, while the work is more specific in its aims, the same principles apply.

5.3 Summary

We have introduced indexed dependence metadata as a generalisation of stream programming and given a hint as to why it is useful. In Chapter 6 and Chapter 7 we will see more concretely how the indexed dependence metadata concept can be used in software development. In particular we will demonstrate that the metadata can be used to

- Provide additional information that would be challenging or impossible to extract from a computation kernel.
- Enable optimisations that take account of this additional information. To this end we discuss a component programming model in Chapter 6 that uses programmer-supplied metadata to enhance inter-component data flow analysis.
- Simplify the development of high-performance data-movement code with little loss from automation. We discuss this in terms of a programming environment for the Cell processor in Chapter 7.

Chapter 6

Indexed dependence metadata in a component programming system

6.1 Introduction

Component-based programming consists of writing software entities to satisfy specified interfaces. Component models allow multiple component implementations to satisfy the same interface, offering flexibility on the choice of implementation for a particular problem or computing platform. However, treating components as black boxes described by their interfaces can limit the scope for optimisation. In particular, whilst individual components can be statically optimised when the component is defined, component compositions can only be optimised at the point of use. This is important both because the optimal set of optimisations for a component may change when placed in context with another component, and because there may be optimisations that can be performed across component interfaces that provide further benefits. To perform cross-component optimisation at composition time requires an element of dynamic optimisation that exploits context information.

Powerful but expensive inter-procedural compiler optimisations such as enabled by the polyhedral framework [PSC+06] could be used once the composite component structure is known. However, the cost of the analysis would have to be paid each time the same components were composed in the same way. Extracting the component structure and providing a workable representation on which to perform optimisations is a high-cost process. This cost is particularly problematic if the component composition is to be optimised at run time.

Adaptive components are explicitly programmed to make use of context information, e.g. knowledge of the components with which they are composed, in order to produce optimised execution schedules. We propose to implement a form of adaptive behaviour through the use of supplied component metadata and to use that metadata to identify dynamic optimisation opportunities at the time of composition. The fact that the metadata is supplied rather than extracted at composition time, obviates the need to analyse a component's code each time it is used, in order to identify whether cross-component optimisation opportunities exist. The cost of the optimisation is therefore reduced and the scope of optimisations increased.

In this chapter we examine the application of *indexed dependence metadata* to component compositions. The metadata defines the set of memory locations that a component may access at a point in

its iteration space. The memory mapping provides information on the parallelisation of data-sets to match parallelisation of components. In addition, the relationship between these mappings in different components serves to define implicitly the communication requirements of their compositions.

By examining the dependence metadata of the components in a composition, we seek to expose opportunities for cross-component optimisation that are not possible by optimising the individual components in isolation.

Specifically, we use the dependence metadata to determine whether two loops occurring separately in the components of a composition can be aligned whilst respecting dependences, in which case the loops can be fused. Fusion in turn may facilitate array contraction, reducing the space requirements of the composition, and inter-processor communication in the case where the components themselves comprise parallel loops. We use CLooG [CLO, Bas04] to generate the code for a fused loop using a scheduling matrix generated from an analysis of the components' metadata and a matrix representation of the iteration space at the core of the components' execution.

The contributions of this chapter are as follows:

- We show how the dependence metadata can be used in conjunction with a representation of the components' iteration spaces to implement loop fusion and array contraction across the component boundaries in a composition (Section 6.5). In particular, we extend this to parallel components, where the contraction reduces inter-processor communication.
- We describe a prototype software component framework incorporating the above ideas, which has potential applications in multi-core software development (Section 6.2 and Section 6.4).
- We illustrate the power of the approach by showing substantial performance improvements through fusion of parallel components in linear algebra and image processing benchmarks and a 3D multigrid solver (Section 6.6). On an eight-core Intel Xeon system, maximum performance improvements on these examples range from 12% to 50%.

This chapter is based on a paper published at the 1st International Workshop on New Frontiers in High-performance and Hardware-aware Computing (HipHaC) [HLKF08b].

6.2 Architecture overview

Our component programming system is designed to select and generate code from a library of components. Each component carries various metadata describing both functional interfaces, as in traditional component management systems, and also data dependence relationships. We identify three elements of the system: *Component*, *Interface* and *Manager*.

The application and individual *components* depend on one or more *interfaces*. Components implement interfaces, satisfying the contract defined by the interface. We must assume that any component that implements a given interface fully satisfies that contract, and that any user of the interface would have no reason to complain about the implementation. A component that depends on an interface implicitly depends on any of a set of components that are capable of satisfying the interface in question.

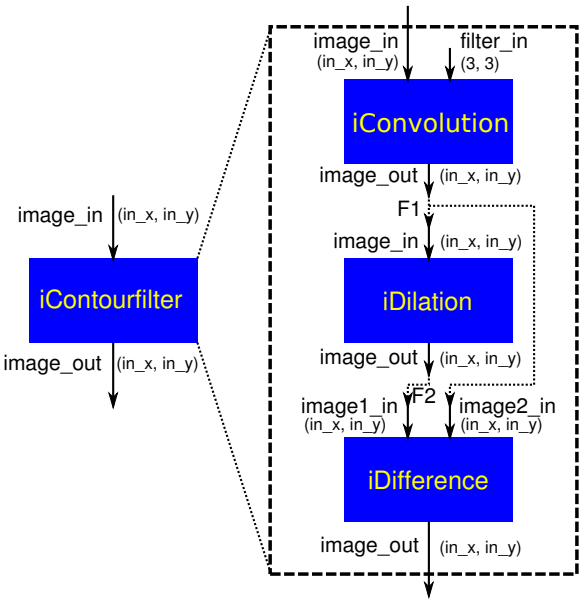


Figure 6.1: A contour filter example showing dependencies, data flows and size descriptions of inputs and outputs.

The *manager* maintains the component dependence graph and allocates component implementations to the interfaces as necessary. If a component $C1$ depends on an interface that is implemented by a component $C2$, we say that $C2$ is a subcomponent of $C1$. We generate the dependence graph of an application by recursively expanding the dependencies in the component graph. The assignment of components to interfaces is performed during a later graph pass.

Figure 6.1 shows the dependency relationships for an image filtering example. In the diagram we see a *iContourfilter* interface with one input and one output, which is implemented by a component that depends on three further interfaces to perform its computation: *iConvolution*, *iDilation* and *iDifference*. The diagram does not include components that implement the interfaces and hence satisfy the dependencies. These interface dependences must be satisfied by the component manager from the component library. Flow annotations *F1* and *F2*, metadata representing dependencies between subcomponents, define data flow dependencies at the composition level such that the metadata removes a further level of necessary analysis where possible.

Figure 6.2 shows the specification for two of the interfaces in Figure 6.1: *iContourfilter* and *iConvolution*. Note that the interface definition specifies the names and data types of the inputs and outputs to the interface. For simplicity we assume that the interface is that of a function-like component, rather than a more complicated stateful component supporting a wider variety of functions.

Figure 6.3 shows part of the component specification for the contour filter (*cf*), including its dependence on its convolution subcomponent.[1] The *cf* component, which implements the *iContourfilter* interface, depends on the *iConvolution* interface, which will be implemented by some other component defined in the system and assigned by the component manager. We name the dimensions of the input and output parameters, and specify a constant 3×3 size for the filter parameter. The *flow-to* keyword names a data flow as in Figure 6.1. The equality constraint defines an equality mapping between two variables. In this case the metadata is stating clearly that the *in_x* variable within the

[1]Note that our implementation currently uses XML to define interfaces, component specifications and dependence metadata. We envisage the use of automated or GUI based tools in the future to manipulate this.

```
<interface id="iContourfilter">
    <input type="float" name="image_in"
        format="array(in_x,in_y)" />
    <output type="float" name="image_out"
        format="array(out_x,out_y)" />
</interface>
<interface id="iConvolution">
    <input type="float" name="image_in"
        format="array(in_x,in_y)" />
    <input type="float" name="filter_in"
        format="array(filter_x,filter_y)" />
    <output type="float" name="image_out"
        format="array(out_x,out_y)" />
</interface>
```

Figure 6.2: Interface specifications for the contour filter and convolution.

```
<component id="cf" >
  <implements id="iContourfilter" />
  <uses name="conv">
    iConvolution(
      image_in(in_x, in_y), filter_in(3, 3),
      image_out(out_x, out_y) flow to F1)
  </uses>
  <constraint type="equality">
    conv.in_x=in_x
  </constraint>
  ...
</component>
```

Figure 6.3: Part of the contour filter component specification.

component *conv* is the same as the *in_x* variable defined in the *iContourFilter* interface itself. As before, this saves having to analyse the code of *cf* to obtain this information.

The implementation language for a given component is flexible. We currently support C/C++, a high level polyhedral representation of loop nests with loop bodies implemented in C++, or pre-compiled binaries. In principle the system can integrate components in any language, given support at the component level.

The conversion of one form of component into another is through a process of representation-lowering. As we can see in Figure 6.4 a component represented as nested polyhedra can be combined with other polyhedron-based components. These components can be converted into a compilable language, generally C++, using the CLooG [CLO, Bas04] library, as we shall see in Section 6.4. The code generation results in a fully specified component implemented in C++, which can then be compiled using a standard C++ compiler to a fully-specified binary level component. At each level the component can be integrated into the system, connecting to other similar components. At each conversion we take a component as input and generate a replacement component as output, with correctly lowered annotations. Components at any level can be cached, allowing full flexibility and provision of optimisations such as fusion and constant propagation at appropriate levels, as well as enabling storage of compiled components to reduce compilation overhead.

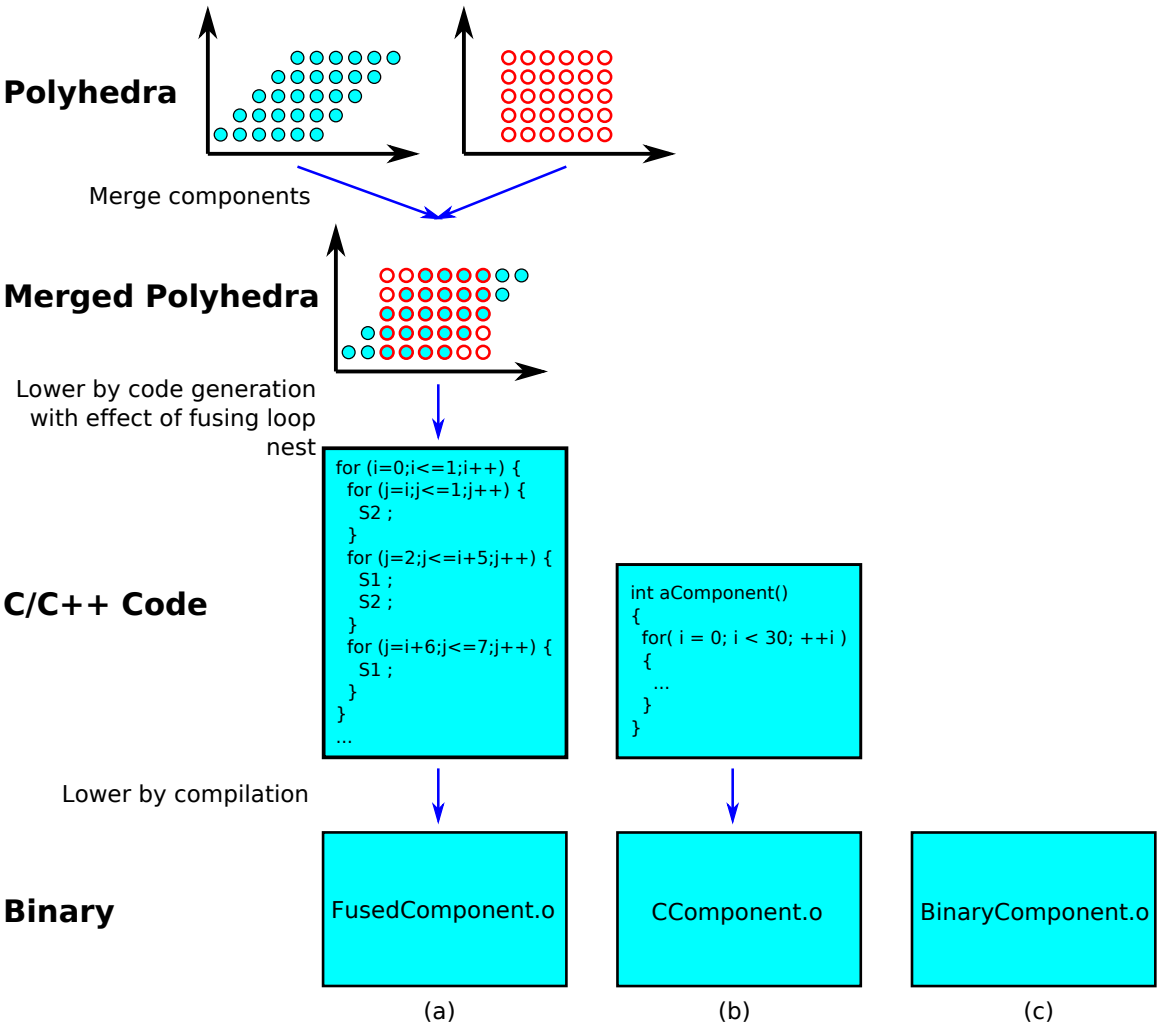


Figure 6.4: Lowering a component from a polyhedral representation to executable binary code. Components can be integrated at different levels of the hierarchy. In (a) we see a set of components represented as polyhedra, that are combined into a single fused source component. In (b) we see a different component created directly in C; and in (c) we see a third component that is created through some means as a binary and integrated directly into the system.

6.3 Component metadata

In general, the input and output variables of components need to interact with those in their subcomponents. For example, variables in subcomponents can be configured to maintain the same value as variables in the parent component. Values known at composition-time can be propagated through the component graph through these connections. To reduce analysis of parent components, additional metadata can be attached to a component specification in order to express these properties. For example, as we saw, Figure 6.3 includes an equality constraint specifying that the value *in_x* in the interface, and hence in the component *cf* that implements the interface in this case, matches the *in_x* in the subcomponent named *conv*.To generalise this, we can specify *inequalities* rather than *equalities* to constraints. Inequalities can be used to restrict the possible ranges subcomponent parameters can have. By improving the knowledge of a subcomponent's requirements and by reducing the range of values the subcomponent might require can allow more specific and efficient subcomponents to be selected.

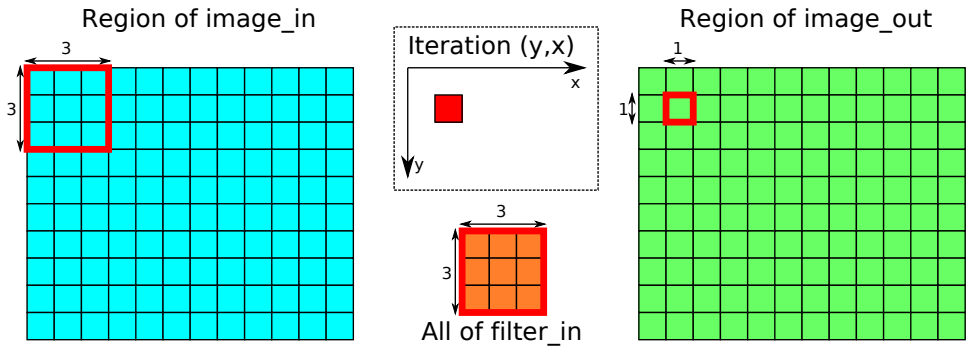


Figure 6.5: Region dependencies at a point in the iteration space. At a point in the iteration space (y, x) (in this case $(1, 1)$), we require a 3×3 region of the input (*image_in*), a 1×1 region of the output (*image_out*), and all of the filter kernel (*filter_in*). The input and output regions are relative to the iteration coordinates, and the filter is a fixed description which is reused on each iteration.

In addition, data can flow from one subcomponent to another, and hence through various levels of the component graph when combined with parent/child relationships. In the example, the *image_out* value of *iConvolution* is connected (*flows to*) the flow *F1*, which will be connected again to an input variable in another dependency of the component. Data flows are defined in the metadata at the component graph level, to avoid composition-time component analysis.

It should be emphasised that the aim is to provide dependence relationships on component inputs and outputs *at composition time*, without analysis of the component code. Indeed, an individual component might be provided in binary form, which would make such analysis very difficult and possibly infeasible.

6.3.1 Indexed Dependence Metadata for components

Indexed dependence metadata defines a set of data elements in input and output data structures that may be accessed by a point in a component's iteration space. In this case we view this in simple terms of arrays and offsets, which map fairly directly to sets of memory addresses. By interpreting the metadata associated with a component, the component manager can map a given set of iteration coordinates onto a set of memory locations. If the execution patterns are predictable and reasonably simple, the manager can then infer dependencies across sets of iterations.

In Figure 6.5 we see the region constraints of our convolution filter from the running example, assuming a 3×3 filter. Figure 6.6 shows the generic component specification for the convolution filter assuming an arbitrary-sized filter. The specification includes various pieces of metadata that the component manager can use to optimise the composition to its context. Note that omitting some or all of the metadata will not break the code; it will simply reduce the amount of information the component manager has available to it and hence limit the scope for optimisation. Incorrect metadata would be a more serious problem, and to this end the component developer must ensure that the provided metadata is correct.

The iteration space of the component corresponds to the dimensions of the input image (*image_in*), as shown. That is to say that the convolution computation will need to be performed for each point in the image, but that each point is logically separable from all other points. For each point in the iteration space a 3×3 rectangular region of *image_in* will be read. This region is addressed with its centre

```
<component id="convolution">
  <iteration_space
    dimensions="(image_in.width,image_in.height)"
  />
  <constraint type="dependentregion"
    shape="rectangle">
    <constraintinput name="image_in"
      placement="relative"
      radius="((filter_in.w-1)/2,(filter_in.h-1)/2)"
    />
    <constraintinput name="filter_in"
      placement="absolute"
      range="(0->filter_in.w-1,0->filter_in.h-1)"
    />
    <constraintoutput name="image_out"
      radius="(0,0)" />
  </constraint>
</component>
```

Figure 6.6: Constraints in the specification of a component.

computed from the iteration space coordinate and its dimensions relative to that coordinate. Given the 3×3 filter, this corresponds to a *radius* around the point of size 1 in each dimension. Additionally, the whole of *filter_in* will be read and the corresponding point in *image_out* (i.e. *radius(0,0)* around the point) will be written. The filter input variables are defined in the interface and their values propagated through the component graph.

6.3.2 Component relationships through metadata

Metadata directly affects the relationships between components. If two components communicate either through a functional dependence, or through a data flow, the metadata will need to be propagated.

A component's metadata must be combined with the metadata of connected components to give a full specification of the component relationship and of the composed component set. For example, in Figure 6.3 the contour filter requires a 3×3 convolution operation, which defines an access region on one of the inputs of the convolution, but also in terms of parallelising the contour filter component itself, on the input of the contour filter. The size of this access region depends on the size of the filter parameter to the convolution, which is specified as a constant in the contour filter specification. Therefore, to specify fully the convolution's metadata we need to propagate the filter size specified by the contour filter through the graph and allow the dimension constants to be propagated into variables within the convolution component's specification. Metadata can be propagated both through parent/child and through data flow relationships, reaching the entire component hierarchy in this fashion.

When the application requests an interface, values are bound to the interface's parameters and to variables contained within the metadata. These values are combined with constraints and dependence metadata throughout the component graph to bind values to variables and define component relationships as accurately as possible. Component selection or composition uses the propagated information to limit the binding of components to interfaces or to define possible composition optimisation opportunities. Given a prospective subcomponent that accepts a specific input parameter, for example

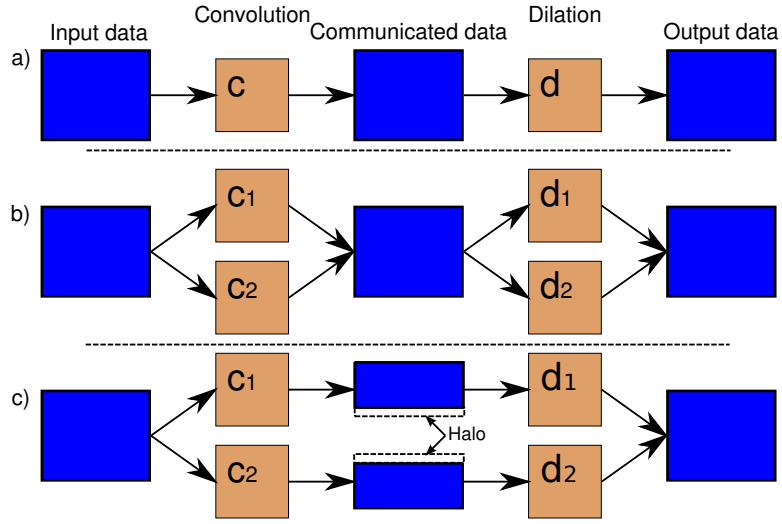


Figure 6.7: The addition of region descriptors enables more efficient parallelism.

a convolution that is designed specifically for 3×3 filters, the component management system can choose this component in preference to a general one if it is likely to be a better performing choice. If, on the other hand, the requirement of the contour filter component were more general and might require a convolutions over a wider range of filter sizes, then that 3×3 convolution component would be an unsatisfactory choice and a more general convolution selected.

Figure 6.7 shows how the information provided by combining region definitions with the size of the datasets can reduce the size of the required communication between two components. In this case we see the convolution and dilation components from Figure 6.1.

Figure 6.7(a) is an example of a simple component composition communicating via an intermediate data set. Unless we know what precise operations these components will perform, we have no way of knowing how much data the dilation component will need. Without knowing how much data it will need, we do not know how much data we should pass to it from the convolution component. Therefore it becomes necessary for us to pass all of the data from one component to the other.

If we assume the components are parallelisable and have some knowledge of how to achieve this (this can be a part of a component's interface) then we can parallelise them individually. However, we can not parallelise the composition, because the second component (in this case the dilation) might need any part of the output of the first. Equivalently, any subset of the second component might need any of the output of any subset of the first. This case is illustrated in Figure 6.7(b).

With full region information we can minimise the communication between parallel components. For example, if the dilation component depends also on a 3×3 filter then parallelisation of the components as shown in Figure 6.7(c) requires only half the data set, plus an additional halo strip, to be sent from each convolution subset to its corresponding dilation subset. As a direct consequence the component management system can keep data data in more localised, and often faster faster, memory for longer. Efficiency and performance can be gained and communication becomes more predictable. If c_i and d_i both execute in the same memory region, only the halo strips would need to pass through higher levels in the memory hierarchy.

6.3.3 Scalability

The component metadata in the examples are currently written by hand. This is practical with reasonably simple components, and indeed it is often the case in reality that the programmer knows more about his code than a compiler is able to extract. In such cases creating the metadata by hand would be beneficial, although with some level of tool support to make the process easier and to support checking that the metadata are correct.

We envisage that in practice the information would be obtained using a suite of tools supporting component analysis at construction time. Clearly, complicated components limit the feasibility of such analysis, whether obtained at construction time or composition/run time. Construction time offers more time for complicated analysis, in the same way that just-in-time compilers tend to be lighter weight than off line compilers. By limiting the dependence information to the input and output data structures of the component, and assuming the contents are correct, we simplify the run time workload, and improve scalability in that manner, ensuring that the scale of individual components does not affect composition time scalability.

Generation time analysis may not be possible for all components. However, the discussed system improves containment of analysis at construction time, and as a result increases the possibility of correct dependence construction over fully-general system-wide analysis of all possible interactions.

6.4 Code Generation

Our system supports components input to the system in various different forms. In the simplest case we can use a pre-compiled binary, which uses a pre-defined simple calling interface and which can be dynamically linked at runtime. Alternatively, we can compile and link a component for which we have the C or C++ code at run time. Delaying compilation to run-time offers scope for performance improvements as the compiler may have more information about the code, or the system, and as a result be able to perform more specific optimisations.

Generating code at run time is an alternative solution. This code will of course then be compiled and linked as in any other case. Earlier work such as Taskgraph [BHKM03] shows that run time code generation and compilation can be effective, enabling a wider degree of run-time optimisations and code specialisations than are available merely from run-time compilation. As we saw in Figure 6.4, in this system we view both run time code generation and compilation as a lowering from one implementation level to another. At each stage a valid component definition is maintained. For example, we can lower from a high level source representation, to C++; then through compilation of C++ to a binary.

We use the CLooG [CLO, Bas04] code generator to construct the code for compilation. The generator constructs C++ code and hence a C++ component in the first stage lowering from a high level representation of the component that is a combination of a declarative definition of the iteration space combined with kernel code.

CLooG is a code generator based on a domain and scheduling subset of the polyhedral model [GLW98]. The polyhedral model in full represents execution domains, schedules, access functions and (optionally) dependencies as polyhedra in multi-dimensional space. CLooG takes as input a matrix repre-

```
COMPONENT_TARGET(difference)
{
  POLYHEDRAL_LOOP(i) [ i >= 0;
      i < image1_in.height(); ] {
    POLYHEDRAL_LOOP(j) [ j >= 0;
        j < image1_in.width(); ] {
      image_out(j,i) = image1_in(j,i)-image2_in(j,i);
    }
  }
}
```

Figure 6.8: A simple polyhedral representation of the iteration space of an image difference operation.

senting a set of inequalities. The inequalities specify a set of affine half-spaces as subsets of a multidimensional space, which together define a polyhedral execution schedule. Each row of CLooG's input matrix represents an individual inequality. An example of the input matrix can be seen in Figure 6.11(b). CLooG processes the matrices and generates the necessary code to allow each statement to visit each integer point within the defined polyhedron, given a specified schedule. It does not perform dependence analysis and so for ill-considered input will generate incorrect output. As a result, our input to the code generator must satisfy all necessary dependencies and be scheduled correctly before generation.

We generate input to CLooG from a component implementation as in Figure 6.8. We could alternatively utilise full dependence analysis of source code, or to use a simplified binary representation of polyhedral code and dependencies. However, full analysis would be prone to inaccuracies and falling short of completeness without programmer hints in the source code. For our purposes, this syntax offers a simple basis to work with for experimentation.

In this representation, we specify the execution polyhedron of the kernel using nested range descriptions to define dimensions for the iteration space. We define ranges for each iteration variable using a list of inequalities, which map almost directly to the polyhedral representation while being easier to read. The inequality syntax is converted into CLooG's input matrices during the process of lowering from CLooG input to C++, allowing CLooG to generate the C++ code from this matrix representation.

The major benefit of using a code generator such as CLooG for this purpose is that CLooG is capable of generating hundreds or thousands of lines of code to cover complicated iteration spaces which would be extremely difficult to write by hand. The transformations are performed on a set of matrix representations that do not change in complexity and the code complexity arises only during code generation. As we can see in Figure 6.9, when we fuse loops we end up with a large number of iteration space fragments that have more or fewer of the kernels executing in a larger wrapping iteration space. Code could be written to support this using conditionals, but a large number of conditionals will severely affect performance of the loop nest. CLooG is capable of generating code that either uses conditionals, or is appropriately unrolled to maintain high efficiency in the loop fragments. This code can easily be regenerated with minor changes to the described iteration spaces, which would be particularly painful to maintain by hand.

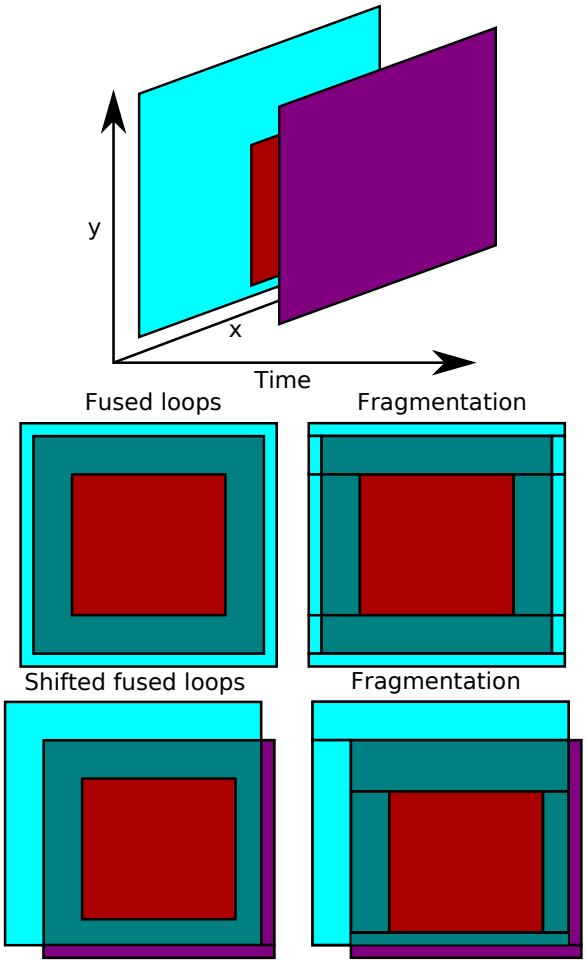


Figure 6.9: Three separate loops operating over overlapping regions of x and y, but consecutively in time. If these loops are trivially fused such that all execute at the same time and in the same loop nest we see a high degree of fragmentation necessary from the loop nest to allow this to take place. The alternative to this fragmentation is to insert conditionals such that the loop nest is bounded by the minimum and maximum in either direction and instructions from the three loops are conditionally executed at each point. Once we sift the loops relative to each other to account for dat-flow dependencies that might be present we see even greater fragmentation. CLooG is designed to efficiently generate code for such loop nests that would be too complicated to write by hand.

6.5 Using metadata for optimisations

The presence of dependence metadata on components allows the *manager* to perform component mapping decisions and, in addition, cross-component optimisations. In this work we illustrate the potential by applying loop fusion, and the array contraction this enables, to a connected subgraph of components.

6.5.1 Increasing temporal locality with loop fusion

Loop fusion [KM94] takes two or more consecutive loops and merges the bodies together as illustrated in Figure 6.10(a). Fusion reduces the number of control instructions. In the right context it improves

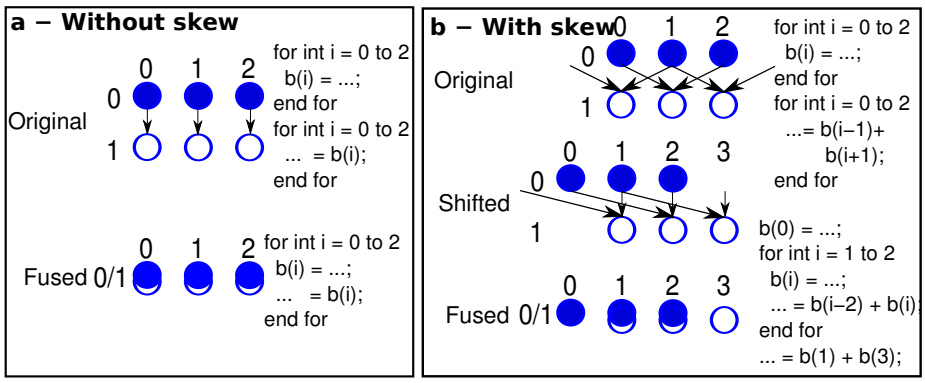


Figure 6.10: A simplified one-dimensional loop fusion example.

the temporal locality of data and, when fusing parallel loops, avoids unnecessary synchronisation. The risk of loop fusion is that cache performance and instruction scheduling can be harmed. As a result it is possible for fusion to reduce performance as the increased working set or number of loop body instructions can harm cache performance, instruction scheduling or add excess control to deal with alignment issues [RK98].

Loop dependencies can complicate fusion. In Figure 6.10(b) for example, statement 1 has a forward data dependence on the output of statement 0. These two statements from the same iteration number of the original loops cannot execute in the same iteration of the fused loop because the inputs to the iteration of statement 1 will not yet have been generated by statement 0. The dependence can be resolved by shifting the iteration space of the second loop such that the loop schedule is delayed by an iteration. The schedule shift allows each loop to perform its given set of iterations with all dependencies satisfied before the data is required. The result of this fusion and shift is a guarded or (for performance, if not code size) partially unrolled loop nest as in Figure 6.10(b), with a necessary loss of parallelism at the edges. With a large enough iteration space, the lack of parallelism should apply to a small subset of the iterations, and the gains in parallelism from fusion would outweigh that considerably. This fuse, shift and unroll combination is sometimes sometimes called “shift and peel” [MA97].

Input and output regions defined in the metadata make the data dependencies explicit. As a result of this metadata, we know which data values may be read or written at a given point in a component’s iteration space. Using this knowledge we can compute the shift necessary to resolve data dependencies between two components’ executing kernels. As previously noted, this is metadata provided on the input and output datasets only - internal data structures are assumed to still be valid as long as the overall execution schedule of the component is within the bounds of parallelisation that it allows. Shifts in the iteration space do not affect accesses to internal data structures, they only change the relative schedule between two components.

We use CLooG to generate code representing the fused set of components. We supply the individual input matrices that define the iteration space for each component. The set of matrices collectively defines the iteration domain of the statement set. We also provide a mapping of points in the iteration space to a logical execution time, known as the *scatter* matrix. This scatter matrix affects how CLooG will schedule the independent iteration spaces of the components and allows us to perform the shifting necessary to satisfy dependencies. As demonstrated in Figure 6.11, we can specify that a point (i, j) in the iteration space (a) of a component can be mapped to (t_i, t_j) in time, where either $t_i = i$ and $t_j = j$ (b), or $t_i = i + 1$ and $t_j = j + 1$ (c), shifting the schedule.

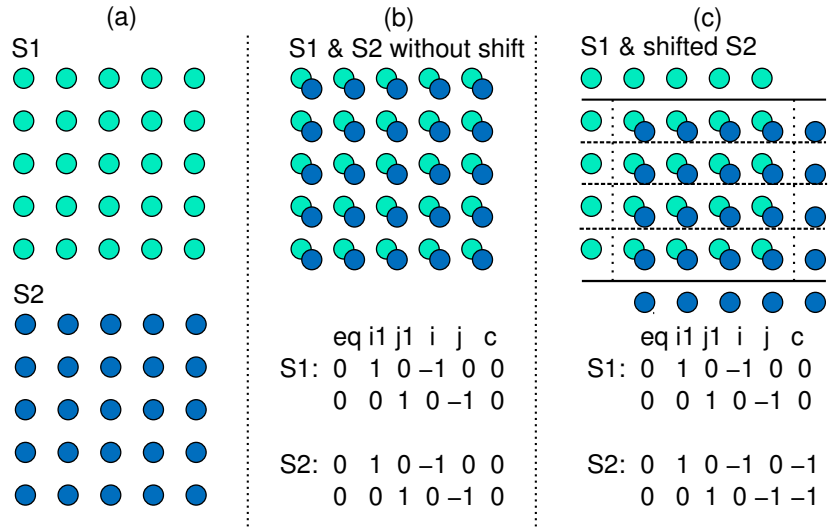


Figure 6.11: The scatter matrix can be used to schedule the loop by changing the logical execution time of a given iteration.

The amount of shift required depends on the dependence relationship between two components. These relationships are computed from the access region metadata. For example, a 3×3 region as input to the second component where that input is sourced from an output of the first component requires a shift of 1 in the iteration space of the second component so that the output of the first is ready when it is needed. In the general case, we need to compute the last iteration in the source component that may generate data required by the matching iteration in the target component. If the maximum dependence distance can be computed as a constant then we can compute a static schedule correction to satisfy this. We parameterise the scatter matrix by a set of shift values computed from the dependence relationships to map the iteration space of the component and therefore of its statements to a later logical execution time. With a correct schedule defined in the scatter matrices, CLooG will generate a series of loops that respects the inter-component data dependencies.

Component selection for fusion depends on the flow of data between components. Unrelated components are easy to fuse having no dependencies to satisfy, but are unlikely to benefit from fusion unless the reduced control overhead gives an adequate benefit. Components that share inputs, or communicate using a intermediate data structure, are more likely to benefit by reducing memory traffic. Having analysed the data flow in the parent component at construction time, we can fuse the children at composition time using the data-flow metadata defined within the parent component's specification. Calls to the subcomponents can be replaced with calls to stub functions that merely prepare data structures and where the execution of the fused component can be delayed until the last subcomponent call. As a result, fusion can occur to the subcomponents without actively changing the parent component, allowing a pre-compiled parent to use a series of fusible children and gain in efficiency through their fusion when this might not have been fully considered at construction time.

The code generated by CLooG from a fused set of loops can expand enormously in the worst case. The generation can be parameterised to reduce this, but not unrolling outer levels of the loop nest, but too many conditionals will reduce efficiency. The more metadata that is present with constant values, or at least maximal values, defining region sizes, the greater the predictability of the code generation. By generating a loop nest with clear maxima and minima there will be fewer special cases to deal with because the code no longer has to deal with interactions at extreme points in the domain that will never really be visited. As a result, the generated code can be simpler than in the unbounded case.

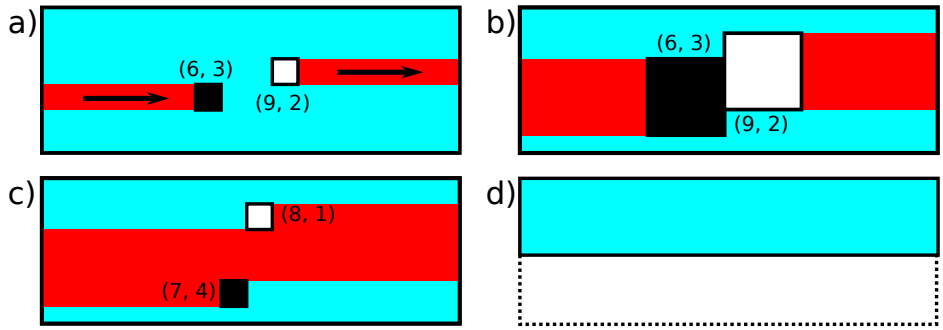


Figure 6.12: A producer and consumer performing a row major traversal over a shared data structure and communicating using a contracted buffer, with iteration space coordinates shown.

6.5.2 Reducing storage through contraction

Loop fusion reduces the period between generation and use of intermediate data values, often leading to more efficient utilisation of the cache and improved performance. *Array contraction* is a further optimisation that reduces the size of intermediate arrays to only store the values that are in flight at any point in time rather than all that will be generated during the execution. Contraction of arrays where data is communicated limits the range of memory addresses that will be read or written during the communication. Limiting the set of memory addresses enables more efficient use of the cache, whereby fewer values will be displaced from the cache to main memory or have to be read in as writes start to occur. In addition, limiting the range of memory addresses reduces the cache pollution effects that a particular data communication might have on other memory accesses in the application. Array contraction can be a key enabler of high performance in large parallel fused loops [CKPN07]. Rather than storing entire intermediate arrays, we reduce the intermediate storage to the minimum required to satisfy data flow requirements.

Contraction can be seen in Figure 6.12. A producer will generate data in a shared data structure between any two points in its iteration space (Figure 6.12a). A consumer with a large input window will consume data over a similarly constructed range of addresses (Figure 6.12b), in this case over the same range of iteration points. To generate this data the producer must cover the range of the iteration space seen in Figure 6.12c. Note the offset between the location of the producer in Figure 6.12(c) and the consumer in Figure 6.12b: this is the required array shift for correct communication. No more data than this is required during this range of iterations and the entire data set can be adequately communicated using a circular buffer smaller than the original data structure Figure 6.12(d).

6.6 Experimental results

We implement three examples using our component framework to demonstrate its capabilities and how we can use it to improve the performance of an application. These examples possess different data flow situations and hence show varied performance after optimisation. The code for these examples can be seen in Section A.1.

To enable fusion, all subcomponents are implemented in a high-level polyhedral representation, as in Figure 6.8. In addition, all components have appropriate dependent region and data flow meta-

data attached to describe the relationships between component inputs and outputs and the component iteration spaces.

We compile the examples using Intel C/C++ 10.1 or GCC 4.2 (whichever performs better) on an eight core, dual-socket Intel Xeon X5355 based machine running at 2.66GHz. We use a 64-bit Linux 2.6 kernel and parallelise using OpenMP. The core code relating to these experiments is listed in Appendix A.1.

6.6.1 Image processing

As an image processing benchmark we use the contour filter Figure 6.1 that has been a running example throughout this chapter. The contour filter operates on four-component (RGBA) data and is vectorised using SSE instructions as single pixel vector operations.

The contour filter comprises three components:

Convolution
A standard convolution filter computing a single result value for each point in the iteration space for each component of the pixel. The computation is computed over a 3×3 rectangular input region.

Dilation
The dilation filter operates on a 3×3 input region much like the convolution. Its purpose is to find the maximum value in that region and use that as the output value. Hence dilating the edges of colour blocks.

Difference
A simple arithmetic operation taking a single input pixel from the outputs of both the convolution and dilation, and computing a single output pixel.

The dependence between the convolution and dilation necessitates a shift in the execution space because the dilation requires a 3×3 access region on the convolution's output. Therefore, for the dilation to access the output of convolution iteration $(1, 1)$ during the same combined iteration, it must be executing iteration $(0, 0)$. We can see this in Figure 6.13. The effect of the delay in the dilation's iteration space to allow it to read the convolution's output is that its output is delayed, and hence the difference operation must be delayed as well. The effect propagates through the dependency graph. The execution of the elements of both the dilation and difference should be delayed by the radius of the dilation's input region to satisfy the dependency.

Figure 6.14 shows performance results for the contour filter with and without SSE optimisations. We can see that there is a substantial reduction in execution time for fusion combined with contraction. For the SSE version, execution time is reduced by 21% for a single thread, 35% for four threads and 48% for eight threads, with similar gains for the non-SSE version. While not plotted on the graphs, fusion alone offers 4%, 11% and 20% respectively on the same computation.

Demonstrating the benefit of contraction in this case we can see Figure 6.15. The improvement from fusion alone is slightly erratic, but tends to decrease with data set size as the larger range of visited addresses increases the chance of an individual element being removed from the cache. A similar

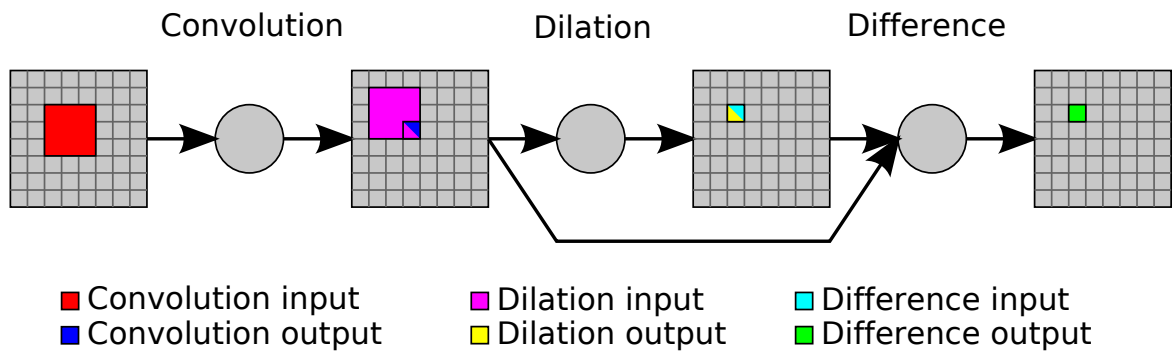
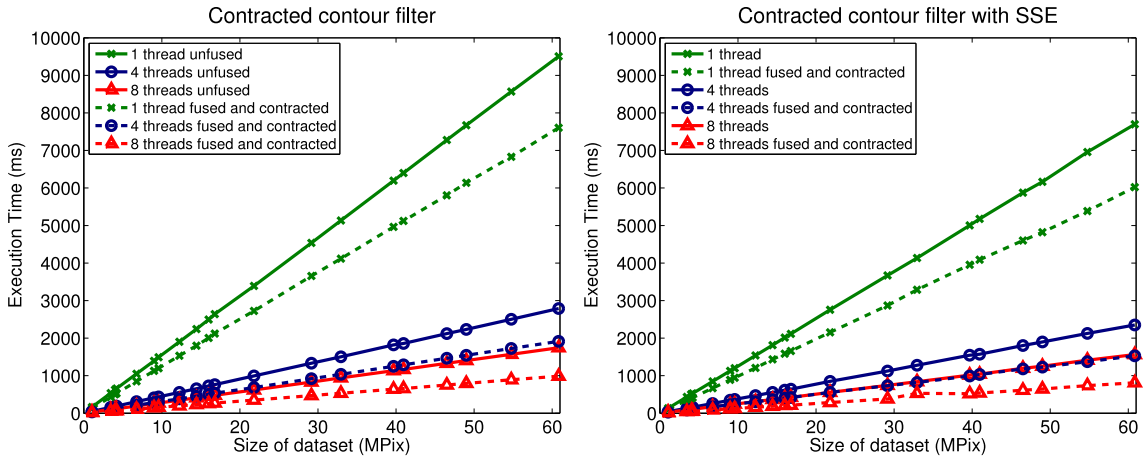


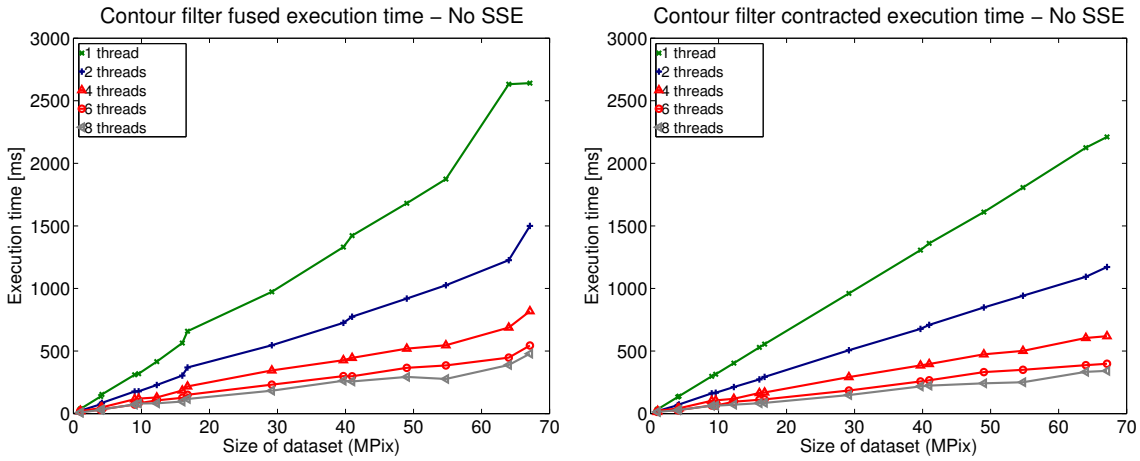
Figure 6.13: Interaction of the three components in the contour filter example. As data flows through the components, note that the dilation and difference operations must execute at an earlier, and hence delayed, part of the iteration space than the convolution.



(a) Without SSE.

(b) With SSE.

Figure 6.14: Execution time of the contour filter example with contraction and fusion.



(a) With fusion alone.

(b) With fusion and contraction.

Figure 6.15: Comparing fusion alone to fusion with contraction for a range of datasets.

effect is not seen with the contracted data sets where the visited address range is reduced to a circular buffer of a few image rows in size and the performance increases are smoothed and improved overall.

6.6.2 Linear Algebra

Our linear algebra example is a biconjugate gradient solver from the Iterative Template Library [LLS], with components defining various aspects of the computation flow. Fur the purpose of experimentation we allow fusion to occur between a standard matrix/vector multiplication, and a transposed matrix/vector multiplication. These components share input matrices, a situation distinct from the earlier example where we saw data flows relating the components. The result of this sharing is that there is no communicated array to contract, and the example only supports fusion. However, we should expect improvements from the improved access locality resulting from accessing the input array twice on the same iteration rather than having to be loaded into the cache on two separate occasions. We can see in Figure 6.16 that the transpose in the iteration space is equivalent to transposing the addressing of the input. The difference between the transposed and non-transposed versions becomes a difference in the way the accumulation into the output is performed. In the untransposed case the output will sum into a scalar, in the fused case into the entire vector element by element. The performance gains come from reusing the cache efficiently as the vectors are small, and the matrix is used in an efficient, simultaneous, streaming manner by both computations.

In this example we use 1×1 access regions because the execution maps a single iteration space point to a single data element from each input. As the input and output vectors are present in a single dimension only, the mapping into the vectors is a flattening from 2 dimensions onto 1.

Figure 6.17a graphs performance results for the fused versions of the biconjugate gradient solver as well as results for Intel's Math Kernel Library (MKL) [Int08] as an optimised version for comparison. In addition the performance of the unfused component-based code on a single thread is shown to give a baseline. We can see that while there is an improvement in performance for all numbers of threads, this improvement is more pronounced for 4 and 8 threads where memory contention between cores

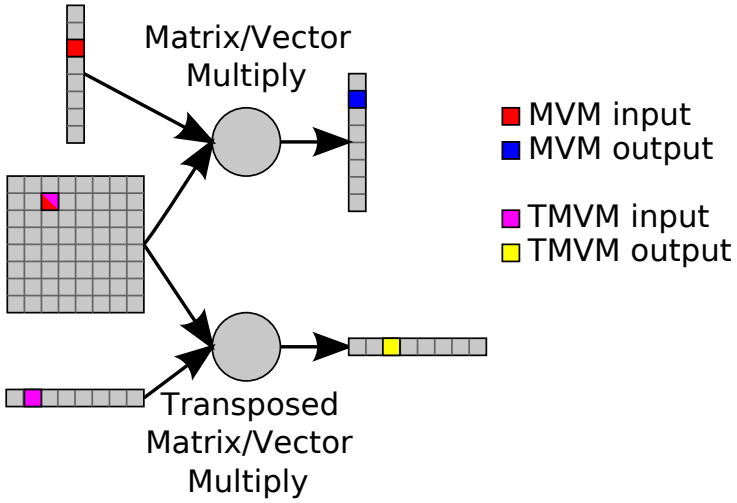


Figure 6.16: Interaction of the two components of the linear algebra example. Note that the indexing is transposed rather than the iteration space. If we transposed the iteration space indexing would be the same in both cases, but the computation would not be fusable. Accumulation is performed into the result vectors.

is reduced by fusion. Figure 6.17b graphs the same results in terms of the benefit from performing fusion. This demonstrates that while a higher number of threads benefits more from fusion, the improvement depends on the size of the dataset, including features such as the row length's relationship to cache parameters.

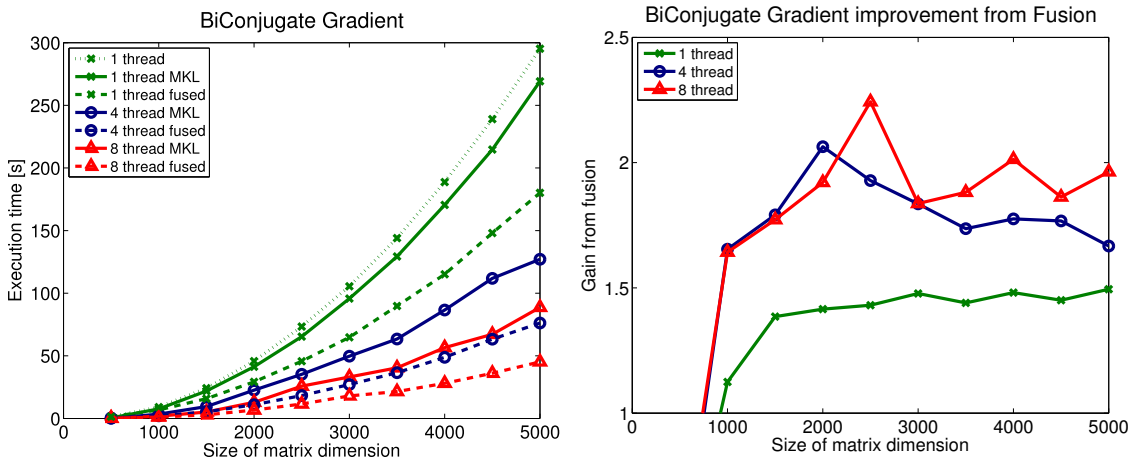
6.6.3 3D Multigrid

Multigrid is a technique for solving a problem using multiple discretisation levels. It can be used to solve differential equations more efficiently than using fixed-grid solution techniques. We adapted this example from the NAS Parallel benchmarks suite [CDS00] using fixed boundary conditions.[2] We created a sequence of dependent components based on the core functions that iterate on the data, and the dependence code for this can be seen in Figure 6.18:

- Data initialisation.
- Interpolation from a lower resolution computation stage.
- Computation of residuals.
- Application of a smoother to the data.

The four components are related by region dependencies that describe regions of input and output data structures accessed by a point in the iteratioin space, and by data-flow dependencies that describe how data flows from one component to another. Together these dependencies describe how a value in the iteration space of one component relates to a value in the iteration space of the next in sequence. We

[2] In the original code the computation is complicated by a cyclic dependency due to a wrap-around boundary condition. While fusion is still possible with the cyclic dependency, performance benefits are lost due to the increased loop shift necessary to support wrapping on all three dimensions.



(a) MKL and custom versions.

(b) Performance scaling

Figure 6.17: Results for the linear algebra example. (a) compares MKL with custom versions for 1, 4 and 8 threads (the custom version without fusion shown only for 1 thread). (b) shows how increasing the number of threads changes the performance.

```
<operation name="mgrid">
  <uses name="zero3">mgrid_zero3.mgrid_zero3(
    z_out structured (GRIDX, GRIDY, GRIDZ) flow to z1,
    n1 = m1_k, n2 = m2_k, n3 = m3_k)
  </uses>

  <uses name="interp">mgrid_interp.mgrid_interp( z_in structured (GRIDX_J, GRIDY_J, GRIDZ_
    u_in structured (GRIDX, GRIDY, GRIDZ) flow from z1,
    u_out structured (GRIDX, GRIDY, GRIDZ) flow to u2,
    mm1, mm2, mm3, n1 = m1_k, n2 = m2_k, n3 = m3_k, k, m )
  </uses>

  <uses name="resid">mgrid_resid.mgrid_resid(
    u_in structured (GRIDX, GRIDY, GRIDZ) flow from u2,
    v_in structured (GRIDX, GRIDY, GRIDZ), r_out structured (GRIDX, GRIDY, GRIDZ),
    n1 = m1_k, n2 = m2_k, n3 = m3_k, k, m, a_in structured (4) )
  </uses>

  <uses name="psinv">mgrid_psinv.mgrid_psinv(
    r_in structured(GRIDX, GRIDY, GRIDZ),
    u_in structured(GRIDX, GRIDY, GRIDZ ) flow from u2,
    u_out structured( GRIDX, GRIDY, GRIDZ ),
    c_in structured( 4 ), n1 = m1_k, n2 = m2_k, n3 = m3_k, k, m )
  </uses>
</operation>
```

Figure 6.18: Dependences on subcomponents in the multigrid example.

require $3 \times 3 \times 3$ regions around the input to each of the the interpolation, residual and smoother components. All three components apply 3-dimensional stencil computations to their input data, with a fairly complicated interaction of data sets at different resolutions as we can see in Figure 6.19.

Each value propagation pass is performed repeatedly from low resolution to high resolution, during which low resolution data is interpolated and smoothed into high resolution data. Residuals are updated at each resolution during the pass and the smoothed higher resolution data is output for the next level of the computation. We fuse components at a given resolution level, and while the interpolation operation is performed at half resolution, this single-level fusion removes the complication of scheduling varied strides through the computation: an overhead which would considerably worsen performance.

In real-world examples we cannot always expect to have highly tuned kernels, and so it is difficult to decide on a level of kernel complexity to implement for fairness. As this is a benchmark from a standard suite it has been tuned for good performance on a reasonable range of architectures. To make the comparison with the benchmark more fair we make the kernel more efficient by absorbing the inner dimension of the loop nest, maintaining the tuned kernel present in the original benchmark. Given such a kernel, our access region specifies an entire row of the data set in one dimension and a 3×3 region in the other two. Note that the component manager need not know that our tuned kernel has a carefully written inner loop, only that it needs to access an entire row of the data set to perform its work. Hence kernels such as this can be integrated into the system at any level and, indeed, can be implemented in various different ways while still satisfying the interfaces correctly.

Figure 6.20 offers performance results for 1, 4 and 8 threads. The improvement from fusion peaks at 4 threads where we see a mean reduction in execution time of 12% over the range shown. For larger data sets the performance of fusion falls off as the amount of data maintained by the loop shift accessing a 3D data set creates stress on the cache and other shared data structures of the CPU. A shift of 1 in the loop requires an entire additional plane of the dataset to be present in the contracted array, and the size of the array quickly fills the cache. The improvement (though not overall performance) peak at 4 threads is explained because the L2 cache on this architecture is shared between pairs of cores. The effective cache size per core is therefore smaller when 8 threads are used than it is when only 4 threads are used, and hence the caching limits are reached earlier.

6.7 Conclusions and Future Work

We have shown in this chapter how interfaces with indexed dependence metadata can be used to improve the performance of component compositions. Our experimental results show that metadata can be used to perform aggressive component fusion, where hundreds of lines of code (200-300 in the contour filter and over 1500 for the multigrid example) can be generated that would be extremely challenging to implement by hand and almost impossible to maintain. We have also confirmed that loop fusion can substantially reduce execution time through improvements in temporal locality of data.

We have only discussed a subset of the possible optimisations in this chapter. The general goal has been to reduce the amount of data being moved between components, and hence reduce memory traffic and improve performance. Metadata allows us to achieve this with reduced analysis. We do not deal with the initial movement of the data to the correct place, in part because the target architecture is

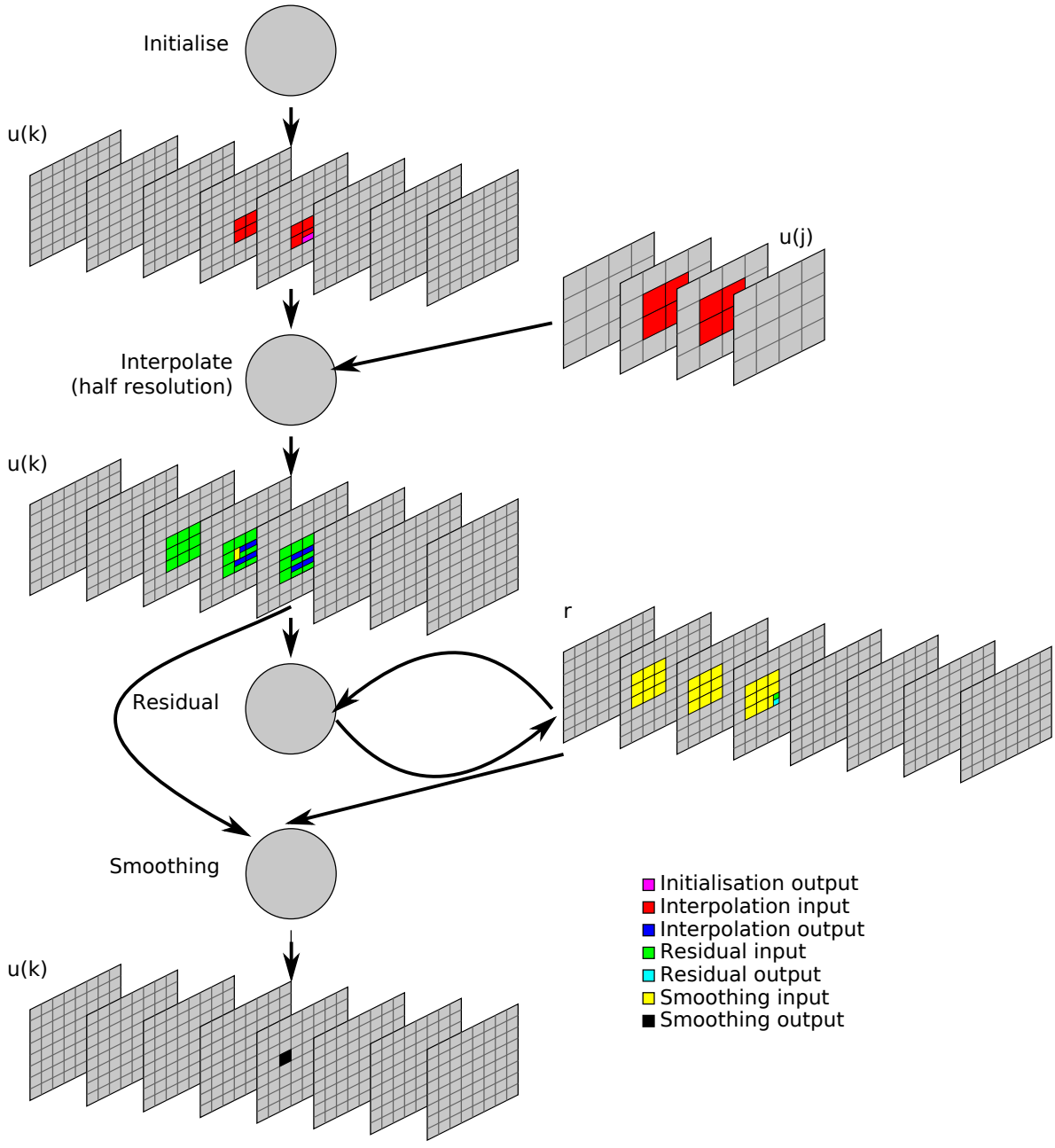


Figure 6.19: Interaction of components in the multigrid example for a single resolution level of the computation. Notice that u is interpolated from a lower resolution to a higher resolution. This interpolation is at the heart of the multigrid method.

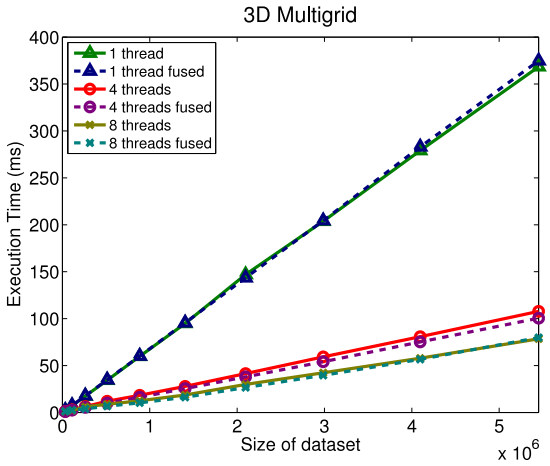


Figure 6.20: Execution time for single and four and eight threaded 3D multigrid solver kernel.

a general purpose CPU. Chapter 7 will discuss the application of dependence metadata to the problem of data movement on an architecture where this is a more serious issue that must be considered in any application written.

The THEMIS [KBFB01] proposal discusses more possibilities for metadata than we have been able to implement to date. In the future further investigation is possible, particularly in the area of applying cross-component optimisation techniques to data layout by adding metadata annotations describing the access patterns for data. By changing the layout of data we envisage that improved access patterns should be possible, but in addition more sophisticated subsetting of connected component iteration spaces should allow improved locality even on data with far more complicated access patterns than we have seen in this work.

The multigrid example shows that in some cases fusion gives only a small benefit and it is likely that there are cases where fusion would reduce performance. In these cases we plan to use adaptive component mapping to use the original components rather than fused sets when a fusion attempt reduces performance. Optimal combinations may include calls to vendor libraries wrapped in components, as used in the MKL comparison for the linear algebra example.

Novel architectures such as heterogeneous multi-core platforms require novel optimisation strategies. Hand coding is often impractical. We envisage that adaptive, metadata-driven optimisation techniques will be of increasing relevance as technology develops.

6.8 Summary

In this chapter we discussed a component programming model that includes both traditional function interfaces and additional metadata describing the iteration space of the computation kernel housed within a component and the mapping of that iteration space onto the input and output data structures. We have seen how this metadata can be used by a combination of a runtime system and composition-time code generation to produce optimised fused components from sequences of joined simpler components. In the next chapter we will see how similar metadata can be used to manage data movement for single computation kernels in an architecture with decoupled memory access.

Chapter 7

Decoupled access/execute software in C++

7.1 Introduction

Architectures with software-managed memories can achieve higher performance and power efficiency than traditional architectures with hardware-managed memories (e.g. caches), but place additional burden on the programmer. For a traditional architecture, the programmer typically designs a computation kernel and specifies the order in which the kernel traverses the iteration space by way of producing an appropriate loop nest. To off-load the kernel to a co-processor equipped with local memory, the programmer must additionally write code to explicitly manage data movement into and out of that local memory. This code will move data on and off the co-processor in such a way that high throughput is maintained.

This step is not trivial and is laden with complications that might not be expected by the algorithm designer. Indeed these may be complications that an algorithm designer need not be conscious of. A programmer aiming for high performance on a co-processor needs to consider not only issues of general algorithm design, frequently already a complicated task to design, but also to consider optimal data transfer sizes, constraints on alignment and format of data transfers, layout of data for reuse and so on. Moreover, when the working set of a processor is too large to fit in its local memory, the programmer has to use low-level optimisation techniques such as double buffering to overlap computation and communication. Any low level optimisations of this sort tend to harm programmer productivity. Unfortunately these optimisations also harm code code portability and maintainability, leading to ever more severe productivity and high costs in the future.

The indexed dependence metadata we discussed earlier that represents the mapping from the iteration space to its input and output data structures can be adapted to support data movement. Utilising these metadata concepts we introduce and discuss in this chapter the decoupled Access/Execute (Æcute, pronounced “acute”) programming model. The Æcute model is designed to allow the programmer to express explicitly:

- The iteration pattern and execution schedule of a computation.
- The memory access pattern of a computation in terms of a mapping from the iteration space to the input and output data spaces.

The iteration space/data mappings can be seen as independent descriptions but may be used to represent programs designed intended to run on both traditional and accelerator-based architectures. We show that, given these descriptions, in many cases the compiler or run-time system can derive efficient data movement, thus removing from the programmer the additional complexity of managing data movement. This data movement derivation should be possible, even if extracting the information from kernel code by automated analysis is difficult or impossible.

In the remainder of this chapter we argue that decoupling access and execute is natural when programming architectures with software-managed memories (Section 7.2) and introduce decoupled Access/Execute specifications (Section 7.3). At the same time we wish to make it clear that the Ækute model is merely an extension of the concept of Indexed Dependence Metadata, discussed in terms of kernel dependences in Chapter 5 and Chapter 6.

We discuss the prototype Ækute framework (Section 7.4) and use examples adapted from linear algebra and signal processing (Section 7.3.1 and Section 7.5) to show the ease of programming using the specifications. We present experimental results for the examples examples (Section 7.6) obtained on a Cell Broadband Engine (BE) processor and compare them against alternative implementations, which use hand-written DMA transfers and software-based caching. These comparison implementations are written using the assistance of Alastair Donaldson at Codeplay Software.

The contributions of this chapter are as follows:

- We propose the Ækute programming model as an alternative approach to developing computational kernels for architectures with software-managed memories using the principles of indexed dependence metadata as we saw in Chapter 5.
- We demonstrate a prototype software implementation to show how the Ækute model could work in practice.
- We demonstrate the application of that model to a set of benchmark examples.

This chapter is based on a paper published at the HiPEAC 2009 conference [HLDK09c] in collaboration with Codeplay Software.

7.2 Background

Since the 1980s, microprocessor designers have worked hard to preserve the *illusion* of fast memory by providing hardware-managed caches. Sadly, increasing the number of transistors dedicated to caches has been found to achieve diminishing effects on performance. Moreover, to achieve high performance even on a caching architecture, the cache structure must be considered carefully. As a result, optimising software for the memory hierarchy has become the principal activity of performance-conscious programmers and compiler writers, who "spend much of their time *reverse-engineering and defeating* the sophisticated mechanisms that automatically bring data on to and off the chip" [Hof05].

Given this unsatisfactory situation, some hardware designers have turned their attention to software-managed memory hierarchies, where data is copied between memories under explicit software control. Examples include the Cell BE architecture from Sony/Toshiba/IBM [Hof05], the CSX SIMD

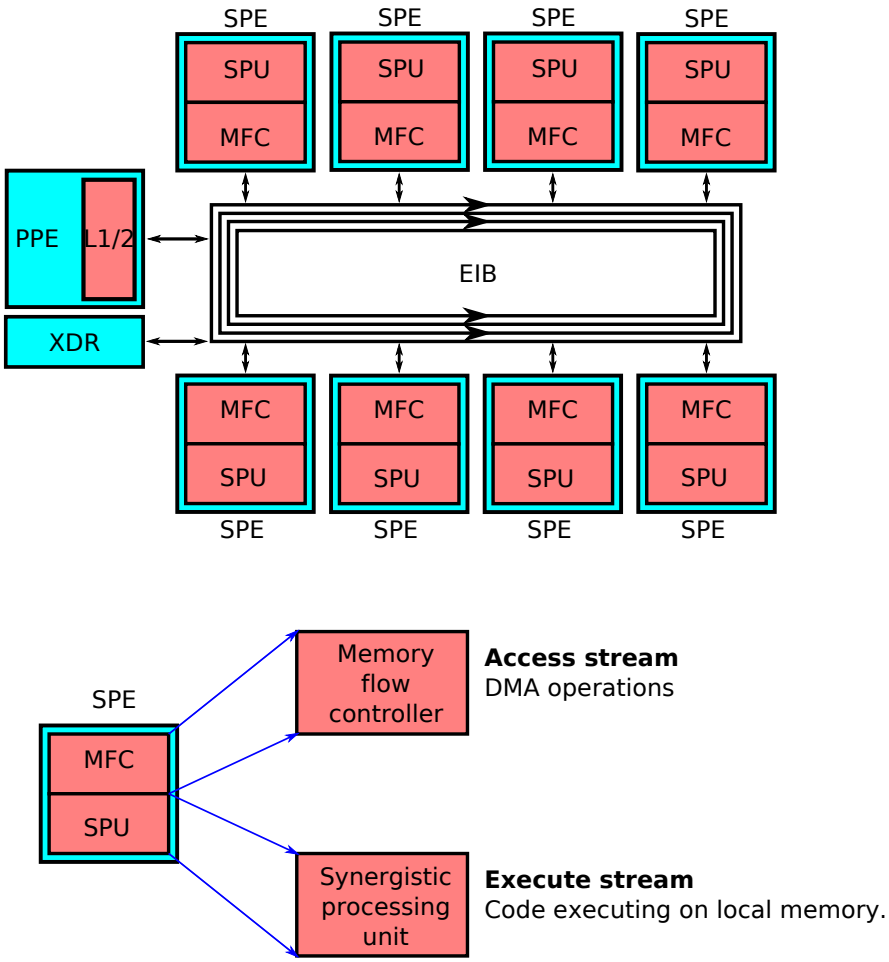


Figure 7.1: The cell processor uses a main, simplified, POWER based core and a number (8 in the blade servers and 6 in the PlayStation 3) of synergistic processing elements. Each SPE combines: a memory flow controller and a processing unit. The memory flow controller executes DMA operations in the memory access instruction stream. The processing unit executes the computation kernel on local data.

array architecture from ClearSpeed [TP05], and massively parallel architectures from NVIDIA and ATI (still habitually called graphics processing units, GPUs).

Local memory is typically cheap to access (e.g. 6 cycles on Cell), and thus is akin to an extended register file. On some architectures (e.g. on Cell and CSX), processing elements can only access local memory, and need to invoke expensive data transfer mechanisms to access remote memory. On other architectures (e.g. on GPUs), exploiting local memory is not obligatory but is essential to performance.

Efficient programs on such architectures are naturally separated into two parts:

Access
: Remote memory access to copy operands in and to copy results out (often asynchronously).

Execute
: Execution in local memory to produce the results.

The access and execute parts can be thought of as two concurrent instruction streams. For example, on Cell the execute part runs on an SPE, while the access part is serviced by its DMA engine (Memory Flow Controller), as we can see in Figure 7.1. On GPUs, programs are usually developed with a copy in procedure at the beginning of a kernel, followed by processing, followed by a copy-out procedure. The separation is reminiscent of decoupled access/execute architectures [Smi84], which run (conceptually or physically) separate access and execute instruction streams, as we discussed in Section 2.2.

Decoupled architectures use either a single original program or two programs derived (manually or automatically) from the original program. We observe, however, that deriving access and execute instruction streams from programs written in mainstream programming languages such as C/C++ is hard, in particular, because of the difficulty of dependence analysis in the presence of aliasing as illustrated in the example in Chapter 5.

7.3 Decoupled Access/Execute Specifications

We propose a declarative programming model that allows the programmer to annotate a computation kernel with both execute (Section 7.3.2) and access (Section 7.3.3) metadata. These metadata describe features of the instruction streams seen in Figure 7.1, supporting code generation and appropriate optimisations.

The *Closest-to-Mean* (CTM) filter [LG97] is an effective mechanism for reducing noise in near Gaussian environments. The closest-to-mean filter preserves edges more effectively than do linear filters and offers better performance than computationally expensive median-based filters, which require sorting operations. For a sample set of vectors V with distance metric δ, the output for the CTM filter is given by the following formula:

$$CTM(V) = \arg\min_{x \in V} \delta(x, \overline{x}),$$

where $\overline{x}$ denotes the sample average value, and $\arg\min_{x \in V}(\textit{expr})$ denotes a value of x that minimises *expr*.

The CTM filter can be applied to a digital $W \times H$ image by mapping each pixel to a CTM value for a $(2K+1) \times (2K+1)$ square sample of neighbouring pixels (for some $K > 0$).

7.3.1 Motivating Example: The Closest-to-Mean Image Filter

Figure 7.2 shows a simple closest-to-mean filter implementation in C++. The computation is split into two phases:

- A mean calculation.
- Finding the pixel in the region with the closest colour value to the mean.

While intermediate values are used, it should be obvious that we can extend these two phases into four by interleaving with necessary I/O, and indeed must do so for the algorithm to work on the Cell processor. These phases can be repeated at the pixel, block or whole-image level.

```
void ctmFilter( Array2D<rgb> &input, Array2D<rgb> &output )
{
  for( int y = 0; y < H; ++y )
  {
    for( int x = 0; x < W; ++x )
    {
      // compute mean
      rgb mean( 0.0f, 0.0f, 0.0f );
      for(int w = -K; w <= K; ++w)
      {
        for(int z = -K; z <= K; ++z)
        {
          mean += input(x+w, y+z);
        }
      }

      mean /= (2*K + 1) * (2*K + 1);

      // Compute closest to mean
      rgb closest = input(x, y);
      for(int w = -K; w <= K; ++w)
      {
        for(int z = -K; z <= K; ++z)
        {
          rgb curr = input(x+w, y+z);
          // Find closest colour value to the computed mean
          if( dist(curr, mean) < dist(closest, mean) )
            closest = curr;
        }
      }
      output(x, y) = closest;
    }
  }
}
```

Figure 7.2: Simple C++ implementation of the closest-to-mean filter. *rgb* is a simple container class with appropriate operator overloading. *Array2D* wraps a 2D data structure to deal efficiently with boundary conditions.

- Read *input* in from main memory.
- A mean calculation.
- Finding the pixel in the region with the closest colour value to the mean.
- Write *output* back to main memory.

The code in Figure 7.3 informally adds the necessary read and write operations to the kernel. The extra work involved in producing this code (and ensuring it is correct) is only the start. To achieve high performance the code must be software pipelined to gain a degree of overlapping of computation with communication, such that the DMA transfer happens asynchronously. This is similar to the optimisations that are performed on a standard CPU when performing blocking of computations such as matrix multiply, but suffers from the necessity to design the code to support such a design earlier in the process: a form of premature optimisation.

```
void ctmFilter( Array2D<rgb> &input, Array2D<rgb> &output )
{
  for( int blockY = 0; blockY < H; blockY += blockSize )
  {
    for( int blockX = 0; blockX < W; blockX += blockSize )
    {
      // DMA block (blockX, blockY) of input in
      // Wait on completion of DMA operation

      for( int y = 0; y < H; ++y )
      {
        for( int x = 0; x < W; ++x )
        {
          // compute mean
          rgb mean( 0.0f, 0.0f, 0.0f );
          for(int w = -K; w <= K; ++w)
          {
            for(int z = -K; z <= K; ++z)
            {
              mean += input(x+w, y+z);
            }
          }

          mean /= (2*K + 1) * (2*K + 1);

          // Compute closest to mean
          rgb closest = input(x, y);
          for(int w = -K; w <= K; ++w)
          {
            for(int z = -K; z <= K; ++z)
            {
              rgb curr = input(x+w, y+z);
              // Find closest colour value to the computed mean
              if( dist(curr, mean) < dist(closest, mean) )
                closest = curr;
            }
          }
          output(x, y) = closest;
        }
      }

      // DMA block (blockX, blockY) of output out
    }
  }
}
```

Figure 7.3: Blocked version of the CTM filter with DMA operations represented informally. Note the extra complexity already being added to a relatively simple kernel here even without any form of software pipelining.

```
class CTMFilter : public StreamKernel {
  Neighbourhood2D_Read inputPointSet(iterationSpace, input, K);
  Point2D_Write outputPointSet(iterationSpace, output);

  CTMFilter( IterationSpace2D &iterationSpace,
    int K, Array2D &input, Array2D &output ) {...}
  ...
  void kernel( const IterationSpace2D::element_iterator &eit )
  {
    // compute mean
    rgb mean( 0.0f, 0.0f, 0.0f );
    for(int w = -K; w <= K; ++w)
    {
      for(int z = -K; z <= K; ++z)
      {
        mean += inputPointSet(eit, w, z); // input[y+z][x+w]
      }
    }
    mean /= (2*K + 1) * (2*K + 1);

    // compute closest to mean
    rgb closest = inputPointSet(eit, 0, 0); // input[y][x]
    for(int w = -K; w <= K; ++w)
    {
      for(int z = -K; z <= K; ++z)
      {
        rgb curr = inputPointSet(eit, w, z); // input[y+z][x+w]
        if( dist(curr, mean) < dist(closest, mean) )
          closest = curr;
      }
    }
    outputPointSet(eit) = closest; // output[y][x]
  }
}
```

Figure 7.4: Æcute implementation code for the CTM filter. *inputPointSet* and *outputPointSet* are Æcute access descriptors through which the data is access.

Figure 7.4 shows a CTM filter implementation in our prototype C++ framework and Figure 7.5 demonstrates the calling of such a kernel.[1] The class method kernel closely resembles the filter's original kernel code, except that accesses to arrays have been replaced with uses of Æcute access descriptors (Section 7.4.1) which localise access to the region planned to be accessible on a given iteration of the kernel, rather than giving free access to the input and output data sets.

7.3.2 Execute Metadata

Definition 7.1 *Execute metadata for a kernel is a tuple $E = (I, R, P)$, where:*

- $I \subset \mathbb{Z}^n$ *is a finite, n-dimensional* iteration space*, for some $n > 0$;*

[1] Note that in all our examples we have compacted construction of member fields into their declarations, to save space.

```
const int K = 2; // 5x5 filter

// 2D iteration space is equivalent to a doubly nested loop:
// parallel for (int x = K; x < W-K; ++x)
//   parallel for(int y = K; y < H-K; ++y)
IterationSpace2D iterationSpace( K, W-K, K, H-K );

// 2D array descriptors
Array2D < rgb >  inputArray( W, H,  &input[0][0] );
Array2D < rgb > outputArray( W, H, &output[0][0] );

// Filter class instantiation
CTMFilter filter( iterationSpace, K, inputArray, outputArray );

// Filter invocation
filter.execute();
```

Figure 7.5: Æcute setup and invocation code for the CTM filter.

- $R \subseteq I \times I$, *is a precedence relation such that* $(i_1, i_2) \in R$ *iff iteration* i_1 *must be executed before iteration* i_2.
- P *is a* partition *of* I *into a set of non-empty, disjoint iteration subspaces.*

We have extended added the definition in comparison to the execute metdata described in Section 5.2 to support an implementation that need not perform analysis of read/write dependencies within the entire iteration space and to provide a structured blocking of the execution.

The precedence relationship R specifies constraints on the execution schedule: if iterations i_1 and i_2 are in the relationship, i_1 must be executed before i_2; otherwise, i_1 and i_2 can be executed in any order.

The partition P indicates sets of iterations that it is sensible to execute on the same processing element (e.g. a set of iterations that exhibit data reuse). In this work, we assume that the working set of each $p \in P$ fits into local memory, given a set number of buffers (e.g. two for double buffering). The programmer has the option of partitioning the iteration space manually. Alternatively he can use an simple automatic partitioning method which computes the maximum iteration subspace size needed to achieve maximal use of local memory. More complicated automatic partitioning methods could be implemented at a future date.

In the CTM filter example, the iteration space is a two-dimensional rectangle with the same dimensions as the image. When the input and output data sets are disjoint, as they are in this example, the execution schedule can be unconstrained. Such a lack of constraints allows for maximal parallelism and for greater variation in choosing the most efficient schedule for execution. The partitioning can be a tiling into rectangular $w \times h$ tiles: a pattern that maximises locality on a 2-dimensional stencil filter such as the closest-to-mean:[2]

- $I = \{(x, y) : K \leq x < W - K, K \leq y < H - K\}$
- $R = \emptyset$

[2]We assume, for simplicity, that the iteration space contains a whole number of tiles.

- $P = \big\{\{(x,y) \in I : w(i-1) \leq x - K < wi, h(j-1) \leq y - K < hj\} : 1 \leq i < (W-2K)/w, 1 \leq j < (H-2K)/h\big\}$

7.3.3 Access Metadata

Let M be a set of memory locations.

Definition 7.2 *Access metadata for a kernel is a tuple $A = (M_r, M_w)$, where:*

- $M_r : I \rightarrow \mathcal{P}(M)$ *specifies the set of memory locations $M_r(i)$ that may be* read *on iteration $i \in I$;*
- $M_w : I \rightarrow \mathcal{P}(M)$ *specifies the set of memory locations $M_w(i)$ that may be* written *on iteration $i \in I$.*

Æcute access metdata is an instance of the data mapping dependence metadata described in Section 5.2. Often, the set of memory locations accessed on a given iteration is a function of the *iteration vector* (in which case we say that the set is *indexed* by the iteration vector). The set can also include locations that are independent of the iteration vector such as scalars or arrays of constants.

In the CTM filter example, and assuming a row major addressing of a C-style array, the input and output memory locations are indexed:

- $M_r = \big\{\text{output}[y+z][x+w] : (x,y) \in I, -K \leq w, z \leq K\big\}$.
- $M_w = \big\{\text{input}[y][x] : (x,y) \in I\big\}$;

We can see in Figure 7.6 how a block of the iteration space maps onto blocks of the input and output datasets. In particular it can be seen that the input region is slightly larger than the output region. The input and output buffers seen in the diagram will be described in Section 7.4.

7.3.4 Æcute Specifications

Definition 7.3 *An Æcute specification for a kernel is a tuple $S = (A, E)$, where A and E are its access and execute metadata.*

Access metadata describes the set of memory locations that may be accessed on any given iteration. Execute metadata describes the iteration spaces and subspaces that are to be executed. The Æcute specification combines the two forms of metadata to provide an overall description of the execution of a computation kernel.

Given an iteration subspace $p \in P$ and access metadata, we can (over) approximate the set of memory locations that the subspace may read and write: $M_r(p) = \{M_r(i) : i \in P\} \in \mathcal{P}(L)$ and $M_w(p) = \{M_w(i) : i \in P\} \in \mathcal{P}(L)$. Combining execute and access metadata in the form of Æcute

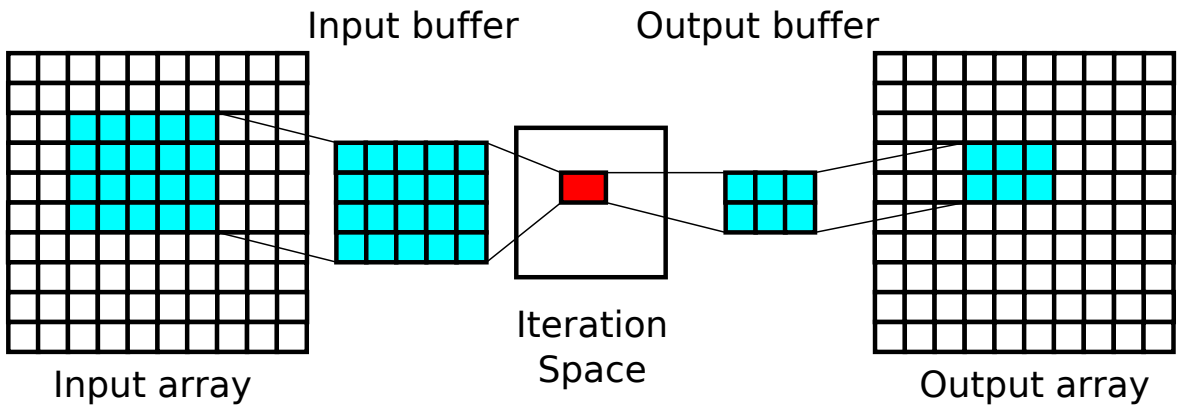


Figure 7.6: The mapping of the iteration space to data for the closest-to-mean filter. A given block of the iteration space maps via buffers of local memory to regions of the input and output data sets. Notice that the output region is the same size as the iteration space region, but the input is larger accounting for the necessary filter radius.

specifications enables powerful optimisations such as software pipelining and exploiting data reuse both within a kernel, as discussed in this chapter, and between kernels using the execution shifting and array contraction techniques discussed in Chapter 6.

In the CTM filter example, Æcute specifications can be used to trigger data prefetching of blocks of the image into local memory, to ensure that the data is delivered with ample time to spare for processing.

7.4 Æcute C++ Framework

We have developed a prototype framework to support the Æcute concept, consisting of:

- A set of C++ descriptor classes (Section 7.4.1) that represent the Æcute metadata using the C++ type system.
- A run-time system (Section 7.4.2) to process the metadata representations, performing the correct iteration schedule and correctly and efficiently moving data between the main memory and the local SPE memories.

The prototype library compiles and executes on the Cell BE architecture.

7.4.1 The Æcute C++ Classes

The formal iteration space I is specified via an instance of an IterationSpace class, which records the number of dimensions and size of each dimension, as in Figure 7.5. Practically useful timestamp functions (T) are available in the prototype in simplified form. Rather than fully flexible timestamps we support the specification of serialised dimensions on iteration spaces, for example using

the COLUMN_SERIAL directive seen in the example code. Partitioning of the iteration space is performed in the current prototype with a call to the setBlockSize function, which is parameterised with the size of a partition in each dimension of the iteration space.

A kernel class contains a main kernel method parameterised by an iterator. The kernel method is executed at each point in the iteration space. An example kernel can be seen in Figure 7.4. The iterator is used to parameterise the access descriptors, communicating the point in the iteration space such that the correct set of data elements are made available.

The memory mappings M_r and M_w are defined by *access descriptor* classes. An access descriptor object is created for each input or output associated with a kernel. These objects are invoked from the kernel code, using the kernel iterator as a parameter, to gain access to data. The prototype implementation supports the following access descriptor classes. For each member of the iteration space:

Point <mapping function>(iteration space, data structure)
: returns a single element of the data structure at the point provided by the computed mapping from the iterator.

Neighbourhood <mapping function>(iteration space, data structure, radius)
: returns a set of memory locations within a given radius of a primary address generated from the iterator which can be addressed through a passed parameter.

Buffer <mapping function>(iteration space, data structure, region size)
: returns a set of points with per point addressing into the data structure based on computing a mapping from the combintation of the iterator and an offset into the region.

In each case the primary address is computed from the iteration space coordinates provided by the Æcute iterator. To these coordinates we may apply a conversion function. The prototype framework can be extended with custom conversion functions for specific applications. In the examples of Section 7.5 we see the following conversion functions:

Project
: Performs an affine scaling of the iteration space coordinates.

Identity
: Performs no conversion.

BitRev
: Performs a bit-reversal conversion of the address passed in.

ReAddress
: Acts as a proxy for applying separate conversion functions to each dimension. In the examples we see this used parameterised with BitRev in one dimension and Identity in the other to perform a bit-reversed addressing in a single dimension only.

7.4.2 The Æcute Run-time System

The Æcute run-time system comprises two components, as we see in Figure 7.7:

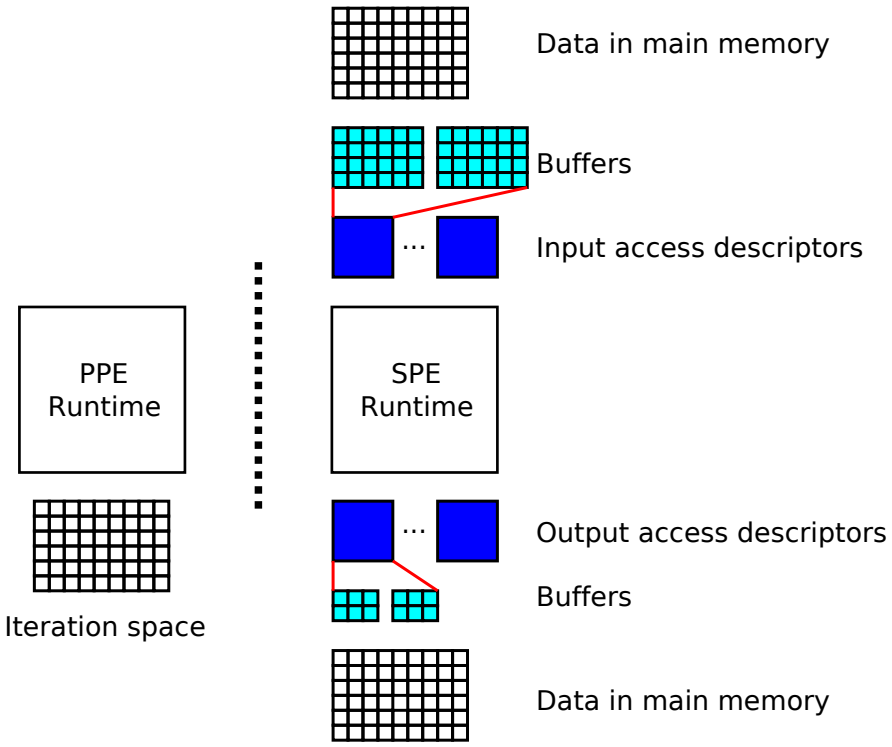


Figure 7.7: A representation of the cell Ækute framework. Associated with the PPE runtime instance is an SPE runtime for each SPE to be used. Each SPE runtime instance possesses a set of access descriptors. Each access descriptor owns a set of buffers which are used to manage the segments of the input and output arrays currently loaded into local memory.

- PPE-side support code that manages the overall execution, spawns SPE processes and synchronises computation.
- An set of SPE run-time instances executing on each SPE. This SPE-side code manages data movement into and out of the SPE's local memory and processing of data blocks.

The PPE run-time spawns an SPE run-time process on each available SPE. Given an iteration space partitioning that is specified either by the programmer or obtained automatically at run-time by querying access metadata, the PPE run-time generates a list of partition identifiers. These partition identifiers are transmitted to the SPE runtime instances, which are then responsible for executing the kernel iterations associated with the transmitted partition identifiers. Once all partitions that can be queued by SPE instances have been assigned, the PPE run-time waits for completion reports from SPE instances. More partition identifiers can later be transmitted to SPEs on completion of earlier computations until such time as all partitions have been executed. Once all partitions have been assigned, the PPE run-time waits for completion reports from all SPEs before returning control to the main program.

The SPE instance is structured using a set of access descriptor objects representing the access descriptors defined in the original C++ code. Each of these objects manages its own set of data buffers to maintain a clean code structure. On initialisation of the SPE instance the access descriptors each create an appropriate series of data buffers based on the maximum partition size they are likely to deal with: this is a constraint in the system and it is assumed that as the PPE maintains similar information, and SPEs are all identical, that the maximum buffer size will not be exceeded by the PPE scheduler.

The chosen maximum size is based on both memory availability and number of buffers to maintain: a double buffering system will maintain two buffers for each input and output, for example. As a result, at least one buffer will be present in each input and output descriptor, and possibly more if the configuration specifies this.

Each SPE run-time instance executes a wait loop on a mailbox, waiting for a notification that the PPE wishes it to process a given set of partitions. These partitions are transmitted to the SPEs as simple identifier lists, which can be looked up in a table or, in the current implementation, converted using simple arithmetic into an appropriate set of partition descriptions. The simple conversion is possible because the SPE code is constructed from the same source as the PPE code and hence minimal data structure interpretation is necessary. A partition description is similar to a full iteration space description: it defines the dimensions of the iteration block, and also the execution directions.

The partition information is processed and passed to the access descriptor code for processing. Each access descriptor selects an available input data buffer and constructs appropriate DMA operations to copy the data in from DRAM to the local memory buffer. When no buffer is available the system will wait on completion of earlier computation and, when appropriate, completion of DMA writes to free buffers. The computation kernel checks that the data it needs for a given identifier is available in each of its input buffers, and that output buffers to take the output data are empty. If input data is unavailable the run-time will block on the DMA read operations, waiting for the decoupled DMA engine to complete its operations. On completion of the computation the access descriptor objects receive information about which partition has been completed by the computation kernel and will initialise DMA write operations to clear the output buffers and free input buffers to allow the next input DMA operations to be initiated.

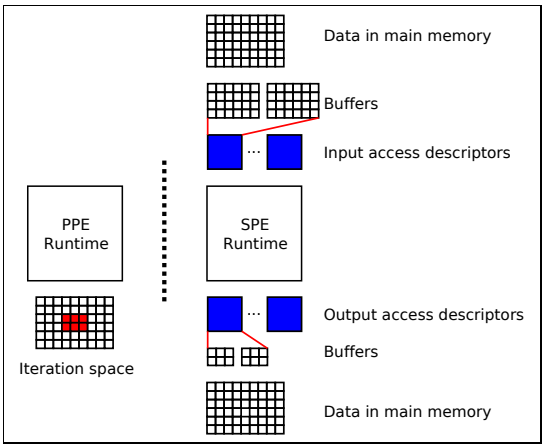
Double or triple buffering naturally occurs through this system. A fixed buffer set is managed automatically to ensure that data is always available, without additional programmer intervention. This multiple buffering enables dynamic software pipelining of the execution to improve the efficiency of memory access, decoupling the movement of data into and out of the buffers from the computation on data in the buffers and allowing computation and data movement to drift apart within synchronisation constraints. Another benefit of this decoupled approach is that the run-time system can maintain data buffers without reloading or writing back early if it detects that data useful to a later requested partition of the iteration space is already resident in a buffer.

In the combined PPE/SPE runtime system we can see the following stages of behaviour, visualised in Figure 7.8:

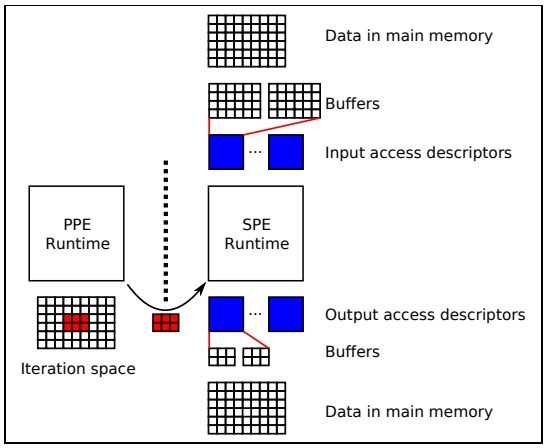
1. The PPE runtime moves onto the next block in the iteration space. This may or may not be a single block, for efficiency reasons blocks can be batched.

2. The block identifier is passed to the appropriate SPE runtime which translates the identifier into the appropriate block coordinates. These block coordinates are passed to the access descriptor objects which convert them into the sets of input or output data values needed for the block to execute.

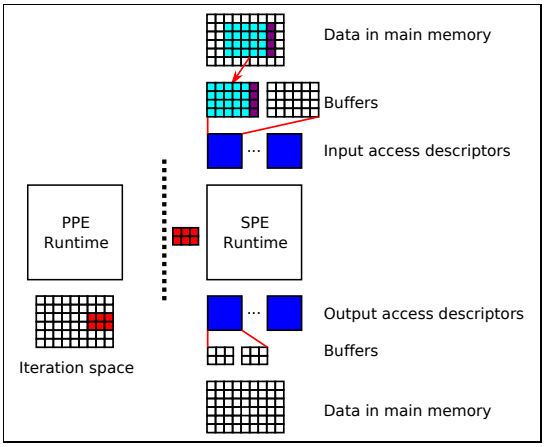
3. The SPE runtime initiates DMA transfers to load the appropriate data for the specified block into one of its input buffers and to configure output buffers to be ready to receive output data. SPE execution can block at this point if output buffers are not yet available, waiting for output DMA operations to complete.



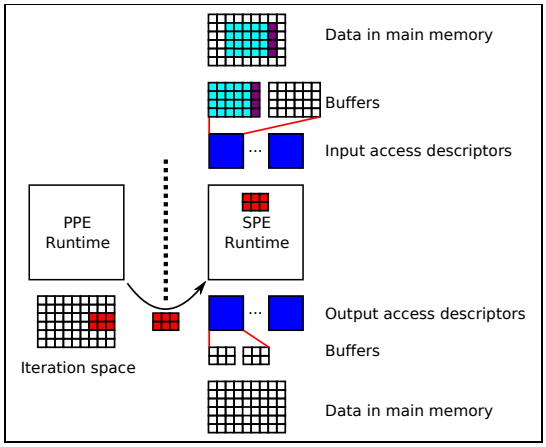
(a) PPE moves to next iteration space block.



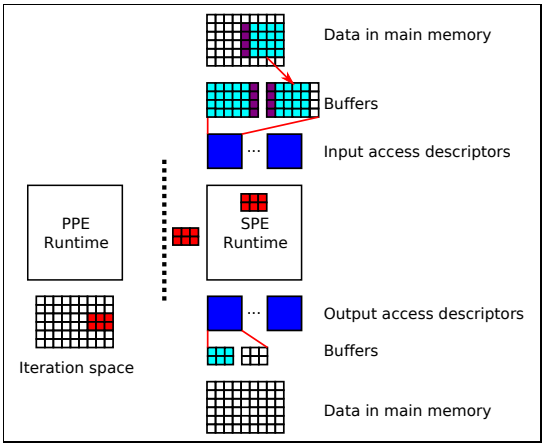
(b) SPE receives specification of iteration space block from PPE.



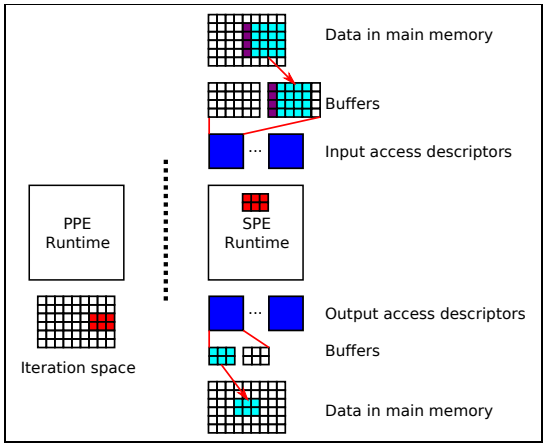
(c) SPE initiates input DMA transfers from locations calculated from the iteration space block.



(d) On DMA completion the SPE starts executing the iteration space block. Meanwhile the PPE may be passing another block to the SPE's queue.



(e) DMA transfers initiated for the next iteration space block in the queue.



(f) On completion of a block, DMA writes are initiated. The next block will be processed.

Figure 7.8: The sequence of operations enacted by the runtime framework to process a block of iterations.

4. The SPE runtime checks for completion of the DMA transfers. At the same time the PPE might have transmitted details of further blocks for the SPE to process. Processing of the block can begin when input DMA transfers have completed. Local output memory is assumed to have been reserved by this point.

5. The later blocks transmitted by the PPE are queued. At an appropriate point the SPE runtime initialises DMA transfers for the queued blocks, while still processing the earlier part of the schedule.

6. On completion of the block's execution, the runtime initiates DMA transfers to write output buffers to main memory. Input buffers can be freed at this point, ready to be used to accept data for future blocks. The SPE can begin processing the next block in its queue.

The cycle will repeat with the PPE continually transmitting blocks to the SPEs until all partitions of the iteration space have been processed.

7.5 Further Examples

7.5.1 Matrix-vector multiply

A matrix-vector multiply $y = Ax$ can be implemented as a two-dimensional iteration space of the dimensions of matrix A. The two vectors, x and y, are one-dimensional, so to obtain the vector indices from the iteration space coordinates we *project* the coordinates onto a single dimension. The dimension we project depends on whether we are dealing with the input or output vector, as we can see in Figure 7.9. The sizes of the input and output buffers depend on how the iteration space will be projected onto the data and are filled from appropriate regions of the input and output datasets.

We can formalise the matrix-vector multiple example in the terms described in Section 7.3.4. As discussed, we tile the iteration space. In this case we assume that local memory can hold the working set for a regular tile of $h \times w$ iterations.

Æcute specification $S = ((M_r, M_w), (I, T, P))$:

- $I = \{(i, j) : 0 \leq i < H, 0 \leq j < W\}$
- $R = \{((i, j), (i, k)) : 0 \leq i < H, 0 \leq j < k < W\}$
- $M_r(i, j) = \{A[i][j], x[j]\}$
- $M_w(i, j) = \{y[i]\}$
- $P = \big\{\{(i, j) \in I : h(k-1) \leq i < hk, w(l-1) \leq j < wl, \} :$
  $1 \leq k < H/h, 1 \leq l < W/w\big\}$

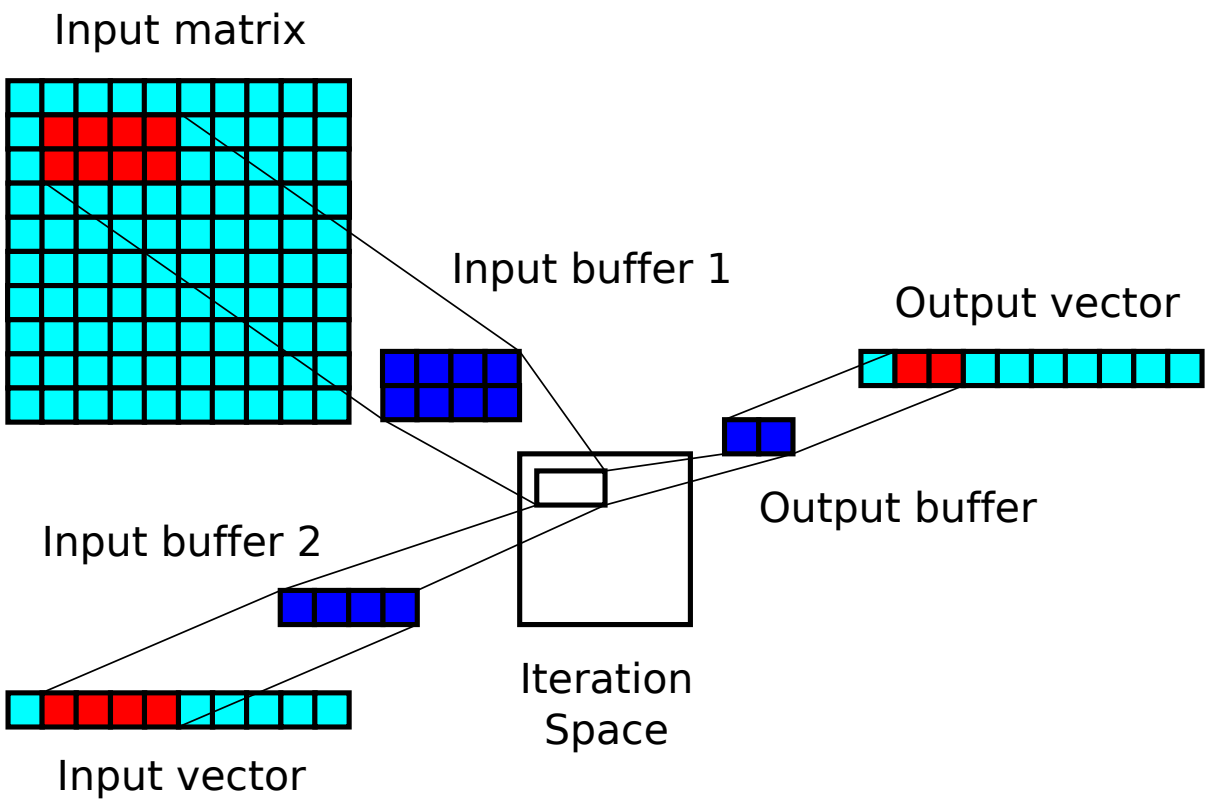


Figure 7.9: Mapping of the iteration space onto data sets for the matrix vector multiply example. The region read from the input matrix maps directly onto the set of iterations in the iteration space. The input and output vectors' regions are obtained through projection of the iteration space region onto one or other dimension, obtaining differently sized data regions.

The precedence relation indicates that the loop indexed by i can be executed in parallel, and that the loop indexed by j is serial. This serialisation removes the requirement for the PPE to perform accumulation of partial results. If the += operator could be guaranteed to be associative then the j loop could also be specified as parallel, by setting $R = \emptyset$. In this case a serialisation, if present, would not affect the correctness of the computation, but might affect efficiency.

Æcute code The kernel operates over the input matrix and vector and the output vector. Note that we specify the column dimension to be serial, which preserves the order of multiply-accumulate operations and matches the above formalisation.

```
IterationSpace2D iterationSpace(W, H, COLUMN_SERIAL);
Array2D < float >  inMatrix(H, W, pInMatrix);
Array1D < float >  inVector(W, pInVector);
Array1D < float > outVector(H, pOutVector);
MatrixVectorMul matvec(iterationSpace, inMatrix, inVector, outVector);
// Matrix-vector multiply invocation
matvec.execute();
```

The associated MatrixVectorMul kernel class is roughly as follows:

```
class MatrixVectorMul : public StreamKernel
{
  Point2D_Read inputMatrix( iterationSpace, inMatrix);
  Point2D_Read < Project2D1D< 1, 0 >  >
```

```
    inputVector( iterationSpace, inVector );
  Point2D_Write < Project2D1D< 0, 1 >  >
    outputVector( iterationSpace, outVector );

  MatrixVectorMul( IterationSpace2D iterationSpace,
    Array2D inMatrix, Array1D inVector, Array1D outVector )
  {
    ...
  }

  void kernel( const IterationSpace2D::element_iterator &eit )
  {
    outputVector( eit ) += inputVector( eit ) * inputMatrix( eit );
  }
};
```

where Project2D1D projects a 2D iteration space coordinate onto a 1D iteration space. For example, Project2D1D<0,1> projects (i, j) onto j as $(i \times 0, 1 \times j)$. Not in particular that the projections onto the two vectors are in different dimensions, arising from the fact that one vector will correspond with the horizontal dimension and one with the vertical dimension of the matrix during the multiplication process.

7.5.2 Bit-reversal

Many radix-2 FFT algorithms start or end their processing with data permuted in bit-reversed order. To complete (or begin) the computation, the data must be rearranged into the correct order. The reordering is typically performed by performing a bit-reversed data copy (often abbreviated, if inaccurately, to *bit-reversal*).

We assume that the subroutine reads an array $x[\,]$ of $N = 2^n$ elements and writes these elements into an array $y[\,]$ of N elements in bit-reversed order, such that x and y do not overlap. That is to say that an element of the source array at the index written in binary as $b_0 \ldots b_{n-1}$, is copied to the target array at the index with reversed digits $b_{n-1} \ldots b_0$. The function $\sigma_n(i)$ reversing bits of index i having n bits can be implemented efficiently as [War02]:

```
unsigned int reverse_bits(unsigned int n, unsigned int i)
{
    i = (i & 0x55555555) << 1 | (i >> 1) & 0x55555555;
    i = (i & 0x33333333) << 2 | (i >> 2) & 0x33333333;
    i = (i & 0x0f0f0f0f) << 4 | (i >> 4) & 0x0f0f0f0f;
    i = (i<<24) | ((i & 0xff00)<<8) | ((i>>8) & 0xff00) | (i>>24);

    return (i >> (32 - n));
}
```

This sequence of bit-wise operations and shifts implies that $y[\,]$ will contain a permutation of $x[\,]$ and hence assignments can be performed in any order. Few programmers will recognise this fact from looking at the code. One cannot expect that a compiler will recognise this either.

In addition to obscuring parallelism, bit-reversed indexing is unfriendly to hardware-managed caches: starting from a certain array size $N = 2^n$, each access to $y[\,]$ results in a cache miss. To avoid caching problems inherent in bit-reversals of large arrays, the best approach, used by Carter and Gatlin in

the so-called Cache Optimal BitReverse Algorithm (COBRA) [CG98], introduces a cache-resident buffer.

If the buffer holds B^2 elements, the iteration space is partitioned into N/B^2 independent subspaces. For each subspace, B source blocks of B elements each are copied into the buffer, permuted *in place*, and then copied out from the buffer into B target blocks of B elements each. In this fashion memory accesses of 1 element, widely spread throughout memory, may be aggregated into fewer accesses of B elements, far more efficiently utilising the memory system.

The permute kernel of the COBRA algorithm can easily be off-loaded to operate on blocks in the local memory of a co-process. This introduces a data movement challenge, where we must implement the copy-in and copy-out loops, where the copy-out loop uses a non-affine mapping to memory.

According to our experience, the complication of implementing data movement code, in particular data movement code that supports double buffering, can take as much time or longer than implementing the kernel itself. A desired alternative, then, is to derive this data movement activity from Ækute specifications.

Ækute specification $S = \big((M_r, M_w), (I, R, P)\big)$:

- $I = \{t : 0 \leq t < N/B^2\}$
- $R = \emptyset$;
- $P = \big\{\{t\} : t \in I\big\}$
- $M_r(t) = \{x[u.t.v] : t \in I, 0 \leq u < B, 0 \leq v < B\}$
- $M_w(t) = \{y[u.\sigma_n(t).v] : t \in I, 0 \leq u < B, 0 \leq v < B\}$.

The iteration space in this case is defined in terms of the larger subspaces mentioned before rather than the full elementwise iteration space of the naive bit-reversed copy operation. Each iteration refers to the computation on a $B \times B$ block of data. The precedence function, R, indicates that the one-dimensional iteration space is unordered because that the computation is a pure permutation. In this case each partition is a single element of the iteration space. We can see the mapping to data in Figure 7.10 including the two-dimensional treatment of the input and output datasets that eases consideration of the addressing scheme. The blocks operated on at each iteration space point are disjoint and fairly large and so do not require grouping into larger partitions for efficiency. In the Ækute code below we see how the programmer can manually set the partition size to match the code to the above specification.

Ækute code As a result of the $B \times B$ blocking, it is natural to think of the input and output arrays of N elements as two-dimensional, having N/B rows of B elements each. Entire rows are read and written by the computation kernel, and these operations (as well as the kernel itself) can be vectorised for efficiency.

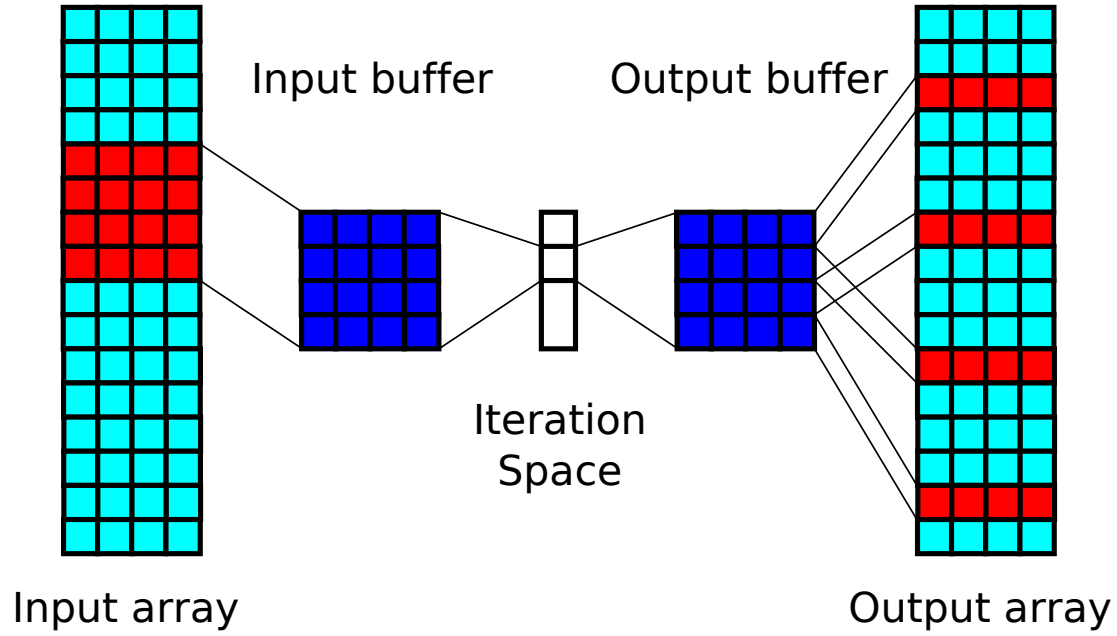


Figure 7.10: Mapping of iteration space to data for the bit-reversed data copy. A point in the iteration space represents a block that will be processed in the local memory according to COBRA from an input buffer to an output buffer. The input buffer maps neatly to a two-dimensional representation of the input array. The output buffer maps line-wise using bit-reversed addressing to a two-dimensional representation of the output array.

```
IterationSpace1D iterationSpace(N/(B*B));
Array2D <float> inputData(B, N/B,  pInputData);
Array2D <float> outputData(B, N/B, pOutputData);
BitReversal bitrev(iterationSpace, inputData, outputData);
bitrev.iterationSpace.setBlockSize( 1 );
// Bit-reversal invocation
bitrev.execute();
```

We iterate over independent subspaces $t \in I$:

1. Copying rows numbered as $u.t, 0 \leq u < B$, from DRAM into the local buffer.
2. Applying the permutation kernel on the $B \times B$ block.
3. Copying rows numbered as $u.\sigma_n(t), 0 \leq u < B$, from the local buffer back to DRAM.

```
class BitReversal : public StreamKernel< BitReversal >
{
  Buffer2D_Read
    input(iterationSpace, inputData, B);
  Buffer2D_Write < ReAddress2D< Identity, BitRev > >
    output(iterationSpace, outputData, B);

  BitReversal( IterationSpace 2D iterationSpace,
    Array2D input, Array2D output ) {...}

  // Do in place permutation
  void kernel( const IterationSpace2D::element_iterator &eit )
  {
```

```
    ...
  }
};
```

ReAddress2D takes the (i, j) coordinate formed from the iteration space point and the buffer coordinates and applies the specified pair of functions to i and j respectively. BitRev reverses bits of the j value to correctly address the destination for the row by calling the reverse_bits function, which represents $\sigma_n()$ as shown earlier, on the appropriate bits of the memory address.

7.6 Experimental Evaluation

We use a 3.2GHz Cell processor on a Sony PlayStation 3 console, running Fedora Linux (2.6.23.17-88.fc7), with IBM Cell SDK 2.1. We compiled the benchmark programs using the highest optimisation settings in the IBM compiler, and executed them on all six SPEs that are available to the programmer on the Cell processor in the PlayStation 3.

We evaluate the benchmarks described in Section 7.3 and Section 7.5. The core code relating to these experiments is listed in Appendix A.2.

7.6.1 Implementation

To evaluate our prototype Ækute framework we compare against alternative implementations that use hand-written DMA transfers and a software-based SPE data cache. The software cache allows remote data to be accessed in a familiar way, which simulates a hardware CPU cache, to enable quick porting of code to run on SPEs. In these experiments we use the standard software cache implementation provided by IBM with Cell SDK 2.1 [Wri08]. We use a 4-way set associative cache with default "write-back" write policy and "round-robin" replacement policy, and vary the number of cache sets and line size on an application-specific basis. It is possible that tuning these parameters for specific examples would improve peak performance, but we do not envisage improvements that change the competitive balance of the evaluation and a high degree of example-specific tuning is infeasible in most cases.

The kernel code is essentially the same in the matrix/vector and bit-reverse cases, with minor changes to support the use of Ækute framework classes and software cache functions. In the closest-to-mean filter the hand-written code uses a rolling implementation that offers higher performance than the block-based approach used by the Ækute prototype. The current implementation of the Ækute prototype cannot partially replace data and hence must reload boundary data. The hand written versions do not have this restriction and can largely reuse buffers giving lower overheads.

7.6.2 Closest-to-mean filter (Section 7.3.1)

Figure 7.11 shows execution time normalised to code with hand-written DMA transfers. We consider two neighbourhood diameters N: 15 and 63, and two image sizes $D \times D$ where D is: 256 and 1024. These represent increasing computation workload. We also consider three different iteration space

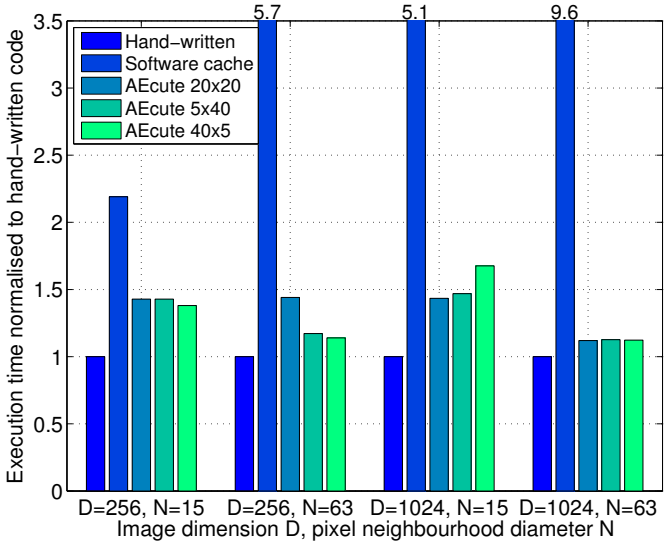


Figure 7.11: Closest-to-mean filter.

tile sizes: 20×20 (default square size, which is calculated automatically under the constraint that the tile footprint must fit into local memory); 5×40; and 40×5.

For $D = 256$ and $N = 15$, the best Æcute code performs within 40% of hand-written code; for $N = 63$, within 15%: the increased workload amortises the overhead of interpreting Æcute specifications at run time. In contrast, the overhead of using the software cache grows with increasing neighbourhood size (which perhaps can be remedied by tuning the cache parameters). For $D = 1024$ and $N = 63$, the overhead drops to 12%.

We observe that no tile size was universally best. Given the simplicity of varying tile sizes, the best tile size could be found by iterative search, possibly even during the execution of the application. In contrast, it is usually more difficult to adapt code safely with tile sizes integrated more tightly with the kernel code itself.

Blocked DMA transfers improve the efficiency of memory traffic and enable both hand-written and Æcute code perform far better than code using the software cache. These are supported naturally by the partitioning and automated buffering in the Æcute system, and implemented directly in the hand-written code,

7.6.3 Matrix-vector multiply (Section 7.5.1)

For this example we hand-vectorised the entire block computation for efficiency: unfortunately compilers can still not be entirely trusted to achieve this, though this is a situation that should improve with time. The hand-written and software-cache-based code are similarly vectorised for fair comparison. While the Æcute model looks promising for automatic vectorisation, it is important that the programmer retains full control over kernel optimisations should automatic optimisations fail. While vectorisation may be simplified through the use of iteration spaces and vectors across clearly defined iterations, it is difficult in the general case. Even in these cases, optimisation of an individual kernel is best left either to the compiler or to the programmer when full control is required.

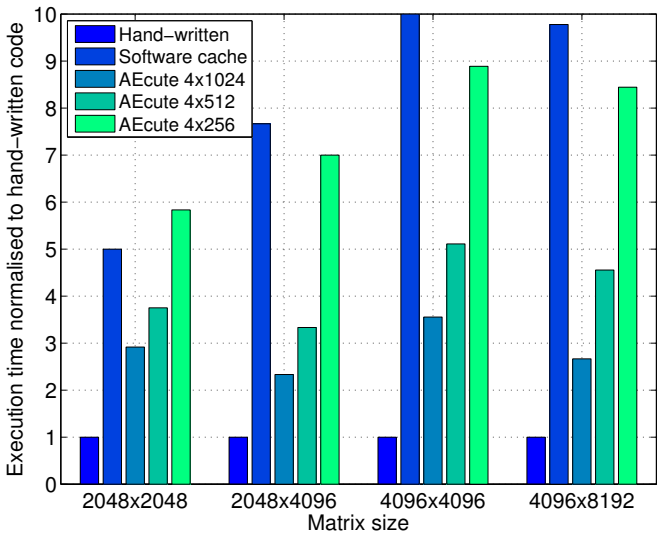


Figure 7.12: Matrix-vector multiply normalised to execution time of hand written code.

Figure 7.12 shows normalised execution time for various matrix sizes. The best tile size is 2–3 times slower than hand-written code, but considerably faster than the software cache implementation. The run-time overhead associated with the Æcute framework is significant for this example due to the low arithmetic density of the matrix-vector multiply operation. The hand-written implementation requires less SPE-PPE communication: the SPEs are able to compute results entirely independently. This fully independent execution plan is a method of implementation that requires a high degree of knowledge of the computation being performed whereas in realistic cases the PPE would have to do some processing to divide the iteration space, and is unlikely to know that the load balance will be easily left to the SPEs.

7.6.4 Bit-reversal (Section 7.5.2)

Figure 7.13 plots data copy throughput against the bitwidth of the array index. We see smooth scaling of performance with the size of the dataset. In addition, the performance of the Æcute implementation tracks that of the hand-written implementation with a near-constant scaling. In this case, while remote memory accesses are inherently non-contiguous due to bit-reversed indexing in the algorithm, the system can construct efficient DMA list transfers from Æcute specifications.

7.7 Conclusion and Future Work

In this chapter we have presented the concept of decoupled Access/Execute specifications and demonstrated their convenience, flexibility and efficiency on three benchmark examples. Our Æcute implementation automates the data-movement element of the accelerator programming task and therefore offers scope for improving programmer productivity and easing code maintenance. The blocking of DMA transfers and construction of DMA lists enabled by separating the memory access from computation results in more efficient memory traffic.

We are looking into extending this work in several ways.

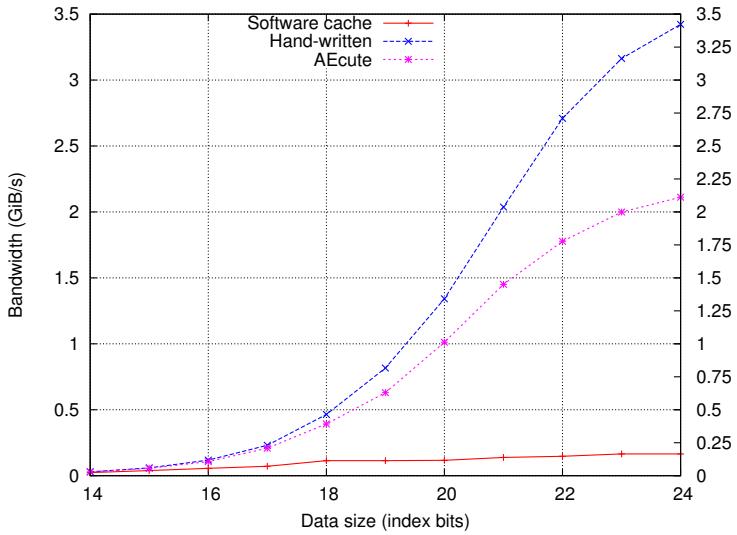


Figure 7.13: Bit-reversal.

First, Æcute specifications may be thought of at a level of compiler intermediate representation rather than a high-level programming language. Thus, we plan to investigate 'front-ends' that will derive Æcute specifications from higher-level abstractions, in particular, from the polyhedral model [Gri04] in combination with programmer hints, possibly through the use of an analysis tool suite. In addition, we wish to investigate 'back-ends' for other accelerator architectures, such as GPUs: a hint of which will be discussed in Chapter 8.

Second, we plan to integrate Æcute specifications into a compiler, to reduce both the overhead of interpreting Æcute specifications at run-time and the size of generated data-movement code, which must be minimised to conserve precious local memory. As in Gaster's streaming extension to OpenMP [Gas08], compiler support can be layered on top of an extended and streamlined version of the current Æcute classes. In this fashion an application could be made to work correctly with or without compiler support. Compiler support would also allow us to combine metadata from series of kernels, generating fused code in the fashion discussed in Chapter 6.

Third, we plan to extend the expressivity of Æcute metadata to handle a larger set of kernels, associated with full-scale applications. The current Æcute implementation supports only a limited range of partitioning options and mappings to data. We can extend this by using a hierarchical partitioning and improving the search options, e.g. for locality. In addition, we wish to support unstructured mesh based computations, such as fluid flow. For unstructured data we need to extend the memory read and write sets to support indirection while maintaining decoupling of access and execute.

One of the issues with the current system lies in its use of C++. C++ eases the development process itself, at least in comparison to a lower level language such as C, by allowing easier code reuse is efficient and clearly defined data structures. Unfortunately, C++ code is unsuited to execution on many embedded devices. Both IBM's compiler and GCC are incapable of adequately minimising code size, and as a result C++ code compiled for the SPE quickly consumes the available local memory - memory which is shared for data and code.

Compiler support for AEcute classes would ease this problem, and seems vital anyway to allow compilation for GPUs where C++ is completely unsupported. By generating complicated decoupled C code from the simpler, easier to read, C++ we can obtain the efficiency of C without the maintenance

issues of the low-level programming model.

7.8 Summary

In this chapter we described a C++ programming model based on the concept of decoupled access/execute metadata: a form of indexed dependence metadata. This model allows us to implement computation kernels that consider only the local data, and separately map that data to the memory system. Localisation of mapping code enables the generation and optimisation of data movement operations, to complement the inter-kernel optimisations discussed in Chapter 6. In the next chapter, coming full circle from our original look at GPUs in Chapter 4 we see how the latest GPUs can benefit from this metadata treatment.

Chapter 8

Decoupled access execute software as a solution for the modern GPU

8.1 Introduction

In this chapter we discuss the difficulties of development for highly parallel SIMD devices: in particular the current generation of GPUs. To this end we describe implementations of several versions of a simple image processing filter. We evaluate these implementations on Intel and AMD multi-core systems equipped with NVIDIA graphics cards using both two vector programming models: NVIDIA's CUDA [NVI] and Intel's Streaming SIMD Extensions (SSE) [Int]. Using a thorough design-space exploration we see the tradeoffs necessary when creating high-performance implementation.

Our experimental results demonstrate that efficiently implementing an algorithm to execute on commodity parallel hardware requires careful tuning to match the hardware characteristics. Even similar systems show a wide variation in performance when varying low-level implementation details such as iteration space tiling and data layout. While such manual tuning is possible, it is not practical. The number of versions to write and maintain grows with the number of target architectures. For applications consisting of multiple kernels, or those with a large variety of target architectures, such development and maintenance can consume a considerable amount of developer time to the point that it is infeasible.

Our findings motivate the need for tools and techniques that decouple a high-level algorithm description from low-level mapping, tuning and code generation. We believe that the issues that make such mapping and tuning difficult can be reduced by allowing the programmer to describe both execution constraints and memory-access patterns using a high level representation.

This chapter is published in part at SAAHPC 2009 [HLDK09a] and at HPPC 2009 [HLDK09b].

8.2 The mean filter

We consider an image mean filter, which is a separable convolution operation[1] for which the output pixel at position (x, y) in the vertical case is given by the formula

$$\mathbf{O}_{x,y} = \frac{1}{D} \sum_{k=0}^{D-1} \mathbf{I}_{x,y+k}, \tag{8.1}$$

where:

- $\mathbf{I}$ is a $W \times H$ grey-scale input image;
- $\mathbf{O}$ is a $W \times (H - D)$ grey-scale output image;
- D is the *diameter* of the filter, i.e. the number of input pixels over which the mean is computed (typically, $D \ll H$);
- $0 \leq x < W, 0 \leq y < H - D$.

Mean filtering is a simple technique for smoothing images, for example to reduce noise [FPWW09].

8.2.1 Unlimited parallelism

The naive example of the mean filter seen in Figure 8.1 allows unlimited parallelism, but also performs an unnecessarily large amount of computation to produce each result.

In this implementation, let N be the number of output pixels: $N = width \times (height - diameter)$. The straightforward C version of the algorithm seen in Figure 8.1 performs N writes to $\mathbf{O}$, $N \times diameter$ reads from $\mathbf{input}$, and $\Theta(N \times diameter)$ arithmetic operations.

Note that in this version of the algorithm, the filter outputs can be computed in parallel as specified by equation (8.1), since the x and y loops carry no dependences. We see this in Figure 8.3a.

8.2.2 Scalable parallelism

An alternative approach is to serialise the implementation. If serialised naively, the implementation in Figure 8.1 performs redundant computation. Observe that $\mathbf{O}_{y,x} = \mathbf{O}_{y-1,x} + \frac{1}{D}\big(\mathbf{I}_{y+diameter-1,x} - \mathbf{I}_{y-1,x}\big)$, for $y \geq 1$. An implementation based on this formula performs considerably less computation than would be performed in Section 8.2.1.

The price for this improved efficiency in the vertical dimension is that the y loop carries a dependence, hence parallelism is limited to processing W columns in parallel as we see in Figure 8.3b.

To achieve both serial efficiency and parallel scalability, we *strip* the filter in the y dimension as in Figure 8.2, where up to T outputs in the same strip are computed serially.

[1] A convolution filter that produces the same result as a single two-dimensional convolution or as two separate one-dimentional convolutions with derived filters.

```
// Parameters
const int width;
const int height;
const int diameter;

// Input and output images
const float *input;
float * output;

...

// for each column
for(int x = 0; x < width; ++x) {
  // for each row
  for(int y = 0; y < height-diamter; ++y) {
    float sum = 0.0f;
    for(int k = 0; k < diameter; ++k)
      sum += input[k*W + x];
    output[y*width + x] = sum / (float)diameter;
  }
}
```

Figure 8.1: Mean filter implementation that is parallel in both x and y dimensions.

```
// for each column
for(int x = 0; x < width; ++x) {
  // for each strip of rows
  for(int y0 = 0; y0 < height-diameter; y0+=T) {
    float sum = 0.0f;
    for(int k = 0; k < diameter; ++k)
      sum += input[(y0+k)*width + x];
    output[y0*width + x] = sum / (float)diameter;

    for(int dy = 1; dy < min(T,height-diameter-y0); ++dy) {
      int y = y0 + dy;
      sum -= input[(y-1)*width + x];
      sum += input[(y-1+diameter)*width + x];
      output[y*width + x] = sum / (float)diameter;
    }
  }
}
```

Figure 8.2: Mean filter implementation that is both efficient and scalable.

Since the x and $y0$ loops carry no dependences, parallelism is scalable to processing $\lceil N/T \rceil$ strips. This implementation performs $\Theta(N + (N \times diameter)/T)$ reads from **input** and the same number of arithmetic operations. In effect we end up with a blocking in the vertical dimension. Due to the parallelism in the horizontal dimension we can create 2D blocks to spread the parallelism efficiently between cores, as we see in Figure 8.3c. Note that since the order of computation defined by the filter equation is undefined, the implementations discussed in Section 8.2.1 and Section 8.2.2 are functionally, if not arithmetically, equivalent.

8.2.3 Vector parallelism

The nature of the 2D vertical filter allows serialisation, and the associated efficiency gain, only in the vertical dimension. As a result, the horizontal dimension maintains full parallelism and can be vectorised, as in Figure 8.3d, leading to utilisation of the parallelism inherent in both the Intel's SSE and NVIDIA's CUDA architectures.

We can see either architecture as a vector design. CUDA has 16-element physical vectors (literally 8 SIMD units pipelined over two cycles) on current architectures. These relatively short SIMD units execute much longer logical vectors, created as thread blocks. SSE vectors are four 32-bit elements in length in current architectures, both logically and physically.

The blocking presented in Figure 8.2 and Section 8.2.2 can efficiently map to the logical vectors in the horizontal dimension, balancing the parallelism on the chosen architecture as necessary.

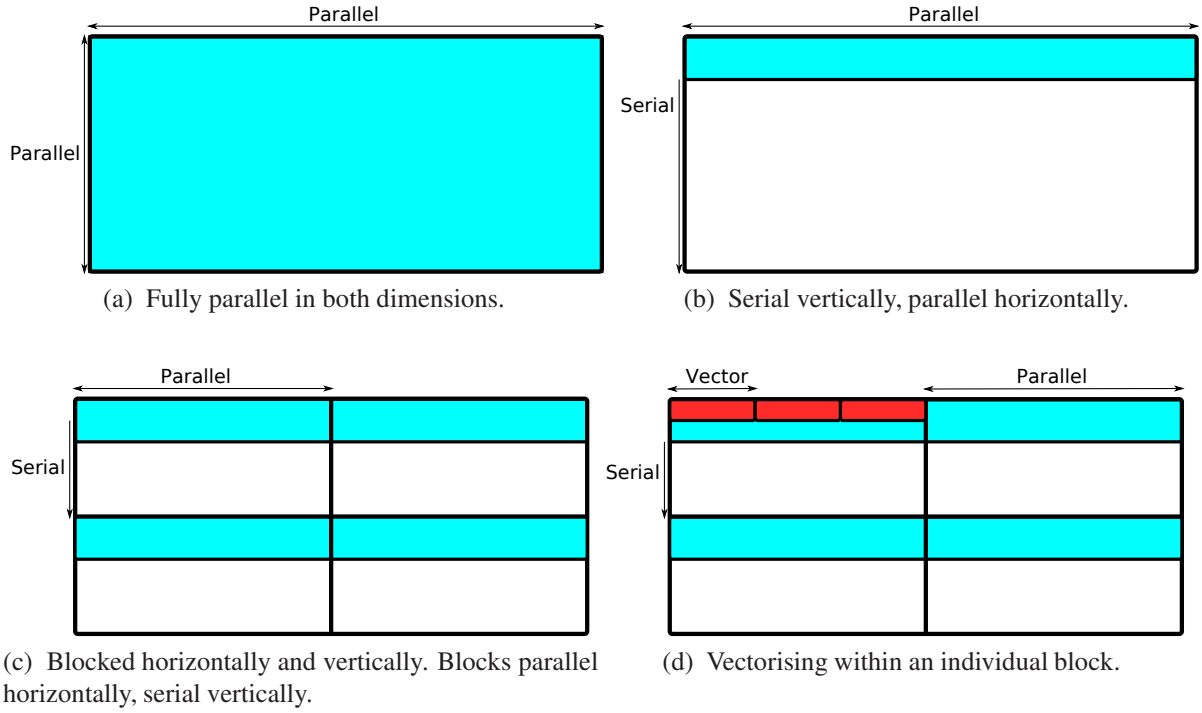
However, as we shall see in Section 8.3, the same is not true of the horizontal filter. In the horizontal case the vector would have to perform a vertical read as only the vertical dimension is parallel. Such reads are highly inefficient on most vector architectures. SSE offers no such capabilities, instead requiring a sequence of scalar reads. CUDA devices divide the read similarly in hardware. As a result, different tradeoffs are necessary for the vertical and horizontal filters.

8.3 Tradeoffs for the vertical filter

8.3.1 Tradeoffs for CUDA architectures

The CUDA programming model abstracts the SIMD nature of NVIDIA's GPU hardware and was briefly discussed in Section 2.2.4. Unlike Intel SSE, CUDA kernels do not consist of vector data types and instructions. Instead, the code of an executable kernel is expressed for a single SIMD element, termed a *thread*. Large logical *blocks* of threads execute together on hardware implemented (at least logically) as short SIMD vectors called *warps*. Warps execute in lock-step; synchronisation can be performed efficiently between warps within a single block. Distinct blocks execute independently during the lifetime of a kernel call with no supported means of synchronisation.

The current high-end architectures, the GeForce GTX280 and Tesla C1060, consist of 30 processing cores, each of which operates on 8-way SIMD vectors. This results in 240 computation elements (somewhat misleadingly also termed *cores*). Warps are pushed through these 8-way vectors over multiple clock cycles.



(a) Fully parallel in both dimensions.

(b) Serial vertically, parallel horizontally.

(c) Blocked horizontally and vertically. Blocks parallel horizontally, serial vertically.

(d) Vectorising within an individual block.

Figure 8.3: Logical parallelism and vectorisation.

Like any vector architecture, and indeed to a lesser extent any cache-based architecture, memory accesses are more efficient when coordinated across a vector. As a result CUDA threads must be designed to coordinate by accessing addresses within a single 128-byte aligned memory unit. On earlier architectures this limitation was even more strict, requiring the 128-byte reads to be aligned with a significant throughput falloff whenever this was not the case.

When discussing the partitioning of an iteration space to be executed by blocks of CUDA threads, we use the following abbreviations:

- WPTX/WPTY – work per thread in X/Y dimension;
- TPBX/TPBY – number of threads per block in X/Y dimension;
- WPBX/WPBY – work per block in X/Y dimension. We have WPBX=WPTX$\times$TPBX and WPBY=WPTY$\times$TPBY.

2D mapping

To maintain throughput a CUDA-based architecture requires a high degree of threading to cover memory latency rather than relying on prefetching as in the considered CPUs. The result is that a large number of vectors are created, each larger than the physical vector length of the architecture. The CUDA model treats logical vectors as groups of cooperating threads, or thread blocks, which are then initialised in a grid layout. The natural mapping, based on the GPUs graphics origins, is to use a two-dimensional grid which maps almost directly to the two-dimensional dataset. We can create such a grid by instantiating a three-dimensional vector (*dim3*) as follows:

```
const dim3 grid( ceil(width/WPBX),  ceil((height-diameter)/WPBY) );
```

This 2D grid consists both of 2D blocks of threads and 2D blocks of data processed by each block: the latter being a multiple of the work per thread and number of threads in a given dimension.

That is, the number of blocks of threads in the X and Y dimensions is determined as the ceiling of the height or width divided by the amount of work per block in that dimension. The CUDA declarations of *block* and *grid* configure the GPU threads as desired. An individual computation element computes its initial coordinates based on its block and thread identifiers, and calls a generic filter method, *vfilter*. Each execution of *vfilter* is essentially restricted to process only those pixels within the working set of the computation element. The call to *vfilter* is guarded to ensure that it is only applied to coordinates which lie within the image:

```
const unsigned int x0 = WPBX*blockIdx.x + threadIdx.x;
const unsigned int y0 = WPBY*blockIdx.y + threadIdx.y;

if(x0 < width && y0 < height-diameter)
  vfilter(g_in, g_out, x0, y0);
```

Note that this computation is performed per vector element. Figure 8.4a graphically illustrates the block 2D, grid 2D partitioning. Each thread block deals with a rectangle of image pixels. Although this provides a natural mapping of GPU threads to image locations, with straightforward computation of starting coordinates, there is likely to be redundancy. As Figure 8.4a shows, significant portions of blocks covering the far right and bottom of the image may be assigned coordinates outside of the image bounds if the block dimensions do not exactly divide the image dimensions. These threads do no useful work, and portions of physical hardware vectors in particular, do significant amounts of unused work thanks to the nature of SIMD computation.

Basic 1D mapping – improving thread utilisation

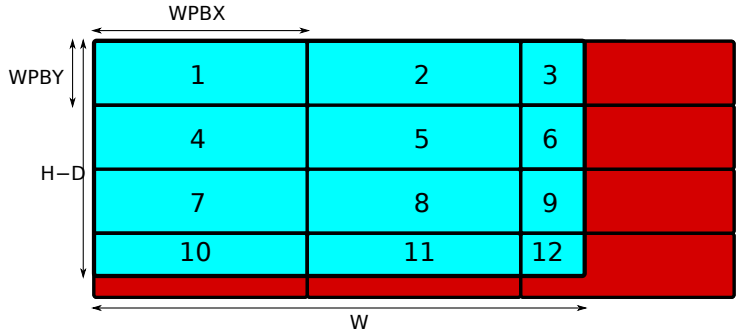
An alternative arrangement is to map the vectors to the data in a one-dimensional fashion. The goal of one-dimensional mapping is to eliminate the redundancy associated with the two-dimensional mapping discussed above. The simplest approach is to generate a one-dimensional block layout directly in CUDA. The long flat one-dimensional vector structure can be wrapped around the image, removing the redundancy at the right hand edge:

```
const int NT = ceil(width/WPTX) * ceil((height-diameter)/WPTY);
const dim3 grid( ceil((N * T)/TPBX) );
```

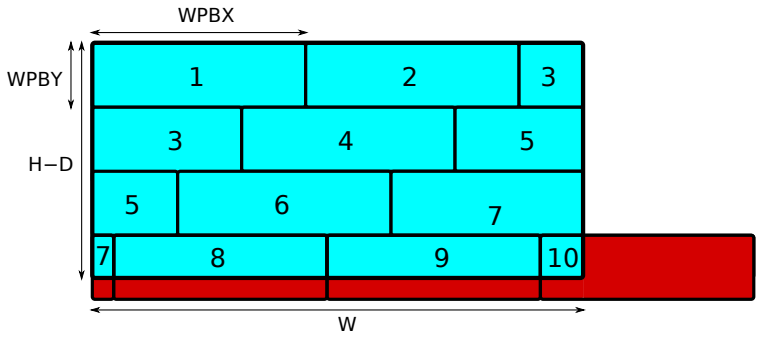
As before, an individual thread calls *vfilter* having computed its starting coordinates from its block (*blockIdx*) and thread (*threadIdx*) ids. The block and thread ids are both vectors that store the address of a computation block or thread within a block in terms of its three dimensions.

```
const unsigned int tid = TPBX * blockIdx.x + threadIdx.x;
const unsigned int x0  = (tid % ceil(width/WPTX)) * WPTX;
const unsigned int y0  = (tid / ceil(width/WPTX)) * WPTY;

if(x0 < width && y0 < height-diameter)
  vfilter(g_in, g_out, x0, y0);
```

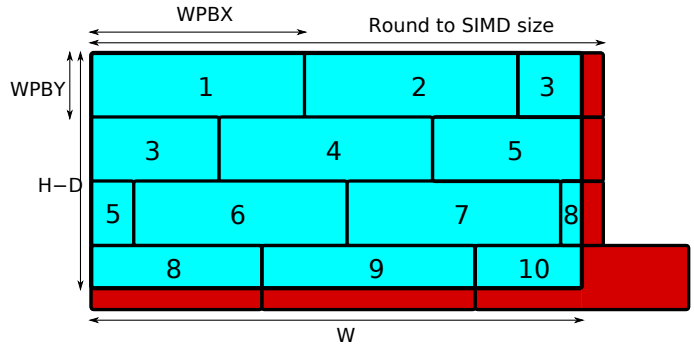
The coordinate calculations in the one-dimensional kernel are more intricate than in the earlier case, since they involve converting a block and thread id into an “absolute” id, then recovering a 2D co-ordinate pair from this absolute id. The expectation is that under the right circumstances, this added overhead will be reduced by the more efficient use of threading resources. The added efficiency then



(a) A 2D grid loses efficiency from inactive threads off the right of the image.



(b) A 1D grid wraps to make most efficient use of its threads, at the cost of more complicated addressing code. Additionally, if the image width is not a multiple of an alignment constraint, some rows become unaligned.



(c) Alignment problems can be rectified by making threads wrap on the alignment constraint, at the cost of slightly reduced thread utilisation. This ensures that each new row of threads starts correctly aligned to the appropriate vector size.

Figure 8.4: Thread mapping strategies for the vertical filter. Different thread mapping strategies result in different utilisation of the compute resources. Light and dark regions of blocks denote utilised and non-utilised threads, respectively.

allows the hardware to maintain a high level of utilisation for a higher proportion of the execution time.

Figure 8.4b illustrates the way threads cover image coordinates using the one-dimensional partitioning. Horizontal redundancy is eliminated by wrapping. There will still be some redundancy across the bottom of the image and where the last block cannot wrap.

1D mapping – maintaining alignment

While the one-dimensional partitioning eliminates much of the redundancy associated with the two-dimensional mapping, the wrapping can introduce unaligned memory accesses if the image width is not a multiple of the SIMD size. In these cases the first line of reads would align correctly, but the following line would start reading part way through a vector. The next line of the image will then suffer from unaligned reads. This is the case even if the data itself is aligned, a feature guaranteed by the `cudaMallocPitch` function call. Once alignment is broken it will continue to be broken across the next image row until, by coincidence, wrapping of the image happens to result in repairing alignment. Each of these unaligned reads requires multiple memory accesses to satisfy and hence operates at a low level of efficiency.

Adjusting the address calculation slightly and reducing thread utilisation by a small amount at the boundary allows us to correct for this problem. This is illustrated in Figure 8.4c. Unfortunately, the additional complexity of the addressing code negatively effects code maintenance.

8.3.2 Tradeoffs for CPUs with SSE support

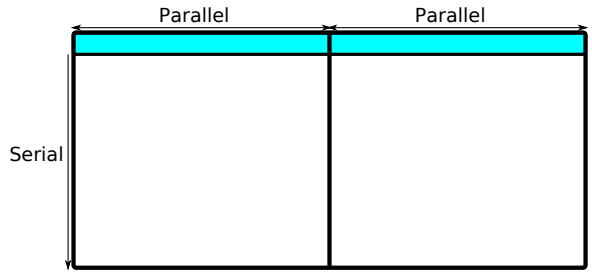
As we saw in Section 8.2.2, serialisation can increase computational efficiency by reducing the number of required arithmetic operations and memory reads required to compute each output value. This is as true on a general purpose CPU as on a CUDA-based device, if not more so. However, different tradeoffs apply. To achieve efficient memory access the outer x loop must be blocked and interchanged with the vertical loop, as we see in Figure 8.5. The innermost computation is always reading consecutive addresses in the horizontal direction, to gain optimum use of the cache and prefetching support provided by the hardware. This new inner loop can be vectorised as in Figure 8.3d using SSE instructions, allowing it to exploit parallelism locally while globally executing serially to gain maximum performance from the CPU design.

As we can see in Figure 8.6 it possible to split the computation among threads vertically or horizontally (or both). Vertical blocking (Figure 8.6b) minimises thread overhead, allowing vectorisation of a longer inner loop with fewer branches to deal with. Horizontal blocking (Figure 8.6b) allows the full vertical serialisation that we hoped to gain from the serial implementation of the algorithm. The small number of threads on current SSE-based systems makes both of these approaches feasible, without the need to block in both directions. Unfortunately, deciding which loop to parallelise adds another variable to the system. In both cases the discussed loop interchange is possible and high memory throughput attainable.

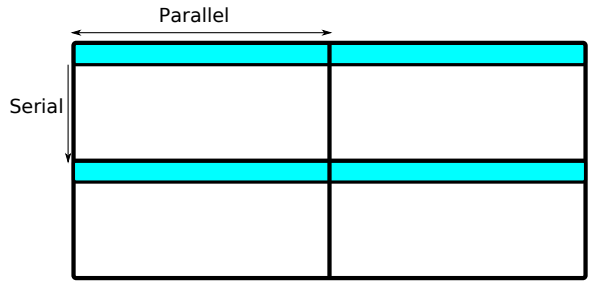
There is no advantage to using more blocks than the number of parallel threads because load balancing is not a concern of the algorithm and a high degree of parallelism to cover memory latency is unnecessary.

```
// for each column
for(int x = 0; x < width; x += blockSize) {
  // for each strip of rows
  for(int y0 = 0; y0 < height-diameter; y0+=T) {
    float sum[blockSize] = {0.0f...0.0f};
    for(int k = 0; k < diameter; ++k)
      for( int bx = 0; bx < blockSize; ++bx )
        sum[b] += input[(y0+k)*width + x + bx];

    for( int bx = 0; bx < blockSize; ++bx )
      output[y0*width + x + bx] = sum[bx] / (float)diameter;

    for(int dy = 1; dy < min(T,height-diameter-y0); ++dy) {
      int y = y0 + dy;
      for( int bx = 0; bx < blockSize; ++bx ) {
        sum[bx] -= input[(y-1)*width + x + bx];
        sum[bx] += input[(y-1+diameter)*width + x + bx];
        output[y*width + x + bx] = sum[bx] / (float)diameter;
      }
    }
  }
}
```

Figure 8.5: Vertical mean filter implementation. The inner bx loops can be vectorised and the Intel compiler does a fairly good job of this.



(a) Horizontal thread blocking maximises serialisation efficiency at the expense of loop overhead.



(b) Vertical thread blocking minimises loop overhead at the expense of a reduction in serialisation efficiency.

Figure 8.6: The SSE implementations of the vertical filter can be blocked into threads vertically or horizontally. The choice between the two offers an additional tradeoff for mapping.

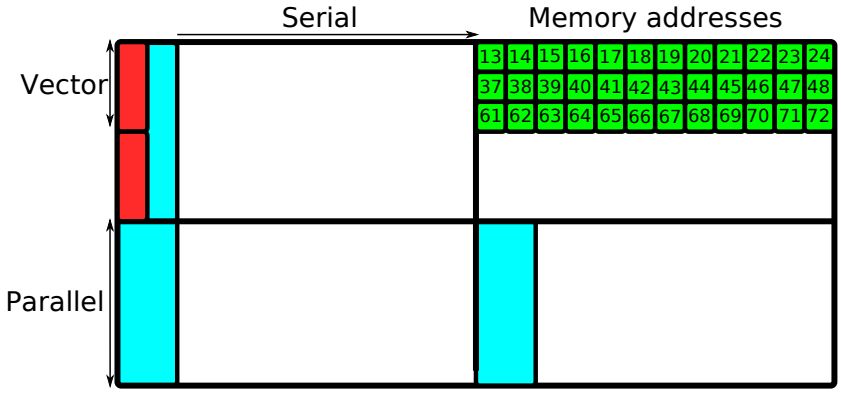


Figure 8.7: The horizontal mean filter naively vectorised. Note that any vector memory-access would involve locations that stride through memory. We can see in the region of the diagram laid out as memory addresses that one vector read might take addresses 13, 37 and 61.

8.4 Tradeoffs for the horizontal filter

For any vector architecture the horizontal filter can be fully vectorised in its completely parallel form. Although each stage of a vectorised read from a filter region might be unaligned without careful tuning. Unfortunately, we know that the fully parallel version of the mean filter is not computationally efficient. The serial version suffers from a severe limitation that we can see in Figure 8.7. While we can run a vector horizontally through the data, the flow dependence means that reading vector data horizontally cannot contribute to the serial computation. Reading vertically suffers from reading gathered data with a large stride through memory. With SSE this is not possible: the reads have to be performed as scalars. For CUDA it is highly inefficient, turning into 16 separate, largely uncached, reads. Other solutions are necessary to obtain high performance on the horizontal filter.

8.4.1 Tradeoffs for CUDA architectures

The current CUDA designs read large 16-way SIMD vectors from memory. However, with little caching, a 16x performance inefficiency is likely when arranging that vector as a vertical read.

A common approach in writing CUDA applications is to read horizontal vectors and transpose in shared memory. In this case we have a challenge. There is 16kBytes of shared memory available per core. For parallelism we wish to run more than one thread block on a given core so we can assume a maximum memory allocation of 8kBytes per thread block. If we read a 16×16 block of memory into shared memory we require 1kByte of shared memory to perform that staging. Unfortunately there are two problems with this:

- To maintain efficiency we require at least one full warp of 32 threads. When performing the horizontal serial summation through the data these threads arrange vertically, therefore we need $n \times 32$ elements in our staging array. It is possible that more threads help with efficiency, so using 64-threads might be better. However, the larger number of threads severely limits the horizontal size of the region we can read as the vertical size must increase with the thread count.

- The parallel convolution stage allows us to read a given region of kernel-size elements in width, offering us reasonable kernel sizes that still fit in 8k of memory. The serial stage of the computation is complicated by having to read data to add to the running sum some horizontal distance in advance of the value needed to subtract from the running sum

These two factors combined complicate the algorithm for processing the data. A side-effect of this addressing complication is that the algorithm requires reading data using the CUDA-threads in one pattern, and then processing the data using a completely different pattern. Thinking about and debugging such algorithms becomes complicated. In Figure 8.8 we can see the algorithm step by step.

The main alternative approach is to transpose the entire dataset on the way in and transpose back again on the way out. Due to the high level of memory bandwidth on the GPU and the use of shared memory to efficiently transpose relatively large blocks of data directly, full dataset transposition is a relatively efficient operation. If this transpose allows a high degree of utilisation of parallelism during the vertical filter which must then be applied to the data the overhead of the transpose operation can be negated by the efficiency gain from performing an efficient vertical filter on the data. In addition, many more complicated filters contain sequences of mean filters. A common high-performance approximation for the Gaussian filter used in image processing is to perform multiple vertical followed by multiple horizontal mean filters. These sequences of horizontal filters would require only a single transpose at each end, thereby amortising the overhead and making the transposing filter a net winner overall.

8.4.2 Tradeoffs for CPUs with SSE support

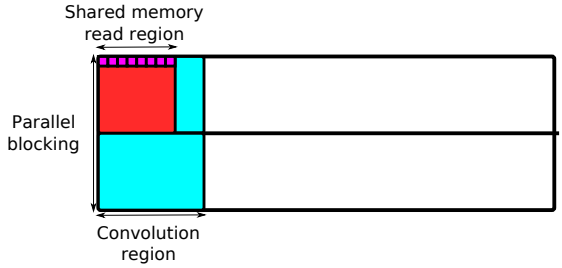
For the horizontal filter on the CPU we have three primary options:

- Naive streaming accumulation loops.
- Transposing the data set followed by a vertical filter.
- In-place transposition of data at the vector level.

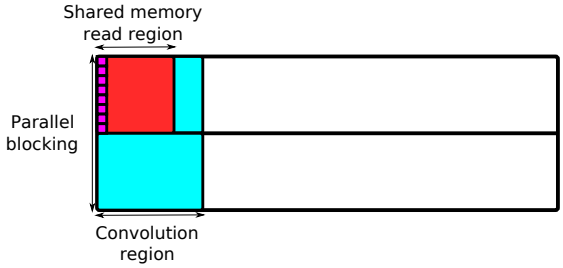
Naive streaming takes an inner loop with a loop-carried dependency and streams through the data in a fashion that is very efficient for the CPU's cache hierarchy. Intel's compiler manages to optimise the loop effectively, improving performance substantially. However, there is a loop-carried dependence in the loop. As a result, hand vectorisation is not feasible. Careful optimisation such as unroll-and-jam of the whole loop nest and relying on the cache prefetcher is possible.

Transposing the data set allows us to utilise the efficient vertical filter code already discussed. The transposition removes the dependence and offers us the ability to optimise again. However, it depends on a high-performance transpose operation and so does not perform well unless a sequence of filters is being performed or there is some other reason to keep the data in a transposed state.

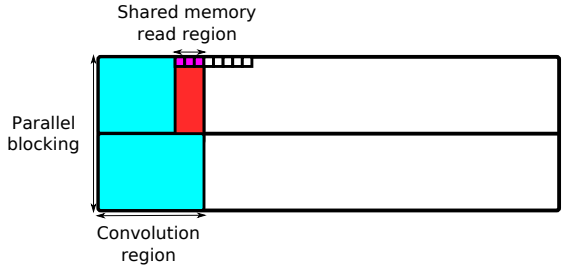
In-place transposition can be seen in Figure 8.9. We read the data as horizontally oriented vectors. These vectors are not efficient for element-wise summation because we would need to read individual vector elements. We can transpose a set of four vectors making a 4×4 square entirely in registers, giving a small vertical filter which can then sum into a vertical summation register. Unfortunately this



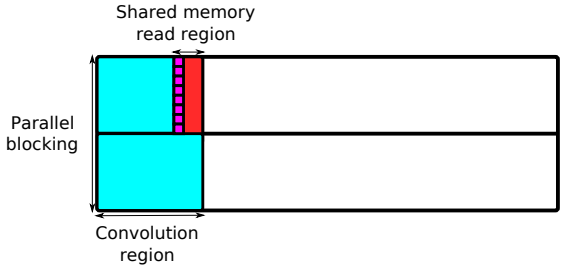
(a) During the convolution stage we read the first part of the filter to the specified vector length. This read should be as wide as possible to maintain memory bus efficiency. The whole region will be read into shared memory in this way, top to bottom.



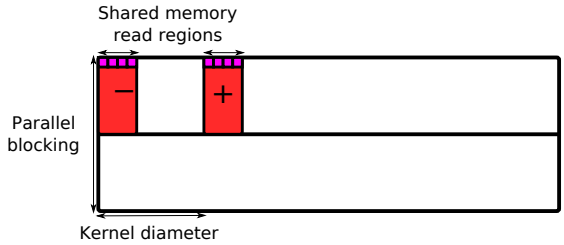
(b) We then orient the threads vertically in the shared memory buffer and perform the computation moving horizontally through the region.



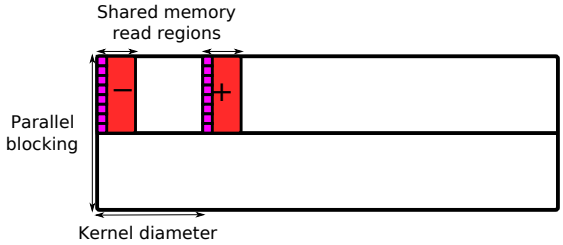
(c) We continue by loading from the next section of the convolution region into the same shared memory buffer. By the end of the convolution region we may not have enough data to satisfy a full vector read so some threads will be wasted.



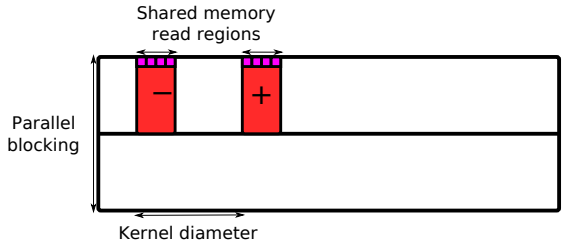
(d) As before, processing continues orienting the threads vertically in the region.



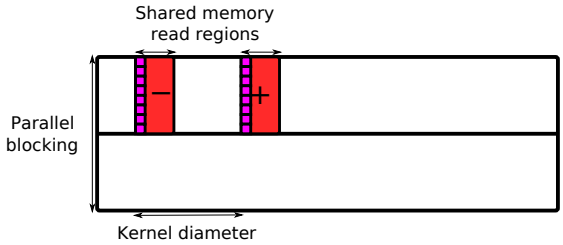
(e) The threads are split over two regions: one at the left hand side of the kernel, and one at the right. We need to read data ahead of the current point in the computation to perform both the additions and subtractions in the running filter.



(f) Computation is performed by each thread performing both an add and a subtract and moving horizontally through both regions together.



(g) The reading continues with the next block in the serialised region of the computation.



(h) The threads continue to add data appropriately, outputting into a shared memory buffer the size of the left-hand read block, which is written out at the end of the block.

Figure 8.8: The four stages involved in a horizontal mean filter algorithm that transposes using shared memory.

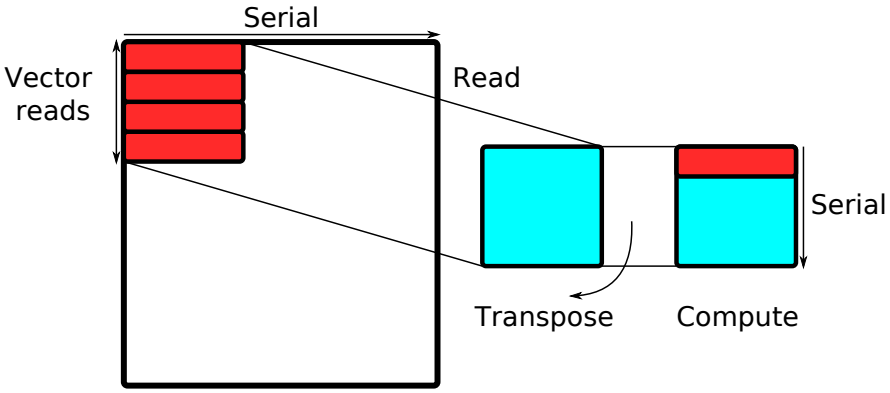


Figure 8.9: Horizontal SSE computations can be performed by reading four short vectors, each of which efficiently reads four consecutive addresses. These four vectors are transposed using a sequence of eight vector operations that interleave the original four into four transposed vectors. These transposed vectors now apply to the intermediate sum variable in traditional vector form.

suffers from two problems. First, it takes 8 operations to perform the transpose. The total number of operations is substantial and overrides the possible advantage from vectorisation. with larger vectors and a big register file this approach might perform well. Additionally, writing the data out requires transposing a single vector and writing it out vertically, or more likely performing the same transpose on four output sum vectors: another 8 operations.

8.5 Performance results

We provide experimental results obtained on:

- An eight-core (dual-socket quad-core) Intel Xeon E5420 at 2.5GHz with 16GiB RAM.
- A quad-core AMD Phenom 9650 at 2.3GHz with 8GiB RAM.
- A dual-core Intel Core 2 Duo E8400 at 3GHz with 2GiB RAM.

All systems are equipped with NVIDIA GTX280 graphics cards, though only the best set of results is quoted as the difference is only in driver overhead. For the SSE results the Intel compiler version 10.1 is used with "-O3 -xHost -fast" optimisation settings. The CUDA results are compiled with the CUDA SDK version 2.2 and GCC 4.2.4 with the "-O3" optimisation setting. We measure the kernel execution time only and record the best throughput out of 50 runs.

8.5.1 CUDA results

CUDA vertical mean filter

Figure 8.10, Figure 8.11, Figure 8.12 and Figure 8.13 present results obtained on the GTX 280 card using the blocking strategies discussed in Section 8.3.1. Parameter TPBX/TPBY records the number

of threads per block in the X/Y dimension.

In all the experiments, we fix the number of threads per block at 128 (128×1), as we nearly achieve the peak memory efficiency with this setting: $\approx$ 10 Gpixel/s $\times$ 4 bytes/pixel $\times$ (2 reads + 1 write) = 120 GB/s (close to the bandwidth of aligned copy on this card). Thus, WPBX = 128 and WPBY = T.

Figure 8.10 shows that the 1D and 2D grid versions are similar in throughput when applied to a 5120×3200 image, where 5120 is a multiple of 128 pixels. The throughput is below 800 Mpixel/s when each thread produces a single pixel. It climbs fast with increasing serial efficiency, achieving (by the 1D grid version) the peak throughput of 9.89 Gpixel/s when $T = 355$. Finally, throughput declines with decreasing parallelism.

When applied to a 5121×3200 image, however, the 2D grid version only achieves 7.02 Gpixel/s, as shown by the bottom line in Figure 8.11. While we allocate memory using the `cudaMallocPitch` function, which pads the image to a multiple of 16 pixels to enable global memory access coalescing (5136 pixels in this case), such allocation leads to DRAM partition conflicts. We remedy the conflicts by manually padding the image to a multiple of 32, 64 and 128. Since the results of padding to a multiple of 64 and 128 are barely distinguishable, we fix the image padding at a multiple of 64 (5184 pixels) for all subsequent experiments.

Figure 8.12 shows that the 1D grid mapping that maximises thread utilisation by wrapping on 5121 pixels only achieves 6.00 Gpixel/s, whilst wrapping on the image padding of 5184 pixels performs worse than wrapping on the warp size multiple of 5152 pixels.

In Figure 8.13 we see a summary of the results from Figure 8.11 and Figure 8.12. For the misaligned image padded to 5184 pixels, the 1D grid version wrapped on 5152 pixels achieves 9.58 Gpixel/s at $T = 396$, whilst the 2D grid version achieves only 9.06 Gpixel/s at $T = 409$; thus, the 1D grid version performs 6% better than the 2D grid one. The 1D version that is uncorrected for alignment performs poorly in comparison.

CUDA horizontal mean filter

As we saw in Section 8.4.1 we can implement a separate transpose or transpose in shared memory. In Figure 8.14 we see the naive horizontal filter, which reads vertically and hence inefficiently from memory, with three transposing versions of the filter.

The first transposing version uses a single transpose of the entire dataset on each end of the computation and then uses the optimal vertical filter to compute the mean. We must remember that this version has the advantage of allowing us to optimise away the transpose operations in the right circumstances.

In comparison we have two versions that transpose in shared memory using different numbers of threads. The number of threads affects the vertical read size, which in turn affects the amount we can read horizontally in the transposition. We can see that a 32-thread shared memory transpose peaks far higher than the other versions.

The overall performance of the horizontal filter is far lower than that of the vertical due to the inefficiency resulting from transposition. We will see below that this is not the case with the CPU version of the code. This is a clear demonstration of how optimisations must be carefully chosen to match the caching structure of a given architecture.

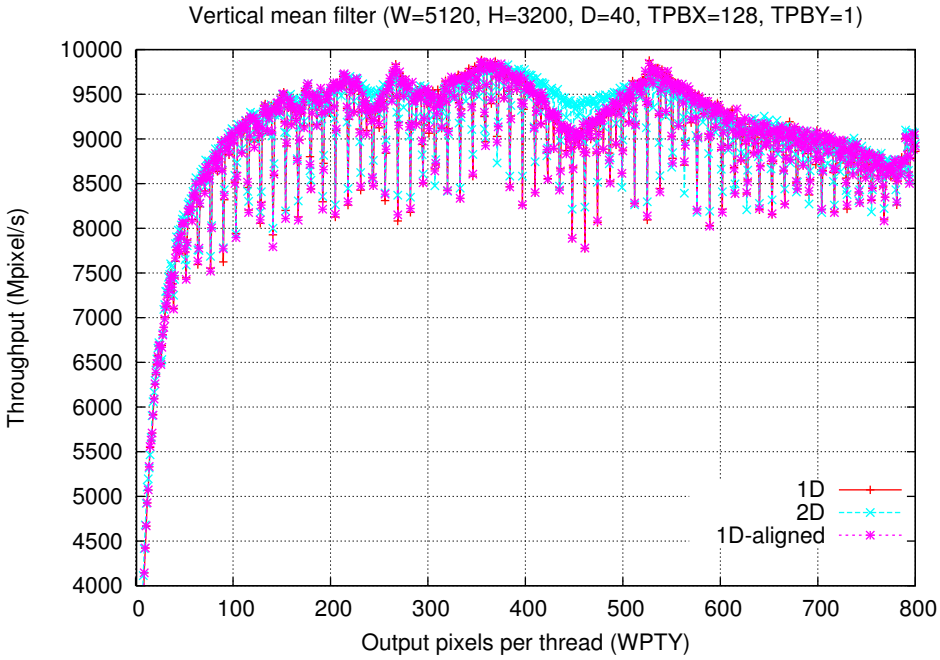


Figure 8.10: A 5120×3200 image. 2D, 1D and 1D-aligned thread to data mappings.

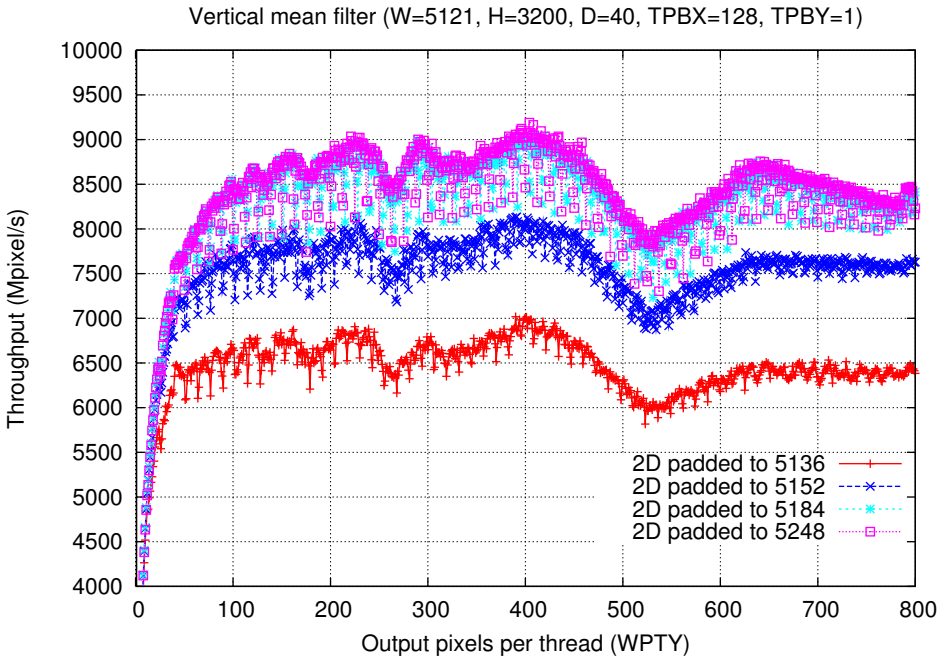


Figure 8.11: A 5121×3200 image. 2D mapping of computation to data but with the data aligned to 16, 32, 64, and 128 pixels.

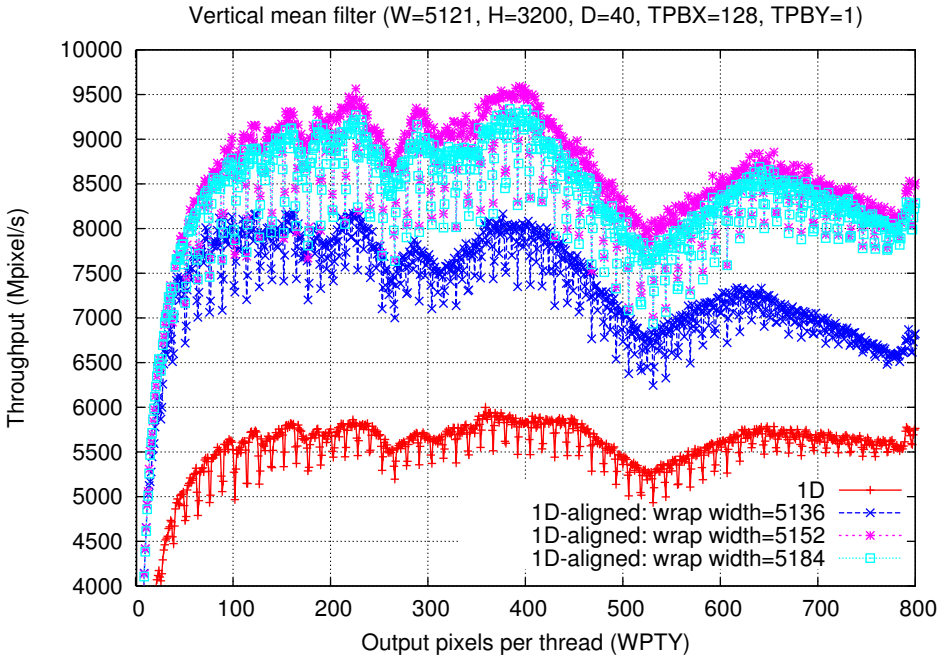


Figure 8.12: A 5121×3200 image. 1D mapping of computation to data, with the data alignment fixed to 5184 pixels, a multiple of the optimal alignment of 64 pixels. In this case the computation is aligned to 16, 32 and 64 pixels with appropriate wrapping overhead.

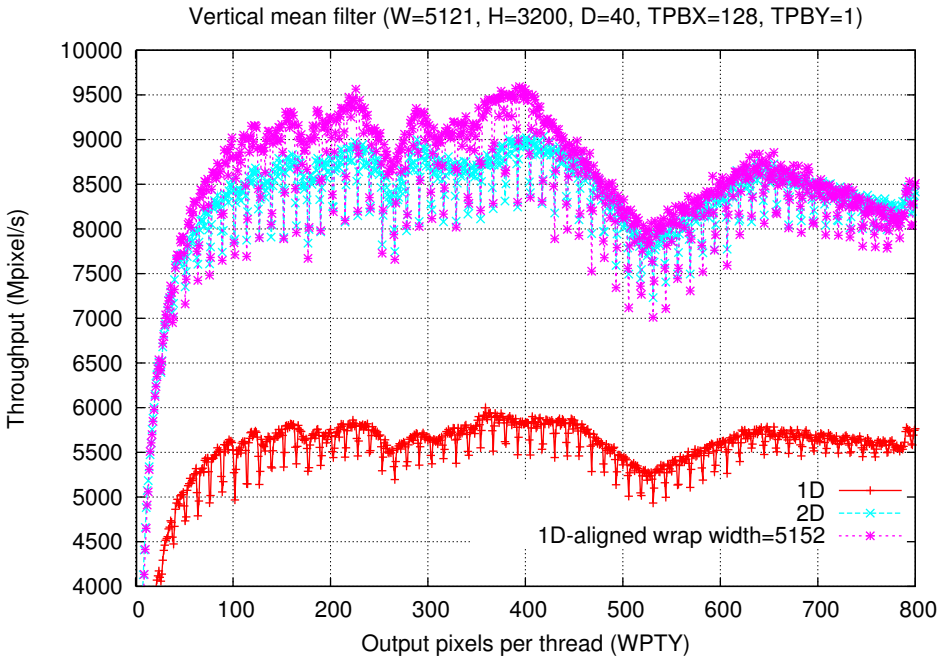


Figure 8.13: A 5121×3200 image. The data is aligned to 5184 pixels. Results compare the 2D, 1D and aligned 1D grid to data mappings.

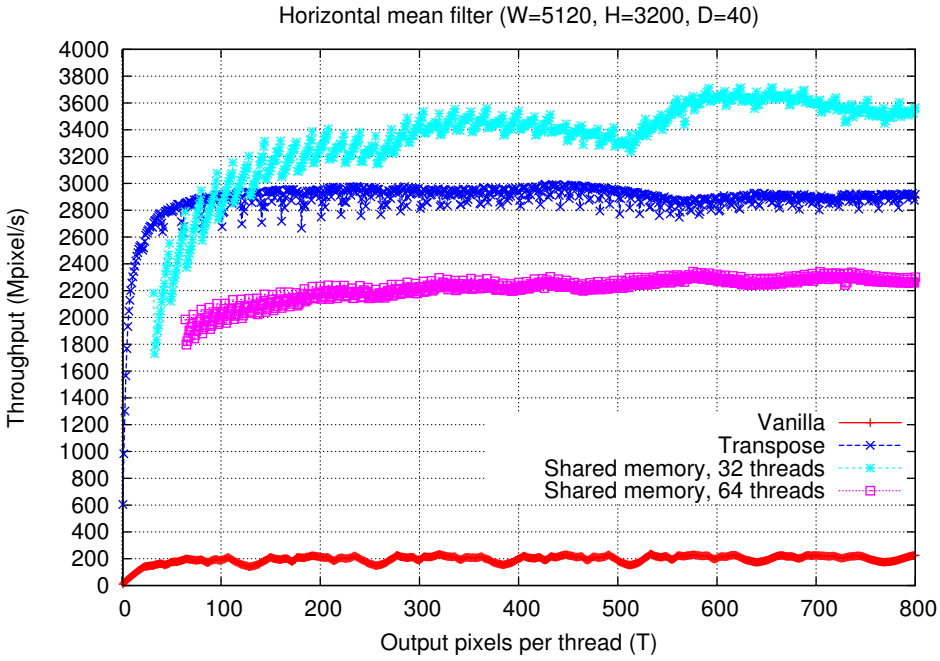


Figure 8.14: Results on a 5120×3200 image. Comparing the naive version with inefficient memory access to versions that transpose the entire dataset before computation and that transpose in shared memory using two different sizes of block.

8.5.2 SSE results

SSE vertical mean filter

In Figure 8.15 we present experimental results for the CPU version of the vertical mean filter, implemented using SSE instructions as discussed in Section 8.3.2. Experiments are performed on three platforms: an 8-core 2.5GHz Intel Xeon E5420 with two sockets and 4 cores-per-socket (Xeon); a 4-core 2.3GHz AMD Phenom 9650 (Phenom) and a 2-core 3GHz Intel Core 2 Duo E8400 (Duo). As a baseline for comparison we use a version where the horizontal and vertical loops have not been interchanged to lead to contiguous memory accesses. We refer to this version as XY, and to versions where loop interchange has been applied as YX.

The YX loop scans horizontally and sums into an intermediate accumulation array. We compare the number of threads and the way the computation is parallelised: columns or horizontal strips. Using more partitions than threads performs worse as load-balancing issues are minimal. The peak number of threads does not necessarily coincide with the number of cores because of interference in the memory system between cores.

SSE horizontal mean filter The optimal algorithm to use for the stand-alone horizontal mean filter is trivially a streaming loop, as we can see in Figure 8.16. This approach efficiently uses the CPU's cache prefetching infrastructure and is the type of loop the CPU is optimised for and the performance in this case rivals that of the vertical scan, even though vectorisation efficiency is reduced due to the horizontal dependency. In the graph we see results for: the simple streamed loop, a bi-directional

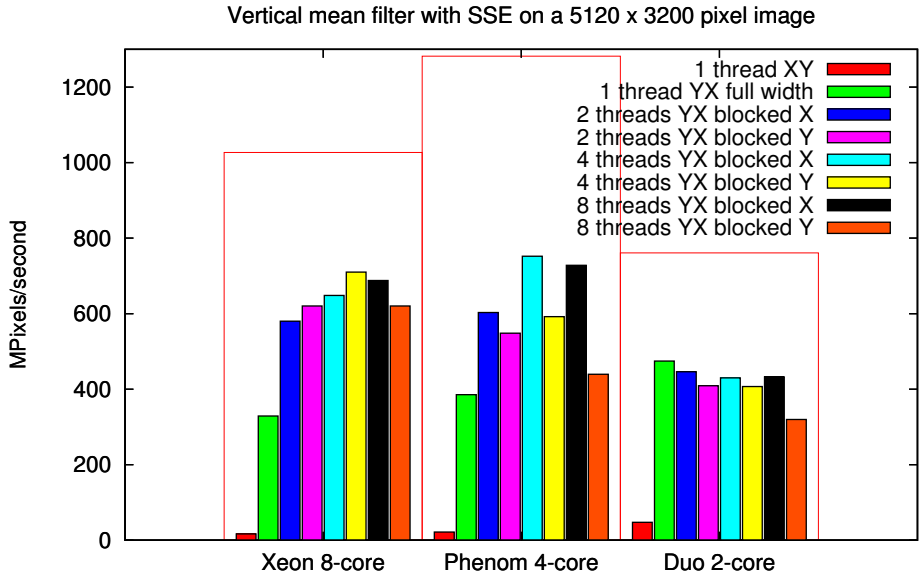


Figure 8.15: Comparison of different blocking strategies for a CPU version of the vertical mean filter. The large surrounding boxes represent the peak memory copy throughput for each of the architectures, as obtained by running the STREAM benchmark [MHMF00].

transpose operation performed by Intel's MKL library, and a combination of the two, giving the correct result. The performance of the transpose alone is considerably less than that of the streaming filter implementation and as a result the combined version is also slow in comparison.

8.6 Conclusions

By exploring the optimization space of several versions of an image-processing filter, evaluated on Intel and AMD architectures equipped with GPUs, we have shown that the strategy used for iteration-space partitioning can have a dramatic effect on the performance of the filter. No single version is suitable for both GPU and CPU architectures, and each version requires quite different code to be written and maintained. We have considered only a simple image processing kernel; clearly the difficulty of creating and maintaining multiple versions of efficient code bases only increases in difficulty when working with more complicated kernels and full HPC applications.

The cost and maintenance problem makes it problematic for programmers to work at this level of development. Cleanly separating the execution schedule of a kernel from its memory access pattern has the potential to facilitate productive and efficient programming of heterogeneous multi-core systems. The Ækute programming model enables representations that can go some way to ease the maintenance burden by decoupling computation kernels from their memory access requirements.

The multiple levels of memory hierarchy present in GPU accelerated systems led to a need to partition the iteration space into multiple levels, in a similar fashion to Stanford's Sequoia language [FHK+06]. As in Sequoia we target systems with software-managed memory hierarchies and seek to separate a high-level algorithm representation from a system-specific mapping. Unlike Sequoia, we base our mapping on partitioning (manually or automatically) an iteration space into disjoint subspaces and infer memory access of subspaces from Ækute metadata.

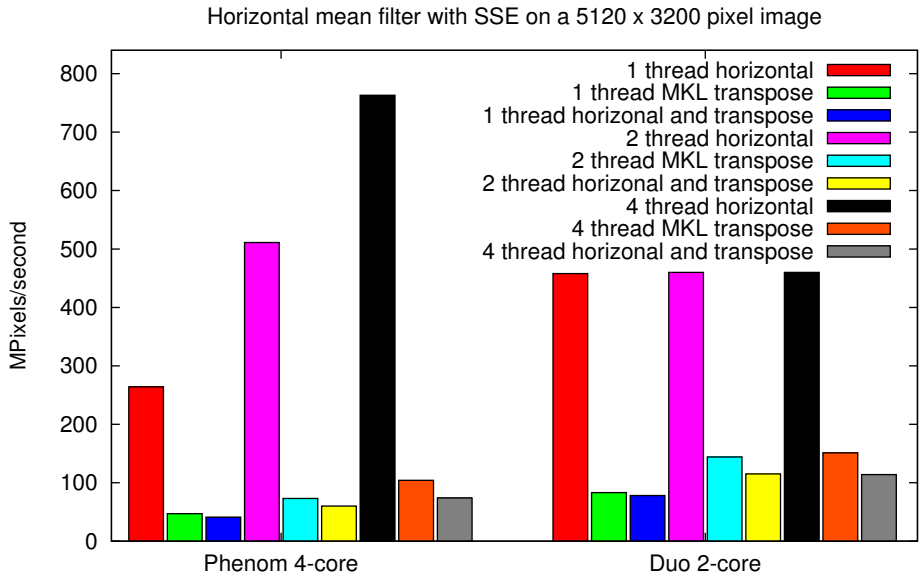


Figure 8.16: Comparison of horizontal filter versions with and without transposition. We see results for the streamed horizontal filter, a transpose performed in optimised form by Intel's MKL library and the combination of the transpose and optimal vertical filter variant.

```
1  // Array descriptors (C array wrappers)
2  Array2D<float> arrayI(&I[0][0], W, H);
3  Array2D<float> arrayO(&O[0][0], W, H-D);
4
5  // Execute metadata: parallel iteration space
6  IterationSpace1D x(0,W);
7  IterationSpace1D y(0,H-D);
8  IterationSpace2D iterXY(x,y);
9
10 // Access metadata: iteration space -> memory
11 VerticalStrip2D_R accessI(iterXY, arrayI, D);
12 Point2D_W accessO(iterXY, arrayO);
```

Figure 8.17: Æcute metadata for the mean filter. The iteration space here is defined in terms of the vertical filter. Moving to support for the horizontal filter is as simple as adding an address conversion function as discussed in Chapter 7.

In Figure 8.17 we see an example of Æcute metadata for the vertical mean filter. In lines 1–3 we wrap accesses to plain C arrays `I[W][H]` and `O[W][H-D]` into Æcute array descriptors `arrayI` and `arrayO` to cleanse the kernel of uncontrolled side-effects. In lines 5–8 we construct a 2D iteration space descriptor `iterXY` from 1D descriptors `x` and `y`, having the same bounds as the output image dimensions. By default, an iteration space is parallel in every dimension. Finally, in lines 10–12 we specify that on each iteration of the 2D iteration space the kernel reads a vertical strip of D pixels from `arrayI` and writes a single pixel to `arrayO`.

This system can be partitioned in multiple levels. For example, we might use the following partitioning:

- at the lowest level, by individual threads:
```
// 1xT outputs per thread
iterXY.partitionThreads(1,T);
```
- at the middle level, by blocks of possibly cooperating threads:
```
// 128xT outputs per block
iterXY.partitionBlocks(128,T);
```
- at the highest level, by possibly cooperating compute devices:
```
// (W/2)x(H-D) outputs per device
iterXY.partitionDevices(W/2,H-D)
```

This partitioning can then be extended to support multiple processing nodes.

The goal of this work is to make up for the limitation of languages such as OpenCL [The09] The OpenCL initiative aims to provide portability across heterogeneous compute devices by providing a detailed, low-level API for describing computational kernels. Although a standard-compliant OpenCL kernel will execute correctly on any standard-compliant implementation, it is clear that performance of the kernel will depend critically on characteristics of the underlying hardware.

High-level models such as Æcute aim to repair the disconnect between the ideal of a cross-platform language and the reality of the need to carefully tune the language for specific architectures. In particular, code generation can be oriented towards effectively orchestrating data movement in software-managed memory hierarchies, including automatically handling such low-level details as data alignment and padding.

8.7 Summary

In this chapter we evaluated various code possibilities and discussed the limitations of development for vector architectures in terms of SSE-based CPUs and CUDA-based GPUs. The chapter has not presented a full solution but demonstrated a series of results looking at the problem, with the aim of leading to a better understanding of how the Æcute model can be applied to such architectures. In the future it would be worth continuing to a full solution to this problem, possibly in terms of Æcute-style extensions to OpenCL itself.

Chapter 9

Conclusions

9.1 Summary of the thesis

In the introduction we claimed the following contributions:

1. We demonstrate, in Chapter 4 the GPU variant of a unified stream-like programming description for GPUs, FPGAs and small vector units on the Sony PlayStation 3. We also show how this programming model only has limited scope and that for more complicated problems it is overly restrictive.
2. In Chapter 5 we introduce the concept of *indexed dependence metadata*. This metadata can be viewed as a generalisation of stream programming that enables a wider range of optimisations and code generation opportunities than we could achieve with the model in Chapter 4.
3. We show how indexed dependence metadata can be used in a component programming system. We demonstrate in Chapter 6 the use of indexed dependence metadata on component interfaces, and implement a framework that uses these data to perform fusion and array contraction of connected components. We then show how these optimisations give performance benefits.
4. Chapter 7 introduces the concept of decoupled access/execute metadata and the Ækute programming model. This variant of indexed dependence metadata is used in a programming system for the Cell processor, automatically executing DMA operations for kernels defined to execute only on local memory regions.
5. In the final contribution chapter, Chapter 8, we show the performance variation possible depending on the implementation of even simple kernels on GPU architectures. We show how code generation, using the Ækute model and indexed dependence metadata, can be used to ease the development process and automate the production of high performance GPU code.

The baseline ASC programming model we discussed in contribution 1 has severe limitations and is not something we would like to continue. The streaming model is applied at the level of individual arithmetic operations, which limits the flexibility of implementation of complicated kernels.

Instead we proposed moving away from the streaming model and suggested indexed dependence metadata as an alternative solution. We discussed this in introductory form in Chapter 5, showing

how metadata can be seen as additional information to aid optimisation of a computation kernel, or as a form of necessary implementation description: in either case it is a declarative mapping of computation to data and offers a separation of concerns. This introduction demonstrated the principle of indexed dependence metadata as a generalisation of stream programming and led to further discussion in Chapter 6 and Chapter 7 on two applications of such metadata.

In Chapter 6 we saw how metadata on component interfaces adds information that a component runtime system can use to optimise component compositions. With access to component implementation models in addition fusion operations can also be performed, using only the metadata to compute data dependencies throughout the composition. These components work perfectly without the metadata, if sub-optimally, and hence in this case we can see indexed dependence metadata as true metadata to the computation. The disadvantage of this approach is that metadata must be added to components by hand. The use of authoring tools would offer opportunities to improve this, particularly if fully integrated into an analysis framework at component-creation time.

Chapter 7 discussed a different approach: embedding the indexed dependence metadata in the form of decoupled access/execute or Æcute metadata in C++ objects. The information embedded in these objects allows data movement to be efficiently performed, precisely specifying the data that is needed over a series of loop iterations in a fashion that would be more difficult to extract from the loop body. At the same time the code is kept reasonably simple without the need for difficult-to-read manual software pipelining. However, the complications of the C++ objects mean that the code is not as simple as it might be with compiler assistence.. Simple compiler-readable annotations in the code providing the same information that is provided by the C++ classes would simplify the model, removing the inevitable overhead of layers of C++.

In both these examples the range of metadata options is limited. Further work is needed in the future to expand the range of possible metadata representations and to widen the scope of applications that can be cleanly developed in this fashion.

Finally, in Chapter 8 we looked at an example kernel on the GPU and demonstrated that a wide range of possible implementations are possible without changing the algorithm itself, and that these implementations are challenging to implement, maintain and understand. We proposed the use of indexed dependence metadata and code generation as a solution to this problem and would like to extend this work towards such an implementation in the future.

9.2 Future work

9.2.1 Future targets for indexed dependence metadata

The implementations discussed in Chapter 7 and Chapter 6 target general CPUs and Cell. In Chapter 8 we discussed targeting CUDA GPUs, but without a full implementation. Targeting CUDA is probably not the best approach. GPUs would be better targeted by using OpenCL as a cross-platform parallel programming API. OpenCL still requires careful tuning of kernel structure and access patterns for a given architecture, and it is not clear how OpenCL will perform as an output medium on the wider range of platforms including general purpose CPUs and AMD's GPUs.

To efficiently generate for OpenCL, CUDA and for the wide range of architectures that exist now and

will exist in the future more sophisticated partitioning strategies are necessary than we have previously discussed. In particular, a degree of automated partitioning is necessary. One possible model that acts as a hint to how this could work is the Sequoia [KPR+07] architectural mapping. In Sequoia this is written by the programmer, but a generalised description of the architecture and efficient mapping schemes that can be extracted from the metadata descriptions and partitioning information should enable efficient partially automated mapping. Automated mapping is vital because the primary goal of any system of this sort is to reduce the maintenance burden on the programmer with as little performance degradation as possible. Achieving the highest possible performance on a given architecture is something that will always be achieved with extended programmer effort.

9.2.2 More sophisticated mapping strategies

In the examples we discussed in this thesis, indexed dependence metadata is used to map affine iteration spaces into directly accessible memory in the form of arrays. This provides benefits where the mapping is separated from computation and hence the computation need not address global memory addresses - allowing us to produce data movement operations as we saw in Chapter 7. However, the assumption of a flat memory representation at some level is limiting and might lead to similar representation problems at the cluster level as we see at the node level with simple C addressing.

If we observe environments such as STAPL [AJR+03], or even the C++ standard template library, we can see that the memory representation is abstracted away by an iterator structure. In the case of STAPL these iterators combine with partitioning representations which allow sophisticated underlying data representations to automate the parallelisation process.

STAPL is still a data-based partitioning and parallelisation model. Indexed dependence metadata can be extended to provide a mapping from the iteration space not to memory addresses, but to iterator addresses. The partitioning of the iteration space can be used to drive the partitioning of the data structures. However the data structures themselves will still maintain an independent underlying representation of the data and merely take hints from the iteration space partitioning and mapping to define how they structure their underlying data. In this way the concept can be expanded to provide a more practical programming model for large-scale application development.

9.2.3 Development environments

In our current work we have a combination of XML-based component and interface descriptions that house indexed dependence metadata and C++ classes that wrap memory accesses with metadata. In either case the programming model is clumsy. The XML code is long winded and hard to deal with, and the C++ classes use an unattractive method of implicitly implementing the kernel as a function contained within a class. This situation suffers from considerable code overhead. The complexity of this C++ approach aims to allow efficient memory movement and code generation while using a standard C++ compiler.

Improving this situation could be achieved in many ways. The first is that we could continue to use C++ classes, but place them directly in the kernel code, written inline as is natural to the programmer. These classes should, in a normal compiler, compile away to efficient but standard memory access code. Under a compiler with specific Ækute support, these classes could be detected and the kernel

extracted to generate good code using the classes as input metadata to the code generation process. This could be considered an extension of current OpenMP approaches.

Alternatively pragmas or other comment-based information could achieve the same purpose within a compiler, with an arbitrarily complicated pragma language allowing full metadata specification, extending the ideas discussed in [Gas08]. The downside of this approach is that the kernel would have to be fully operational code to run through a normal compiler and hence utilise complicated memory access functions throughout. By not limiting the kernel's memory access to the small set of data elements we allow in the metadata for a given iteration we raise the risk of finding kernels that the compiler is again unable to extract the necessary information from, losing the benefits we might have gained from using metadata.

If we wish to allow components written using another method and obtained as binaries then another approach is to extend standard interface definition languages such as those use for Microsoft's COM or Java RPC to include more sophisticated metadata. This would allow the creation of large bodies of metadata-enhanced computation kernels, allowing code reuse but maintaining the benefits of code fusion and optimisation to a new environment when possible.

9.2.4 Serialised computation kernels

One problem that has to be solved is how to represent kernels such as the box filter discussed in Chapter 8. We know that the optimal kernel partially serialises, and the experiments in the chapter demonstrate what a range of optimisations are possible. What is less clear is how to represent such a kernel.

The simplest option would be to represent the block of iterations as a single kernel, where the serialised loop is internal to the kernel and the iteration space is blocked appropriately. The size of the inner blocks can still be flexible in this case, and the bounds of the kernel loops can depend on the size of the blocks.

The downside of this approach is that we have lost the polyhedral representation at the inner level that we might obtain automatically from the iteration space definition. This polyhedral representation was vital in Chapter 6 to enable fused code generation from sets of components. The alternative solution to support this is to support explicitly phased computations, such that the parallel and serial phases of the kernel are represented directly and can be mapped to the polyhedral iteration space without the complex kernel analysis that we wish to avoid by adding such metadata. We could achieve this by supporting two separate kernels:

```
boxFilter
{
  iteration(0, y) {
    sum = 0;
    foreach i in (0:radius) {
      sum += data(i, y);
    }
    output(0, y) = sum / radius;
  }
  iteration(x(>0), y) {
    sum -= data(x, y);
    sum += data(x+radius, y);
    output(x,y) = sum/radius;
```

```
    }
}
```

Alternatively we might embed conditionals into a single kernel that trigger at different points in the iteration space. Given that these conditionals would relate only to the iterator variables we might even rely on the compiler to optimise the conditionals away in the right circumstances.

The optimal choice would depend on the overall representation of kernels, and these decisions need to be made before a clean programming model can be developed. If we wish to take this indexed dependence metadata and Æcute work forward into a full programming model such issues must be solved.

9.2.5 When is metadata metadata and when is it implementation?

It is important to define carefully when metadata is really metadata and when it is implementation. We discussed this briefly in Section 5.2. There are two primary options:

- Added information that is optional where the kernel will compile and work correctly without such information.
- Information that is vital to the correct execution of the kernel.

It might be argued that only for former case can correctly be called metadata, but in either case the principle is the same. The information discussed in Chapter 6 fits the former category, and the Æcute notation in Chapter 7 is the latter, if only by implementation rather than by necessity. If we agree that such information is useful to a productive programming environment, the optimum approach depends on the situation.

Extending the indexed dependence metadata work into a useful programming environment will require basic decisions on the principles. Should the kernels be stand-alone in a compiler? Should they only work with language extensions? Should they compile fine, but using wrapper classes that achieve both goals at the cost of a high degree of complexity, as we used in Chapter 7?

In either case we should investigate ways to use static analysis to check the correctness of metadata specifications.

9.3 Final conclusions

This chapter concluded the thesis with a summary of the main contributions and a discussion of directions for future work. The goal of the thesis was to investigate and promote metadata-enhanced programming with the aim to finding an optimal model to apply to development, allowing programmers to maintain a high level of efficiency which maintaining high software performance. This thesis is an attempt to realise the THEMIS [KBFB01] proposal and we think contributes substantially towards this goal and hope that work on the concepts in the proposal continue, particularly in the directions of the future work mentioned in this chapter.

Appendix A

Code associated with examples

This appendix provides, for completeness, the code associated with the examples presented in Chapter 6 and Chapter 7.

A.1 Components with metadata

In this section we see representations of the components used in Chapter 6.

A.1.1 The contour filter master component

The interface specification for the contour filter.

```
<interface id="fourcomponentcontourfilter">
  <operation name="contourfilter">
    <input type="float" format="array(in_x, in_y)" name="image_in" />
    <output type="float" format="array(out_x, out_y)" name="image_out" />
  </operation>
</interface>
```

The definition of a specific instantiation of the component.

```
<component id="fourcomponentcontourfilter">
  <implementation type="c">
  <location>4componentcontourfilter.c.componentsource</location>
  </implementation>

  <implements id="fourcomponentcontourfilter" />

    <operation name="contourfilter">
    <uses name="convA">
      fourcomponentconvolution.convolution(
        image_in structured (in_x2, in_y2),
        filter_in structured (3, 3),
        image_out structured (out_x2, out_y2) flow to convolutionout)
```

```
    </uses>

    <uses name="dilationA">
      fourcomponentdilation.dilation(
        image_in structured
          (fourcomponentcontourfilter.out_x,
          fourcomponentcontourfilter.out_y)
          flow from convolutionout,
        filter_x = 3,
        filter_y = 3,
        image_out structured
          (fourcomponentcontourfilter.out_x,
          fourcomponentcontourfilter.out_y)
          flow to dilationout)
    </uses>

    <uses name="arithA">
      fourcomponentarithmetic.arithmetic_operation(
        image_in structured
          (fourcomponentcontourfilter.in_x,
          fourcomponentcontourfilter.in_y)
          flow from dilationout,
        image2_in structured
          (fourcomponentcontourfilter.in_x,
          fourcomponentcontourfilter.in_y)
          flow from convolutionout,
        image_out structured
          (fourcomponentcontourfilter.out_x,
          fourcomponentcontourfilter.out_y)
          flow to arithout1)
    </uses>


  </operation>

  <constraint type="equality">
    fourcomponentcontourfilter.in_x=in_x2
  </constraint>
  <constraint type="equality">
    fourcomponentcontourfilter.in_y=in_y2
  </constraint>
  <constraint type="equality">
    fourcomponentcontourfilter.out_x=out_x2
  </constraint>
  <constraint type="equality">
    fourcomponentcontourfilter.out_y=out_y2
  </constraint>
</component>
```

The implementation of the contour filter component.

```
#include <stdlib.h>
#include <stdio.h>

COMPONENT_TARGET(contourfilter)
{
  Array2d< float > filter( 3, 3 );
  Array2d< float > convolutionout( image_in.width(), image_in.height() );
```

```
  Array2d< float > dilationout( image_in.width(), image_in.height() );

  for( int i = 0; i < (filter.width()); i++ )
    for( int j = 0; j < (filter.height()); j++ )
      filter(i, j) = 1.0/(filter.width()*filter.height());

  COMPONENT_CALL(fourcomponentconvolution.convolution{convA})[
    image_in -> image_in;
    filter -> filter_in;
    convolutionout -> image_out];

  COMPONENT_CALL(fourcomponentdilation.dilation{dilationA})[
    convolutionout -> image_in;
    3 -> filter_x;
    3 -> filter_y;
    dilationout -> image_out];

  COMPONENT_CALL(fourcomponentarithmetic.arithmetic_operation{arithA})[
    dilationout -> image_in;
    convolutionout -> image2_in;
    SUBTRACT -> operation;
    image_out -> image_out];

}
```

A.1.2 The convolution sub-component used by the contour filter

The interface specification for the convolution component.

```
<interface id="fourcomponentconvolution">
  <operation name="convolution">
    <input type="float" format="array(ACONV, BCONV)" name="image_in" />
    <input type="float" format="array(CCONV, DCONV)" name="filter_in" />
    <output type="float" format="array(ACONV, BCONV)" name="image_out" />
  </operation>
</interface>
```

The component instantiation of the convolution component.

```
<component id="fourcomponentcloogconvolution">
  <implementation type="cloog">
    <location>4componentcloogconvolution.cloog.componentsource</location>
  </implementation>

  <implements id="fourcomponentconvolution" />

  <operation name="convolution">
    <constraint type="dependentregion" shape="orthotope">
      <!-- These are "radius" values.
      the total diameter is (radius*2 + 1) -->
      <constraintinput
        name="image_in"
        placement="relative"
        ranges="((fourcomponentconvolution.CCONV-1)/2,
          (fourcomponentconvolution.DCONV-1)/2)" />
      <constraintinput
        name="filter_in"
```

```
        placement="absolute"
        ranges="(0->fourcomponentconvolution.CCONV-1,
          0->fourcomponentconvolution.DCONV-1)" />
      <!-- note that (0,0) is an inclusive range.
      So actually that's (0->0, 0->0)
      or a 1x1 region, single pixel value -->
      <constraintoutput name="image_out" ranges="(0, 0)" />
    </constraint>
  </operation>
</component>
```

The polyhedral source for the convolution component

```
CODE_BLOCK(conv)
{
  static int sum;
  static int xoffset = 1;
  static int yoffset = 1;

  static void inline init(
    Array2d<float> &filter_in,
    Array2d<float> &image_in,
    Array2d<float> &image_out)
  {
    xoffset = (filter_in.width()-1)/2;
    yoffset = (filter_in.height()-1)/2;
  }

  static void inline convolve(
    int y, int x, Array2d<float> &filter_in,
    Array2d<float> &image_in, Array2d<float> &image_out)
  {
    float outvalue1 = 0;
    float outvalue2 = 0;
    float outvalue3 = 0;
    float outvalue4 = 0;
    for( int yy = 0; yy < filter_in.height(); yy++ )
    {
      for( int xx = 0; xx < (filter_in.width()); xx++ )
      {
        int yaddr = (y + (yy) - yoffset);
        int xaddr = (x + (4*xx) - (4*xoffset));

        // Correct for edges
        if( yaddr < 0 ) yaddr = 0;
        if( xaddr < 0 ) xaddr = 0;
        if( yaddr >= image_in.height() ) yaddr = image_in.height()-1;
        // Bound to a group of 4, 4 in from the edge
        if( xaddr >= image_in.width() ) xaddr = image_in.width()-4;

        // Convolve
        // Don't keep doing region check as we're assuming image
        // is a multiple of 4.
        // Assume we just need the first value of the filter here as they
        // are repeated across the 4 elements
        float filterVal = filter_in(xx, yy);
        outvalue1 += image_in(xaddr, yaddr) * filterVal;
        outvalue2 += image_in(xaddr+1, yaddr) * filterVal;
```

```
        outvalue3 += image_in(xaddr+2, yaddr) * filterVal;
        outvalue4 += image_in(xaddr+3, yaddr) * filterVal;
      }
    }

    image_out(x, y)  = outvalue1;
    image_out(x + 1, y)  = outvalue2;
    image_out(x + 2, y)  = outvalue3;
    image_out(x + 3, y)  = outvalue4;
  }
}

COMPONENT_TARGET(convolution)
{

  INITIALISE(conv) init();
  POLYHEDRAL_LOOP(i)
  [
    i < image_in.height();
    // i must be greater than 0
    i >= 0;
  ]
  {
    POLYHEDRAL_LOOP(j)
      [
       j < image_in.width();
       j   >= 0;
      stride = 4;
       ]
    {
      OP(conv) convolve();
    }

  }
}
```

A.1.3 The dilation sub-component used by the contour filter

The interface specification for the dilation component.

```
<interface id="fourcomponentdilation">
  <operation name="dilation">
    <input type="float" format="array(ADIL, BDIL)" name="image_in" />
    <input type="integer" format="scalar{FX}" name="filter_x" />
    <input type="integer" format="scalar{FY}" name="filter_y" />
    <output type="float" format="array(ADIL, BDIL)" name="image_out" />
  </operation>
</interface>
```

The component instantiation of the dilation component.

```
<component id="fourcomponentcloogdilation">
  <implementation type="cloog">
    <location>4componentcloogdilation.cloog.componentsource</location>
  </implementation>

  <implements id="fourcomponentdilation" />
```

```
  <operation name="dilation" >
    <constraint type="dependentregion" shape="rectangle">
      <!-- These are "radius" values,
      where the total diameter is (radius*2 + 1) -->
      <constraintinput
        name="image_in"
        placement="relative"
        ranges="((fourcomponentdilation.FX-1)/2,
          (fourcomponentdilation.FY-1)/2)" />
      <!-- note that (0,0) is an inclusive range.
      So actually that's (0->0, 0->0) or a 1x1 region,
      single pixel value -->
      <constraintoutput name="image_out" ranges="(0, 0)" />
    </constraint>
  </operation>
</component>
```

The polyhedral source for the dilation component

```
CODE_BLOCK(dilation)
{
  static int sum;
  static int xoffset = 1;
  static int yoffset = 1;

  static void inline init(
    int filter_x,
    int filter_y,
    Array2d<float> &image_in,
    Array2d<float> &image_out)
  {
    xoffset = (filter_x-1)/2;
    yoffset = (filter_y-1)/2;
  }

  static void inline dilate(
    int y,
    int x,
    int filter_x,
    int filter_y,
    Array2d<float> &image_in,
    Array2d<float> &image_out)
  {
    float max1 = 0;
    float max2 = 0;
    float max3 = 0;
    float max4 = 0;
    float imagevalue = 0;

    for( int yy = 0; yy < (filter_y); yy++ )
    {

      int yaddr = (y + yy - (yoffset));
      if( yaddr >= 0 &&  yaddr < image_in.height() )
      {
        for( int xx = 0; xx < (4*filter_x); xx+=4 )
        {
```

```
          int xaddr = (x + xx - (4*xoffset));

          if( xaddr >= 0 && xaddr < image_in.width() )
          {
            imagevalue = image_in(xaddr, yaddr);
            if( imagevalue > max1 )
              max1 = imagevalue;
            imagevalue = image_in(xaddr + 1, yaddr);
            if( imagevalue > max2 )
              max2 = imagevalue;
            imagevalue = image_in(xaddr + 2, yaddr);
            if( imagevalue > max3 )
              max3 = imagevalue;
            imagevalue = image_in(xaddr + 3, yaddr);
            if( imagevalue > max4 )
              max4 = imagevalue;
          }
        }
      }
    }

    image_out(x, y)  = max1;
    image_out(x + 1, y)  = max2;
    image_out(x + 2, y)  = max3;
    image_out(x + 3, y)  = max4;
  }
}

COMPONENT_TARGET(dilation)
{

  INITIALISE(dilation) init();

  POLYHEDRAL_LOOP(i)
  [
    i < image_in.height();
    // i must be greater than 0
    i >= 0;
  ]
  {
    POLYHEDRAL_LOOP(j)
      [
       j < image_in.width();
       j   >= 0;
      stride = 4;
       ]
    {
      OP(dilation) dilate();
    }
  }
}
```

A.1.4 The arithmetic sub-component used by the contour filter

The interface specification for the arithmetic component.

```
<interface id="fourcomponentarithmetic">
```

```
  <operation name="arithmetic_operation">
    <input type="float" format="array(AARITH, BARITH)" name="image_in" />
    <input type="float" format="array(AARITH, BARITH)" name="image2_in" />
    <output type="float" format="array(AARITH, BARITH)" name="image_out" />
  </operation>
</interface>
```

The component instantiation of the arithmetic component.

```
<component id="fourcomponentcloogarithmetic">
  <implementation type="cloog">
    <location>4componentcloogarithmetic.cloog.componentsource</location>
  </implementation>

  <implements id="fourcomponentarithmetic" />
</component>
```

The polyhedral source for the arithmetic component

```
CODE_BLOCK(arithmeticOperation)
{
  static void inline arithmeticOperation(
    int y,
    int x,
    Array2d<float> &image_in2,
    Array2d<float> &image_in,
    Array2d<float> &image_out)
  {
    image_out(x, y)  = image_in(x, y) - image_in2(x, y);
    x++;
    image_out(x, y)  = image_in(x, y) - image_in2(x, y);
    x++;
    image_out(x, y)  = image_in(x, y) - image_in2(x, y);
    x++;
    image_out(x, y)  = image_in(x, y) - image_in2(x, y);
  }
}

COMPONENT_TARGET(arithmetic_operation)
{
  POLYHEDRAL_LOOP(i)
  [
    i < image_in.height();
    // i must be greater than 0
    i >= 0;
  ]
  {
    POLYHEDRAL_LOOP(j)
      [
       j < image_in.width();
       j   >= 0;
      stride = 4;
       ]
    {
      OP(arithmeticOperation) arithmeticOperation();
    }
  }
}
```

A.1.5 The core component for the multigrid example.

The interface specification for the multigrid component.

```
<interface id="mgrid">
  <operation name="mgrid">
    <input type="integer" format="scalar" name="m1_j" />
    <input type="integer" format="scalar" name="m2_j" />
    <input type="integer" format="scalar" name="m3_j" />
    <input type="integer" format="scalar" name="m1_k" />
    <input type="integer" format="scalar" name="m2_k" />
    <input type="integer" format="scalar" name="m3_k" />
    <input type="double" format="array(4)" name="a_in" />
    <input type="double" format="array(4)" name="c_in" />
    <input type="integer" format="scalar" name="k" />
    <input type="integer" format="scalar" name="m" />
    <input type="double" format="array(GRIDX, GRIDY, GRIDZ)" name="u_j_in" />
    <input type="double" format="array(GRIDXK, GRIDYK, GRIDZK)" name="r_in" />
    <output type="double" format="array(GRIDX, GRIDY, GRIDZ)" name="u_out" />
    <output type="double" format="array(GRIDXK, GRIDYK, GRIDZK)" name="r_out" />
  </operation>
</interface>
```

The component instantiation of the multigrid component.

```
<component id="mgrid">
  <implementation type="c">
    <location>mgrid.c.componentsource</location>
  </implementation>
  <implements id="mgrid" />
  <operation name="mgrid">
    <uses name="zero3">
      mgrid_zero3.mgrid_zero3(
        z_out structured (GRIDX, GRIDY, GRIDZ) flow to z1,
        n1 = m1_k,
        n2 = m2_k,
        n3 = m3_k)
    </uses>

    <uses name="interp">
      mgrid_interp.mgrid_interp(
        z_in structured (GRIDX_J, GRIDY_J, GRIDZ_J),
        u_in structured (GRIDX, GRIDY, GRIDZ) flow from z1,
        u_out structured (GRIDX, GRIDY, GRIDZ) flow to u2,
        mm1, mm2, mm3,
        n1 = m1_k, n2 = m2_k, n3 = m3_k, k, m )
    </uses>

    <uses name="resid">
      mgrid_resid.mgrid_resid(
        u_in structured (GRIDX, GRIDY, GRIDZ) flow from u2,
        v_in structured (GRIDX, GRIDY, GRIDZ),
        r_out structured (GRIDX, GRIDY, GRIDZ),
        n1 = m1_k, n2 = m2_k, n3 = m3_k,
        k, m, a_in structured (4) )
    </uses>
```

```
    <uses name="psinv">
      mgrid_psinv.mgrid_psinv(
        r_in structured(GRIDX, GRIDY, GRIDZ),
        u_in structured(GRIDX, GRIDY, GRIDZ ) flow from u2,
        u_out structured( GRIDX, GRIDY, GRIDZ ),
        c_in structured( 4 ),
        n1 = m1_k, n2 = m2_k, n3 = m3_k, k, m )
    </uses>


  </operation>

  <constraint type="inequality">m1_k>5</constraint>
  <constraint type="inequality">m2_k>5</constraint>
  <constraint type="inequality">m3_k>5</constraint>
</component>
```

The C source for the multigrid component

```
CODE_BLOCK(mgrid)
{
}


COMPONENT_TARGET(mgrid)
{

  COMPONENT_CALL(mgrid_zero3.mgrid_zero3{zero3})[
    u_out -> z_out;
    m1_k -> n1;
    m2_k -> n2;
    m3_k -> n3 ];

  COMPONENT_CALL(mgrid_interp.mgrid_interp{interp})[
    u_j_in -> z_in; u_out -> u_in; u_out -> u_out;
    m1_j -> mm1; m2_j -> mm2; m3_j -> mm3;
    m1_k -> n1; m2_k -> n2; m3_k -> n3;
    k -> k; m -> m ];

  COMPONENT_CALL(mgrid_resid.mgrid_resid{resid})[
    u_out -> u_in; r_in -> v_in; r_out -> r_out;
    m1_k -> n1; m2_k -> n2; m3_k -> n3;
    k -> k; m -> m; a_in -> a_in ];

  COMPONENT_CALL(mgrid_psinv.mgrid_psinv{psinv})[
    r_out -> r_in;  u_out -> u_in;
    m1_k -> n1; m2_k -> n2; m3_k -> n3;
    k -> k; m -> m; c_in -> c_in; u_out -> u_out ];
}
```

A.1.6 The zeroing component for the multigrid example.

The interface specification for the zeroing component of the multigrid example.

```
<interface id="mgrid_zero3">
  <operation name="mgrid_zero3">
    <input type="integer" format="scalar" name="n1" />
```

```
    <input type="integer" format="scalar" name="n2" />
    <input type="integer" format="scalar" name="n3" />
    <output type="double" format="array(GRIDX, GRIDY, GRIDZ)"
      name="z_out" />
  </operation>
</interface>
```

The component instantiation of the zeroing component for the multigrid example.

```
<component id="mgrid_zero3">
  <implementation type="cloog">
    <location>mgrid_zero3_el.cloog.componentsource</location>
  </implementation>

  <implements id="mgrid_zero3" />

  <operation name="mgrid_zero3" >
    <constraint type="dependentregion" shape="rectangle">
      <constraintoutput name="z_out" ranges="(0, 0, 0)" />
    </constraint>
  </operation>
</component>
```

The polyhedral source for the zeroing component of the multigrid example.

```
CODE_BLOCK(mgridzero3)
{
  static void inline mgridzero3(
    int z, int y,
    int n1, int n2, int n3,
    Array3d<double> &z_out, int block)
  {
    for( int x = 0; x < n1; ++x )
    {
      z_out(x, y, z) = (double)0.0;
    }
  }
}

COMPONENT_TARGET(mgrid_zero3)
{
  POLYHEDRAL_LOOP(z)
  [
    z < n3;
    z >= 0;
  ]
  {
    POLYHEDRAL_LOOP(y)
    [
      y < n2;
      y >= 0;
    ]
    {
      OP(mgridzero3) mgridzero3();

    }
  }
}
```

A.1.7 The interpolation component for the multigrid example.

The interface specification for the interpolation component of the multigrid example.

```
<interface id="mgrid_interp">
  <operation name="mgrid_interp">
    <input type="integer" format="scalar" name="mm1" />
    <input type="integer" format="scalar" name="mm2" />
    <input type="integer" format="scalar" name="mm3" />
    <input type="integer" format="scalar" name="n1" />
    <input type="integer" format="scalar" name="n2" />
    <input type="integer" format="scalar" name="n3" />
    <input type="integer" format="scalar" name="k" />
    <input type="integer" format="scalar" name="m" />
    <input type="double" format="array(GRIDX, GRIDY, GRIDZ)"
      name="u_in" />
    <input type="double" format="array(GRIDX, GRIDY, GRIDZ)"
      name="z_in" />
    <output type="double" format="array(GRIDX, GRIDY, GRIDZ)"
      name="u_out" />
  </operation>
</interface>
```

The component instantiation of the interpolation component for the multigrid example.

```
<component id="mgrid_interp">
  <implementation type="cloog">
    <location>mgrid_interp_el.cloog.componentsource</location>
  </implementation>

  <implements id="mgrid_interp" />

  <operation name="mgrid_interp" >
    <constraint type="dependentregion" shape="rectangle">
      <constraintinput
        name="u_in" placement="relative"
        ranges="(-1->1, -1->1, -1->1)" />
      <constraintinput
        name="z_in" placement="relative"
        ranges="(-1->1, -1->1, -1->1)" />
      <constraintoutput
        name="u_out" ranges="(-1->1, -1->1, -1->1)" />
    </constraint>
  </operation>
</component>
```

The polyhedral source for the interpolation component of the multigrid example.

```
CODE_BLOCK(mgridinterp) THREADPRIVATE( z1, z2, z3 )
{

  const int correction = -1;

  double *z1;
  double *z2;
  double *z3;
```

```
volatile int waitvariables[32];

static void inline init(
  int k, int m, int mm1, int mm2, int mm3,
  int n1, int n2, int n3,
  Array3d<double> &u_in, Array3d<double> &z_in, Array3d<double> &u_out,
  double **z1, double **z2, double **z3)
{
  *z1 = new double[m];
  *z2 = new double[m];
  *z3 = new double[m];
  for( int i = 0; i < 32; ++i ) waitvariables[i] = 0;
}

static void inline finish(
  int k, int m, int mm1, int mm2, int mm3,
  int n1, int n2, int n3,
  Array3d<double> &u_in, Array3d<double> &z_in, Array3d<double> &u_out,
  double **z1, double **z2, double **z3)
{
  delete [] *z1;
  delete [] *z2;
  delete [] *z3;
}

static void inline completed(
  int z, int y, int k, int m,
  int mm1, int mm2, int mm3,
  int n1, int n2, int n3,
  Array3d<double> &u_in, Array3d<double> &z_in, Array3d<double> &u_out,
  double *z1, double *z2, double *z3, int block)
{
  waitvariables[ block ] = 1;
}

static void inline docontinue(
  int z, int y, int k, int m,
  int mm1, int mm2, int mm3,
  int n1, int n2, int n3,
  Array3d<double> &u_in, Array3d<double> &z_in, Array3d<double> &u_out,
  double *z1, double *z2, double *z3, int block)
{
  // Check that next block has finished
  int counter = 0;
  int theBlock = block -1;
  if( theBlock < 0 )
    (n3>>6) + 1;
}

static void inline mgridinterp(
  int z, int y, int k, int m,
  int mm1, int mm2, int mm3,
  int n1, int n2, int n3,
  Array3d<double> &u_in, Array3d<double> &z_in, Array3d<double> &u_out,
  double *z1, double *z2, double *z3, int block)
{
  if( (y == 1) && ((z & 31) == 31) )
    docontinue(
```

```
        z, y, k, m, mm1, mm2, mm3,
        n1, n2, n3, u_in, z_in, u_out,
        z1, z2, z3, block );
    if( (y == 1) && ((z & 31) == 3) )
      completed(
        z, y, k, m, mm1, mm2, mm3,
        n1, n2, n3, u_in, z_in, u_out,
        z1, z2, z3, block );

    int zi3 = (z-1)/2;
    int zi2 = (y-1)/2;
    for( long i1 = 1; i1 < (mm1+1); ++i1 )
    {
      z1[i1+correction] =
        z_in(i1+correction, (zi2+1), (zi3))
        + z_in((i1+correction), (zi2), (zi3));
      z2[i1+correction] =
        z_in((i1+correction), (zi2), (zi3+1))
        + z_in((i1+correction), (zi2), (zi3));
      z3[i1+correction] =
        z_in((i1+correction), (zi2+1), (zi3+1))
        + z_in((i1+correction), (zi2), (zi3+1)) + z1[i1+correction];
    }
    for( long i1 = 1; i1 < (mm1); ++i1 )
    {
      u_out((2*i1-1+correction), y, z-1) =
        u_in((2*i1-1+correction), y, z-1)
          + (double)0.5 * z1[i1+correction];
      u_out((2*i1+correction), y, (z-1)) =
        u_in((2*i1+correction), y, (z-1))
          + (double)0.25 * ( z1[i1+correction] + z1[i1 + 1+correction] );
    }
    for( long i1 = 1; i1 < (mm1); ++i1 )
    {

      u_out((2*i1-1+correction), (y-1), (z)) =
        u_in((2*i1-1+correction), (y-1), (z))
        + (double)0.5 * z2[i1+correction];
      u_out((2*i1+correction), (y-1), (z)) =
        u_in((2*i1+correction), (y-1), (z))
        + (double)0.25 * (z2[i1+correction] + z2[i1 + 1 + correction] );
    }
    for( long i1 = 1; i1 < (mm1); ++i1)
    {
      u_out((2*i1-1+correction), y, z) =
        u_in((2*i1-1+correction), y, z)
        + (double)0.25 * z3[i1+correction];
      u_out((2*i1+correction), y, z) =
        u_in((2*i1+correction), y, z)
        + (double)0.125 * (z3[i1+correction] + z3[i1 + 1 + correction]);
    }
  }

}

COMPONENT_TARGET(mgrid_interp)
{
  INITIALISE(mgridinterp) init();
```

```
  POLYHEDRAL_LOOP(z)
  [
    z < n3;
    z >= 1;
    stride = 2;
    stridefrom = 1;

  ]
  {
    POLYHEDRAL_LOOP(y)
    [
      y < n2;
      y >= 1;
      stride = 2;
      stridefrom = 1;
    ]
    {
        OP(mgridinterp) mgridinterp();
    }
  }
  CLEANUP(mgridinterp) finish();
}
```

A.1.8 The residual component for the multigrid example.

The interface specification for the residual component of the multigrid example.

```
<interface id="mgrid_resid">
  <operation name="mgrid_resid">
    <input type="integer" format="scalar" name="n1" />
    <input type="integer" format="scalar" name="n2" />
    <input type="integer" format="scalar" name="n3" />
    <input type="integer" format="scalar" name="k" />
    <input type="integer" format="scalar" name="m" />
    <input type="double" format="array(4)" name="a_in" />
    <input type="double" format="array(GRIDX, GRIDY, GRIDZ)"
      name="u_in" />
    <input type="double" format="array(GRIDX, GRIDY, GRIDZ)"
      name="v_in" />
    <output type="double" format="array(GRIDX, GRIDY, GRIDZ)"
      name="r_out" />
  </operation>
</interface>
```

The component instantiation of the residual component for the multigrid example.

```
<component id="mgrid_resid">
  <implementation type="cloog">
    <location>mgrid_resid_el.cloog.componentsource</location>
  </implementation>

  <implements id="mgrid_resid" />

  <operation name="mgrid_resid" >
    <constraint type="dependentregion" shape="rectangle">
      <constraintoutput name="a_in" ranges="(0->3)" />
      <constraintinput
```

```
        name="u_in" placement="relative"
        ranges="(-1->1, -1->1, -1->1)" />
      <constraintinput
        name="v_in" placement="relative"
        ranges="(0, 0, 0)" />
      <constraintoutput name="r_out" ranges="(0, 0, 0)" />
    </constraint>
  </operation>

</component>
```

The polyhedral source for the residual component of the multigrid example.

```
CODE_BLOCK(mgridresid) THREADPRIVATE( u1, u2, u3 )
{
  double *u1, *u2, *u3;

  volatile int waitvariables[32];

  static void inline init(
    Array1d<double> &a_in,
    int k, int m, int n1, int n2, int n3,
    Array3d<double> &u_in, Array3d<double> &v_in, Array3d<double> &r_out,
    double **lu1, double **lu2, double **lu3)
  {
    *lu1 = new double[m];
    *lu2 = new double[m];
    *lu3 = new double[m];

    for( int i = 0; i < 32; ++i ) waitvariables[i] = 0;
  }

  static void inline finish(
    Array1d<double> &a_in,
    int k, int m, int n1, int n2, int n3,
    Array3d<double> &u_in, Array3d<double> &v_in, Array3d<double> &r_out,
    double **lu1, double **lu2, double **lu3)
  {
    delete [] *lu1;
    delete [] *lu2;
    delete [] *lu3;
  }

  static void inline completed(
    int z, int y,
    Array1d<double> &a_in,
    int k, int m, int n1, int n2, int n3,
    Array3d<double> &u_in, Array3d<double> &v_in, Array3d<double> &r_out,
    double *u1, double *u2, double *u3, int block)
  {
    waitvariables[ block ] = 1;
  }

  static void inline docontinue(
    int z, int y,
    Array1d<double> &a_in,
    int k, int m, int n1, int n2, int n3,
```

```
    Array3d<double> &u_in, Array3d<double> &v_in, Array3d<double> &r_out,
    double *u1, double *u2, double *u3, int block)
  {
    // Check that next block has finished
    int counter = 0;
  }



  static void inline mgridresid(
    int z, int y,
    Array1d<double> &a_in,
    int k, int m, int n1, int n2, int n3,
    Array3d<double> &u_in, Array3d<double> &v_in, Array3d<double> &r_out,
    double *u1, double *u2, double *u3, int block)
  {
    for( int x = 0; x < n1; ++x )
    {
      u1[x] = u_in(x, y-1, z) + u_in(x, y+1, z)
        + u_in(x, y, z-1) + u_in(x, y, z+1);
      u2[x] = u_in(x, y-1, z-1) + u_in(x, y+1, z-1)
        + u_in(x, y-1, z+1) + u_in(x, y+1, z+1);
    }

    for( int x = 1; x < (n1-1); ++x )
    {
      r_out(x, y, z) = v_in(x, y, z)
        - a_in(0) * u_in(x, y, z)
        - a_in(2) * ( u2[x] + u1[x-1] + u1[x + 1] )
        - a_in(3) * ( u2[x-1] + u2[x+1] );

    }
  }
}

COMPONENT_TARGET(mgrid_resid)
{
  INITIALISE(mgridresid) init();

  POLYHEDRAL_LOOP(z)
  [
    z < n3-1;
    z >= 1;
  ]
  {
    POLYHEDRAL_LOOP(y)
    [
      y < n2-1;
      y >= 1;
    ]
    {
      OP(mgridresid) mgridresid();
    }
  }
  CLEANUP(mgridresid) finish();
}
```

A.1.9 The smoothing component for the multigrid example.

The interface specification for the smoothing component of the multigrid example.

```
<interface id="mgrid_psinv">
  <operation name="mgrid_psinv">
    <input type="integer" format="scalar" name="n1" />
    <input type="integer" format="scalar" name="n2" />
    <input type="integer" format="scalar" name="n3" />
    <input type="integer" format="scalar" name="k" />
    <input type="integer" format="scalar" name="m" />
    <input type="double" format="array(4)" name="c_in" />
    <input type="double" format="array(GRIDX, GRIDY, GRIDZ)"
      name="r_in" />
    <input type="double" format="array(GRIDX, GRIDY, GRIDZ)"
      name="u_in" />
    <output type="double" format="array(GRIDX, GRIDY, GRIDZ)"
      name="u_out" />
  </operation>
</interface>
```

The component instantiation of the smoothing component for the multigrid example.

```
<component id="mgrid_psinv">
  <implementation type="cloog">
    <location>mgrid_psinv_el.cloog.componentsource</location>
  </implementation>

  <implements id="mgrid_psinv" />

  <operation name="mgrid_psinv" >
    <constraint type="dependentregion" shape="rectangle">
      <constraintoutput
        name="c_in" ranges="(0->3)" />
      <constraintinput
        name="u_in" placement="relative"
        ranges="(0, 0, 0)" />
      <constraintinput
        name="r_in" placement="relative"
        ranges="(1, 1, 1)" />
      <constraintoutput name="u_out" ranges="(0, 0, 0)" />
    </constraint>
  </operation>
</component>
```

The polyhedral source for the smoothing component of the multigrid example.

```
CODE_BLOCK(mgridpsinv)  THREADPRIVATE( r1, r2 )
{
  double *r1, *r2;
  volatile int cyclecomm;

  static void inline init(
    Array1d<double> &a_in,
    int k, int m, int n1, int n2, int n3,
    Array3d<double> &u_in, Array3d<double> &v_in, Array3d<double> &r_out,
```

```
    double **r1, double **r2)
  {
    cyclecomm = 0;
    *r1 = new double[m];
    *r2 = new double[m];
  }

  static void inline finish(
    Array1d<double> &a_in,
    int k, int m, int n1, int n2, int n3,
    Array3d<double> &u_in, Array3d<double> &v_in, Array3d<double> &r_out,
    double **r1, double **r2)
  {
    delete [] *r1;
    delete [] *r2;
  }


  static void inline mgridpsinv(
    int z, int y, Array1d<double> &c_in,
    int k, int m, int n1, int n2, int n3,
    Array3d<double> &r_in, Array3d<double> &u_in, Array3d<double> &u_out,
    double *r1, double *r2, int block)
  {
    if( (z == n3-3 && y == 1) ) {
      cyclecomm = 1;
    }


    for( int x = 0; x < n1; ++x )
    {
      r1[x] = r_in(x, y-1, z) + r_in(x, y+1, z)
        + r_in(x, y, z-1) + r_in(x, y, z+1);
      r2[x] = r_in(x, y-1, z-1) + r_in(x, y+1, z-1)
        + r_in(x, y-1, z+1) + r_in(x, y+1, z+1);
    }


    for( int  x = 1; x < (n1-1); ++x )

    {
      u_out(x, y, z) = u_in(x, y, z)
        + c_in(0) * r_in(x, y, z)
        + c_in(1) * (r_in(x-1, y, z) + r_in(x+1, y, z)
          + r1[x] )
        + c_in(2) * (r2[x] + r1[x-1] + r1[x+1] );
    }

    // Wrap the end elements correctly
    u_out(0, y, z) = u_out(n1-2, y, z);
    u_out(n1-1, y, z) = u_out(1, y, z);
  }

}

COMPONENT_TARGET(mgrid_psinv)
{
  INITIALISE(mgridpsinv) init();
```

```
  POLYHEDRAL_LOOP(z)
  [
    z < n3-2;
    z >= 1;
  ]
  {
    POLYHEDRAL_LOOP(y)
    [
      y < n2-1;
      y >= 1;
    ]
    {
        OP(mgridpsinv) mgridpsinv();
    }
  }

  CLEANUP(mgridpsinv) finish();
}
```

A.2 The Æcute model

In this section we can see the code samples that link to the results discussed in Chapter 7. In this code we can particularly see a certain amount of overhead from the class library, which would be cleanly removed with compiler support for Æcute types.

A.2.1 The median filter.

Host code for the median filter example.

```
// Timing and test code removed for space reasons.

int main()
{
  using namespace ICStreams;

  MedianFilter::rgb  *inputDataOrig =
    new  MedianFilter::rgb[height*width + 16];
  MedianFilter::rgb  *outputDataOrig =
    new  MedianFilter::rgb[height*width + 16];
  MedianFilter::rgb  *compareDataOrig =
    new  MedianFilter::rgb[height*width + 16];

  MedianFilter::rgb  *inputData = inputDataOrig;
  MedianFilter::rgb  *outputData = outputDataOrig;
  MedianFilter::rgb  *compareData = compareDataOrig;

  if( (reinterpret_cast< int >(inputData)) % 16 )
    inputData = reinterpret_cast< MedianFilter::rgb* >(
      reinterpret_cast< char* >(inputData) +
      (16-((reinterpret_cast< int >(inputData)) % 16)));
  if( (reinterpret_cast< int >(outputData)) % 16 )
    outputData = reinterpret_cast< MedianFilter::rgb* >(
```

```
      reinterpret_cast< char* >(outputData) +
      (16-((reinterpret_cast< int >(outputData)) % 16)));


  for( int j = 0; j < height; ++j )
  {
    for( int i = 0; i < width; ++i )
    {
      inputData[ j * width + i ].r = i;
      inputData[ j * width + i ].g = j;
      inputData[ j * width + i ].b = j;
      outputData[ j * width + i ].r = 0;
      outputData[ j * width + i ].g = 0;
      outputData[ j * width + i ].b = 0;
      compareData[ j * width + i ].r = 0;
      compareData[ j * width + i ].g = 0;
      compareData[ j * width + i ].b = 0;
    }
  }

  // Slightly smaller iteration space than the entire data set
  IterationSpace2D iterationSpace(
    WIDTH-2*int(KERNEL_DIM/2),
    HEIGHT-2*int(KERNEL_DIM/2) );
  Array2D < MedianFilter::rgb > inputArray( WIDTH, HEIGHT, inputData );
  printf("input: %p\n", (void*)&inputArray);
  Array2D < MedianFilter::rgb > outputArray( WIDTH, HEIGHT, outputData );
  MedianFilter k( iterationSpace, KERNEL_DIM/2, inputArray, outputArray );

  k.execute();
}
```

Kernel class for the median filter.

```
#define OVERLAP_READS
#define OVERLAP_WRITES

#include "kernel.hpp"

namespace ICStreams
{
  class MedianFilter : public StreamKernel< MedianFilter >
  {
    public:
    typedef struct
    {
      float r;
      float g;
      float b;
      float padding;

    } rgb;
    static float EuclideanDistance(rgb p1, rgb p2)
    {
      float dr, dg, db;

      dr = p2.r - p1.r;
      dg = p2.g - p1.g;
```

```
    db = p2.b - p1.b;

    return dr*dr + dg*dg + db*db;
  }

  protected:
  int regionRadius;
  // Translate accesses by KERNEL_DIM/2 to shift correctly in data set
  Region2DPrefetch_R <
    rgb,
    Array2D< rgb >,
    IterationSpace2D,
    Translate2D< int(KERNEL_DIM/2), int(KERNEL_DIM/2) >
    > inputPointSet;
  Point2DPrefetch_W <
    rgb,
    Array2D< rgb >,
    IterationSpace2D,
    Translate2D< int(KERNEL_DIM/2), int(KERNEL_DIM/2) >
    > outputPointSet;

  public:
  MedianFilter(
    const IterationSpace2D &it,
    int regionRadius,
    Array2D< rgb > &inputArray,
    Array2D< rgb > &outputArray ) :

    StreamKernel< MedianFilter >( it ),
    inputPointSet( iterationSpace, inputArray, regionRadius ),
    outputPointSet( iterationSpace, outputArray )
  {
    this->regionRadius = regionRadius;

    // Store point sets so we can use them correctly
    addToSet( &inputPointSet );
    addToSet( &outputPointSet );

    this->finishSetup();
  }

  void kernel( const IterationSpace2D::block_iterator::element_iterator &eit )
  {

    rgb closest;
    rgb average;
    float closestDistance;

    // compute average
    average.r = 0.0f;
    average.g = 0.0f;
    average.b = 0.0f;

    int z;
    z = 0;

    for(z = 0-regionRadius; z < (regionRadius+1); ++z)
    {
```

```
      int w;
      for(w = 0-regionRadius; w < (regionRadius+1); ++w)
      {
        rgb pixel = inputPointSet( eit, w, z );

        average.r += pixel.r;
        average.g += pixel.g;
        average.b += pixel.b;
      }
    }

    int area = (2*regionRadius+1)*(2*regionRadius+1);
    average.r /= area;
    average.g /= area;
    average.b /= area;

    closest = inputPointSet( eit, 0, 0 );
    closestDistance = EuclideanDistance(average, closest);

    for(z = 0-regionRadius; z < (regionRadius+1); ++z)
    {
      int w;
      for(w = 0-regionRadius; w < (regionRadius+1); ++w)
      {
        rgb pixel = inputPointSet( eit, w, z );

        float currentDistance;

        currentDistance = EuclideanDistance(average, pixel);
        if(currentDistance < closestDistance)
        {
          closestDistance = currentDistance;
          closest = pixel;
        }
      }
    }
    outputPointSet( eit, closest);
  }

  // State structure for all the regions in the kernel
  struct StateStruct
  {
    StreamKernel< MedianFilter >::StateStruct streamKernel;
    Region2DPrefetch_R <
      rgb,
      Array2D< rgb >,
      IterationSpace2D,
      Translate2D< int(KERNEL_DIM/2), int(KERNEL_DIM/2) >
      >::StateStruct inputPointSet;
    Point2DPrefetch_W <
      rgb,
      Array2D< rgb >,
      IterationSpace2D,
      Translate2D< int(KERNEL_DIM/2), int(KERNEL_DIM/2) >
      >::StateStruct outputPointSet;
    int regionRadius;
  };
```

```
    MedianFilter( const StateStruct &state ) :
      StreamKernel< MedianFilter >( state.streamKernel ),
      regionRadius( state.regionRadius ),
      inputPointSet( iterationSpace, state.inputPointSet ),
      outputPointSet( iterationSpace, state.outputPointSet )
    {
      // Store point sets so we can use them correctly
      addToSet( &inputPointSet );
      addToSet( &outputPointSet );
    }

    void saveState( StateStruct & state )
    {
      StreamKernel< MedianFilter >::saveState( state.streamKernel );
      inputPointSet.saveState( state.inputPointSet );
      outputPointSet.saveState( state.outputPointSet );
      state.regionRadius = regionRadius;
    }
  };
}

#define KERNEL_LIST ("MedianFilter")
```

A.2.2 The matrix/vector multiplication example.

Host code for the matrix/vector multiplication example.

```
using namespace ICStreams;

int main()
{
  using namespace ICStreams;

  float *pInputMatrix = new  float[height*width];
  float  *pInputVector = new  float[width];
  float  *pOutputVector = new  float[height  * 4];
  float  *pCompareVector = new  float[height * 4];

  for( int j = 0; j < height; ++j )
  {
    for( int i = 0; i < width; ++i )
    {
      pInputMatrix[ j * width + i ] = i + 1;
    }
  }

  for( int i = 0; i < width; ++i )
  {
    pInputVector[ i ] = i + 1;
  }

  for( int i = 0; i < height; ++i )
  {
    pOutputVector[ i ] = 0.0;
    pCompareVector[ i ] = 0.0;
  }
```

```
  // Slightly smaller iteration space than the entire data set
  IterationSpace2D iterationSpace( WIDTH, HEIGHT, true );
  Array2D < float > inputMatrix( WIDTH, HEIGHT, pInputMatrix);
  Array2D < float > inputVector( WIDTH, 1, pInputVector );
  Array2D < float > outputVector( 1, HEIGHT, pOutputVector );
  MatrixVectorMul k( iterationSpace, inputMatrix, inputVector, outputVector );

  k.execute();

  double diffsum = 0;
  for( int i = 0; i < height; ++i )
  {
    double diff = (pOutputVector[i] - pCompareVector[i]);
    diffsum += diff * diff;
  }
}
```

Kernel class for the matrix/vector multiplication.

```
namespace ICStreams
{
  class MatrixVectorMul : public StreamKernel< MatrixVectorMul >
  {
    protected:
    // Translate accesses by KERNEL_DIM/2 to shift correctly in data set
    Point2DPrefetch_R < float, Array2D< float >, IterationSpace2D > inputMatrix;
    Point2DPrefetch_R <
      float, Array2D< float >, IterationSpace2D, Scale2D< 1, 0 >
      > inputVector;
    // This one is read and write
    Point2DPrefetch <
      float, Array2D< float >, IterationSpace2D, Scale2D< 0,1 >
      > outputVector;

    public:
    MatrixVectorMul(
      const IterationSpace2D &it,
      Array2D< float > &aInputMatrix,
      Array2D< float > &aInputVector,
      Array2D< float > &aOutputVector ) :
      StreamKernel< MatrixVectorMul >( it ),
      inputMatrix( iterationSpace, aInputMatrix ),
      inputVector( iterationSpace, aInputVector ),
      outputVector( iterationSpace, aOutputVector)
    {
      // Store point sets so we can use them correctly
      addToSet( &inputMatrix );
      addToSet( &inputVector );
      addToSet( &outputVector );

      this->finishSetup();
    }

    void kernel( const IterationSpace2D::block_iterator::element_iterator &eit )
    {
      outputVector( eit ) += inputVector( eit ) * inputMatrix( eit );
    }
```

```
    // State structure for all the regions in the kernel
    struct StateStruct
    {
      StreamKernel< MatrixVectorMul >::StateStruct streamKernel;

      Point2DPrefetch_R <
        float, Array2D< float >, IterationSpace2D
        >::StateStruct inputMatrix;
      Point2DPrefetch_R <
        float, Array2D< float >, IterationSpace2D, Scale2D< 1, 0 >
        >::StateStruct inputVector;
      Point2DPrefetch <
        float, Array2D< float >, IterationSpace2D, Scale2D< 0, 1 >
        >::StateStruct outputVector;
    };
    MatrixVectorMul( const StateStruct &state ) :
      StreamKernel< MatrixVectorMul >( state.streamKernel ),
      inputMatrix( iterationSpace, state.inputMatrix ),
      inputVector( iterationSpace, state.inputVector ),
      outputVector( iterationSpace, state.outputVector )
    {
      // Store point sets so we can use them correctly
      addToSet( &inputMatrix );
      addToSet( &inputVector );
      addToSet( &outputVector );
    }
    void saveState( StateStruct & state )
    {
      StreamKernel< MatrixVectorMul >::saveState( state.streamKernel );
      inputMatrix.saveState( state.inputMatrix );
      inputVector.saveState( state.inputVector );
      outputVector.saveState( state.outputVector );
    }
  };
}
```

A.2.3 The bit-reversal example.

Host code for the bit-reverse example.

```
const int blockCount = _N/(_B*_B);

int main()
{
  using namespace ICStreams;
  TYPE  *pInputData = new  TYPE [height*width];
  TYPE   *pOutputData = new  TYPE [height  * width];

  pInputData[0] = 3;

  // Slightly smaller iteration space than the entire data set
  IterationSpace2D iterationSpace( 1, blockCount );
  Array2D < TYPE > inputData( width, height, pInputData);
  Array2D < TYPE  > outputData( width, height, pOutputData );
  BitReverse k( iterationSpace, inputData, outputData );

  k.iterationSpace.setBlockSize( 1, 1 );
```

```
  k.execute();
}
```

Kernel class for the bit-reverse example.

```
template < class additionalConverter = Identity< Coordinate2D > >
  class BitReverseY : public AggregateConverter<  additionalConverter  >
{
  public:
  Coordinate2D local( const Coordinate2D &in )
  {
    return this->_conv.local( in );
  }

  Coordinate2D globalRead( const Coordinate2D &in )
  {
    int block = (in.y & (~(((1<<_b)-1))))>>_b;
    int row = in.y & ((1<<_b)-1);

    int newaddr = (row << (_n-2*_b)) | block;

    char string1[33];
    char string2[33];
    char string3[33];

    Coordinate2D ret( in.x, newaddr );

    return ret;
  }

  Coordinate2D globalWrite( const Coordinate2D &in )
  {
    int block = (in.y & (~(((1<<_b)-1))))>>_b;
    int row = in.y & ((1<<_b)-1);

    int newaddr = (row << (_n-2*_b)) | reverse_bits(_n-(2*_b),  block);

    Coordinate2D ret( in.x, newaddr );

    return ret;
  }
};

namespace ICStreams
{
  // Permutation constants
  const vec_uchar16 lo4 = ...
  const vec_uchar16 hi4 = ...

  const vec_uint4 vincr = ...
  const vec_uint4 vsize = ...

  class BitReverse : public StreamKernel< BitReverse >
  {
    public:

    protected:
```

```
// Translate accesses by KERNEL_DIM/2 to shift correctly in data set
Region2DFrom0PrefetchBDScatter <
  TYPE, Array2D< TYPE >,
  IterationSpace2D,
  BitReverseY< Scale2D< 1, BLOCKHEIGHT >  >
  > data;
// Bit-reversed indices
uint32 sigma[_Q] ALIGN(128);

public:
BitReverse( const IterationSpace2D &it,
  Array2D< TYPE > &aInputData,
  Array2D< TYPE > &aOutputData ) :
  StreamKernel< BitReverse >( it ),
  data( iterationSpace,
    aInputData, aOutputData,
    (unsigned int)(BLOCKWIDTH), (unsigned int)(BLOCKHEIGHT) )
{
  // Store point sets so we can use them correctly
  addToSet( &data );
  this->finishSetup();
}

void kernel( const IterationSpace2D::block_iterator::element_iterator &eit )
{
  TYPE *t = data._currentBuffer;

  vector TYPE T[8];
  TYPE * t0, * t1;

  // Do in place permutation
  unsigned int l, rev_l;
  for(l = 0; l < _Q; ++l) {
    rev_l = sigma[l];

    if(rev_l < l) {
      continue; // already processed
    }
    // else not yet processed
    t0 = t + (l << _w);

    // Load
    T[0] = LOAD(o0, t0);
    T[1] = LOAD(o1, t0);
    T[2] = LOAD(o2, t0);
    T[3] = LOAD(o3, t0);

    // Interleave (round 1)
    T[4] = ILVlo(T[0], T[1]);
    T[5] = ILVhi(T[0], T[1]);
    T[6] = ILVlo(T[2], T[3]);
    T[7] = ILVhi(T[2], T[3]);

    // Interleave (round 2)
    T[0] = ILVlo(T[4], T[6]);
    T[2] = ILVhi(T[4], T[6]);
    T[1] = ILVlo(T[5], T[7]);
    T[3] = ILVhi(T[5], T[7]);
```

```
    if(rev_l == l) {

      // Store
      STORE(T[0], o0, t0);
      STORE(T[1], o1, t0);
      STORE(T[2], o2, t0);
      STORE(T[3], o3, t0);

    } else { // rev_l > l, swap

      t1  = t + (rev_l << _w);

      // Load
      T[4] = LOAD(o0, t1);
      T[5] = LOAD(o1, t1);
      T[6] = LOAD(o2, t1);
      T[7] = LOAD(o3, t1);

      // Store
      STORE(T[0], o0, t1);
      STORE(T[1], o1, t1);
      STORE(T[2], o2, t1);
      STORE(T[3], o3, t1);

      // Interleave (round 1)
      T[0] = ILVlo(T[4], T[5]);
      T[1] = ILVhi(T[4], T[5]);
      T[2] = ILVlo(T[6], T[7]);
      T[3] = ILVhi(T[6], T[7]);

      // Interleave (round 2)
      T[4] = ILVlo(T[0], T[2]);
      T[6] = ILVhi(T[0], T[2]);
      T[5] = ILVlo(T[1], T[3]);
      T[7] = ILVhi(T[1], T[3]);

      // Store
      STORE(T[4], o0, t0);
      STORE(T[5], o1, t0);
      STORE(T[6], o2, t0);
      STORE(T[7], o3, t0);
    }
  }
}


// State structure for all the regions in the kernel
struct StateStruct
{
  StreamKernel< BitReverse >::StateStruct streamKernel;
  Region2DFrom0PrefetchBDScatter <
    TYPE, Array2D< TYPE >,
    IterationSpace2D,
    BitReverseY< Scale2D< 1, BLOCKHEIGHT > >
    >::StateStruct data;
};
```

```
    void init_sigma(unsigned int q
           , unsigned int * sigma)
    {
      unsigned int k, l;
      unsigned int n0; // $N_{k-1}$
      unsigned int n1; // $N_k$

      sigma[0] = 0;
      sigma[1] = 1 << (q-1);
      sigma[2] = 1 << (q-2);
      sigma[3] = 3 * sigma[2];

      n0 = 3;
      for(k = 3; k <= q; ++k) {
        n1 = (1 << k) - 1;
        sigma[n1] = sigma[n0] + (1 << (q - k));
        for(l = 1; l <= n0; ++l) {
          sigma[n1 - l] = sigma[n1] - sigma[l];
        }
        n0 = n1;
      }
    }

    BitReverse( const StateStruct &state ) :
      StreamKernel< BitReverse >( state.streamKernel ),
      data( iterationSpace, state.data )
    {
      // Store point sets so we can use them correctly
      addToSet( &data );
      init_sigma(_q, sigma);

    }

    void saveState( StateStruct & state )
    {
      StreamKernel< BitReverse >::saveState( state.streamKernel );
      data.saveState( state.data );
    }

  };
}
```

Bibliography

[AC93] B Alpern and L Carter. Towards a Model for Portable Parallel Performance: Exposing the Memory Hierachy. In *Workshop on Potability and Performance for Parallel Processing*, July 1993.

[ACF95] B Alpern, L Carter, and J Ferrante. Space-limited procedures: a methodology for portable high-performance. In *Programming Models for Massively Parallel Computers, 1995*, pages 10–17, 1995.

[AJR+03] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato, and Lawrence Rauchwerger. STAPL: An adaptive, generic parallel C++ library. pages 195–210, 2003.

[AK02] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, San Francisco, CA, USA, 2002.

[AKO] T. J. Ashby, A. D. Kennedy, and M.F.P. O'Boyle. Cross component optimisation in a high level category-based language. *Lecture Notes in Computer Science*.

[atl09] ATLAS homepage. http://math-atlas.sourceforge.net/, 2009.

[BACD97] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and James Demmel. Optimizing matrix multiply using phipac: A portable, high-performance, ANSI C coding methodology. In *International Conference on Supercomputing*, pages 340–347, 1997.

[Bas04] Cedric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society.

[BBK+08] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 1–10, New York, NY, USA, 2008. ACM.

[BCG+03] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. Putting polyhedral loop transformations to work. In *The 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, volume 2958 of *Lecture Notes in Computer Science*, pages 209–225. Springer, October 2003.

[BD02] Tom Boyd and Partha Dasgupta. Process migration: A generalized approach using a virtualizing operating system. In *ICDCS '02: Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, page 385, Washington, DC, USA, 2002. IEEE Computer Society.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[Ber00] G. Berry. The foundations of Esterel. In *Proof, language, and interaction: essays in honour of Robin Milner*, pages 425–454. MIT Press, Cambridge, MA, USA, 2000.

[BFH+04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream computing on graphics hardware. In *SIGGRAPH*, 2004.

[BGS94] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[BGT08] Luciano Baresi, Sam Guinea, and Giordano Tamburrelli. Towards decentralized self-adaptive component-based systems. In *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems (SEAMS)*, pages 57–64, New York, NY, USA, 2008. ACM.

[BHKM03] O. Beckmann, A. Houghton, P. H. J. Kelly, and M. Mellor. Run-time code generation in C++ as a foundation for domain-specific optimisation. In *Domain-Specific Program Generation International Seminar*, 2003.

[BHRS08] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Not.*, 43(6):101–113, 2008.

[BJK+96] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *J. Parallel Distrib. Comput.*, 37(1):55–69, 1996.

[BJK02] Olav Beckmann, Thiyagalingam Jeyarajan, and Paul Kelly. A review of data placement optimisation for data-parallel component composition. In *2nd international workshop on constructive methods for parallel programming, Pont de Lima, Portugal*, 2002.

[Ble93] Guy E. Blelloch. Nesl: A nested data-parallel language (version 2.6). Technical report, Pittsburgh, PA, USA, 1993.

[BPBL06] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. CellSs: a programming model for the Cell BE architecture. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 86, New York, NY, USA, 2006. ACM.

[BRT93] Peter L. Bird, Alasdair Rawsthorne, and Nigel P. Topham. The effectiveness of decoupling. In *ICS '93: Proceedings of the 7th international conference on Supercomputing*, pages 47–56, New York, NY, USA, 1993. ACM.

[Bud88] Timothy A. Budd. Composition and compilation in functional programming languages. Technical report, Corvallis, OR, USA, 1988.

[BWSF06] Carsten Benthin, Ingo Wald, Michael Scherbaum, and Heiko Friedrich. Ray Tracing on the CELL Processor. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, 2006.

[CBK+] Phillip Colella, John Bell, Noel Keen, Terry Ligocki, Michael Lijewski, and Brian van Straalen. Performance and scaling of locally-structured grid methods for partial differential equations. Presented at SciDAC 2007 Annual Meeting.

[CCDS04] Bradford L. Chamberlain, Sung-Eun Choi, Steven J. Deitz, and Lawrence Snyder. The high-level parallel language zpl improves productivity and performance. In *Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing*, 2004.

[CCLHa] Ben Cope, Peter Y.K. Cheung, Wayne Luk, and Lee Howes. Performance comparison of graphics processors to reconfigurable logic: A case study. *Transactions on Computers*.

[CCLHb] Ben Cope, Peter Y.K. Cheung, Wayne Luk, and Lee Howes. A systematic design space exploration approach to customising multi-processor architectures: Exemplified using graphics processors. *SAMOS journal*.

[CCZ04] D. Callahan, B.L. Chamberlain, and H.P. Zima. The cascade high productivity language. In *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004*, pages 52–60, April 2004.

[CCZ07] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007.

[CDS00] Bradford L. Chamberlain, Steven J. Deitz, and Lawrence Snyder. A comparative study of the NAS MG benchmark across parallel languages and architectures. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, page 46, Washington, DC, USA, 2000. IEEE Computer Society.

[CG98] Larry Carter and Kang Su Gatlin. Towards an optimal bit-reversal permutation program. In *Proceedings of Foundations of Computer Science (FOCS)*, pages 544–555, 1998.

[CGT04] Albert Cohen, Sylvain Girbal, and Olivier Temam. A polyhedral approach to ease the composition of program transformations. In *Euro-Par '04*, number 3149 in Lecture Notes in Computer Science. Springer-Verlag, August 2004.

[CHK+09] Jay L.T. Cornwall, Lee Howes, Paul H.J. Kelly, Phil Parsonage, and Bruno Nicoletti. High-performance simt code generation in an active visual effects library. In *CF '09: Proceedings of the 6th ACM conference on Computing frontiers*, pages 175–184, New York, NY, USA, 2009. ACM.

[CKPN07] Jay L. T. Cornwall, Paul H. J. Kelly, Phil Parsonage, and Bruno Nicoletti. Explicit dependence metadata in an active visual effects library. In *Proceedings of the 20th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*. Springer LNCS, October 2007.

[CLLS98] Bradford L. Chamberlain, E. Christopher Lewis, Calvin Lin, and Lawrence Snyder. Regions: an abstraction for expressing array computation. *SIGAPL APL Quote Quad*, 29(2):41–49, 1998.

[CLO] CLooG. http://www.cloog.org/, retrieved 2009.

[CLPT02] Siddhartha Chatterjee, Alvin R. Lebeck, Praveen K. Patnala, and Mithuna Thottethodi. Recursive array layouts and fast matrix multiplication. *IEEE Trans. Parallel Distrib. Syst.*, 13(11):1105–1123, 2002.

[CM08] Gilberto Contreras and Margaret Martonosi. Characterizing and improving the performance of the intel threading building blocks runtime system. In *International Symposium on Workload Characterization (IISWC 2008)*, September 2008.

[Cod] Codeplay Software. Portable high-performance compilers. http://www.codeplay.com/, retrieved 2008.

[Coma] DRC Computer. DRC RPU100. http://www.drccomputer.com/pdfs/DRC_RPU100_datasheet.pdf, retrieved 2008.

[Comb] SRC Computers. SRC-7 product sheet. http://www.srccomp.com/Product Sheets/, retrieved 2008.

[Cor04] Randima Fernando, NVIDIA Corporation. Trends in GPU evolution. In *Eurographics*, September 2004.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: a declarative language for real-time programming. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 178–188, New York, NY, USA, 1987. ACM Press.

[CRM+07] Paul Carpenter, David Rodenas, Xavier Martorell, Alex Ramirez, and Eduard Ayguadá. A streaming machine description and programming model. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 107–116, 2007.

[CRM+08] Paul Carpenter, Alex Ramirez, Xavier Martorell, David Rodenas, and Roger Ferrer. Report on streaming programming model and abstract streaming machine description. Technical report, UPC, 2008.

[CSG+05] Albert Cohen, Marc Sigler, Sylvain Girbal, Olivier Temam, David Parello, and Nicolas Vasilache. Facilitating the search for compositions of program transformations. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 151–160, New York, NY, USA, 2005. ACM.

[CTHL03] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing multi-stage languages using ASTs, Gensym, and reflection. In *GPCE '03: Proceedings of the 2nd international conference on Generative programming and component engineering*, pages 57–76, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[DBR+05] Sebastien Donadio, James Brodman, Thomas Roeder, Kamen Yotov, Denis Barthou, Albert Cohen, Mara Jesus Garzaran, David Padua, and Keshav Pingali. A language for the compact representation of multiple program versions. In *Languages and Compilers for Parallel Computing (LCPC)*, 2005.

[DC01] Jim Dowling and Vinny Cahill. The k-component architecture meta-model for self-adaptive software. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 81–88, London, UK, 2001. Springer-Verlag.

[DFH+93] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While. Parallel programming using skeleton functions. In A. Bode, M. Reeve, and G. Wolf, editors, *Parallel Architectures and Languages Europe (PARLE)*, pages 146–160. Springer-Verlag, Berlin, DE, 1993.

[DLD+03] William J. Dally, Francois Labonte, Abhishek Das, Patrick Hanrahan, Jung-Ho Ahn, Jayanth Gummaraju, Mattan Erez, Nuwan Jayasena, Ian Buck, Timothy J. Knight, and Ujval J. Kapasi. Merrimac: Supercomputing with streams. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 35, Washington, DC, USA, 2003. IEEE Computer Society.

[DM97] James Dundas and Trevor Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 68–75, New York, NY, USA, 1997. ACM.

[DM98] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, 1998.

[Edw00] Stephen A. Edwards. Compiling esterel into sequential code. In *DAC '00: Proceedings of the 37th conference on Design automation*, pages 322–327, New York, NY, USA, 2000. ACM Press.

[EGD+05] Arkady Epshteyn, Maria Garzaran, Gerald DeJong, David Padua, Gang Ren, Xiaoming Li, Kamen Yotov, and Keshav Pingali. Analytic models and empirical search: A hybrid approach to code optimization. In *Languages and Compilers for Parallel Computing*, 2005.

[EGL98] Nils Ellmenreich, Martin Griebl, and Christian Lengauer. Applicability of the polytope model to functional programs. In *Languages and Compilers for Parallel Computing (LCPC)*, May 1998.

[FBK98] Stephen J. Fink, Scott B. Baden, and Scott R. Kohn. Efficient run-time support for irregular block-structured applications. *J. Parallel Distrib. Comput.*, 50(1-2):61–82, 1998.

[FC07] Grigori Fursin and Albert Cohen. Building a practical iterative interactive compiler. In *1st International Workshop on Statistical and Machine Learning Approaches Applied to Architectures and Compilation*, January 2007.

[Fea96] Paul Feautrier. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, pages 79–103, London, UK, 1996. Springer-Verlag.

[FHK+06] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: programming the memory hierarchy. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 83, New York, NY, USA, 2006. ACM Press.

[FHPM02] Fang Fang, James C. Hoe, Markus Püschel, and Smarahara Misra. Generation of custom DSP transform IP cores: Case study Walsh-Hadamard transform. In *Proceedings of HPEC*, 2002.

[FJ98] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998.

[FJ05] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[FMM+02] Nathalie Furmento, Anthony Mayer, Stephen McGough, Steven Newhouse, A. J. Field, and John Darlington. ICENI: Optimisation of component applications within a Grid environment. *Parallel Computing*, 28(12):1753–1772, December 2002.

[FPWW09] Robert Fisher, Simon Perkins, Ashley Walker, and Erik Wolfart. Image processing learning resources. `http://homepages.inf.ed.ac.uk/rbf/HIPR2/`, 2000–2009.

[Gas08] Benedict R. Gaster. Streams: Emerging from a shared memory model. In Rudolf Eigenmann and Bronis R. de Supinski, editors, *Proceedings of the 4th International Workshop on OpenMP (IWOMP)*, Lecture Notes in Computer Science, pages 134–145. Springer, 2008.

[GC99] Kang Su Gatlin and Larry Carter. Architecture-cognizant divide and conquer algorithms. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 25, New York, NY, USA, 1999. ACM Press.

[GC05] Jonathan Greene and Robert Cooper. A parallel 64k complex FFT algorithm for the IBM/Sony/Toshiba Cell Broadband Engine processor. Technical report, Mercury Computer Systems, 2005.

[GG07] Kazushige Goto and Robert A. Van De Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 2007.

[Ghu07] Anwar Ghuloum. Ct: channelling NeSL and SISAL in c++. In *CUFP '07: Proceedings of the 4th ACM SIGPLAN workshop on commercial users of functional programming*, pages 1–3, New York, NY, USA, 2007. ACM.

[GL05] Samual Guyer and Calvin Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. *Proceedings of the IEEE: Special Issue on Program Generation, Optimization, and Platform Adaptation*, 2005.

[GLS] OpenGL Shading Language. http://www.opengl.org/documentation/glsl/, retrieved 2008.

[GLW98] M. Griebl, C. Lengauer, and S Wetzel. Code generation in the polytope model. In *Parallel Architectures and Compilation Techniques*, October 1998.

[GM02] H.-G. Gross and N.F. Mayer. Search-based execution-time analysis in component-oriented real-time application development. In *13th Intl. Symposium on Software Reliability Engineering*, November 2002.

[Got] Kazushige Goto. Goto BLAS FAQ. http://www.tacc.utexas.edu/resources/software/gotoblasfaq.php retrived 2008.

[gpg] General-purpose computation using graphics hardware. http://www.gpgpu.org, retrived 2008.

[GPM04] Aca Gacic, Markus Pschel, and Jos Moura. Automatically generated high-performance code for discrete wavelet transforms. In *Proc. International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2004.

[Gri96] Martin Griebl. *The Mechanical Parallelization of Loop Nests Containing while Loops*. PhD thesis, University of Passau, 1996. Also available as technical report MIP-9701.

[Gri04] Martin Griebl. *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. University of Passau, 2004. habilitation thesis.

[Grö09] Armin Größlinger. Precise management of scratchpad memories for localising array accesses in scientific codes. In *Proceedings of the International Conference on Compiler Construction (CC 2009)*, LNCS 5501. Springer, 2009.

[Gus02] Jan Gustafsson. Worst case execution time analysis of object-oriented programs. In *WORDS '02: Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, page 71, Washington, DC, USA, 2002. IEEE Computer Society.

[GVB+06] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Int. J. Parallel Program.*, 34(3):261–317, 2006.

[GvdG02] Kazushige Goto and Robert van de Geijn. On reducing TLB misses in matrix multiplication. Technical report, University of Texas at Austin, 2002.

[HBM+06] Lee Howes, Olav Beckmann, Oskar Mencer, Oliver Pell, and Paul Price. Comparing FPGAs to Graphics Accelerators and the Playstation 2 Using a Unified Source Description. In *International Conference on Field-Programmable Logic*, 2006.

[HLDK09a] Lee Howes, Anton Lokhmotov, Alastair F. Donaldson, and Paul H.J. Kelly. Decoupled Access/Execute metaprogramming for GPU-accelerated systems. In *Symposium on Application Accelerators in High Performance Computing (SAAHPC)*, 2009.

[HLDK09b] Lee Howes, Anton Lokhmotov, Alastair F. Donaldson, and Paul H.J. Kelly. Towards metaprogramming for parallel systems on a chip. In *Proceedings of the 3rd Workshop on Highly Parallel Processing on a Chip (HPPC)*, Lecture Notes in Computer Science. Springer, 2009.

[HLDK09c] Lee W. Howes, Anton Lokhmotov, Alastair F. Donaldson, and Paul H.J. Kelly. Deriving efficient data movement from decoupled access/execute specifications. In *Proceedings of the 4th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC)*, volume 5409 of *Lecture Notes in Computer Science*, pages 168–182. Springer, 2009.

[HLKD08] Lee W. Howes, Anton Lokhmotov, Paul Kelly, and Alastair Donaldson. Automating generation of data movement code for processors with distributed memories. In *Presented at the 5th HiPEAC industrial workshop*, June 2008.

[HLKF08a] Lee W. Howes, Anton Lokhmotov, Paul H.J. Kelly, and A. J. Field. Optimising component composition using indexed dependence metadata. In *Proceedings of the 1st International Workshop on New Frontiers in High-performance and Hardware-aware Computing*, 2008.

[HLKF08b] Lee W. Howes, Anton Lokhmotov, Paul H.J. Kelly, and A. J. Field. Optimising component composition using indexed dependence metadata. In *Proceedings of the 1st International Workshop on New Frontiers in High-performance and Hardware-aware Computing*, 2008.

[Hof05] H. Peter Hofstee. Power efficient processor architecture and the Cell processor. In *Proceedings of the 11th International Conference on High-Performance Computer Architecture (HPCA)*, pages 258–262. IEEE Computer Society, 2005.

[How05] Lee Howes. ASC2GPU - Stream Compilation with Graphics Cards. Master's thesis, Imperial College London, 2005.

[HPMB06] Lee Howes, Oliver Pell, Oskar Mencer, and Olav Beckmann. Accelerating the Development of Hardware Accelerators. In *Workshop on Edge Computing, North Carolina, USA, May 2006*, 2006.

[HT07] Lee Howes and David Thomas. Efficient random number generation and application using CUDA. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 37, pages 805–830. Addison Wesley, 2007.

[Inc] Cray Inc. Cray XD1. http://www.cray.com/downloads/Cray_XD1_Datasheet.pdf, retrived 2009.

[Int] Intel. *Intel SSE4 Programming Reference*. http://software.intel.com/file/18187/, retrieved 2009.

[Int08] Intel. Intel math kernel library, 2008. http://www.intel.com/cd/software/products/asmona/eng/307757.htm, retrieved 2008.

[Joe96] C. F. Joerg. The Cilk system for parallel multithreaded computing. Technical report, Cambridge, MA, USA, 1996.

[KBFB01] Paul Kelly, Olav Beckmann, A. J. Field, and Scott Baden. THEMIS: Component dependence metadata in adaptive parallel computations. *Parallel Processing Letters*, 11(4):455–470, December 2001.

[KM94] Ken Kennedy and Kathryn S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 301–320, London, UK, 1994. Springer-Verlag.

[KMW67] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, 1967.

[KP06] Volodymyr Kindratenko and David Pointer. A case study in porting a production scientific supercomputing application to a reconfigurable computer. In *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*, pages 13–22, Washington, DC, USA, 2006. IEEE Computer Society.

[KPR+07] Timothy J. Knight, Ji Young Park, Manman Ren, Mike Houston, Mattan Erez, Kayvon Fatahalian, Alex Aiken, William J. Dally, and Pat Hanrahan. Compilation for explicitly managed memory hierarchies. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 226–236, New York, NY, USA, 2007. ACM Press.

[Len93] Christian Lengauer. Loop parallelization in the polytope model. In *CONCUR '93: Proceedings of the 4th International Conference on Concurrency Theory*, pages 398–416, London, UK, 1993. Springer-Verlag.

[LG97] Daniel Leo Lau and Juin Guillermo Gonzalez. The closest-to-mean filter: an edge preserving smoother for Gaussian environments. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2593–2596. IEEE Press, 1997.

[LG05] Xiaoming Li and Mara Jess Garzarn. Optimizing matrix multiplication with a classifier learning system. In *Languages and Compilers for Parallel Computing, 16th International Workshop, (LCPC)*, 2005.

[LLS] Andrew Lumsdaine, Lie-Quan Lee, and Jeremy Siek. Iterative template library. http://www.osl.iu.edu/research/itl/, retrived 2008.

[LME09] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to gpgpu: a compiler framework for automatic translation and optimization. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 101–110, New York, NY, USA, 2009. ACM.

[LMR07] Anton Lokhmotov, Alan Mycroft, and Andrew Richards. Delayed side-effects ease multi-core programming. In *Proceedings of the 13th European Conference on Parallel and Distributed Computing (Euro-Par)*, volume 4641 of *Lecture Notes in Computer Science*. Springer, 2007.

[LSSP02] Stefan M. Larson, Christopher D. Snow, Michael Shirts, and Vijay S. Pande. Folding@home and genome@home: Using distributed computing to tackle previously intractable problems in computational biology. *Computational Genomics*, 2002.

[MA97] Naraig Manjikian and Tarek S. Abdelrahman. Fusion of loops for parallelism and locality. *IEEE Trans. Parallel Distrib. Syst.*, 8(2):193–209, 1997.

[MCFH97] Nicholas Mitchell, Larry Carter, Jeanne Ferrante, and Karin Hogstedt. Quantifying the multi-level nature of tiling interactions. In *Languages and Compilers for Parallel Computing*, pages 1–15, 1997.

[McK04] Sally A. McKee. Reflections on the memory wall. In *CF '04: Proceedings of the 1st conference on Computing frontiers*, page 162, New York, NY, USA, 2004. ACM.

[MD06] Michael D. McCool and Bruce D'Amora. M08—programming using rapidmind on the cell be. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 222, New York, NY, USA, 2006. ACM Press.

[Men02] Oskar Mencer. PAM-Blox II: Design and evaluation of C++ module generation for computing with FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2002.

[Men06] Oskar Mencer. ASC, a stream compiler for computing with FPGAs. In *IEEE Transactions on CAD*, 2006.

[MGAK03] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. In *SIGGRAPH*, 2003.

[MGS+01] M. Meißner, S. Grimm, W. Straßer, J. Packer, and D. Latimer. Parallel volume rendering on a single-chip SIMD architecture. In *PVG '01: Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics*, pages 107–113, Piscataway, NJ, USA, 2001. IEEE Press.

[MHMF00] Oskar Mencer, Heiko Huebert, Martin Morf, and Michael J. Flynn. StReAm: Object-oriented programming of stream architectures using PAM-Blox. In *Plenary Session, Field-Programmable Logic (FPL)*, September 2000.

[MPHL03] Oskar Mencer, David J. Pearce, Lee W. Howes, and Wayne Luk. Design space exploration with a stream compiler. In *Proceedings of IEEE International Conference on Field Programmable Technology (FPT)*, Dec 2003.

[MQP02] Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. Shader metaprogramming. In *SIGGRAPH/Eurographics Graphics Hardware Workshop*, September 2002.

[MRA+06] Lois C. McInnes, Jaideep Ray, Rob Armstrong, Tamara L. Dahlgren, A. Malony, Boyana Norris, Sameer Shende, Joseph P. Kenny, and Johan Steensland. Computational quality of service for scientific CCA applications: Composition, substitution, and reconfiguration. Technical Report ANL/MCS-P1326-0206, Argonne National Laboratory, February 2006.

[MSWP03] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *HPCA '03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, page 129, Washington, DC, USA, 2003. IEEE Computer Society.

[MYBC03] Hui Ma, I-Ling Yen, Farokh Bastani, and Kendra Cooper. Composition analysis of qos properties for adaptive integration of embedded software components. In *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE)*, page 383, Washington, DC, USA, 2003. IEEE Computer Society.

[NR95] Kelvin D. Nilsen and Bernt Rygg. Worst-case execution time analysis on modern processors. *SIGPLAN Not.*, 30(11):20–30, 1995.

[NR05] Robert W. Numrich and John Reid. Co-arrays in the next Fortran standard. *SIGPLAN Fortran Forum*, 24(2):4–17, 2005.

[NVI] NVIDIA Corporation. *CUDA Programming Guide*.

[ODK+00] John D. Owens, William J Dally, Ujval J. Kapasi, Scott Rixner, Peter Mattson, and Ben Mowery. Polygon rendering on a stream architecture. In *Proceedings of 2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, November 2000.

[OLG+07] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[PAB+05] D. Pham, S. Asano, M. Bolliger, M.N. Day, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation Cell processor. In *Solid-State Circuits Conference*, February 2005.

[PB00] Peter Puschner and Alan Burns. A review of worst-case execution-time analysis. *Journal of Real-Time Systems*, 18(2/3):115–128, May 2000.

[PMJ+05] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gačić, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232–275, 2005.

[pol] Polylib. http://www.irisa.fr/polylib/, retrieved 2008.

[PSC+06] Sebastian Pop, Georges-Andr Silber, Albert Cohen, Cdric Bastoul, Sylvain Girbal, and Nicolas Vasilache. GRAPHITE: Polyhedral analyses and optimizations for GCC. In *GNU Compilers Collection Developers Summit*, 2006.

[PSG06] Mark Peercy, Mark Segal, and Derek Gerstmann. A performance-oriented data parallel virtual machine for GPUs. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*, page 184, New York, NY, USA, 2006. ACM Press.

[QRW00] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *Int. J. Parallel Program.*, 28(5):469–498, 2000.

[RFS+00] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: component composition for systems software. In *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, pages 24–24, Berkeley, CA, USA, 2000. USENIX Association.

[RK98] Gerald Roth and Ken Kennedy. Loop fusion in high performance Fortran. In *ICS '98: Proceedings of the 12th international conference on Supercomputing*, pages 125–132, New York, NY, USA, 1998. ACM.

[RPSV08] T. Ramirez, A. Pajuelo, O.J. Santana, and M. Valero. Runahead threads to improve smt performance. In *IEEE 14th International Symposium on High Performance Computer Architecture, 2008. HPCA 2008.*, pages 149–158, February 2008.

[RVOA08] Ram Rangan, Neil Vachharajani, Guilherme Ottoni, and David I. August. Performance scalability of decoupled software pipelining. *ACM Trans. Archit. Code Optim.*, 5(2):1–25, 2008.

[SDD+07] David E. Shaw, Martin M. Deneroff, Ron O. Dror, Jeffrey S. Kuskin, Richard H. Larson, John K. Salmon, Cliff Young, Brannon Batson, Kevin J. Bowers, Jack C. Chao, Michael P. Eastwood, Joseph Gagliardo, J. P. Grossman, C. Richard Ho, Douglas J. Ierardi, István Kolossváry, John L. Klepeis, Timothy Layman, Christine McLeavey, Mark A. Moraes, Rolf Mueller, Edward C. Priest, Yibing Shan, Jochen Spengler, Michael Theobald, Brian Towles, and Stanley C. Wang. Anton, a special-purpose machine for molecular dynamics simulation. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 1–12, New York, NY, USA, 2007. ACM.

[SGI] SGI. RASC. http://www.sgi.com/pdfs/3920.pdf, retrieved 2009.

[SLAT+07] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. Sketching stencils. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 167–178, New York, NY, USA, 2007. ACM.

[SMC91] Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. Run-time parallelization and scheduling of loops. *IEEE Trans. Comput.*, (5):603–612, 1991.

[Smi84] James E. Smith. Decoupled access/execute computer architectures. *ACM Trans. Comput. Syst.*, 2(4):289–308, 1984.

[Tah04] Walid Taha. A gentle introduction to multi-stage programming. 2004.

[TC08] Marc Tremblay and Shailender Chaudhry. A third-generation 65nm 16-core 32-thread plus 32-scout-thread CMT SPARC processor. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*, 2008.

[The09] The Khronos Group. OpenCL. `http://www.khronos.org/opencl`, 2008–2009.

[THL09] David B. Thomas, Lee Howes, and Wayne Luk. A comparison of cpus, gpus, fpgas, and masssively parallel processor arrays for random number generation. In *Proceedings of FPGA*, 2009.

[TJ01] Deependra Talla and Lizy K. John. Mediabreeze: a decoupled architecture for accelerating multimedia applications. *SIGARCH Comput. Archit. News*, 29(5):62–67, 2001.

[TKA02] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, London, UK, 2002. Springer-Verlag.

[TP05] ClearSpeed Technologies Plc. CSX processor architecture whitepaper. 2005.

[TRM+95] Nigel Topham, Alasdair Rawsthorne, Callum McLean, Muriel Mewissen, and Peter Bird. Compiling and optimizing for decoupled architectures. In *Proceedings of Supercomputing (SC)*, page 40, 1995.

[TS97] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *PEPM '97: Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 203–217, New York, NY, USA, 1997. ACM Press.

[UBLmH08] Sain-Zee Ueng, Sara Baghsorkhi, Melvin Lathara, and Wen mei Hwu. CUDA-lite: Reducing GPU programming complexity. In *Proceedings of Languages and Compilers for Parallel Computing (LCPC)*. Springer, 2008.

[VBCG04] Girish Venkataramani, Mihai Budiu, Tiberiu Chelcea, and Seth Copen Goldstein. C to asynchronous dataflow circuits: An end-to-end toolflow. In *International Workshop on Logic synthesis (IWLS)*, pages 501–508, Temecula, CA, June 2004.

[VD08] Vasily Volkov and James W. Demmel. Benchmarking gpus to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[Ven03] Suresh Venkatasubramanian. The graphics card as a stream computer. In *SIGMOD-DIMACS Workshop on Management and Processing of Data Streams*, 2003.

[VMQ91] Hervé; Le Verge, Christophe Mauras, and Patrice Quinton. The ALPHA language and its use for the design of systolic arrays. *J. VLSI Signal Process. Syst.*, 3(3):173–182, 1991.

[War02] Henry S. Warren. *Hacker's Delight*. Addison-Wesley, 2002.

[Wil97] D. K. Wilde. A library for doing polyhedral operations. Technical Report RR-2157, Brigham Young University, 1997.

[WM95] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, 1995.

[WP05] R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005. http://www.cs.utsa.edu/ whaley/papers/spercw04.ps, retrieved 2009.

[WPD01] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[WR95] Ian Watson and Alasdair Rawsthorne. Decoupled pre-fetching for distributed shared memory. In *Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS)*, pages 252–261, Washington, DC, USA, 1995. IEEE Computer Society.

[Wri08] Cornell Wright. IBM software development kit for multicore acceleration. Roadrunner tutorial LA-UR-08-2819. http://www.lanl.gov/orgs/hpc/roadrunner, retrieved 2009, 2008.

[WSO+06] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM Press.

[Xtr] XtremeData. xd1000. http://www.xtremedatainc.com/xd1000_brief.html, retrieved 2009.

[YLR+05] K. Yotov, Xiaoming Li, Gang Ren, M.J.S. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance BLAS? In *Proceedings of the IEEE*, volume 93, February 2005.

[ZBN93] N. Zhang, A. Burns, and M. Nicholson. Pipelined processors and worst case execution times. *Real-Time Systems*, 5(4):319–343, 1993.