Imperial College London
Department of Computing

A metadata-enhanced framework for high performance visual effects

Jay L. T. Cornwall

April 2010

Submitted in partial fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London.
The work within this thesis is my own except where explicity cited.

Abstract

This thesis is devoted to reducing the interactive latency of image processing computations in visual effects. Film and television graphic artists depend upon low-latency feedback to receive a visual response to changes in effect parameters. We tackle latency with a domain-specific optimising compiler which leverages high-level program metadata to guide key computational and memory hierarchy optimisations. This metadata encodes static and dynamic information about data dependence and patterns of memory access in the algorithms constituting a visual effect – features that are typically difficult to extract through program analysis – and presents it to the compiler in an explicit form. By using domain-specific information as a substitute for program analysis, our compiler is able to target a set of complex source-level optimisations that a vendor compiler does not attempt, before passing the optimised source to the vendor compiler for lower-level optimisation.

Three key metadata-supported optimisations are presented. The first is an adaptation of space and schedule optimisation – based upon well-known compositions of the loop fusion and array contraction transformations – to the dynamic working sets and schedules of a runtime-parameterised visual effect. This adaptation sidesteps the costly solution of runtime code generation by specialising static parameters in an offline process and exploiting dynamic metadata to adapt the schedule and contracted working sets at runtime to user-tunable parameters. The second optimisation comprises a set of transformations to generate SIMD ISA-augmented source code. Our approach differs from autovectorisation by using static metadata to identify parallelism, in place of data dependence analysis, and runtime metadata to tune the data layout to user-tunable parameters for optimal aligned memory access. The third optimisation comprises a related set of transformations to generate code for SIMT architectures, such as GPUs. Static dependence metadata is exploited to guide large-scale parallelisation for tens of thousands of in-flight threads. Optimal use of the alignment-sensitive, explicitly managed memory hierarchy is achieved by identifying inter-thread and intra-core data sharing opportunities in memory access metadata.

A detailed performance analysis of these optimisations is presented for two industrially developed visual effects. In our evaluation we demonstrate up to 8.1x speed-ups on Intel and AMD multicore CPUs and up to 6.6x speed-ups on NVIDIA GPUs over our best hand-written implementations of these two effects. Programmability is enhanced by automating the generation of SIMD and SIMT implementations from a single programmer-managed scalar representation.

Acknowledgements

The road to this thesis has been shaped by many people to whom I am indebted. Some I have never had the pleasure of meeting: their lives' work is trivialised in citations and off-hand remarks littered throughout these chapters with little more than a nod of acknowledgement. To you I am eternally grateful for allowing me to stand on your shoulders. The view is breathtaking. I hope that I am one day able to return the favour.

To my supervisor, Paul Kelly, I owe much of my success and all of my gratitude. His patience, enthusiasm and insights helped to carve this path of research and navigate its many obstacles with deft ability. The incredible network of minds and personalities he has built throughout his distinguished career has had an invaluable effect on the research contained within in this thesis. To my second supervisor, Tony Field, I am grateful for his guidance on writing good papers, which I hope has influenced the text you are now reading.

Of course, we did not work alone. The input of our researchers, past and present, in the Software Performance Optimisation group, and many of my colleagues in the Computer Systems group, ensured that success was the only possible outcome of our research endeavours. I must name those to whom I worked especially closely and enjoyed many academic and technical conversations with: Lee Howes, Ashley Brown, Francis Russell, Will Osborne and Michael Mellor.

I am beholden to my colleagues at The Foundry for the ideas and discussions we shared in finding common ground between our goals. In particular, I would like to thank Phil Parsonage and Bruno Nicoletti for stepping far beyond their lines of duty and endeavouring to believe that our daring academic–industrial collaboration could succeed. I owe thanks to Bill Collis for proposing the collaboration and arranging the funding to realise it.

I must also thank the Engineering and Physical Sciences Research Council (EPSRC) for partly funding our work through an Industrial CASE award. This was a completely new experiment for our research group and we were very pleased with the outcome. I hope the results of our research have met the funding goals spot on.

To my family, who supported me throughout this period of my life while I was unable to give them the time they deserved, I am more grateful than I am able to show. Amidst personal tragedy, my father and sister stood beside me with a devotion that gave me adamant resolve to complete this thesis. To my mother, I am grateful for life. I hope you find peace in your next.

Contents

1 Introduction **1**

1.1 Motivation . . . 1
1.2 Objectives . . . 2
1.3 Contributions . . . 2
1.4 Publications . . . 3
1.5 Thesis Organisation . . . 4

2 Visual Effects Optimisation **6**

2.1 Introduction . . . 6
2.2 Visual Effects . . . 6
2.2.1 Digital Image . . . 6
2.2.2 Digital Composition . . . 7
2.3 Software Optimisation . . . 11
2.3.1 Memory Hierarchy Management . . . 11
2.3.2 Component-Based Programming . . . 13
2.3.3 Generative Programming . . . 15
2.3.4 Metadata-Guided Optimisation . . . 16
2.3.5 Polyhedral Schedule Transformation . . . 16
2.4 Software Parallelisation . . . 19
2.4.1 Symmetric Multiprocessing (SMP) . . . 19
2.4.2 Single Instruction, Multiple Data (SIMD) . . . 21
2.4.3 Single Instruction, Multiple Threads (SIMT) . . . 21
2.5 Related Work . . . 23
2.6 Concluding Remarks . . . 26

3 Metadata-Augmented Single-Source Framework **28**

3.1 Introduction . . . 28
3.2 Constraints upon the Visual Effects Domain . . . 28
3.3 Metadata Definition . . . 31
3.3.1 Primitive Context . . . 32
3.3.2 Data Dependence . . . 32
3.3.3 Memory Access . . . 34

3.4 Framework Design . . . 35
3.4.1 Visual Primitive Representation . . . 36
3.4.2 Visual Effect Construction . . . 37
3.5 Execution Strategy . . . 39
3.5.1 DAG Serialisation . . . 39
3.5.2 DOD Propagation . . . 40
3.5.3 Cache-Aware Iteration . . . 41
3.5.4 Multicore Parallelisation . . . 42
3.6 Code Generation . . . 43
3.6.1 Static Specialisation . . . 44
3.7 Performance Analysis . . . 45
3.8 Concluding Remarks . . . 53

4 Space and Schedule Optimisation 54
4.1 Introduction . . . 54
4.2 Schedule Optimisation . . . 54
4.2.1 Constructing the Constraint Matrices . . . 55
4.2.2 Minimising Polyhedral Scanning Complexity . . . 57
4.2.3 Constructing the Scattering Matrices . . . 60
4.2.4 Impact of Schedule Optimisation on Parallelisation . . . 61
4.2.5 Optimising the Polyhedral Schedule . . . 64
4.3 Space Optimisation . . . 66
4.4 Performance Analysis . . . 67
4.5 Concluding Remarks . . . 75

5 SIMD Code Generation and Optimisation 76
5.1 Transformation Phases . . . 77
5.1.1 Strip Mining . . . 78
5.1.2 Scalar Promotion . . . 80
5.1.3 Divergent Conditional Predication . . . 83
5.1.4 Memory Access Realignment . . . 85
5.1.5 Contracted Load/Store Rescheduling . . . 89
5.2 Performance Analysis . . . 90
5.3 Concluding Remarks . . . 95

6 SIMT Code Generation and Optimisation 98
6.1 Transformation Phases . . . 99
6.1.1 Syntax-Directed Translation . . . 100
6.1.2 Constant and Shared Memory Staging . . . 102
6.1.3 Memory Access Transposition . . . 103
6.1.4 Memory Access Realignment . . . 105

6.1.5 Maximising Parallelism . . . 107
6.1.6 Scheduling Overhead Reduction . . . 108
6.2 Thread Block Size, Shape and Count Selection . . . 108
6.3 Challenges in SIMT Space/Schedule Optimisation . . . 110
6.4 Performance Analysis . . . 112
6.5 Concluding Remarks . . . 117

7 Conclusions and Further Work **120**
7.1 Review of Objectives . . . 120
7.2 Technical Achievements . . . 121
7.3 Critical Analysis . . . 122
7.4 Further Work . . . 123
7.4.1 Compiler-Assistive Metadata . . . 123
7.4.2 Dynamic Runtime Optimisation . . . 124
7.4.3 Heterogeneous Multiprocessing . . . 125
7.5 Conclusions . . . 125

A Framework Ontology **127**
A.1 The Functor Class . . . 127
A.2 The Indexer Class . . . 129

Bibliography **131**

List of Tables

2.1 A condensed matrix representation of the systems of linear equations defining two convex hulls in a two-dimensional iteration space, as shown in Figure 2.4. 18

3.1 Hardware specifications for the three benchmarking platforms used throughout the performance analyses in Section 3.7. The Xeon was in 2-chip SMP configuration. Both Intel chips share partitions of the L2 cache between pairs of cores. 45
3.2 Image resolutions tested in the benchmarks throughout this thesis to identify cache spills and row-to-row memory aliasing effects. 46

4.1 Constraint matrices for the loop nests shown in Listing 4.1. Each loop nest has one constraint matrix, bounding its multidimensional iteration space. Columns representing the coefficients of loop control variables are separated from the coefficients of parameters to the polytope. 57
4.2 Global context matrix used in all loop nest generations. Each matrix row defines a relation between parameters to the constraint matrices and integers. 58
4.3 Scattering matrices for generating an unoptimised loop schedule for the loop nests shown in Listing 4.1. 60
4.4 Scattering matrices for generating a fused loop schedule for the loop nests shown in Listing 4.2. 60

5.1 Constraint matrix for the strip-mined loop nest shown in Listing 5.1. A new iteration variable x' is introduced and an additional constraint links x to a multiple of x'. Since the statement will only execute for integral configurations of the iteration variables, the unused variable x' spaces values of x apart by its multiple. 80

6.1 Three NVIDIA GPUs based upon the CUDA architecture used in our performance evaluation. Our optimisations were developed for the GeForce 8800 GTX but we evaluate them on two newer devices as well. 113

List of Figures

2.1 Two competing data storage layouts for digital images. The packed layout is better suited for display on digital hardware whilst the planar layout is more cache-efficient in chromatically oblivious image processing algorithms. (Image source: *FreeDigitalPhotos.net*) 8

2.2 Part of a Nuke compositing workflow to orchestrate a three-dimensional scene for 360° dome projection technology. A small segment of the digital composition DAG is visible in the upper-left hand corner. The lower-right corner shows a hint of the zoomed-out DAG.
(© Heribert Raab, Softmachine. Courtesy of Heribert Raab.) 9

2.3 Graphical illustration of the construction of a commercial wavelet-based photographic grain removal [SCW05] visual effect. Nodes represent visual primitives and directed edges indicate the flow of image data between them. Leaf nodes pass image data to/from the effect. 10

2.4 A polytope representation of the domains of the two statements from Listing 2.1 and a set of corresponding linear constraints on their bounds. The shaded region is common to both polyhedra and suggests potential value in a loop fusion [BGS94] schedule optimisation. 17

3.1 An alternative image indexing method with which we have begun preliminary experimentation. An 8x8 image is accessed through a 14x14 ROI defined by $(-3,-3) \rightarrow (10,10)$. The X and Y coordinates are clamped via two small lookup tables (LUTs) in place of expensive conditional branches. In practice, the Y LUT can directly supply an image row pointer. Similar formulations exist for other edge methods such as *set-to-black*, *reflect*, etc. 31

3.2 DAG constructions of two commercially developed visual effects. The visual effect DAG is one piece of metadata recorded in the frontend and used in optimisation. Each node is associated with a parameterised visual primitive. The edges between nodes indicate the flow of image data from primitive to primitive. 33

3.3 Local memory access patterns currently permitted by our memory access metadata. Each pattern is localised around a current iteration point in the image. For spatial filters, both the axis (in 1D patterns) and size of the region inside which memory access is permitted are recorded dynamically. This allows a uniform representation of horizontal/vertical spatial filters with dynamically tunable filter bounds to e.g. change a brush size. 34

3.4 A block-level overview of the domain-specific VFX single-source representation, code generation and optimisation framework. 35

3.5 Vertical moving average kernels have true dependence along column iterations but only output dependence between the end of one column and the start of the next. By replicating one state variable per column the output dependence can be eliminated, permitting row-major, cache-friendly iteration. 42

3.6 Throughput of the wavelet-based degraining and diffusion filtering visual effects in straightforward execution of the object-oriented representation discussed in Section 3.4.1 on a *single core* of each of three benchmarking platforms. Y-axis throughput measures the number of output pixels generated per second with variation of the input image size on the X-axis. 47

3.7 Throughput of the wavelet-based degraining and diffusion filtering visual effects in execution of flattened, statically-specialised generated code (from the object-oriented representation) on a *single core* of each of three benchmarking platforms. Y-axis throughput measures the number of output pixels generated per second with variation of the input image size on the X-axis. 49

3.8 Relative speed-ups (1x = no speed-up) of the wavelet-based degraining and diffusion filtering effects on a *single core* of each benchmarking platforms for a fixed input image size of 12 MPixels, following code generation and static specialisation (Section 3.6) from the object-oriented representation. Speed-ups are attributable to a combination of simplified code analysis and optimisations enabled by the presence of specialised values, such as loop unrolling. 50

3.9 Relative speed-ups (1x = no-speed-up) over single-core throughput of the wavelet-based degraining and diffusion filtering visual effects for a fixed input image size of 12 MPixels, in execution of the statically-specialised generated code as the number of active cores is scaled to the processor limit. Cache and memory resource contention leads to sublinear scalability. 51

3.10 Breakdown of the per-primitive contribution to execution time of the wavelet-based degraining and diffusion filtering visual effects on all cores of the statically-specialised generated code for a fixed input image size of 12 MPixels. These serialisations are space-optimal and derived through exhaustive search. 52

4.1 Reducing loop bound complexity by merging filter radii variables that are known to be identical. This is a critical polyhedral math optimisation in loop fusion. . . 59

4.2 Reduction in loop fragment generation following loop shifting of a superposition of spatial filter primitives. Dotted lines indicate necessary breaks in the x and y loops and the enclosed regions represent generated loop fragments. Loop shifting reduces fragmentation from 25 loops to just 10. 59

4.3 A parallelised spatial filter primitive requires partial data produced by a different core in the preceding primitive. Barrier synchronisation ensures that this data is available to each thread. In the optimised schedule it is not possible to use a barrier in this way. 63

4.4 A graphical representation of the 2D iteration spaces constituting a serialisation of the wavelet-based degraining [SCW05] visual effect. 65

4.5 Throughput of the wavelet-based degraining and diffusion filtering visual effects with maximal fusion (but no space optimisation) on all cores of each of three benchmarking platforms. Y-axis throughput measures the number of output pixels generated per second with variation of the input image size on the X-axis. 69

4.6 Throughput of the wavelet-based degraining visual effect with varying levels of schedule optimisation on all cores of each of three benchmarking platforms for a 12 MPixel image. Y-axis throughput measures the number of output pixels generated per second. Referring back to Listing 3.3, the X-axis records the number of recursive calls to the DeGrainRecursive function which space/schedule optimisation takes place across. 0 indicates no fusion. 1 indicates fusion within the function. 2 indicates two fusions across two levels of recursion. 3 indicates fusion across three levels and fusion within the last. 4 indicates maximal fusion. 70

4.7 Throughput of the wavelet-based degraining and diffusion filtering visual effects with optimal fusion and space optimisation on all cores of each of three benchmarking platforms. Y-axis throughput measures the number of output pixels generated per second with variation of the input image size on the X-axis. 71

4.8 Relative speed-ups (1x = no speed-up) of the wavelet-based degraining and diffusion filtering effects on all cores of each benchmarking platform for a fixed input image size of 12 MPixels, following schedule- and space-/schedule- optimisation. Speed-ups are attributable to reduced temporal data reuse distance and loop control overhead (Section 4.2). Slow-downs arise from increased cache pressure and memory bus traffic, due to the increased working set size, and from higher overheads in parallelisation (Section 4.2.4). 73

4.9 Relative speed-ups (1x = no-speed-up) over single-core throughput of the wavelet-based degraining and diffusion filtering visual effects for a fixed input image size of 12 MPixels, following space and schedule optimisation. Memory optimisation improves scalability over the previous experimental results in Figure 3.9. 74

5.1 Illustration of divergence in a vector-conditional branch, resolved with scalar-wise conditional predication. The boolean result vector consists of true and false elements. In predication, both branches are executed and the live-out variables of each branch are masked with this vector and merged. 84

5.2 A proposed, but as yet unimplemented, solution to the problem of vector load alignment for an indexer with horizontal (in this case 2D) spatial freedom. Aligned loads are made to stage the whole spatial filter window into registers and individual indexer loads are reconstructed by combining the staged registers. 88

5.3 Array contraction on a non-scatterable SIMD device. Some data is lost and misread off the array edge because only the first vector element is addressable. . . . 89

5.4 A scheme for correct SIMD array contraction. Stores are made to two arrays, one shifted by 180°, so data is not lost. Loads choose a safe array to read from. 90

5.5 Throughput of the wavelet-based degraining visual effect with an SSE intrinsic implementation for all constituent visual primitives, both with and without space/schedule optimisation, on all cores of each of three benchmarking platforms. Y-axis throughput measures the number of output pixels generated per second with variation of the input image size on the X-axis. 91

5.6 Throughput of the diffusion filtering visual effect with an SSE intrinsic implementation for 9 out of 12 constituent visual primitives, both with and without space/schedule optimisation, on all cores of each of three benchmarking platforms. Y-axis throughput measures the number of output pixels generated per second with variation of the input image size on the X-axis. 93

5.7 Relative speed-ups (1x = no speed-up) of the wavelet-based degraining and diffusion filtering effects on all cores of each benchmarking platform for a fixed input image size of 12 MPixels, following space/schedule optimisation and SSE vectorisation. In (a) we apply space/schedule optimisation first. In (b) we apply SSE vectorisation first. The phase ordering of the optimisation composition is fixed in both cases: space/schedule optimisation first followed by SSE vectorisation. Speed-ups from SSE are attributable to the increased computational throughput of the ISA over scalar paths and are particularly effective in rebalancing the computational/memory bound once space and schedule optimisation has been applied. . 94

5.8 Relative speed-ups (1x = no-speed-up) over single-core throughput of the wavelet-based degraining visual effect for a fixed input image size of 12 MPixels, following SSE vectorisation both with and without space/schedule optimisation. Scalability has worsened slightly over the scalar code from Figure 3.9a as SSE instructions saturate memory bandwidth more quickly. 96

5.9 Relative speed-ups (1x = no-speed-up) over single-core throughput of the diffusion filtering visual effect for a fixed input image size of 12 MPixels, following SSE vectorisation both with and without space/schedule optimisation. Scalability has worsened slightly over the scalar code from Figure 3.9b as SSE instructions saturate memory bandwidth more quickly. 97

6.1 A simulated cache for primitives with spatial filter access. Threads cooperate to stage the elements of a region of memory into local, manually managed memory, from which overlapping accesses between multiple threads can be served with lower latency and higher bandwidth. 102

6.2 SIMT threads in a moving average execution. Memory accesses made in the horizontal moving average do not satisfy coalescing rules due to discontinuity. . . . 104

6.3 A scheme for memory access transposition without explicit transpose operations. Small blocks are transposed on the fly into shared memory, where coalescing is a non-issue, and processed by a horizontal filter implementation. 105

6.4 Global memory load/store misalignment occurs when the region of interest is smaller than the input image. Coalescing can be mostly restored by redistributing the thread:data mapping so that thread 0 is aligned on a boundary. 106

6.5 Serialisation in moving average (Definition 2.11) kernels leads to underutilisation of SIMT parallel resources. Parallelism can be enhanced by splitting serialised rows/columns into parallel groups of serialised sub-rows/columns, exploiting the explicit roll-up function in the visual primitive front-end. 107

6.6 A coalescing-safe work distribution method to assign multiple work items to each thread. This is useful in simple kernels where the scheduling costs of high thread counts can dominate the computational work being done. Increased register usage to manage the mapping operation typically has no impact on parallelism due to the low register requirements of the kernel. 108

6.7 A proposed wavefront scheduling method for executing space/schedule optimised effects on a SIMT architecture with hardware support for inter-block shared memory communication. The first step of the fused kernel executes in parallel across all thread blocks. The second step does not begin in each block until its neighbouring blocks have completed the first step and synchronised to exchange computed data. 112

6.8 Cumulative execution time of the wavelet-based degraining and diffusion filtering effects on a fixed-size single-precision floating-point images in CUDA on an 8800 GTX (CC 1.0). Kernel execution times are cumulative across the DAG serialisation and optimisations are applied incrementally down the graph to improve performance. Gaps indicate transpose primitives that are not present in a particular composition of optimisations. 114

6.9 Cumulative execution time of the wavelet-based degraining and diffusion filtering effects on a fixed-size single-precision floating-point image in CUDA on a GTX 260 (CC 1.3). Kernel execution times are cumulative across the DAG serialisation and optimisations are applied incrementally down the graph to improve performance. Gaps indicate transpose primitives that are not present in a particular composition of optimisations. 116
6.10 Relative speed-ups (1x = no speed-up) of the wavelet-based degraining and diffusion filtering effects on each GPU benchmarking platform for a fixed input image size of 12 MPixels, with a fractional breakdown of the contributions of each optimisation phase outlined in Section 6.1 117
6.11 Throughput of the wavelet-based degraining and diffusion filtering visual effects with a SIMT implementation of all constituent visual primitives, with all optimisations described in Section 6.1, on each of three GPU benchmarking platforms. Y-axis throughput measures the number of output pixels generated per second with variation of the input image size on the X-axis. 118

Chapter 1

Introduction

1.1 Motivation

Visual effects (VFX) is a stage in film and television post-production in which artificial enhancements are made to pieces of captured film, most commonly in a digitised form. These can be as simple as changing the lighting or focus of a single frame or as complicated as removing film set rigging, smoothly slowing down motion or adding convincing CG characters to a sequence of frames. The scale of data processing is staggering: Industrial Light and Magic (ILM), one of the VFX companies responsible for the 2008 CG-heavy movie *"The Spiderwick Chronicles"*, maintains a renderfarm [Row08] of 3000 multicore processors, dynamically expandable to 5000 processors with the use of its artists' workstations outside of office hours, with more than 100 terabytes of storage. This method of batch processing is well-suited to producing the successive iterations of a final render of hundreds of thousands of film frames over a period of many months.

Digital artists, however, must work with far less powerful workstation hardware – such as the Apple Mac Pro favoured by ILM – and have a greater need for interactive processing to tune their choices of effects and the numerous parameters to each one. This is traditionally tackled by generating preview renders at much lower resolutions than the final render, or by working on small regions of frames at a time. In recent years many of the improvements in workstation hardware have been heavily underutilised by VFX software. VFX developers have been slow to make use of emerging computational technologies, such as Graphics Processing Units (GPUs) and high-throughput vector CPU ISAs such as Streaming SIMD Extensions (SSE). Our collaboration with a commercial VFX developer and research partner, The Foundry[1], has identified difficulties in exploiting hardware performance and the high development and maintenance costs of supporting multiple devices to be the primary barriers to uptake.

To this end, there is a need to find solutions which isolate hardware programming expertise from the VFX algorithm developers and automate device exploitation for a wide range of algorithms. This is a domain-specific problem of automating the mapping of data processing tasks to different parallel programming models. Implementation errors, which would otherwise inflate

[1] The Foundry are commercial developers of VFX software. Website: `www.thefoundry.co.uk`.

maintenance costs, could be tackled in a small piece of expert software and corrected by automatically regenerating each algorithm implementation at little to no cost. Automatic code generation for parallel computational paths is a familiar problem in software engineering with a diverse set of proposed solutions, e.g. [BBK+08, BGGT01, BHRS08, CDK+01, Col91, DRLC08, EOO+06, KPR+07, LA00, McC06, NH06, PMJ+05, Rei07]. Our industrial partners feel that existing research is too ineffective (e.g. C autovectorisation) or too disruptive to development workflow, e.g. by requiring new languages, annotations and toolchains, or by maintaining insufficient separation of expert knowledge from the algorithm representation. We see an opportunity to test and improve upon the space and schedule optimisations which our research group has previously pursued [RMKB08] to deliver high performance software. These optimisations are delivered in an active library package, via runtime code generation, which is completely transparent to the application and needs no modifications of the client code.

Working in partnership with a commercial developer of VFX software provides us with access to an industrial VFX code base on which to test our optimisation suite. Beginning from dusty-deck software plug-ins for digital compositing applications, we will demonstrate recovery of the elegant underlying algorithms from code which has become obfuscated by orthogonal issues, such as iteration and multicore parallelisation. The revised representation, a minimal expression of the algorithm and its important properties, is integrated back into plug-in software and evaluated with a suite of our metadata-supported optimisations on two commercial effects. We believe this collaboration helps to focus the study and enhances the practical value of our results.

1.2 Objectives

The objectives of this thesis are as follows:

- To devise a single-source modular representation of a visual effect which closely resembles the algorithm structure and integrates seamlessly with the development workflow.
- To develop a methodology for automatically deriving SIMD and SIMT parallel implementations from the scalar single-source representation.
- To identify performance-critical optimisations for these implementations and to devise a strategy for automatic optimisation of known and previously untested visual effects.
- To evaluate the efficacy of these ideas on industrial VFX software.

1.3 Contributions

The main contributions of this work are as follows:

- An adaptation of space and schedule optimisation to the dynamic working sets and schedules of a runtime-parameterisable visual effect. This adaptation sidesteps the costly solution

of runtime code generation by specialising static parameters in an offline process and exploiting dynamic metadata to adapt the schedule and contracted working sets at runtime to user-tunable parameters. Chapter 4 documents this work.

- A set of analysis-free transformations on a scalar data-parallel kernel to produce optimised non-scatter/gather SIMD vector-parallel implementations. Static dependence metadata is substituted for dependence analysis to identify parallelism within the kernel. Dynamic memory access metadata enables data layout transformations for aligned vector load/stores. Chapter 5 describes this process in detail.

- A set of analysis-free transformations on a scalar data-parallel kernel to produce optimised SIMT massively parallel implementations. Static dependence metadata is used to guide parallelism-enhancing transformations for tens of thousands of in-flight threads. Use of the alignment-sensitive, explicitly managed memory hierarchy is optimised by identifying inter-thread and intra-core data sharing opportunities in dynamic memory access metadata. Chapter 6 covers these innovations.

- An experimental evaluation of the first three contributions on two industrially developed VFX algorithms, demonstrating up to 8.1x speed-ups on Intel and AMD multicore CPUs and up to 6.6x on NVIDIA GPUs over our best hand-written implementations. Sections 3.7, 4.4, 5.2 and 6.4 (penultimate sections of each chapter) present a progressive evaluation of each optimisation.

1.4 Publications

In chronological order of publication, the papers which have contributed to the research in this thesis are as follows:

- IET Conference on Visual Media Production (2005). In [Cor05] we presented early work on a GPU-accelerated implementation of the wavelet-based degraining effect using the OpenGL shading language, predating the launch of compute-oriented GPU languages. It laid the groundwork for keeping the intermediate data of a chain of GPU kernels inside video memory, avoiding the slow data path to system memory and back. This paper does not directly form part of the material in this thesis.

- Workshop on Performance Optimisation for High-Level Languages and Libraries (2006). In [CBK06] we presented early work on a prototype parallelising source-to-source code generator which converted C++ loop nests into GPU kernels in the OpenGL shading language. We explored the viability of automatic parallelisation of the loop nests in VFX algorithms and identified the key challenges. Those challenges later guided our research towards assisted parallelisation with high-level metadata. This paper does not directly form part of the material in this thesis.

- Workshop on Languages and Compilers for Parallel Computing (2007). In [CKPN07] we presented a metadata-supported approach to space and schedule optimisation in DAGs of parallel VFX algorithms. We demonstrated an early version of our metadata-augmented parallel programming framework for scalar CPU and hand-vectorised SSE code generation, with a packed-component image layout which differed from our final choice of planar layout. Significant speed-ups of 3.4–5.3x were made on the wavelet-based degraining visual effect through space/schedule optimisation. The space and schedule optimisations discussed in Chapter 4 are based upon this paper.

- ACM Conference on Computing Frontiers (2009). In [CHK+09] we presented a metadata-supported approach to single-source code generation and optimisation for SIMT architectures, with an emphasis upon NVIDIA's CUDA GPU programming model. The single source representation formed the basis for the current version of our metadata-augmented parallel programming framework and is shared by scalar, SIMD and SIMT code generators. We demonstrated how metadata could support a variety of parallelism- and bandwidth-enhancing optimisations which are critical for high-performance GPU implementations of wavelet-based degraining and diffusion filtering effects. The SIMT code generation and optimisation methods discussed in Chapter 6 are based upon this paper.

1.5 Thesis Organisation

The remaining chapters in this thesis are organised as follows:

Chapter 2 introduces the domain of computational VFX and defines the fundamental concepts and terminology. A survey of the key literature in software optimisation and parallelisation for high-throughput ISAs is presented in order to establish a foundation for the research in this thesis. The chapter concludes with a summary of the research pathways that we chose to build upon, with reconciliation with the objectives outlined in Section 1.2.

Chapter 3 formalises the single-source representation of a visual effect and the expression of its associated metadata used in Chapters 4, 5 and 6. A set of constraints upon general VFX theory is defined and justified in order to focus our study on an important subset of problems. Next, we demonstrate a use of the framework with implementations of two commercial visual effects. Finally, we describe a source-to-source code generator that implements the optimisations discussed in Chapters 4, 5 and 6 and illustrate the complete toolchain from single-source representation to optimised device code for a visual effect. Simple inlining and static specialisation optimisations are automated to improve generated code performance. Significant performance gains are obtained and evaluated at this early stage.

Chapter 4 builds upon the theory of polyhedral loop representations to model and illustrate the iterative patterns and data dependence structures commonly found in VFX algorithms. We discuss the performance implications of different scheduling choices and the dependence-preserving transformations which facilitate transitions between them. Next, we demonstrate how the temporal locality of transient data can be made optimal through polyhedral manipulation, and how

dependence can be preserved without program analysis through the use of metadata. Dynamic metadata is used to adapt the schedule to runtime configurable parameters. This is followed by a discussion of an important bandwidth-enhancing optimisation to confine transient data to registers or to the lowest cache levels, which leverages memory access metadata to simplify analysis. Finally, we show how an optimised polyhedron is translated into code for a CPU and contrast the effectiveness of our approach with a vendor compiler of similar schedule optimisation capability.

Chapter 5 begins with an analysis of the parallelism requirements and programming constraints of the SIMD model in a multicore architecture. We evaluate different parallelisations of the program polyhedra in terms of their computational redundancy, generated code complexity and conformance to SIMD constraints. Next, parallelisation is broken down into a series of code transformations starting from the single-source representation, including vectorisation, divergent conditional predication and data realignment. Each transformation is analysis-free – except for a simple scope analysis in conditional predication – and uses metadata to guide its application and to ensure correctness. We then compare our results with the SSE autovectorisation capabilities of modern vendor compilers and identify the obstacles which our methods are able to overcome. The interaction with space and schedule optimisation is discussed and concluded with a novel extension of the array contraction optimisation to SIMD processors. Finally, we evaluate the speed-ups and performance characteristics of generated SSE code in two commercial visual effects and demonstrate a performance-critical link to the space optimisation described in Chapter 4.

Chapter 6 introduces the SIMT model with an analysis of its unique parallelism and bandwidth requirements in modern GPU implementations. We reconcile the first of these two needs through parallelisation of the program polyhedra and, where this alone is insufficient, implementation selection. Next, we demonstrate CUDA code generation from simple polyhedra and provide an in-depth discussion of the challenges in code generation from arbitrary polyhedra, particularly those arising from the schedule optimisations discussed in Chapter 4. We address the memory bandwidth requirements of SIMT architectures with a series of analysis-free code transformations, using metadata to identify the memory access patterns of each kernel and to ensure that data dependence is preserved. This is followed by a detailed examination of the dynamic scheduling parameters controlling the execution of a CUDA kernel. The chapter concludes with an experimental evaluation of the generated code performance with two commercial visual effects, in terms of their relative speed-up and a comparison with the SSE CPU implementation.

Chapter 7 concludes the thesis with a summary of the main results and a discussion of the application of our work in other domains and to general compiler theory. We present arguments for different levels of integration of our research into vendor compilers and propose new directions for the analysis-free code transformations prototyped in this work. The chapter concludes with a reflection upon post-research developments in the industrial implementation of our framework.

Chapter 2

Visual Effects Optimisation

2.1 Introduction

This chapter begins with an overview of the field of computational visual effects (VFX). It introduces the key concepts, defines standard terminology and illustrates a visual effect construction. Next, we review the key background literature and related work in optimising the computations constituting a visual effect. This survey covers both software optimisation techniques and systems for mapping computations onto high-throughput parallel instruction set architectures (ISAs). The chapter concludes with a study of related approaches to the problem and a summary of the research pathways we chose to build upon, with justification for each of the thesis objectives set out in Section 1.2 of the preceding chapter.

2.2 Visual Effects

Our formalisation of the VFX domain borrows heavily from the work of Brinkmann [Bri08] on digital compositing. Digital compositing is not a strict superset of VFX but the two disciplines share much in common. We begin by quoting Definition 2.1 to broadly define the domain which we are about to describe.

Definition 2.1 (Digital Compositing) *The digitally manipulated combination of at least two source images to produce an integrated result. [Bri08]*

Much of the VFX domain involves the composition of multiple images, although some effects operate upon a single image; hence VFX is not a strict subset of digital compositing. Before delving more deeply into the algorithms and structures constituting a visual effect, we take the time to define the term *image* more precisely.

2.2.1 Digital Image

The following three definitions collectively define the concept of an image in digital compositing, which we will refer to as a *digital image* in this section.

Definition 2.2 (Component) *A data element carrying information about a discrete location in a digital image, often a primary colour intensity or a mask coefficient.*

Definition 2.3 (Pixel) *A group of components constituting information associated with a discrete location in a digital image.*

Definition 2.4 (Digital Image) *An ordered collection of pixels with a spatial correspondence of information to a static two-dimensional visual depiction.*

In addition to this hierarchical structure of visual information, a fourth term is often used to refer to an orthogonal subset of the information contained within a digital image.

Definition 2.5 (Channel) *A channel is an ordered collection of components with a spatial correspondence of a single type of information to a static two-dimensional visual depiction.*

This fourth term may at first appear redundant but its necessity will become clear as we consider the physical construction of the digital image. There exist two complementary data storage layouts for digital image data: *planar layout* and *packed layout*. Both are stored as dense two-dimensional arrays with spatial correspondence to the two-dimensional visual depiction. Figure 2.1 illustrates the pixel-level storage differences. The planar layout is stored as a set of contiguous channels, while the packed layout is formed by a single contiguous interleaving of components. One tends to gravitate towards the packed layout because it feels more natural; indeed, most graphical display devices, including graphics processing units (GPUs), require this storage layout for image data to be sent to a display.

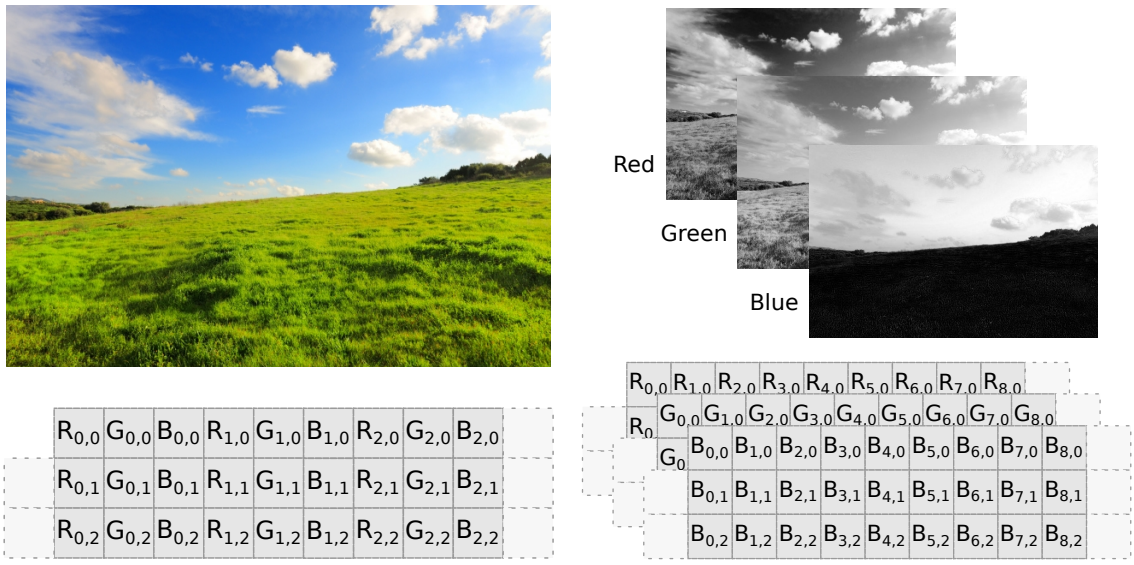
However, the packed layout has two key disadvantages:

- **Cache pollution**. When an image processing algorithm can be broken down into multiple passes, each over a single channel at a time – to reduce the working set size – the unused components of each pass persist between the useful data elements within cache lines. This inflates the effective working set size, thus reducing the useful cache space.
- **Non-uniform data processing**. Divisions of the data set into small pieces (e.g. for single-instruction, multiple-data (SIMD) parallelism) may lead to different phases of component sequences in each iteration, e.g. *RGBR GBRG BRGB* for a three-channel image. Avoiding this change in phase requires a loop unrolling [BGS94] transformation. No transformation is required in the planar case.

2.2.2 Digital Composition

Before looking at the construction of the visual effect in detail, some knowledge of the context in which they are used may be helpful. Visual effects are used within a *digital composition*.

Definition 2.6 (Digital Composition) *A directed acyclic graph (DAG) in which nodes represent visual effects and directed edges indicate the flow of image data between them. Leaf nodes are of type Source or Sink, carrying image data into and out of the composition respectively.*



(a) Packed Layout (b) Planar Layout

Figure 2.1: Two competing data storage layouts for digital images. The packed layout is better suited for display on digital hardware whilst the planar layout is more cache-efficient in chromatically oblivious image processing algorithms. (Image source: *FreeDigitalPhotos.net*)

The digital composition is a weaving of visual effects with elements of film footage. It is the link between the work of digital artists, VFX developers and the resulting feature film, television programme or advertisement. A digital composition is orchestrated within a *compositing application*, such as the contemporary Nuke compositor shown in Figure 2.2.

If we constrain our study to just two dimensions, the process of digital composition can be thought to operate within an infinite plane. Each set of image data has *bounds* enclosing a subregion of this plane. Where overlap of these bounds occurs, composition can reasonably be expected to resolve this conflict in a composited frame; perhaps by allowing parts of one image through the transparent regions of another, or by blending the two together. Each visual effect has two well-defined parameters in a digital composition which control this overlap.

Definition 2.7 (Region of Interest) *The Region of Interest (ROI) of a visual effect is a subregion of the infinite compositing plane inside which the effect consumes data from overlapping subregions of bounded input images.*

Definition 2.8 (Domain of Definition) *The Domain of Definition (DOD) of a visual effect is a subregion of the infinite compositing plane inside which the effect produces data into overlapping subregions of bounded output images.*

We will now define the visual effect. This construct is the focus of optimisation in this thesis.

Definition 2.9 (Visual Effect) *A connected DAG (e.g. Figure 2.3) in which nodes represent visual primitives and directed edges indicate the flow of image data between them. Leaf nodes are of type*

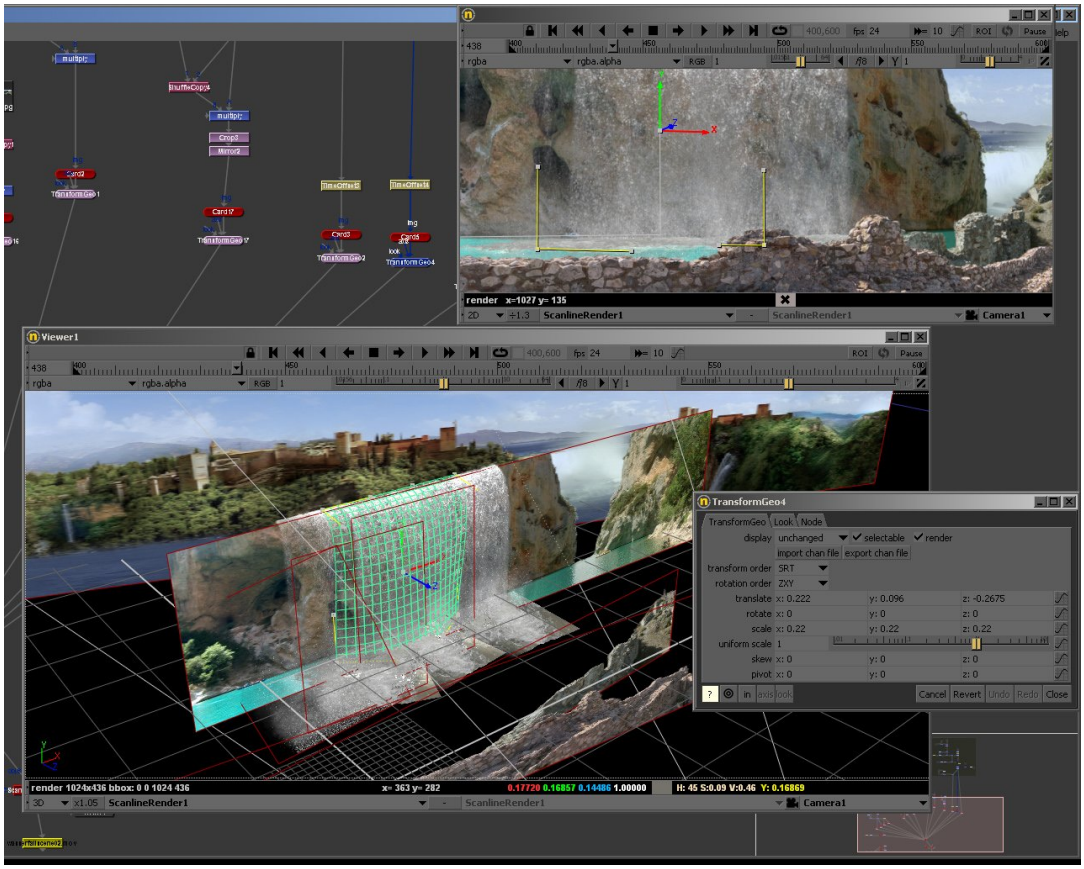


Figure 2.2: Part of a Nuke compositing workflow to orchestrate a three-dimensional scene for 360° dome projection technology. A small segment of the digital composition DAG is visible in the upper-left hand corner. The lower-right corner shows a hint of the zoomed-out DAG.
(© Heribert Raab, Softmachine. Courtesy of Heribert Raab.)

Source or Sink, carrying image data into and out of the effect respectively. An effect can define an arbitrary number of parameters to tune its behaviour in a given composition.

Definition 2.10 (Visual Primitive) *An algorithm which consumes data from one set of images and produces data into another set. A primitive can define an arbitrary number of parameters to tune its behaviour in a given visual effect.*

The reader may notice the similarity of Definition 2.9 to that of a digital composition. Indeed, the difference is primarily one of granularity. A visual effect is a self-supporting composite algorithm with applications in many different digital compositions. It presents fine-grained tunable parameters to the digital artist without overwhelming them with its construction from smaller visual primitives. The primitives themselves are often reusable inside different visual effects.

Figure 2.3 illustrates a commercial visual effect for the automatic removal of photographic grain, based upon a wavelet decomposition algorithm [SCW05]. Each node in this graph, excluding the Source and Sink nodes, corresponds to a reusable visual primitive, There are just four unique primitives in the whole graph. Effects such as this one correspond to a single node – or to multiple nodes if used more than once – in the digital composition graph shown in Figure 2.2.

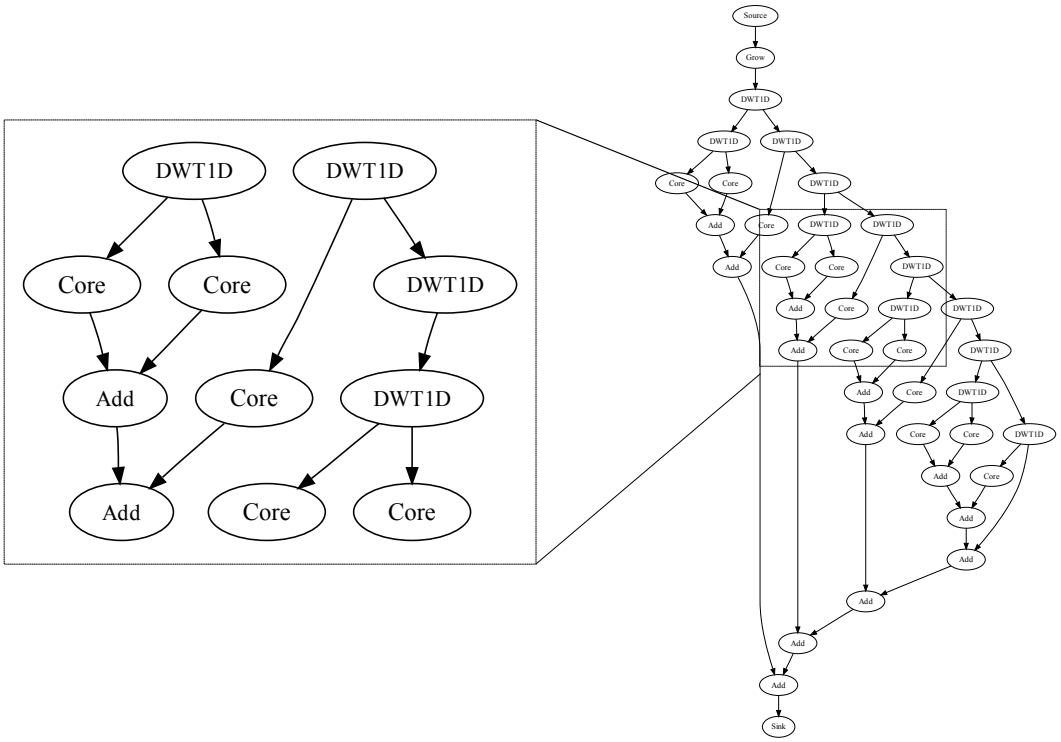


Figure 2.3: Graphical illustration of the construction of a commercial wavelet-based photographic grain removal [SCW05] visual effect. Nodes represent visual primitives and directed edges indicate the flow of image data between them. Leaf nodes pass image data to/from the effect.

Many visual primitives share common features of data dependence and patterns of access to image data. We make use of two such classifications throughout this thesis and these are outlined below:

- **Data Dependence**. The most basic algorithms contain no loop-carried dependencies and are, thus, fully parallel. An interesting class of algorithms, called *moving averages*, have a loop-carried dependence in a single level of their loop nests. Many more complex data dependence structures can be found in primitive algorithms.

- **Patterns of Memory Access**. In many algorithms there is a spatial correspondence between the pixels of output images and the pixels of the input images used to compute them. Basic algorithms may take a single correspondingly located pixel from an input image and use it to produce an output pixel. More sophisticated algorithms, called *spatial filters*, use a correspondingly centred 1D or 2D region of pixels to compute a single output pixel. More complex patterns can be identified before descending into true random access.

Definition 2.11 (Moving Average) *A moving average is a visual primitive in which each output pixel is dependent upon state initialised at the start of, and updated along, the horizontal or vertical axis. This creates a loop-carried dependence in one level of the algorithm's loop nest.*

Definition 2.12 (Spatial Filter) *A spatial filter is a visual primitive in which each output pixel is dependent upon a correspondingly centred $M \times N$ region of pixels in one or more of the input*

images. This region may be one-dimensional, in a horizontal or vertical configuration, or two-dimensional.

This brief definition of the VFX domain is sufficient to prepare the reader for the discussion of our computational VFX framework described in Chapter 3. Further examples and illustrations of VFX constructs may be found in that chapter.

2.3 Software Optimisation

In this section we survey the key literature in software optimisation techniques which are of relevance to the research in this thesis. Notably absent is a discussion of the low-level optimisations employed by established compiler technology and we refer the reader instead to [BGS94] for a survey of these techniques. A discussion of parallelisation, an important software optimisation technique for modern multicore processors, and its interaction with other optimisations is deferred until Section 2.4 where the key advances in research are described in detail. The subsections which follow review the optimisations that are of most significance to our contributions to the field of software optimisation.

2.3.1 Memory Hierarchy Management

The divide between computational and memory performance has grown large [WM95], and continues to widen, as developments in memory technology have failed to keep up with increases in computational throughput. The extrema of this phenomenon is known as the "memory wall". Worse still, the increases in memory bandwidth which have been delivered come at the expense of significant rises in latency [SPN96]. Multicore hardware in particular has driven unsustainable increases in memory throughput requirements [Moo08]. The consequence of this technological performance chasm is that algorithms with poor computation:memory access ratios are frequently bounded by memory performance on modern compute architectures. Tackling the inadequacies of the memory system is a key focus of many of the optimisations discussed in this thesis.

Contemporary memory systems are divided into multiple levels of disks, caches and registers. There is a trend of increasing capacity and latency and decreasing bandwidth as each level of the memory hierarchy is placed further away from the computational resources [Prz90]. An important consequence of this design is that an algorithm with a small working set may take advantage of cache levels with high bandwidth and low latency, whilst minimising spills to larger, slower levels of the memory hierarchy. Memory reduction [SXWL01] is a compiler-based technique that exploits spatial locality and a fixed temporal data reuse distance to reduce the working set size so that it fits into a lower cache level. This optimisation relies on a combination of loop shifting, loop fusion and array contraction [BGS94] transformations. The necessity for memory reduction arises from the natural structuring of composite array-based computations as series of independent loop nests with primitive kernel bodies. This is an elegant expression in imperative languages but leads to large reuse distances between the elements of arrays processed by successive kernels.

Data storage layout transformations may enable more efficient use of caches. Recursive, blocked storage layouts, such as the Z-Morton [Sag94], mitigate the performance impact of column-major array accesses experienced in a row-major storage layout. Cache misses and unuseful prefetches are reduced for algorithms which exhibit multidimensional spatial locality by using a logical-to-physical address mapping which unidimensionally localises elements in multidimensional localities. Recursive layouts are particularly useful because they can embed multiple localities of different sizes to fit into different levels of the memory hierarchy [PHP03]. This comes at a small cost for algorithms with only unidimensional spatial locality, which may experience better cache utilisation with a row- or column-major storage layout. A consequence of non-row-/column-major layout is that address computation becomes more expensive: particularly for non-power-of-two sized arrays, which may need padding to power-of-two sizes. These additional costs to performance may only be offset by improved cache utilisation for large arrays [BJK03].

A novel approach to memory hierarchy management is leveraged by Sequoia [FHK+06], a C++-based parallel programming language and associated compiler. Sequoia exploits an analytical model of the memory hierarchy to generate specialised code with localised computation within each level of the memory hierarchy, with explicit data communication constructs between levels. The algorithm is expressed as a hierarchical set of divide-and-conquer tasks, with parallel language augmentations to mark pieces of kernels which may be mapped onto parallel processors. For this reason, the memory hierarchy models may extend beyond main memory to cluster-level communication. This approach to memory hierarchy optimisation is a hybrid of cache-aware and cache-oblivious [FLPR99] algorithms: the algorithm expression is cache-oblivious divide-and-conquer but the memory model specifications are cache-aware. The programmer must supply both to orchestrate a program implementation for each device.

A brute force approach to memory hierarchy management is employed by self-tuning software such as the ATLAS [WD98] linear algebra library and the FFTW [FJ98] Fast Fourier Transform (FFT) library. Parameters to various compile-time code transformations, including loop tiling for the memory hierarchy, are derived through a heuristic-guided search process with dynamic feedback of performance characteristics on a per-system basis. Such libraries typically require a long pre-execution process to perform a sufficiently large search of the optimisation space before obtaining suitable parameters for normal execution. This approach works well for libraries of widely used algorithms, such as BLAS [LHKK79] routines and FFTs in different point configurations, which can be tuned once per system and then shared by many applications.

Architecture-cognizant algorithms [GC99] hybridise the divide-and-conquer approach with runtime tuning to achieve efficient use of the memory hierarchy. The algorithm implementation provides variants of the divide and combine components, to move data down and up the hierarchy respectively. The authors suggest two examples of variants: to copy the divided problem during division or not, trading the expense of a copy for better cache associativity in the subproblem, and to combine or postpone recombination of subproblems to alleviate the overhead of small recombinations. An exhaustive search for the optimum selection of variants grows exponentially with the number of levels of problem division. A dynamic programming solution is proposed to

find a good variant selection in lower complexity time.

An important improvement to the memory hierarchy model [GG02] was suggested to account for the indirect, but performance critical effects of the Translation Lookaside Buffer (TLB). In matrix multiplication, the authors observe that one operand is normally kept in L1 while the others are streamed through L2. Hence filling the L2 cache may incur TLB misses – and likely will, since one operand (accessed row-major in column-major layout, or vice versa) may span many pages – which stalls the computation's overall progress. Hence, the memory hierarchy model is augmented with an additional level between the L1 and L2 cache to encourage tiling for the TLB. The method of tiling, as for other levels of the hierarchy, can involve copying the tile or simply visiting the tile's elements in order. This decision is dependent upon the matrix size in the ATLAS library. Only once the matrix grows large enough for TLB misses inside the tile to have a significant impact does the library introduce a copy of that tile into a contiguous memory region.

The use of analytical memory models and empirical search for tuning factors is a topic of recent debate. Conventional wisdom suggests that empirical search is more effective than model-driven optimisation because accurate models are very expense to construct and use. However, in the case of BLAS one team presents evidence [YLR+05] to the contrary. The global optimisation engine in ATLAS is replaced with a model that is sensitive to the computational and memory hierarchy requirements of BLAS. Their results demonstrate an order of magnitude reduction in configuration time across many architectures and, with a few exceptions discussed in detail, closely matches the performance of global search-tuned parameters. A compromise of the two approaches is explored through a hybrid of model and search in [EGD+05]. This approach leverages a high degree of confidence in some parts of the model and augments the remainder with empirical search. An evaluation of this idea on the aforementioned model-driven ATLAS implementation documents an increase in configuration time of 3–4x but with configured performance that is comparable to, or better than, both model-driven and empirical search schemes.

2.3.2 Component-Based Programming

A software component is a "binary unit of independent production, acquisition and deployment which interacts with other components to form a functioning system" [Szy02]. It is an advancement of the object concept from object-oriented programming in that it has no externally observable state and conforms to a standard interface (within a class of components) for interoperating with other components. The benefits of a reusable component design are self-evident and contribute to our focus on this paradigm. It is, in fact, a natural structuring of many visual effects algorithms as discussed in Section 2.2.2. Our interest mainly concerns the role of this abstraction in assisting in the optimisation of each component, its communication with other components and the composition of multiple components into a single entity.

Algorithmic skeletons [Col91] form a programming methodology for imperative languages which borrows from the higher order function concept of functional languages to implement software components. Each skeleton defines a template for a pattern of computation and/or communication which can be specialised with details to implement an algorithm which exhibits the

corresponding pattern. A template may typically separate iteration control from the body of an iterated kernel, for example. The specialisation may leverage all of the features of the imperative language to define the kernel and is thus limited in the least restrictive way. The author discusses one possible goal of leveraging an algorithmic skeleton design: to automate the parallel decomposition of a computation for a multiprocessor system. By manipulating the iteration space at a high level of abstraction, parallelisation becomes a simple and well-defined problem for each skeleton. It should be emphasised that additional parallelism may exist within the kernel itself: however, exploiting this parallelism is not a goal of the algorithmic skeleton design. To do so, with no clear methodology for describing or manipulating parallelism within the kernel, is to build a universal parallelisation tool for the language which is inherently difficult for all but the simplest languages.

Poor data locality is a common problem in component-based designs [AKO04]. Strong encapsulation of components isolates successive accesses to the same data in different components, leading to large data reuse distances and, as a result, poor temporal locality. Encapsulation must be broken following component composition in order to optimise compositions for data locality as a whole. The authors discuss a language called Aldor that is well-suited to component expressions. Its compiler generates an intermediate representation which is subsequently optimised and translated to C for lower-level optimisation. Domain-specific improvements to the compiler for a BLAS library enable fusion of vector:vector, stencil:vector and stencil:stencil operations in the decomposed components within the intermediate representation (IR). By delaying these intrusive cross-component optimisations to a compilation phase, the modularity of the original program, and the benefits that brings, is maintained. These optimisations allow the performance of a modular client application to surpass highly tuned ATLAS performance by up to 1.4x.

Another opportunity a component-based design presents lies in optimising the placement of data sets in a distributed memory architecture [BJK02]. The authors describe a domain of data-parallel components which are composed into program graphs. The data placement problem is to distribute computed subarrays (directly or through redistribution between components) in a way which minimises the overall execution time of the graph. Components have well-defined inputs, outputs and computational boundaries and this lends itself well to a formal definition of the problem. The result of most interest to our domain is that optimal solutions to this problem for DAGs are NP-complete in the general case and only heuristic solutions can attain lower complexity.

A domain-specific instance of cross-component optimisation has been studied in the MayaVi visualisation tool [BFG+04]. The MayaVi software is constructed from visualisation and data filter components. There is an opportunity to reduce rendering latency by partitioning large data sets and visualising them per-partition but this is incompatible with the implementation of many of the software's components. The authors argue that rewriting the software to support the partitioning optimisation would be a substantial amount of work and would interfere with the modular design. Instead, calls to each component are redirected through a new Python interface and delayed to construct an execution plan. When a computation is forced (by a component being interrogated for rendered results) the partitioning optimisation is applied to the execution plan and the revised, partitioned plan is run instead. The results are functionally equivalent to the original plan and

deliver substantial speed-ups for small volumes of interest. In larger volumes, where the partitioning optimisation is less useful, overheads from node duplication at partition boundaries dominate and reduce the overall throughput. This method required no modification to the original MayaVi components and resembles, in many ways, an aspect-oriented programming [KLM+98] approach without the need for language or compiler support.

2.3.3 Generative Programming

Generative programming is a metaprogramming technique [CDG+06] with considerable value in domain-specific optimisation. One option is to implement algorithms in a metaprogramming language with templated parameters specifying factors to be optimised, such as loop unrolling amounts. A compiler is then free to instantiate these parameters with different values, found e.g. through empirical search during iterative compilation [KKO02], to produce a fully specialised code implementation. The choices of algorithm parameterisation form a domain-specific decision. A benefit of this approach over string-based code manipulation is that parameters of arbitrary complexity may be substituted with the type-safe guarantees that the meta-language provides. The authors present a case for a pathway from this meta-language to a more pragmatic, performance-oriented language such as C.

The active library paradigm [CEG+00] is a classification of software libraries which play a metaprogramming role in the client application, most commonly in optimisation. An active library typically remains transparent to the client application by exposing a well-defined, often domain-specific, interface behind which complex optimisations can be orchestrated while fulfilling the interface requirements. Runtime code generation [VG98] is a key enabling technique in active libraries which leverages the availability of runtime context, captured through the client / application interface, to generate and compile specialised code fragments for specific tasks on-demand. Delayed evaluation is a common technique used to increase the availability of runtime context in an active library. By postponing evaluation of a progressively constructed computation until the client application requests the computed data, an internal view of the computation up to that point enables optimisations such as data placement/layout manipulation and cross-component fusion.

The TaskGraph library [BHKM03] is an example of an active library augmented with runtime code generation for domain-specific optimisation in a component-based application. Algorithms are written in a C++ metaprogramming language which, when executed, constructs an abstract syntax tree (AST) representation of the algorithm via the SUIF [HAA+96] IR. The authors present two optimisations for an image filtering operation in this framework: runtime specialisation of a loop's trip count to a dynamic parameter and loop unrolling for the architecture on which the application is run. Performance is poorer or no better than an unoptimised implementation of convolution for data sets smaller than 1024x1024, due to the significant cost of compilation. However, for larger data sets the authors observe a speed-up which tends towards 3-4x as the cost of compilation becomes insignificant in proportion to the execution time. Related approaches employ a code cache to store optimised fragments but the authors opted instead for the flexibility offered by full specialisation and arbitrary optimisation with regard to dynamic parameters.

2.3.4 Metadata-Guided Optimisation

Program metadata is a means of communicating information about a program by encoding and embedding the information in the program implementation. Our focus is on metadata which communicates information about algorithm structure alongside its code implementation, as a means of enhancing optimisation of the program. For example, metadata has been used to encode data dependence information in a component-based programming infrastructure [KBFB01] to support analysis-free optimisations, such as loop fusion, loop tiling and data placement. This approach is domain-agnostic and supports a large class of applications by encapsulating low-level dependence constructs, such as iteration spaces, domains, and uses/definition of data collections. The strength of this approach lies in its generality, but this is also its weakness: the expression of metadata for a simple 2D Jacobi operation is quite verbose. A code sketch for the metadata suggests several ways to make a more concise expression: by condensing entire arrays of uniform data dependencies as shapes and by using an overloaded C++ class to capture data dependencies between uses and definitions.

Object-oriented abstractions may incur performance penalties if they obscure the underlying program constructs, such as arrays, during compiler optimisation. One means of resolving this is to introduce explicit programmer/compiler metadata communication constructs [YQ04], which describe the semantics of high-level collection classes to the compiler so that they can be treated as, e.g. arrays for low-level optimisation. Key semantics, such as array indexing and member aliasing, are encoded and embedded in metadata attached to the abstraction interface. Vendor compilers do not provide a path for retrieval and use of this information, so the authors leveraged the ROSE [SQ03] source-to-source compiler infrastructure to collect and use the metadata in a custom optimisation phase.

A different use of metadata can be found in the ICENI grid architecture [FMM+02], which leverages the component-based programming design to express graphs of pluggable application components which can be mapped to grid compute resources. A component may consist of multiple implementations from which one must be selected for execution on a particular resource. Metadata is attached to each implementation to describe its performance properties and affinities for different compute resources. In this role, metadata is used to optimise grid application execution by guiding component implementation selection for mapped compute resources. The authors go further in suggesting that this metadata may even be derived empirically through black box performance analysis.

2.3.5 Polyhedral Schedule Transformation

The polyhedral model is a formalism for reasoning about parallel computations, using systems of affine recurrence equations defined over polyhedral shaped domains [QRW00]. A more concrete definition of the polyhedral model is a matrix representation of the n-dimensional iteration spaces of loop nests in a program and the data dependencies between the statements contained within. Its primary role in software optimisation lies in the systematic parallelisation of com-

```
for(y = 0; y <= 4; ++ y)
    for(x = y; x <= (y + 4); ++ x)
        S1;

for(y = 2; y <= 8; ++ y)
    for(x = 1; x <= 8; ++ x)
        S2;
```

Listing 2.1: Two example loop nests with corresponding polyhedral domains shown in Figure 2.4.

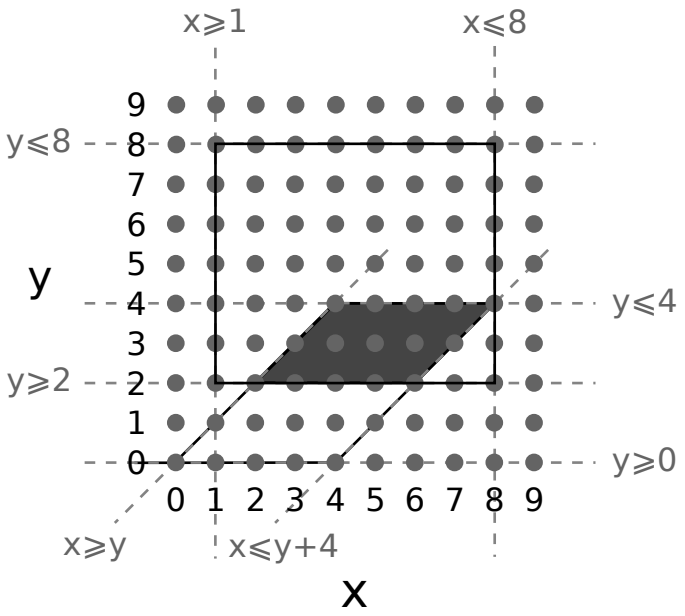


Figure 2.4: A polytope representation of the domains of the two statements from Listing 2.1 and a set of corresponding linear constraints on their bounds. The shaded region is common to both polyhedra and suggests potential value in a loop fusion [BGS94] schedule optimisation.

plex codes [Len93] but has useful application in other loop transformations [BCG+03] as well. Improvements to the original polyhedral model have been leveraged to assist in the composition of loop transformations [CGT04] and the search for appropriate transformation compositions [CSG+05] in iterative compilation.

Consider the two loop nests shown in Listing 2.1. The iteration domains of statements S1 and S2 can be expressed as a system of linear equations defining two (overlapping) convex hulls in a two-dimensional space. The corresponding polytopes encompassing these two iteration domains are illustrated in Figure 2.4. Statement S1 executes at each of the integer points contained within the lower parallelogram, which corresponds to a skewed loop. S2 executes at the integer points within the upper rectangular polytope. There is some overlap between the domains of both statements and they could, if dependence would not be violated, execute together in a single loop fragment at these points. This would correspond to a loop fusion [BGS94] schedule optimisation.

A more compact and computable representation stores the systems of linear equations as matrices. Such a representation for the two polyhedra from Figure 2.4 is given in Table 2.1. There is a straightforward translation of these matrices into loop bounds to reconstruct the domains of

$=/\geq$	y	x	1	constraint
1	1	0	0	$y \geq 0$
1	-1	0	4	$y \leq 4$
1	-1	1	0	$x \geq y$
1	1	-1	4	$x \leq y+4$

$=/\geq$	y	x	1	constraint
1	1	0	-2	$y \geq 2$
1	-1	0	8	$y \leq 8$
1	0	1	-1	$x \geq 1$
1	0	-1	8	$x \leq 8$

Table 2.1: A condensed matrix representation of the systems of linear equations defining two convex hulls in a two-dimensional iteration space, as shown in Figure 2.4.

```
for(y = 0; y <= 1; y++)
  for(x = y; x <= y+4; x++)
    S1;
for(y = 2; y <= 3; y++) {
  for(x = 1; x <= y-1; x++)
    S2;
  for(x = y; x <= y+4; x++) {
    S1;
    S2;
  }
  for(x = y+5; x <= 8; x++)
    S2;
}
y = 4;
for(x = 1; x <= 3; x++)
  S2;
for(x = 4; x <= 8; x++)
  S1;
  S2;
for(y = 5; y <= 8; y++)
  for(x = 1; x <= 8; x++)
    S2;
```

Listing 2.2: Code generated by the CLooG [Bas04] polyhedral library from the polyhedra shown in Figure 2.4. Loop fusion leads to code inflation of the original source from Listing 2.1.

each statement, illustrated by the constraint columns. However, consider the two matrices in aggregate. It is possible to construct two independent loop nests from the matrix set or, dependence permitting, a single loop nest with fragmented loop bodies shared by statements S1 and S2. The latter may improve temporal locality if statement S1 produces output for statement S2 to consume. Constructing this aggregated, fused loop nest is a much harder problem. In general, the process requires a scanning method to identify the integral points that fall within the polyhedra and to construct a corresponding set of loop fragments which visits them.

CLooG [Bas04] is one such polyhedral scanning tool. It accepts the dense matrix format shown in Table 2.1 as input and, given assurances that no dependence exists between statements S1 and S2, generates the fused code shown in Listing 2.2. The inflation in code size evident in this output is a minor example of a serious problem in the generation of optimised loop schedules. In practice, symmetry and other properties of everyday loop structures can be exploited to avoid code explosion [VBC06]. For example, the current release version of the GNU Compiler Collection exploits the polyhedral model for loop nest optimisation [PCB+06].

Fusing loops that contain independent statements is easy but of limited practical application. Loop control overhead may be reduced but there will be no improvement in temporal locality. It is far more interesting to fuse loops containing dependent statements. To achieve this, the polyhedral model allows for expression of data dependence, which will be respected during code generation. CLooG uses a manually-programmed *scheduling matrix* for this purpose, which is a condensed representation of data dependence and loop transformation as scheduling constraints. Another polyhedral code generation framework, PLUTO [BHRS08], expresses data dependence as an explicit set of polyhedra derived from dataflow analysis.

Compositions of traditional loop optimisations may be more easily managed [GVB+06] in the polyhedral loop domain. The authors argue for separation of traditionally related program transformations: those affecting the iteration space, the statement schedule, array accesses and the data layout. Each set of transformations is represented as a composition of operations on a matrix. Code is only generated once a full composition has been selected; this avoids the increase in code complexity between traditional transformation phases. The phase ordering problem is made simpler by improvements in the commutativity of transformations in the isolated matrix form. It is also easy to identify compositions of different transformations which give the same result, by comparing the composed matrices, to reduce unnecessary retesting in iterative compilation. Finally, the matrix form can be used directly in empirical search to locate good compositions of transformations. Legal transformation sequences can be identified in the matrix form and the search process may even be augmented to optimise for legitimate matrix configurations directly.

2.4 Software Parallelisation

In the preceding section we focused on optimisations which improve the performance of serial code. Having established a foundation for generating efficient code, we now explore the problem of exploiting task and data parallelism to leverage multiprocessor, multicore and vector compute architectures. We focus on three technologies available to us in workstation hardware: symmetric multiprocessing (across and within multicore CPUs), SIMD (used by Streaming SIMD Extensions (SSE) and other vector ISAs) and SIMT (an emerging latency-tolerant design in GPUs). A discussion of cluster-level parallelism is beyond the scope of this work.

2.4.1 Symmetric Multiprocessing (SMP)

Multiprocessor and multicore devices are programmed via software threads. Most CPUs execute a single hardware thread (or two in hyperthreaded designs) per core and rely on a software scheduler to timeslice software threads to each hardware thread. Likewise, systems with multiple CPUs require a scheduler to balance threads across the whole system. Some esoteric designs (e.g. Sun's "Niagara" UltraSPARC T1) exploit a simultaneous multithreading (SMT) [TEL98] architecture to provide multiple hardware threads per core to the OS. In all cases, the OS scheduler manages these compute resources and the programmer need only supply enough software threads to keep the hardware threads occupied. Additional complications arise in SMP and multicore systems,

where some resources – such as I/O buses and cache – may be shared between multiple hardware threads. The programmer must consider cache partitioning [CRM07] and thread affinity [KFA08, TaMS+08] in these cases in order to use the hardware efficiently.

OpenMP [CDK+01] is an industry standard application programming interface (API) for exploiting SMP and cluster-level parallelism. Rather than managing threads directly, the application programmer annotates their C program with *pragma* constructs to indicate loops which should be compiler-parallelised for SMP. This sidesteps the autoparallelisation problem by placing the burden of loop selection and data parallelism guarantees upon the programmer. In exchange, OpenMP provides a low-maintenance, scalable path to parallelising an existing application for a wide range of parallel compute devices supported by the OpenMP runtime library. Additional annotation constructs enable implicit management of message passing on parallel clusters.

STAPL [AJR+03] is a library of Standard Template Library (STL)-like templated containers and algorithms which support automatic parallelisation of regular vector-based computations and irregular computations on more complex data structures. The parallelisation process is automated and hidden from the user through a sequential, iterator-based programming model. Each algorithm implementation comprises a set of specialisations for architectural (e.g. memory hierarchy) and algorithmic optimisations, giving rise to dynamically adaptive performance to hardware and problem through model-driven selection. Empirical performance analysis takes place after installation to construct adaptation models for each algorithm. A number of scheduling policies are provided, including fixed-range static parallelisation and dynamic-range work queue parallelisation.

The STAPL library influenced the design of an Intel project, called Threading Building Blocks (TBB) [Rei07], for mapping common data-parallel patterns [HS86], such as *parallel for*, *parallel reduce* and *parallel scan*, onto SMP architectures. The client application selects a pattern from a set of C++ template classes and implements a kernel function (or set of functions) to define the parallel operation. In this manner, TBB exhibits elements of a skeleton [Col91] design. The library maintains a thread pool to avoid the costs of creating and destroying threads in finer-grained computations. Threads are woken when a parallel computation is available and placed into a sleeping state for the remainder of the execution. In addition to data parallel constructs, TBB provides a mechanism for building task graphs to exploit task-level parallelism. SMP architectures are well-suited to task parallelism; SIMD and SIMT designs may only exploit data parallelism.

The ideal path to programming SMP architectures is parallelisation of the program at the compiler-level, with no programmer assistance. This remains a hot research topic with inherent difficulties. In the SUIF research compiler [HAA+96], autoparallelisation relied on a set of classic compiler transformations and analyses. This process is closely tied with data locality optimisation because different valid parallelisations may lead to schedules with varying degrees of data locality. Contemporary research suggests that existing static analysis techniques are unable to detect parallelism in important cases [TWFO09] and propose a profile-driven solution to detect the absence of data dependence at runtime. In the polyhedral model, the parallelisation problem becomes a scheduling choice [BHRS08] and a viable parallelising prototype compiler exploiting this property has recently been constructed.

2.4.2 Single Instruction, Multiple Data (SIMD)

The SIMD data-parallel model is popular among vector ISAs, such as SSE [TH99] and VMX [IBM05]. A defining characteristic of these ISAs is that vector loads/stores can only be addressed by the first scalar; thus scatter/gather operations are not supported and parallelisation techniques must consider the contiguity of data being processed in a vector-parallel manner. Another feature of these ISAs is that they typically exhibit higher memory access performance when vectors are loaded or stored on aligned address boundaries [SJV06]. In many computations, including those considered within this thesis, this is a non-trivial property to attain. The authors discuss several techniques to avoid misaligned data access.

Programming vector units is often a case of using ISA-level compiler intrinsics to express parallel data operations by hand. Hand-vectorised code is prevalent in a number of key scientific and multimedia algorithms [HOM08]. CPU vendor-optimised algorithm libraries are another path to leveraging SIMD ISAs, e.g. Intel's IPP [Ste04] and MKL and AMD's complementary Framewave and ACML libraries. Algorithms may alternatively be expressed in a parallel-aware language to receive compiler assistance in the vectorisation process. Intel's Ct metaprogramming language [GSW+07] captures parallelism through C++ collection classes with overloaded operators and exploits SIMD instructions through a runtime delayed evaluation, code generation library. The Open Compute Language (OpenCL) [Mun09] expresses data parallelism through fully parallel kernels and fixed-vector constructs and operators within. The former is sufficient to derive SIMD implementations but, at the time of writing, only explicit vector-parallel constructs are vectorised by contemporary CPU backends.

Autovectorisation of serial code is a better developed compiler technique than autoparallelisation, because the data parallelism required may be much finer grained without incurring thread management overheads. GCC exploits the static single assignment (SSA) form [Nai04] to build a data dependence graph from which vector parallelism can be identified. The Intel compiler [BGGT02] leverages a set of progressively more complex data dependence analyses to identify parallelism and uses classic transformations to expose vector-level parallelism. Notably, the compiler attempts to realign data with unknown static alignment through dynamic loop peeling. IBM's Cell compiler [EOO+06] uses a mix of unroll-and-jam [BGS94] and loop transformation methods to exploit SIMD parallelism. A technique called superword level parallelism [LA00] is a promising generalisation of classic autovectorisation for short-vector architectures, with improvements in parallelism identification on the SPEC benchmark suite. The deep jam transformation [CCJ05] can be used to expose vector parallelism in coarse-grain parallel code where irregular loop structures block classic vectorising transformations.

2.4.3 Single Instruction, Multiple Threads (SIMT)

The SIMT architecture [NBGS08] was borne out of a need for a parallel processing design which could scale trivially from a few cores to an arbitrarily large number. Several grains of parallelism are defined to accommodate different hardware structures: small numbers of threads execute in

lockstep on the scalar cores of a multiprocessor; warps are grouped into blocks according to their ability to share information and communicate efficiently; blocks are grouped into a grid which defines the domain of a computation. This highly scalable design was needed to form an effective solution to the increasingly high latencies of modern memory systems, which run into several hundreds of cycles. By maintaining a large set of concurrently executing threads with a much larger backlog of "stalled" threads, which can be switched in and out of the multiprocessors efficiently, the design can hide memory access latency by swapping in threads whose requested data has arrived. This design can even be cache free, although current implementations define a small, manually managed cache per multiprocessor to facilitate intra-block communication and data sharing.

Until recently, NVIDIA's Compute Unified Device Architecture (CUDA) [NVI08] language and compiler formed the dominant path for building SIMT applications. Algorithms are written as data-parallel kernels with limited synchronisation constructs. Data placement is managed by the application programmer through an explicit communication API. The OpenCL language [Mun09], a cross-vendor solution to SIMT programming, resembles CUDA in many aspects; its design was influenced by NVIDIA and other hardware vendors. A key advance is the integration of a compiler into the runtime OpenCL library, enabling runtime code generation and the benefits that may bring. The availability of CPU backends for OpenCL presents an opportunity for uniform cross-device parallel algorithm development. Preoptimised vendor SIMT libraries are in an early stage of development. Notable examples include the CUBLAS [BCI+08] linear algebra library and the CUFFT Fourier transform library. The CUDPP library [HOS+07] simplifies the construction of kernels with common data-parallel patterns by providing templates and operators to synthesise optimised kernels through library calls.

To simplify data placement and the implementation of common parallel patterns, such as reduction, higher level languages which compile to CUDA (or a lower-level assembly language) have been developed. The RapidMind platform [McC06] extends the C++ language to include data-parallel containers and operators. A metaprogramming, delayed evaluation, code generation approach intercepts operations on these containers and generates appropriate CUDA code to implement the computation. Data marshalling from/to host memory is handled transparently. CUDA-lite [ULBH08] is a CUDA to CUDA compiler which automates SIMT memory optimisations, such as coalescing and shared memory staging. The programmer retains the flexibility of the CUDA language but only uses naive global memory access in their algorithm implementation: the CUDA-lite suite optimises this form.

There is little established work in autoparallelisation for SIMT architectures. In theory, any existing parallelisation approach which exposes sufficient parallelism could be used to generate code for a SIMT device. However, the methodology for generating memory-optimal kernels is more akin to long-vector SIMD code generation. The contemporary PLuTo polyhedral compiler has been adapted [BBK+08] to generate CUDA code with the performance characteristics of a GPU in mind; this is discussed in more detail in Section 2.5. PGI has recently developed a parallelising Fortran to CUDA compiler [PGI09] based upon their existing parallelising compilers.

2.5 Related Work

Russell et al. [RMKB08] describe a scheme called DESOLA for transparent cross-component optimisation of client applications of a linear algebra library, through loop fusion [BGS94], array contraction [SXWL01] and parameter specialisation [CN96] optimisations. The authors implement a runtime code generation and caching method based on the active library [CEG+00, VG98] paradigm, in which the performance of a client application is improved through runtime context-sensitive optimisations within the library. A key innovation here, and one which inspired our work, is the use of delayed evaluation to construct a DAG of primitive operations at runtime from which cross-component optimisations can be planned. By intercepting library calls which return concrete results from a computation – forced evaluation points – optimised code can be executed instead of the delayed chain of primitives. Our visual effects library methodology in Chapter 3 and the space/schedule optimisation work in Chapter 4 shares much in common with this approach but differs in two regards. Firstly, we move the code generation stage to a pre-deployment step where the costly process of generating and compiling optimised code can be done offline. The implication of this is that we are unable to specialise parameters that may change post-deployment, while DESOLA is able to do this. However, since such specialisation would involve code generation and compilation the costs could not be amortised by caching unless they changed infrequently. We do specialise parameters that are identifiably static as discussed in Section 3.6.1. Secondly, our schedule optimisations are more advanced in using loop shifting [BGS94] to mutate dependence structures in a way which allows majority loop fusion with few unfusible fragments. Similarly, we contract transient arrays to smaller arrays – rather than simply to scalars – when that is the optimum achievable contraction. The method of optimisation also differs in an interesting way; DESOLA uses the TaskGraph [BHKM03] library which incorporates code transformation features from the SUIF [WFW+94] compiler infrastructure. Our approach relies on the ROSE [SQ03] source-to-source compiler for constructing and transforming kernel representations, and for code generation.

Howes et al. [HLKF08, HLDK09] tackle the management of data movement to/from devices with software-managed memories, focusing on those with limited memories that require continuous runtime streaming to keep computational units occupied. Devices in this class include the Cell, with a small memory region local to each Synergistic Processing Element (SPE) [Hof05], and CUDA GPUs, with a small shared memory region in each multiprocessor [LNOM08, NBGS08]. The authors pursue a method for automating the management of data movement into and out of these limited memories, called Ækute. Their approach has similar requirements of information about data dependence and memory access patterns to our own. These needs are satisfied by programmer-supported metadata annotations, in the form of *access/execute descriptors*, which capture information about the memory access patterns and scheduling requirements of computational kernels. These are analogous to our constructs of *indexers* and *functors* (Chapter 3) and are similarly expressed in C++ source code. Metadata is generalised in this work beyond our approximations of loop-carried dependence and spatial filter memory access patterns, into arbitrarily complex mappings, but no practical applications for this level of detail are described. Our

work is indistinguishable in ideals but divergent in implementation and in the role of metadata. Æcute is implemented as a runtime library that intercepts data accesses and initiates asynchronous DMA transfers to keep a continuous stream of data in the small amount of available space. We opted instead for a hybrid runtime/static code generation scheme due to our needs for schedule and kernel transformations, which would be expensive to apply and recompile at runtime without a caching mechanism that would complicate software distribution and deployment. Convergence of our research lies in the generalisation of data dependence and memory access metadata. We are still getting to grips with the applications of metadata at different levels of abstraction (see Sections 3.3 and 7.5) but believe there is a generalised definition which could meet the needs of both threads of research.

Baskaran et al. [BBK+08] present parallelisation and optimisation extensions to the PLuTo [BHRS08] compiler framework for SIMT [NBGS08] architectures. PLuTo exploits the polyhedral model (Section 2.3.5) for unassisted C parallelisation and data locality enhancement. Extension to SIMT code generation is primarily a task of programming the hierarchical memories efficiently and modifying tiling and loop unrolling optimisations [BGS94] to suit the SIMT execution model. The approach is very different from our own and relies on a fusion of traditional compiler analyses and low-level transformations with the polyhedral model to assist higher level transformations. Similarities to our work lie mainly in the SIMT memory optimisations targeted to improve the bandwidth and latency of data accesses from global and shared memory; e.g. through coalescing and shared memory staging. The authors exploit the low-level optimisation capabilities of the PLuTo framework to perform loop unrolling and sub-thread-block tiling. This is a weakness of our approach since we do not have enough information about a kernel to tackle kernel optimisations of this kind. There is no evidence, however, to suggest that the parallelisation functionality is able to match the serialisation-breaking parallelism enhancements discussed in Section 6.1.5; a key strength of our methodology. Results are presented for simple variations of transpose, matrix and vector multiplication; codes that are well-suited to traditional compiler analysis. It is not clear if the ideas are scalable to the complex kernels that we deal with in this thesis.

Püschel et al. [PMJ+05] describe a domain-specific optimised code generation framework for signal processing algorithms, which they call SPIRAL. Their key innovation is to begin from an algebraic description of the algorithm, in which analysis and optimising transformations are easy because the representation is at a high-level. For example, rather than generating code for two spatial filters and then attempting to fuse them, one could compose the mathematical representation before any implementation has been created. Implementation-level optimisations, such as parallelisation and loop unrolling, benefit from the clear information about the algorithm recorded in its algebra. This is similar to our use of structured metadata to record information about algorithms that would otherwise be difficult to recover from code. SPIRAL is more restrictive in the algorithms it can express – for example, there is no support for moving averages – but the strong domain-specificity gives rise to a large array of static and dynamic optimisations for discrete linear signal transforms, with very favourable performance. The authors discuss a similar dilemma to the one which we face: of finding a more generic structured representation of algorithms with a

set of portable optimisations. Our research is one step in this direction and a good place to begin exploring the difficulties in transitioning domain-specific optimisations to wider applications.

Donaldson et al. [DRLC08] present a semantic annotation methodology and optimising compiler for the parallelisation of C++ code fragments. The key innovation is the use of block semantics to mark statements involving accesses to externally defined arrays as free of true dependence, in order to sidestep the difficulties of precise dependence analysis in the presence of pointer aliasing. This is similar to our use of constructs to communicate information about data dependence directly to the compiler, but with far less rigid requirements on application structure. Parallelisation is implemented very differently from our own methodology; user annotations hint at parallelisable loops and speculative execution is used to create parallelism where iterative patterns cannot be determined statically. These semantics are further used to assist in staging data through hierarchical memories, which we achieve with memory access metadata. This approach is less invasive to the client application code than our own method and would, in that regard, better meet our goals. However, the substantially modified language semantics implied by an annotated block are not necessarily well represented by only a minor modification of the source code. Furthermore, information about the patterns of memory access is limited with these annotations alone.

Seinstra et al. [SK04] describe the Parallel-Horus project, a drop-in replacement for the established Horus [Koe02] image processing library with data- and task-parallel extensions. The authors employ an algorithm representation based upon image algebra [RW96], citing previous work [MCB+99] which exploited this in an active library [CEG+00, VG98] for SMP parallelisation. They note concerns about this earlier work in the limited scope of collected graph fragments between delayed evaluation force points, which may inhibit whole-program optimisation if they are of significant number. Our research has similar limitations in scope between visual effects which, in a complete digital composition, may be written by competing vendors and would not be likely to take part in cross-component optimisation. Image algebra is cited as a domain-specific subset of algebra in which a small number of primitive classes can express the majority of image processing algorithms; analogous to our classification of algorithms with a set of common data dependence and memory access patterns. The implementation described shares features with our own framework, through a common representation of algorithms as templated C++ classes with parameterised features and a kernel. The approach described in the paper is distinguished by the use of hardware performance models to adapt the parallelisation scheme for different architectures. We currently use a limited set of runtime parameters (e.g. the CUDA Compute Capability parameters described in Section 6.2) to tune parallelisation but feel that performance models are too rigid. In the future we plan to explore self-tuning optimisations.

Jamieson et al. [JDW+92] undertook an early study of a cross-architecture framework for the parallelisation of computer vision algorithms. The authors were interested in mapping these algorithms to contemporary parallel architectures, which consisted of SIMD CPUs in SMP configurations, while isolating parallelisation expertise in a software framework. They make the familiar observation, albeit in a slightly different domain, that the regular structure of computer vision algorithms could assist in the construction of a domain-specific optimising programming

framework. A number of algorithm characteristics are identified which could assist in parallelisation and algorithm-device mapping; including patterns in data dependence and memory access patterns which we record in metadata. Further similarity can be found in the representation of tasks – analogous to a visual effect (Section 2.2.2) – as cyclic directed graphs of primitives. Our work is distinguished by optimisations beyond parallelisation, and by parallelising transformation for many more threads than were needed at the time of this publication.

Sérot et al. [SG02] revisited the problem of parallelisation of computer vision algorithms with the SKIPPER project, building upon algorithmic skeletons [Col91] to present a functional programming interface with well-defined semantics. Algorithmic skeletons inspired, and are analogous to, our concepts of *functors* with *indexers* (Chapter 3). Our representation is a hybrid of functional and imperative programming, to selectively exploit benefits of both paradigms, whereas the authors of this paper pursued a purely functional approach. The most recent incarnation of SKIPPER records a data-flow graph of primitives and the dependencies between them; identical to our approach. As in earlier work, this paper is primarily concerned with the problems of distributed parallelisation and communication; not of local kernel optimisation. Our research does not focus on distributed memory architectures because this field is very mature, with highly scalable frameworks already in industrial use.

2.6 Concluding Remarks

We have covered a wide selection of background material and related research in this chapter. A subset of these publications have been chosen to form the foundations for this research project from which we can build upon to achieve our objectives. What follows is a selection of publications that inspired the key underlying features of our computational framework described in Chapters 3, 4, 5 and 6, with justification for the achievement of our goals in each case.

- **Bastoul** [Bas04]. The CLooG library embodies a substantial amount of polyhedral scanning research to deliver a fast, production-quality (see GCC 4.4) loop transformation framework. We translate a metadata representation of loop nests and data dependence into the polyhedral form, in which the performance-critical space/schedule optimisation is an instance of the already solved scanning problem.

- **Beckmann et al.** [BHKM03]. Runtime code generation plays a key role in a semi-offline optimisation phase of our framework. While we chose not to move the complete optimisation process to runtime, as advocated by the authors in this paper, a partial runtime phase greatly simplifies the collection of program metadata and specialisation of appropriate algorithm implementations. By limiting specialisation to non-variable parameters, and developing some novel ideas to adapt to some variable parameters, we reap many of the benefits of runtime code generation without the cost of compilation.

- **Cole** [Col91]. Our visual primitive design borrows heavily from the ideas developed in algorithmic skeleton research. Skeletons carry the benefits of functional programming to

imperative languages, where the clear computation and communication structures trivialise the problem of parallelisation for modern SIMD and SIMT architectures.

- **Czarnecki et al.** [CEG+00]. The metaprogramming active library concept underpins our methodology of applying complex optimisations to client applications without modifying them or requiring performance expertise on the developer's part. Leveraging the active library interface at runtime, through delayed evaluation, enables our library to build information about algorithm context without source code analysis.

- **Kelly et al.** [KBFB01]. Previous work on cross-component optimisation and dependence metadata heavily influences our component-oriented visual effect design and metadata encapsulation in indexers. Reusable component-based programming matches the structure of a visual effect well and is thus easy and natural for algorithm developers to work within. Dependence metadata plays a key role in numerous optimisations developed in this thesis.

- **Schordan et al.** [SQ03]. ROSE forms the parsing, transformation and unparsing framework used to implement high-level source code translations and optimisations. A source-to-source approach makes for simple integration into any development workflow and allows us to leverage the existing low-level optimisations provided by vendor compilers.

- **Song et al.** [SXWL01]. The memory reduction optimisation proved to be a turning point in our optimisation research. Visual effects algorithms are frequently memory intensive and naturally structured as graphs of independent loop nests. This was an ideal opportunity to explore the role of metadata in memory reduction and the resulting performance gains were highly beneficial to our goal.

Chapter 3

Metadata-Augmented Single-Source Framework

3.1 Introduction

This chapter presents our first contribution: a domain-specific data-parallel programming framework with metadata augmentations to support the code generation and optimisation techniques discussed in Chapters 4, 5 and 6. We begin by formalising a set of constraints upon the visual effects domain to focus our study upon a subset of key algorithmic patterns. Next, we outline a set of metadata which encapsulates the data dependence and memory access structures of each algorithm for use in analysis-free code transformations. This is followed by a detailed discussion of the framework implementation and the scalar C code generation process employed in the associated source-to-source compiler; in subsequent chapters we build upon this compiler infrastructure. The chapter concludes with a study of two visual effects implemented within this framework and demonstrates speed-ups of 1.4x–2.2x from identity code generation and static specialisation. The material in this chapter is based upon a development of previously published work [CKPN07, CHK+09].

Whilst reading the forthcoming sections, the reader may wish to refer forwards to Figures 3.2, 3.3 and 3.4 to see how the ideas discussed within fit into our framework design. Section 3.4 explains this design in detail and illustrates how these ideas are realised in our prototype software.

3.2 Constraints upon the Visual Effects Domain

In Section 2.2 of the preceding chapter we outlined the classes of algorithms constituting a visual effect. This definition is suitable for expressing a wide range of visual effects but is overwhelming in its scope for optimisation. Fortunately, a large class of effects can be expressed within a much narrower domain definition. Throughout this thesis our study will focus upon this subdomain, and consideration of the full domain will be limited to contextual discussion and proposals for further work. We now formalise the constraints which define this subdomain and justify our decisions.

Constraint 3.2.1 (Data Type) *A component in a digital image is of type IEEE 754 single-precision floating-point.*

Digital images are typically comprised of one of a wide range of data types – such as *unsigned char*, *signed short*, *signed int*, *float* or *double* – in disk storage and in memory. The type is always uniform throughout the digital image. Constraint 3.2.1 narrows our focus to *float* data types. This is by far the most common data type used in VFX pipelines, owing to its favourable balance of dynamic range, size and processing throughput. There is no fundamental limitation in the ideas presented in this thesis which would prevent them from being applied to other data types, but *float* is the most suitable type in terms of space and performance for the CPU and GPU instruction sets we wish to program. A possible branch of further work might explore the performance opportunities of reduced precision (e.g. *unsigned char* on a CPU or *half* on a GPU) when *float* is unnecessarily precise, as it often is in VFX algorithms.

Constraint 3.2.2 (Digital Image Layout) *A digital image is stored as a 1D array of pointers to 2D channel arrays, whose rows may be padded to achieve correct row-base alignment.*

We restrict our interest to the planar image layout discussed in Section 2.2.1 for the reasons outlined in that section. In addition, we permit image row *padding* to avoid row-to-row address aliasing problems and to satisfy the data alignment requirements of some ISAs. Most VFX processing is already done with padded planar image formats so this constraint does not significantly limit the applicability of our research.

Constraint 3.2.3 (Image Reference Aliasing) *References to digital images within a visual primitive must not alias each other.*

A common space and performance optimisation in image processing is to perform *in-place* processing operations: reading from an image and overwriting the data within it. This reduces memory utilisation and cache pressure, but optimises prematurely by introducing additional data dependencies which can block other optimisations. Constraint 3.2.3 enforces unique images to each input and output of a visual primitive; sharing can only occur between visual primitives. Should an in-place optimisation be desirable, we can reintroduce it in code generation.

Constraint 3.2.4 (ROI / DOD Alignment) *The ROIs and DOD of a visual primitive are centred around location (0,0).*

In generalised digital composition the alignment of a visual primitive's ROI with respect to its DOD is unrestricted; this is useful in e.g. shifting an image region. Furthermore, there need be no correlation between the ROIs and DODs of different visual primitives within a composition. This is typical of a need for varying overlaps of effects on different parts of an image. Constraint 3.2.4 enforces a common centre for all ROIs and DODs at the expense of this flexibility. It is a necessary constraint to minimise the combinatorial explosion encountered in the shifted loop fusion transformations described in Chapter 4. One might be concerned about the limiting impact this

may have on arbitrary digital composition. In fact, any such composition can be broken down into subcompositions in which the DODs of constituent primitives share a common centre. Additionally, the correspondence between the ROIs and the DOD of a visual primitive can be restored by offsetting input images by an appropriate amount, as this would not affect the loop structures in the generated code.

Constraint 3.2.5 (ROI / DOD Size) *The ROIs of a visual primitive must be either the same size as its DOD or symmetrically grown in one or both dimensions.*

In Constraint 3.2.5 we are establishing the input data needs of an algorithm to produce a specified amount of output. Algorithms with a 1:1 mapping of input to output data elements will have ROIs of identical size to their DODs. The provision for a ROI to be symmetrically larger – that is, larger by equal amounts on both sides – than the DOD is specifically intended for spatial filters (Definition 2.12), which consume a small region of data beyond the DOD within their filter radii. Symmetricity is a natural property of spatial filters and a requirement to avoid combinatorial explosion in the shifted loop fusion transformation discussed in Chapter 4. One can certainly conceive of algorithms which do not satisfy this constraint – and we may simply have to avoid schedule optimisation in those cases – but it is appropriate for a wide range of visual primitives, including those studied in this thesis.

Constraint 3.2.6 (ROI / Image Correspondence) *The ROIs of a visual primitive must be equal to or smaller in size than their corresponding images.*

It is common for visual primitives to require more image data than is available to a digital composition. This data is artificially constructed on-demand through a predefined edge method – such as *set-to-black*, *clamp-to-edge*, *reflect*, etc. – associated with the primitive. Constraint 3.2.6 requires that there is sufficient real image data for the visual primitive to process; where this is not the case, input images are grown beforehand with a built-in visual primitive. We chose this constraint to alleviate the computationally expensive conditional tests – or fragmented loop structures – required to construct missing data on-demand. The disadvantage of this approach is that it requires more memory, which may be unsuitable for primitives with large ROI:DOD ratios.

We have recently experimented (in work not presented in the thesis) with using lookup tables in place of conditional tests or loop fragmentation to good effect, and in future work this constraint may be alleviated. Figure 3.1 illustrates the use of a lookup table to determine when memory accesses fall outside of the image data and to provide pre-generated edge values in these cases. Since the lookup tables are small, even for large images, we could reasonably expect them to reside in a low cache level. Preliminary experiments showed that algorithm performance with this access method was very close to that of the pre-grown image method, although the impact of increased cache contention has not been assessed in a wider context.

Constraint 3.2.7 (Visual Primitive Parallelism) *The kernel of a visual primitive must have no loop-carried dependence when iterated across all pixels of an image, or it must exhibit loop-carried dependence in one axis only.*

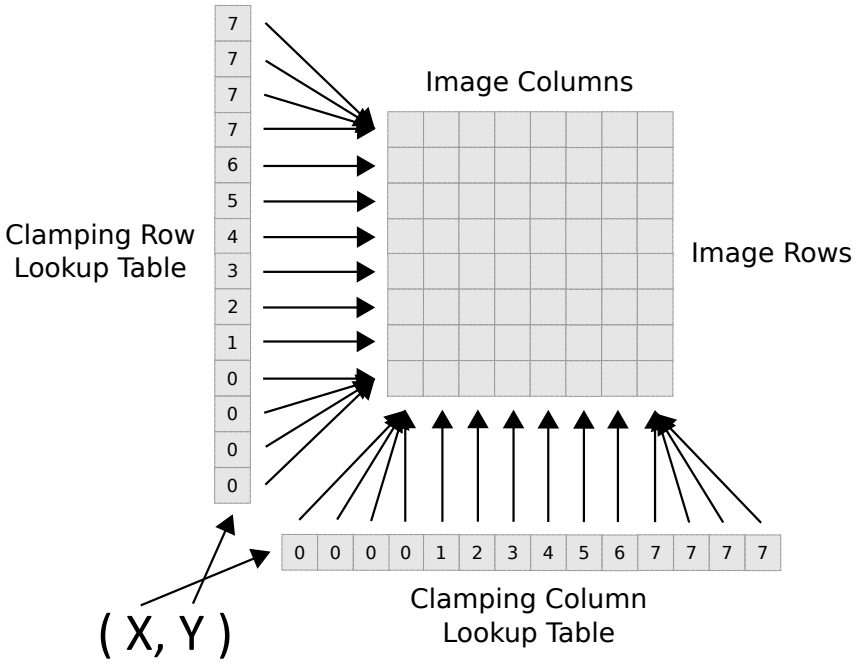


Figure 3.1: An alternative image indexing method with which we have begun preliminary experimentation. An 8x8 image is accessed through a 14x14 ROI defined by $(-3,-3) \rightarrow (10,10)$. The X and Y coordinates are clamped via two small lookup tables (LUTs) in place of expensive conditional branches. In practice, the Y LUT can directly supply an image row pointer. Similar formulations exist for other edge methods such as *set-to-black*, *reflect*, etc.

Our final constraint is the most restrictive of all. VFX algorithms exhibit a wide variety of data dependence structures. Most of these algorithms – or, at the very least, alternative formulations of them – exhibit substantial amounts of parallelism. However, only a subset is entirely absent of loop-carried dependence. Constraint 3.2.7 provisions for one class of algorithm that is not fully parallel: the moving average (Definition 2.11). We expanded our study to incorporate this class because one particular moving average algorithm, the box blur, is used frequently in VFX. This algorithm can be formulated as a fully parallel spatial filter, but a variant with optimal memory access complexity needs to maintain serialised state along the rows or columns of an image. Much of our planned work involves relaxing Constraint 3.2.7 to support a wider class of visual effects. However, it is not entirely clear how well the optimisations described in Chapters 4, 5 and 6 would work in the presence of more complex dependence structures and this requires further study. In Section 7.4 we briefly discuss some post-thesis industrial developments in this area.

3.3 Metadata Definition

Before exploring the framework in detail it is worth outlining our design goals. The key innovation in this research is the use of *metadata*: high-level information about an algorithm embedded alongside its implementation. By integrating metadata into the framework frontend we are able to collect it during code generation and use it to direct and assist optimising code transformations. A

key measure of success is for these optimisations to be *analysis-free*. Code analysis – recovering high-level information from a program implementation – is the primary barrier to optimisation of object-oriented programs in modern compilers. By sidestepping analysis with metadata we are able to focus on the optimisations themselves, rather than the establishment of their necessity or correctness. It is important to note that the correctness of this metadata for a given algorithm can be checked at runtime to assist debugging. Each of the following subsections outlines a class of algorithm metadata which we found useful in code generation and optimisation. All of this metadata is embedded within the framework frontend, described in Section 3.4.1, and is used in the optimisations discussed in Chapters 4, 5 and 6.

3.3.1 Primitive Context

Recall from Definition 2.9 that a visual effect is comprised of a connected DAG of visual primitives and interconnecting images. Two such DAGs are shown in Figure 3.2: one which we saw earlier in Chapter 2, of the wavelet-based degraining visual effect, and another for an (unpublished) diffusion filtering visual effect. Both of these effects were developed commercially by our industrial collaborators, The Foundry, and have constructions that are representative of many different visual effects. The diffusion filtering effect contains proprietary algorithms and thus for reasons of non-disclosure we limit their descriptions to point/filter/moving average classifications.

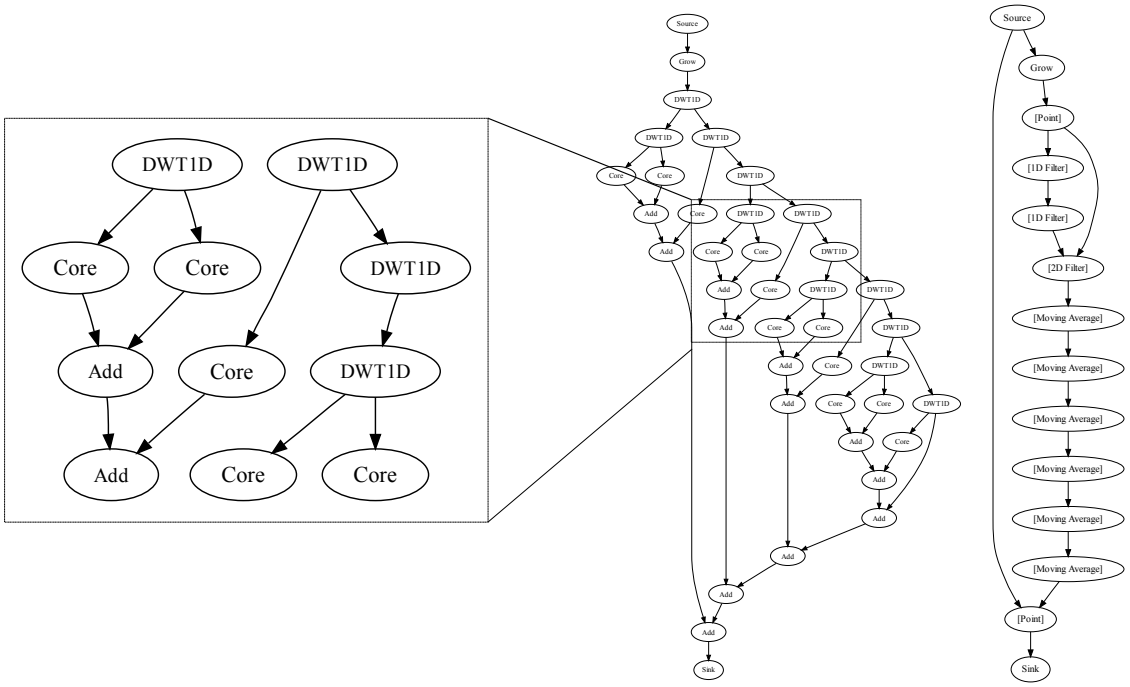
Metadata 3.3.1 (Visual Effect DAG) *A DAG structure with node properties referencing visual primitives and algebraic expressions of their DODs, and edge properties referencing image handles and the ordinals of the input and output in the pairs of visual primitives which they connect.*

The visual effect DAG constitutes an important piece of metadata because it provides context for the execution of a visual primitive. Inter-primitive optimisations, such as schedule optimisation (Chapter 4) and redundant transpose elimination (Chapter 6), require this information to determine where and how to apply their transformations. For example, a loop fusion optimisation between a spatial filter and an adjacent point primitive might be beneficial while a similar optimisation between adjacent spatial filters may not, due to the larger temporal reuse distance and contracted working set size. DAG metadata is stored internally as a dynamically constructed, traversable graph structure. It is implicitly correct due to its method of construction and needs only cycle detection to assert conformity.

The extent of the DAG is bounded by evaluation force points in the client application. An important consequence of this is that within effects whose DAGs grow dynamically until a data-derived termination condition is met, the complete DAG cannot be known and optimised in aggregate. The smaller DAGs between evaluation force points can still be optimised in isolation.

3.3.2 Data Dependence

As specified by Constraint 3.2.7, a visual primitive must either have no loop-carried dependencies or have loop-carried dependence in the horizontal or vertical axis. Constrained dependence blocks



(a) Wavelet-based Degraining [SCW05]

(b) Diffusion Filtering

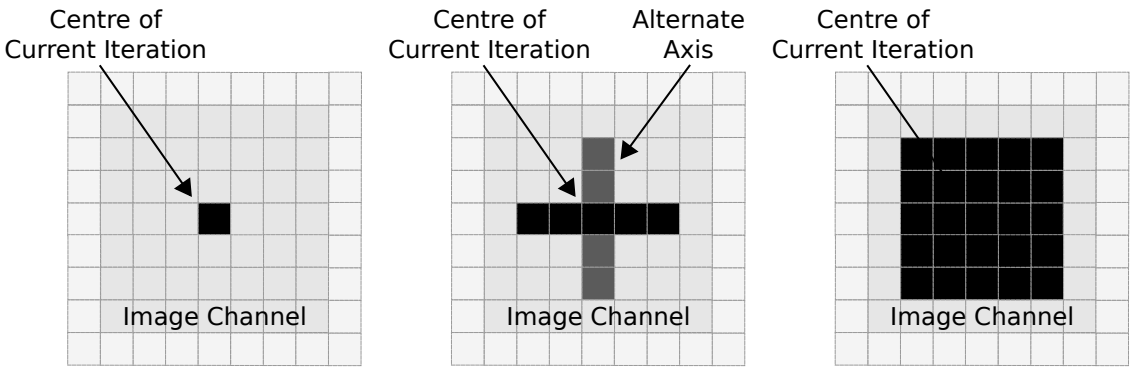
Figure 3.2: DAG constructions of two commercially developed visual effects. The visual effect DAG is one piece of metadata recorded in the frontend and used in optimisation. Each node is associated with a parameterised visual primitive. The edges between nodes indicate the flow of image data from primitive to primitive.

some optimisations, such as vectorisation (Chapter 5) and coalescing (Chapter 6), and necessitates others, such as split- row/column parallelisation (Chapter 6). However, it plays an important role in reducing algorithm complexity by permitting partial data reuse. By encoding the algorithm dependence – as fully parallel, or with a loop-carried dependence and its corresponding axis – we can ensure that optimisations do not violate dependence and, where necessary, employ additional optimisations to counteract its negative effects.

Metadata 3.3.2 (Visual Primitive Dependence) *A static classification of each visual primitive as fully parallel or moving average (Definition 2.11) and, in the latter case, a dynamic record of the serialised axis.*

To verify that dependence metadata is correct, one needs checks to assert that:

- Input images are read from and not written to, and vice versa for outputs.
- No variables external to the primitive representation (e.g. globals, variables accessed through pointers, etc.) are written to.
- Only non-const class member variables are written to by the visual primitive representation (see Section 3.4.1) when moving average metadata is specified. None must be written in the absence of stated loop-carried dependence.



(a) Point (0D) (b) Spatial Filter (1D) (c) Spatial Filter (2D)

Figure 3.3: Local memory access patterns currently permitted by our memory access metadata. Each pattern is localised around a current iteration point in the image. For spatial filters, both the axis (in 1D patterns) and size of the region inside which memory access is permitted are recorded dynamically. This allows a uniform representation of horizontal/vertical spatial filters with dynamically tunable filter bounds to e.g. change a brush size.

Simple syntactic analyses in the code generator are sufficient to check each of these points and to warn the programmer if the metadata does not agree with the implementation.

3.3.3 Memory Access

Constraints 3.2.4 and 3.2.5 restrict the possible patterns of memory access in a visual primitive. We further constrain memory access by annotating the inputs and outputs of each visual primitive with metadata describing the patterns of access which will be made to image data through them. Our pattern classification is illustrated in Figure 3.3. Note that only the bounds of permissible localised access is defined; the algorithm is free to make random accesses within that region. The choice of dimensionality is made statically, since the semantics of the kernel must change to support different degrees of freedom. Both the axis (for a 1D region) and the bounds of a region are chosen at runtime to enable user-tunable primitive variants.

Metadata 3.3.3 (Visual Primitive Memory Access Pattern) *For each input/output of a visual primitive: a static choice of per-pixel or per-component and the dimensionality of the local access region, and a dynamic record of the axis of freedom (for one degree of dimensionality) and size of the access region (for one and two degrees of dimensionality).*

Verification of this metadata is most easily performed by intercepting image accesses at runtime and checking that they lie within the specified local region. Where the specified size of the region appears to be conservative, the runtime system may issue a warning to the user that this may reduce the effectiveness of temporal optimisations.

Memory access metadata is used by space and schedule optimisation (Chapter 4) to determine optimal loop shifts and array contraction sizes. SIMD optimisation (Chapter 5) makes use of this metadata to determine vector load alignment. The metadata is further used by SIMT optimisation (Chapter 6) to identify memory hierarchy staging opportunities and to bound them precisely.

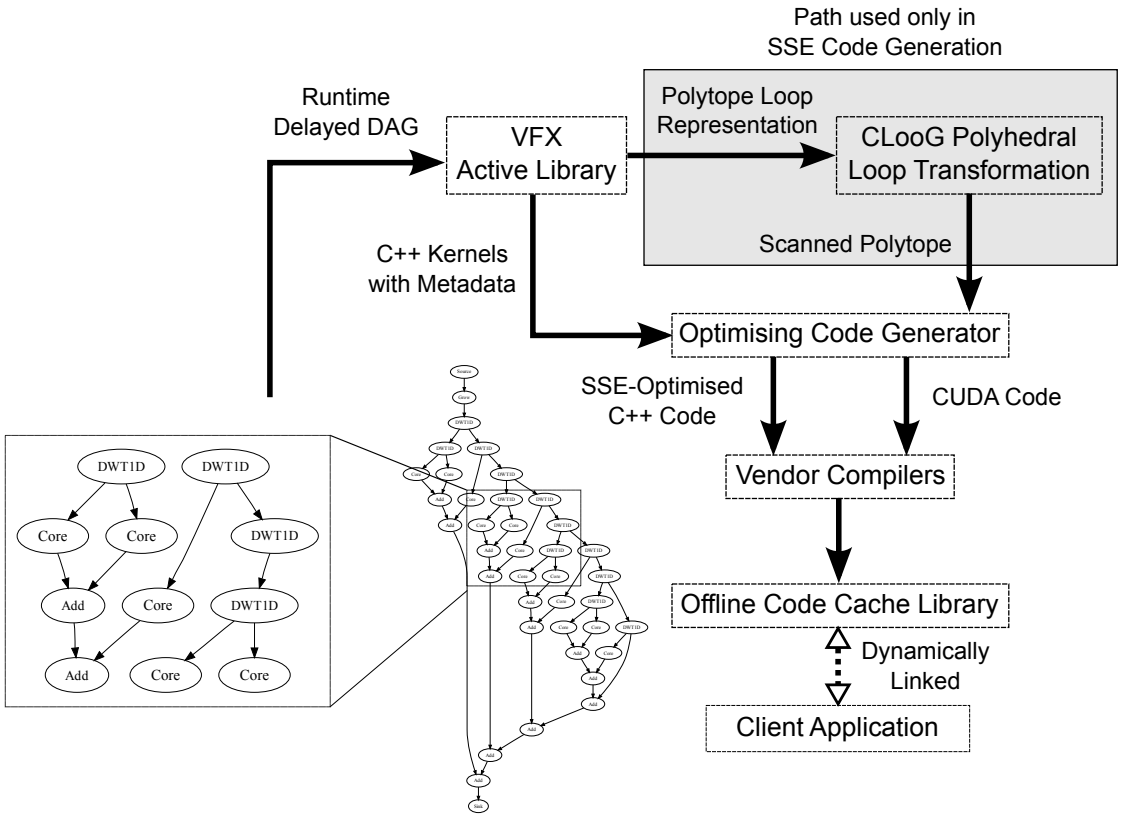


Figure 3.4: A block-level overview of the domain-specific VFX single-source representation, code generation and optimisation framework.

3.4 Framework Design

With a clear definition of the constraints and goals of our framework set out in the preceding section, we now turn our attention to the framework design and its implementation. During the three year period of research leading to the creation of this thesis we evaluated a number of different designs and progressively refined them to support new algorithms, backends and optimisations, and to incorporate feedback from our industrial partners. The following description is of the final prototype framework, at the time of writing of this thesis, and the reasoning behind its design decisions. Substantial refinements have been made to the framework during its transition to industrial software, mainly to support more complex algorithms, but these are not discussed here and have not yet been subjected to our rigorous performance tests.

A high-level overview of the framework is given in Figure 3.4. Code generation and optimisation is built around a hybrid runtime/static paradigm. We have built upon ideas from pure runtime solutions (e.g. [RMKB08]) but added an offline phase to alleviate the expense of runtime compilation. The offline code cache is populated during application development and shipped with the product in binary form, from which composite fragments for different devices can be located and executed at runtime. Population occurs during development test runs of the client application, where the availability of dynamically collected DAG metadata (Metadata 3.3.1) enables specialisation of fixed sequences of visual primitives in the effect. The polyhedral loop optimisation path

is only invoked during CPU code generation since we have not yet developed an automatic mapping of arbitrary loop nests to the threads of a SIMT architecture (see Section 6.3 for discussion). Preliminary work is underway on mapping a subset of loop nests – for example those with perfect nesting – so that simple schedule optimisations can be translated to the GPU.

We now describe the main features of the framework frontend in detail.

3.4.1 Visual Primitive Representation

For a formal definition of the framework constructs, please refer forwards to Appendix A.

Our choice of algorithm and metadata expression was driven by the objective which required our framework to integrate seamlessly with the development workflow (see Section 1.2). Existing algorithms in the industrial development process were implemented and debugged in C++. We chose an embedded domain-specific language approach [Hud96] to maintain this workflow whilst augmenting it with the facilities we needed to encode metadata. Other possible approaches, such as an external domain-specific language or preprocessor annotations, were considered too disruptive to have a realistic prospect of industrial take-up. The disadvantage of the embedded DSL is that C++ is a difficult language to parse (both syntactically and semantically) and we would need a powerful tool to extract the metadata.

A visual primitive is expressed as a C++ class with a data-parallel kernel and object metadata annotations. The representation for the discrete wavelet transform (DWT) [She92] is shown in Listing 3.1. DWT is normally implemented as a two-dimensional finite impulse response (FIR) filter to divide a signal (in this case an image) into two frequency domains. Here, we use a more cache-efficient one-dimensional form which can be composed in both axes to implement the Haar transform [Chu94]. This is an example of an algorithmic optimisation which our framework is not designed to automate. Once the developer conforms an algorithm to a particular implementation, we control only the execution schedule and the movement of associated data. SPIRAL [PMJ+05] is an example of a framework which does implement high-level mathematical algorithmic transformations of this kind by encoding the mathematical form and its transformation rules.

The data-parallel kernel serves a similar purpose to the Intel Threading Building Blocks' (TBB) *parallel_for* [Rei07] callback function. It is invoked iteratively across the DOD to compute each pixel in the output image(s). Separating iteration control from the algorithm body in this way enables loop optimisations, including parallelisation and fusion, within strictly controlled and explicitly annotated iteration structures. The iteration structure in Listing 3.1 is the simplest – annotated as eParallel for fully parallel – of a number of possible structures, such as moving average (Definition 2.11) or reduction (e.g. TBB's *parallel_reduce* [Rei07]). Memory access is similarly abstracted through metadata-annotated *Indexer* objects to form a ROI/DOD correspondence in the absence of explicit iteration. Each indexer is annotated with read/write semantics (eInput or eOutput), granularity (eChannel or ePixel) and degrees of local freedom (e0D, e1D or e2D). The notation *Indexer()*, *Indexer(ds)* and *Indexer(ds,dt)* indicates read or write array access, with optional *ds* and *dt* offsets in the case of spatial filter indexers (see Figure 3.3).

In addition to metadata annotations describing the static structure of the algorithm, two dy-

```
class DWT1D : public Functor<DWT1D, eParallel> {
  Indexer<eInput,  eChannel, e1D> Input;
  Indexer<eOutput, eChannel, e0D> HighOutput;
  Indexer<eOutput, eChannel, e0D> LowOutput;
  mFunctorIndexers(Input, HighOutput, LowOutput);

  DWT1D(Axis axis, Radius radius) : Input(axis, radius) {}

  void Kernel() {
    float edge1  = Input(-Input.Radius);
    float centre = Input();
    float edge2  = Input(Input.Radius);

    float high = (centre - (edge1 + edge2) * 0.5f) * 0.5f;

    HighOutput() = high;
    LowOutput() = centre - high;
  }
};
```

Listing 3.1: The one-dimensional discrete wavelet transform [She92] in a C++ visual primitive representation. It is valid, compilable code and operates in the runtime-parameterised horizontal or vertical axis. Static and dynamic object metadata is underlined.

namic parameters – axis and radius – modify the behaviour of the *Input* indexer to apply the local offset *ds* in the horizontal or vertical axis and to adjust the spatial limits of local access. The former improves maintainability by enabling a uniform implementation of axis-agnostic spatial algorithms, instead of requiring two separate but almost identical primitives for the horizontal and vertical implementations. The latter is used in schedule optimisation (see Chapter 4) to dynamically tune loop shifting and array contraction factors in order to minimise data reuse distance.

A second visual primitive, for the box blur algorithm, is shown in Listing 3.2. This kernel has eMoving dependence, representing a moving average (Definition 2.11) with a serialised axis. The serialised axis is chosen dynamically when constructing the *BoxBlur* object to enable a uniform representation for both horizontal and vertical forms. In addition to the kernel function, which reads and writes the non-constant *MovingSum* state member variable, a *roll-up* function is executed at the beginning of each serialised iteration to initialise state variables. For a box blur, this involves constructing a partial sum which can be incrementally modified by subtracting a value that falls outside the filter window and adding a new value that falls within at each iteration point.

3.4.2 Visual Effect Construction

A single visual primitive is rarely complex enough to implement a complete visual effect by itself. Rather, it is intended to encapsulate a minimal reusable unit whose inputs and outputs may be chained with other primitives – statically or dynamically – to build a complex composite operation. The visual effect DAGs in Figure 3.2 are comprised of groups of primitives with image connections between them. The only form of connection permitted between primitives are images carrying intermediate data. Similarly, the source and sink nodes of a DAG carry image data into

```
class BoxBlur : public Functor<BoxBlur, eMoving> {
  Indexer<eInput, eChannel, e1D> Input;
  Indexer<eOutput, eChannel, e0D> Output;
  mFunctorIndexers(Input, Output)

  BoxBlur(Axis axis, int radius)
    : Functor<BoxBlur, eMoving>(axis),
      Input(axis, radius),
      MultBy(1.0f / float(radius * 2 + 1))
  {}

  void RollUp() {
    MovingSum = 0.0f;
    for(int i = -Input.Radius; i <= Input.Radius; ++ i)
      MovingSum += Input(i);
  }

  void Kernel() {
    MovingSum -= Input(-Input.Radius - 1); // Subtract element outside filter.
    MovingSum += Input(Input.Radius);      // Add new element inside filter.
    Output() = MovingSum * MultBy;         // Divide for mean of filter window.
  }

  const float MultBy;
  float MovingSum;
};
```

Listing 3.2: The one-dimensional box blur expressed as a visual primitive. An optimal implementation is expressed as a moving average (Definition 2.11), hence the roll-up and kernel functions to build and maintain a partial sum respectively along the axis.

and out of a visual effect.

Listing 3.3 shows the complete construction of the wavelet-based degraining [SCW05] visual effect from visual primitives. The *DeGrainRecursive* function takes image data as input (to form a source node) and produces image data as output (to form a sink node). As its name suggests, the function calls itself recursively to chain small groups of primitives into a larger group. Parameters to some of these primitives change at each level of recursion to select different wavelet frequency bands. The *Image* object does not necessarily carry concrete data. A client application would supply concrete image data to this function, but within the function, and between levels of recursion, empty image handles declared on the first line of the function connect the primitives together. The precise dimensions of these images are not computed until runtime and, indeed, following space optimisation (Chapter 4) may not exist at all.

Each primitive object is of the class type described in the preceding section and uses overloaded C++ operators to implement delayed evaluation. Runtime construction of the visual effect DAG bypasses the difficult problem of deriving primitive connections through static code analysis. For example, in Listing 3.3 the connections between levels of recursion cannot be determined statically without induction variable analysis [Wol95] on the *level* counter. By exploiting runtime traces of delayed evaluation calls, the problem of building the DAG becomes greatly simplified. A disadvantage of this approach is that conditional branches in the DAG – for example, to switch

```
Image DeGrainRecursive(Image input, int level = 0) {
  Image HY, LY, HH, HL, LH, LL, HHC, HLC, LHC, LLC, pSum1, pSum2, output;

  DWT1D hDWT(eHorizontal, 1 << level); // Horizontal, radii [1, 2, 4, 8]
  hDWT(input, HY, LY);

  DWT1D vDWT(eVertical, 1 << level);   // Vertical, radii [1, 2, 4, 8]
  vDWT(HY, HH, LH);
  vDWT(LY, LH, LL);

  Core core;
  core(HH, HHC);
  core(LH, LHC);
  core(HL, HLC);

  Add add;
  add(HHC, LHC, pSum1);
  add(HLC, pSum1, pSum2);

  // Go to the next level of recursion or terminate.
  LLC = (level < 3) ? DeGrainRecurse(LL, level + 1) : LL;

  add(pSum2, LLC, out);
  return out;
}
```

Listing 3.3: The recursive wavelet-based degraining [SCW05] visual effect expressed in C++. Visual primitives are chained together with images and placeholder image handles to form a DAG.

a subset of primitives on or off at runtime – require multiple invocations with different configurations to generate code for every path, whereas static analysis could conceivably build multiple DAGs concurrently. Furthermore, neither static nor dynamic analysis can cope with graphs that extend until a dynamic condition is reached – e.g. to implement an iterative solver. In the dynamic case an evaluation force point is reached before the next part of the graph can be constructed.

3.5 Execution Strategy

The intended use of this framework is to construct optimised code sequences corresponding to parts of a visual effect DAG or whole DAGs for deployment and reuse at runtime. In addition to this, we implemented a simple execution method which could be used to validate the images produced by the optimised code and provide a baseline performance measure from which we may estimate the gains made by different optimisations. This method makes no use of the metadata described in Section 3.3 and resembles existing industrial solutions with some enhancements. These enhancements are described in detail in the following subsections.

3.5.1 DAG Serialisation

Given a visual effect DAG construction, such as the ones in Figure 3.2, it is trivial to construct a graph algorithm which will visit each of the constituent visual primitives and execute them

```
Evaluate(SinkNode)

function Evaluate(Node) {
  if(Node.Evaluated)
    return

  // Any order, e.g. first-to-last
  foreach(InputNode in Node.Inputs)
    Evaluate(InputNode)

  AllocateOutputs(Node)
  Execute(Node)
  FreeUnusedInputs(Node)

  Node.Evaluated = true
}
```

Listing 3.4: Pseudocode for a space-oblivious visual effect DAG scheduling algorithm.

while respecting the dependence ordering. One such algorithm is given in Listing 3.4. This typically biases execution backwards along the first inputs of each node but produces a valid serialisation of the DAG. The biggest problem with this algorithm is that it is not sensitive to the differences in peak memory requirements of each possible serialisation which, depending upon resource availability, may impact performance. These differences in memory requirements arise from input nodes whose outputs are used by more than one node – so *FreeUnusedInputs()* cannot immediately release intermediate data in some serialisations – or input nodes which generate additional output images that are not relevant to the node we are trying to execute.

Finding a space-optimal serialisation of the DAG – that is, one that minimises the peak memory consumption – is a task scheduling problem with exponential complexity. One simply evaluates every possible serialisation of the DAG and simulates the execution to determine how much space it would require. For example, the DAG in Figure 3.2a has 3072 unique serialisations. Its optimal serialisation has a peak memory requirement of 432MB when using $\simeq$ 73MB images, while the least optimal solution needs 728MB. This large disparity arises in DAGs with many branches and necessitates smart scheduling to minimise resource usage.

We found the space-optimal DAG serialisation algorithm to run in only a few milliseconds for the 37-primitive wavelet-based degraining effect. However, it is clearly not scalable to larger effects and we are looking towards explore heuristic-guided scheduling algorithms to find good solutions within reasonable complexity. This problem is not considered in the remainder of this thesis and lies outside the scope of our optimisations.

3.5.2 DOD Propagation

Recall that the DOD (Definition 2.8) of a visual primitive places spatial bounds upon the image data being produced, while the ROI (Definition 2.7) bounds the consumed data in a similar manner. In a typical visual effect the client application might expect the source and sink images to be of the same size. Thus, the DOD of the sink node should equal the ROI of the source node. Because

```
Inverse Dominator Walk:
  Evaluate(Node)

function Evaluate(Node) {
  if(Node.Evaluated or IsSource(Node))
    return

  Rectangle NewDOD

  foreach((Edge, OutputNode) in Node.Outputs) {
    Indexer ConsumingIndexer = OutputNode.IndexerFromEdge(Edge)
    Rectangle ROI = OutputNode.DOD

    if(IsSpatialFilter(ConsumingIndexer))
      ROI.Grow(ConsumingIndexer.RadiusX, ConsumingIndexer.RadiusY)

    NewDOD.Union(ROI)
  }

  Node.DOD = NewDOD
}
```

Listing 3.5: Pseudocode for an automatic DOD computation algorithm. The output DOD is propagated backwards from the sink node and grown through the ROIs of spatial filters.

visual primitives with spatial filters indexers (see Figure 3.3) have larger ROIs than their DODs this can be difficult to effect in a chain of primitives. Furthermore, managing the ROI/DOD correlation between primitives is difficult and error-prone. Getting this wrong leads to out-of-bounds memory accesses and other undesirable problems.

For these reasons, our framework automates DOD computation within a visual effect; hence the use of unsized intermediate image handles in Listing 3.3. We use a graph algorithm with an inverse dominator traversal to walk the visual effect DAG from the sink node – where the desired DOD is known – back to the source node. This algorithm is shown in Listing 3.5. Spatial filters cause the ROIs to inflate and this correspondingly inflates the DODs of the visual primitives feeding them.

This algorithm is flawed, however. It is likely that the ROIs of visual primitives attached to the source node will now need more data than there is available to them. This is a well-known problem in image processing and is a consequence of the finite nature of practical data sets. In our description of Constraint 3.2.6 we discussed two solutions to this problem with different advantages. In the prototype software developed for this thesis we choose to grow source images to match the required ROIs with the clamp-to-black edge method. This is trivially achieved by inserting extra visual primitive nodes after the source node of a pre-written Grow type, whose DODs are larger than their ROIs, which segments the available and missing data regions.

3.5.3 Cache-Aware Iteration

Separation of iteration control from visual primitive kernels gives us flexibility in choosing an execution order that generates the complete set of output data. To benefit most from cache prefetching

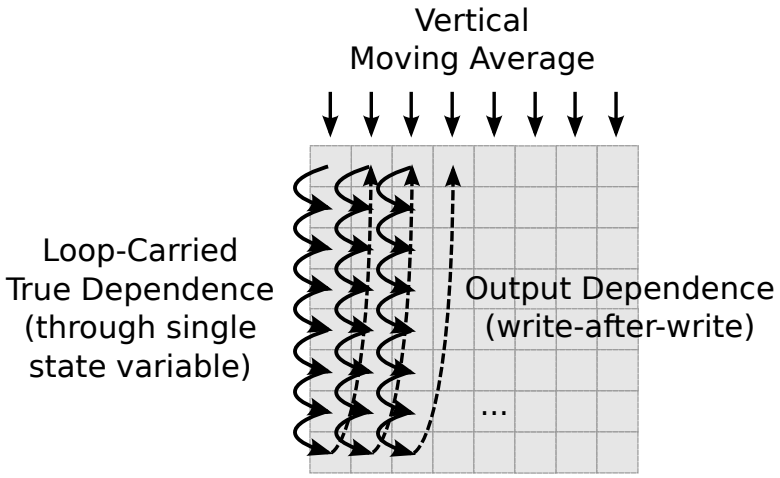


Figure 3.5: Vertical moving average kernels have true dependence along column iterations but only output dependence between the end of one column and the start of the next. By replicating one state variable per column the output dependence can be eliminated, permitting row-major, cache-friendly iteration.

we try to iterate across image data in row-major order. In the case of vertical 1D or 2D spatial filters there will be local column-major data access, but we can reasonably expect our working set to contain a cache line from each row, without spilling the lower cache levels, for most smaller spatial filters.

Moving averages (Definition 2.11) are the main barrier to row-major iteration. A vertical moving average maintains a single piece of state which it carries from the top to the bottom of each image column. This loop-carried dependence is violated in row-major iteration order, thus we are forced to choose the cache-suboptimal column-major execution order. For reasonably sized images it is likely that cache lines read near the top of the image will be expelled before they can be reused in the next column.

To alleviate this problem we note that there is a true dependence, between statements in successive iterations of the kernel, only along the column. See Figure 3.5 for illustration. Between columns there is a much simpler output (write-after-write) dependence because there is no data carried across. Thus, we may still iterate across columns (i.e. row-major order) if we create one copy of each modifiable member variable in the visual primitive for every image column and direct accesses to the correct one. This comes at the cost of slightly raised cache pressure, from the state variable array, but in practical cases is always beneficial to performance.

3.5.4 Multicore Parallelisation

Clearly an important concern in the multicore era is our parallelisation strategy. Most applications must choose from the two pillars of parallelism: task-parallel and data-parallel processing. Data parallelism is generally easier to exploit when it is available, and hybrids of the two approaches can often be employed to maximise processor utilisation. In fact, both kinds of parallelism are readily exploitable in the VFX domain but data parallelism is in such abundance that there is no need

to consider task parallelism. We are careful to draw a distinction between the two kinds of data parallelism commonly found in visual effects: cluster-level high-latency many-frame rendering and low-latency workstation single/few frame preview rendering. The latter, which is the focus of our performance research, requires sub-frame parallelism to minimise latency.

Within a single image our parallelisation strategy is to divide the DOD of each visual primitive into horizontal or vertical strips, each of which is computed by a different CPU core. Row-striding strips are preferred to columnwise strips because they minimise the number of boundary cache lines which may be contended by pairs of cores or prefetched unnecessarily. Of course, careful alignment of the boundary lines could also eliminate this. Moving averages may be partitioned along their parallel axis; with the state replication optimisation described in the preceding section this could be horizontal or vertical. Partitioning parallel to the state array requires one array per thread. Partitioning at right angles needs only one state array for all threads, but cache contention may occur at boundaries within the state array. Thus it is normal for each thread to allocate its own local subset of the array.

3.6 Code Generation

Our code generator builds upon the ROSE [SQ03] source-to-source compiler and CLooG [Bas04] polyhedral loop scanning library (described in Chapter 4). The majority of code generation and optimisation techniques are deferred until later chapters of this thesis. For now, we restrict our discussion to the task of generating C code from the C++ visual primitives. This is largely an identity syntactic translation with simple flattening of the C++ object-oriented representation to loops and arrays. In Section 3.6.1 we discuss a simple static specialisation optimisation which is appropriate for this stage. An example of the code generated by this process is given in Listing 3.6: this is a small fragment of the wavelet-based degraining visual effect.

Code generation begins by instructing ROSE to parse the effect's C++ source code and produce an AST. For each visual primitive in the runtime-delayed visual effect DAG, the associated primitive class in the AST is identified and passed through a syntactic translation process. The resulting code is compiled with the Intel C/C++ 11.0 compiler, using the flags *-xHost -O3 -ansi-alias -restrict -openmp -no-prec-div*, and bundled into a library for the client application to link against. The key translation steps proceed as follows:

- **Loop nest construction**. Form a loop nest with a component loop (R, G, B, etc.) on the outside – to minimise the cache working set by isolating unrelated data – a Y loop and an X loop. Primitives with per-pixel output access do not need the component loop. The Y loop is annotated with an OpenMP *parallel for* construct for stripwise parallelisation.

- **Variable localisation and hoisting**. All parameters and state variables in a primitive are hoisted outside of the loop nest and made *const* where possible. This localises the scope of all variables accessed by the loop nest to a single function; in particular we have observed performance gains by localising the constant bound of the X loop.

```
  float *restrict *restrict arr0x131cf50 = new float *[nPlanes];
  for(c = 0; c < nPlanes; ++ c) {
    arr0x131cf50[c] = f17.Indexers[1]->Image->PlanesZeroShifted[c];
  }
  const int arr0x131cf50Stride = f17.Indexers[1]->Image->RowStrideElems;
  float *restrict *restrict arr0x131cfc0 = new float *[nPlanes];
  for(c = 0; c < nPlanes; ++ c) {
    arr0x131cfc0[c] = f17.Indexers[2]->Image->PlanesZeroShifted[c];
  }
  const int arr0x131cfc0Stride = f17.Indexers[2]->Image->RowStrideElems;
  gettimeofday(&before, 0);
  for(c = 0; c <= nPlanes+-1; ++ c) {
    #pragma omp parallel for
    for(y = y1+-8; y <= y2+8; ++ y) {
      for(x = x1+-12; x <= x2+12; ++ x) {
{
  float centreVal = arr0x131b960[c][y * arr0x131b960Stride + x];
  float highVal = ((centreVal - ((arr0x131b960[c][(y + 4) * arr0x131b960Stride
    + x] + arr0x131b960[c][(y + -4) * arr0x131b960Stride + x])*0.5F))*0.5F);
  arr0x131cf50[c][y * arr0x131cf50Stride + x] = highVal;
  arr0x131cfc0[c][y * arr0x131cfc0Stride + x] = (centreVal - highVal);
}
      }
    }
  }
  gettimeofday(&after, 0);
  codeTimings.AddTiming(functors[17], before, after);
  f17.Evaluated = true;
  f17.FreeUnusedInputs();
```

Listing 3.6: Fragment of raw C++ code generated by our compiler (see Section 3.6) for the wavelet-based degraining visual effect. This fragment shows one of the 37 primitive executions in the effect: the vertical DWT1D primitive from Listing 3.1. The effects of static specialisation can be observed here: loop bounds are partially constant (from filter radii) and array lookups are offset by constant radii (4 and -4). This is derived from the dynamic construction in Listing 3.3.

- **State variable replication**. See Section 3.5.3 for details: we make hoisted, thread-local arrays of writable state variables and modify accesses within the kernel to reference them.
- **Output allocation and input deallocation**. Statements are inserted before and after the loop nest to allocate output images before writing to them and to free input images that are no longer in use. Because input images may be shared with other primitives, we use the visual effect DAG metadata to determine the earliest time to safely free them.
- **Indexer access substitution**. A simple syntactic translation replaces centralised indexer object accesses with their corresponding reads and writes from/to image arrays.

Once this translation is complete we apply the optimisation described in the next section.

3.6.1 Static Specialisation

Static specialisation is a program optimisation which provides additional information to the compiler to help it generate better code. In some cases this information can be extracted by the com-

CPU	Cores	L1 Data/Inst (KiB)	L2 (MiB)	L3 (MiB)	RAM (GiB)
Xeon E5420	8 (2x4)	32/32	12 (6 per 2 cores)	0	12
Phenom 9650	4	64/64	0.5	2	8
C2D E6600	2	32/32	4 per 2 cores	0	4

Table 3.1: Hardware specifications for the three benchmarking platforms used throughout the performance analyses in Section 3.7. The Xeon was in 2-chip SMP configuration. Both Intel chips share partitions of the L2 cache between pairs of cores.

piler through program analysis but our code generator presents an easy opportunity to ensure that this information is available. We are primarily interested in specialising the values of variables containing spatial filter radii and the axes along which they act. Radii are used as offsets for primitive loop bounds and as bounds for intra-kernel loops, while the axes determine the form of local offsets in array access expressions. Specialising their values enables better loop optimisations, such as loop unrolling [BGS94], and simplifies array access logic.

We talked earlier about the dynamic nature of spatial filter radii, which can be used to parameterise a spatial primitive algorithm without modifying its implementation. This is a case where specialisation cannot occur; or, at least, not without knowing which specialisations will be needed ahead of time.

There are other cases in which the radii of spatial filters are fixed. A good example can be found in wavelet-based degraining, shown earlier in Listing 3.3, where the radius of the DWT spatial filter primitive is a static parameter. $1 << level$ evaluates to 1, 2, 4 and 8 as the recursive procedure executes. Thus, we could make four specialisations of this primitive. In fact, we make eight by specialising the horizontal/vertical filter axis as well, which simplifies the array access logic. Overspecialisation can lead to code explosion and must be done with care. In order to derive the specialised radii from Listing 3.3 we would need induction variable analysis to determine the extent of recursion. To avoid this complication we exploit our runtime code generation phase to interrogate a primitive's indexer objects for their radii.

The results of specialisation can be observed in Listing 3.6. The values 4, 8 and 12 have been specialised and combined from static radii and substituted directly into the generated code.

3.7 Performance Analysis

To conclude this chapter we present an analysis of the framework's performance on two commercial visual effects, both before and after code generation and static specialisation. The two effects – wavelet-based degraining, discussed in part throughout this chapter, and diffusion filtering – are developed and sold commercially and were provided for performance experimentation by our industrial partners, The Foundry. Each has a very different computational pattern but is similarly constructed from smaller, reusable primitives. The visual effect DAGs for both effects were shown earlier in Figure 3.2. Wavelet-based degraining consists of a network of smaller primitives while diffusion filtering has fewer primitives and its performance is dominated by one, as we will see.

Our benchmarking platforms are summarised in Table 3.1. Each platform ran Ubuntu Linux

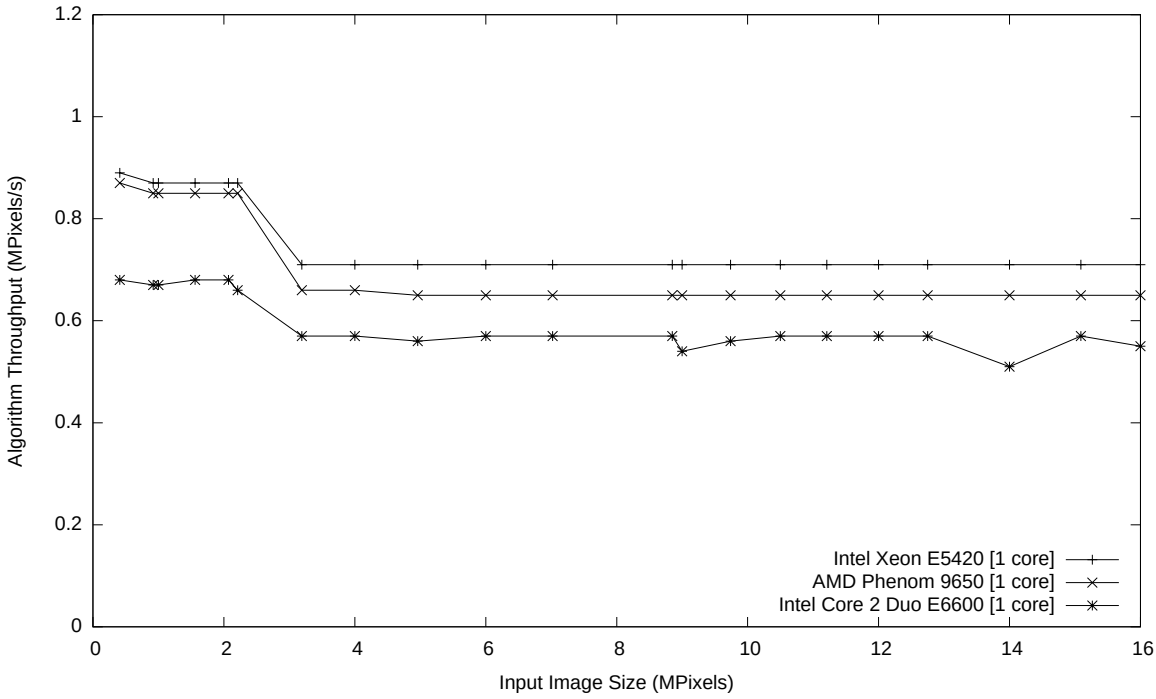
MPixels	Dimensions	Reason	MPixels	Dimensions	Reason
0.41	720x576	SD PAL	8.85	4096x2160	HD (4K, Sony)
0.92	1280x720	HD (720p)	9.00	3000x3000	Near Half Pow2
1.00	1000x1000	Near Pow2	9.74	3656x2664	HD (4K, Acad)
1.56	1440x1080	Sony HDCAM	10.50	3500x3000	Fill Gap
2.07	1920x1080	HD (1080p, HDTV)	11.21	3449x3251	Prime
2.21	2048x1080	HD (2K, 1.85:1)	12.00	4000x3000	Near Pow2
3.19	2048x1556	HD (2K, 1.33:1)	12.75	4096x3112	HD (4K, 1.33:1)
4.00	2000x2000	Near Pow2	14.00	4000x3500	Fill Gap
4.96	2293x2161	Prime	15.09	4639x3253	Prime
6.00	3000x2000	Near Half Pow2	16.00	4000x4000	Fill Gap
7.02	4096x1714	HD (4K, 2.39:1)			

Table 3.2: Image resolutions tested in the benchmarks throughout this thesis to identify cache spills and row-to-row memory aliasing effects.

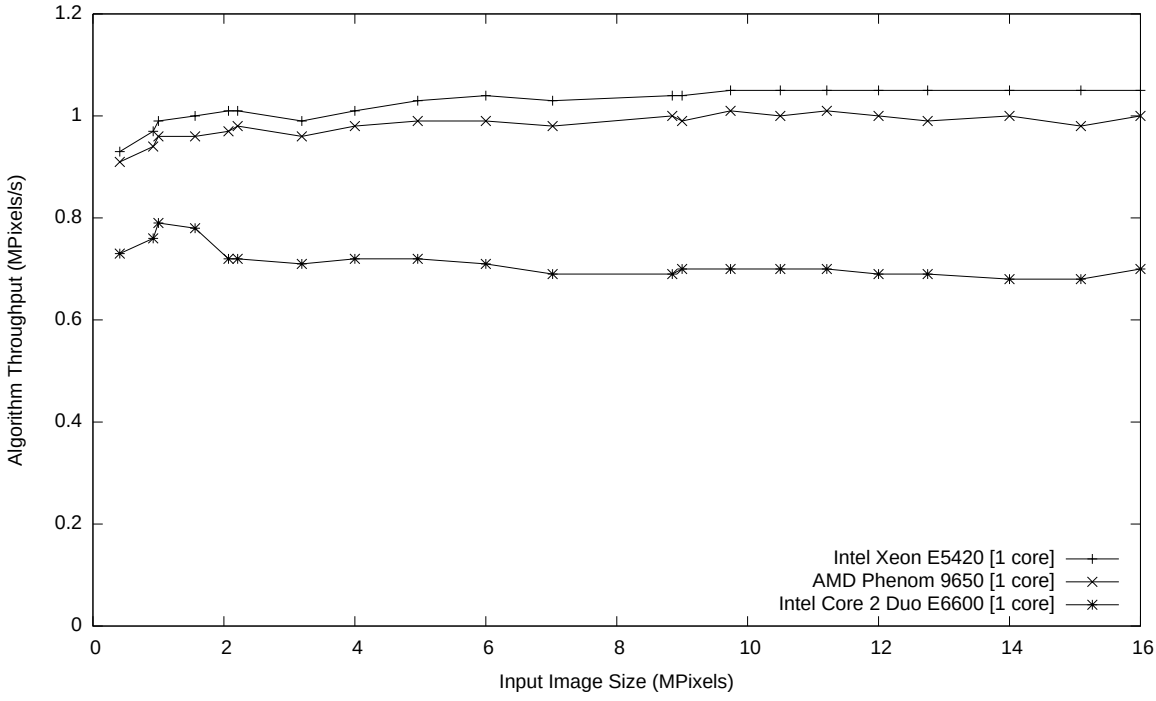
8.10 64-bit. Software benchmarks were compiled with the Intel C/C++ 11.0 compiler with flags *-xHost -O3 -ansi-alias -restrict -openmp -no-prec-div*. OpenMP [DM98] was used to exploit data parallelism and to control the number of cores in use; we confirmed through experimentation that as threading was scaled the most favourable CPU cores (i.e. those in least contention for shared resources) were used first. For all data points there was sufficient free RAM to avoid paging to disk, which would otherwise inflate our performance gains in unrealistic circumstances. The bottleneck in VFX processing is CPU and memory throughput: capacity is trivial to scale.

The results of our first performance experiments are summarised in Figures 3.6a and 3.6b. We began by exploring the throughput of wavelet-based degraining and diffusion filtering under direct execution of the visual primitive representation with no code generation. Furthermore, we restricted our study to a single core of each processor. Throughput on the Y axis is measured in millions of output pixels generated per second for the complete effect. On the X axis we vary the size of the input image (and correspondingly the output image) and summarise this by recording the total number of pixels. Because the width of the image can lead to performance degradation through row-to-row memory aliasing, particularly near powers of two, we tested a number of realistic and interesting image resolutions. These are summarised in Table 3.2. Furthermore, we were careful to select image data sets with features appropriate to each effect in order to accommodate data-dependent performance characteristics. The performance of the diffusion filtering effect changes substantially with degrees of brightness in the image. Thus, we ensured that the image data tested had realistic brightness levels for the application of this effect.

The first feature of interest in these results is the consistent ranking of CPU performance between each effect. The Xeon E5420 and Core 2 Duo E6600 led both experiments in throughput with only a small disparity between them; likely due to the similar clock rates (2.5GHz vs 2.4GHz). The Phenom lagged slightly in wavelet-based degraining but substantially in diffusion filtering. The diffusion filtering effect contains a cache intensive primitive and we speculate that it spills the cache more often due to the smaller capacity in the Phenom: 2.5MB vs 4MB and 12MB on the Intel processors.



(a) Wavelet-based degraining



(b) Diffusion filtering

Figure 3.6: Throughput of the wavelet-based degraining and diffusion filtering visual effects in straightforward execution of the object-oriented representation discussed in Section 3.4.1 on a *single core* of each of three benchmarking platforms. Y-axis throughput measures the number of output pixels generated per second with variation of the input image size on the X-axis.

Throughput is quite stable for all image sizes, although wavelet-based degraining exhibits an interesting spike in performance with small images. Frequent exchange of images between its many primitives is likely to benefit from smaller images fitting within the cache, although this argument is not well supported by the data: all CPUs see a drop in performance between 2 MPixels (∼24MiB) and 3 MPixels (∼36MiB). Both of these sizes are too large for the caches of any CPU tested and we would expect to see the Xeon benefit for longer than the Phenom, due to their different cache capacities.

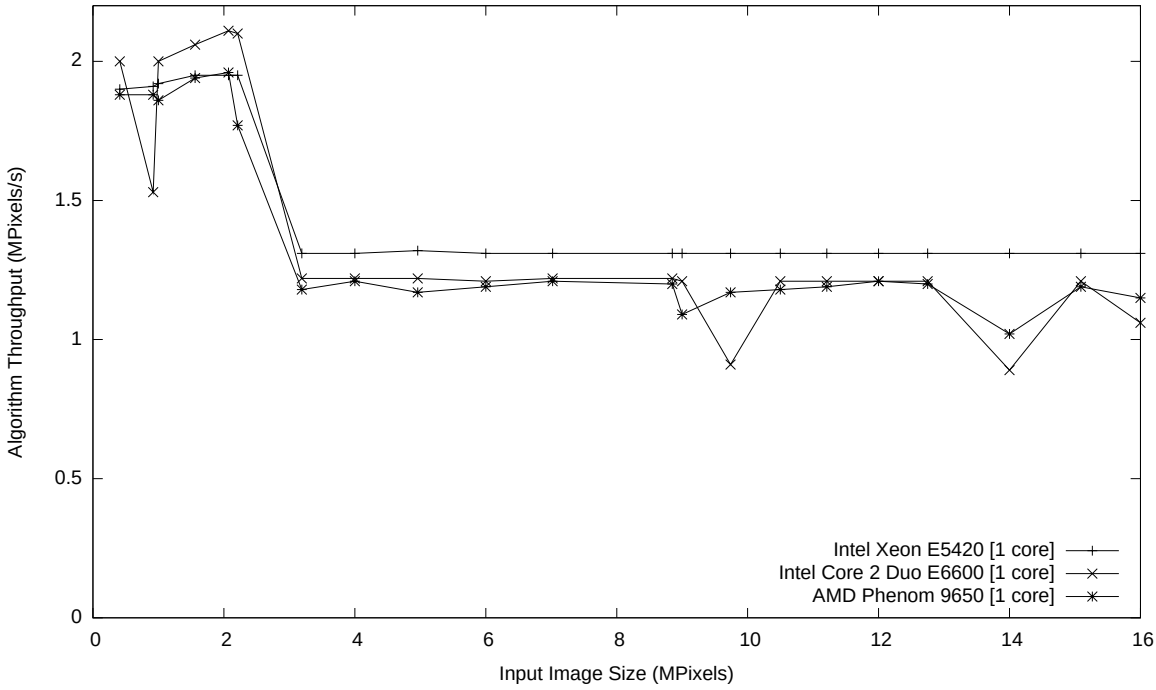
Consider now the performance of generated and statically specialised code for the same visual effects, shown in Figures 3.7a and 3.7b. Relative CPU performance rankings remain the same but the Phenom is now much closer to the Intel CPUs' throughput. Our speculation about the Phenom's performance degradation through cache spilling is not supported by this new set of data. The effect has been removed by the transformations described in Section 3.6 or the static specialisation optimisation discussed in Section 3.6.1.

The throughput spike with small images has become more pronounced: there is now an approximate 1.5x difference compared with 1.2x before. New effects have appeared with image sizes of 0.9 MPixels (but notably not at 1.0 MPixels), 9-10 MPixels and 14 MPixels. The Core 2 Duo and Phenom both experience small drops in throughput, curiously at similar image sizes, whilst the Core 2 Duo also experiences a similarly proportioned drop at 0.9 MPixels. At all of these sizes the Xeon's throughput remains very stable. The widths involved are 1280 (but not 1000), 3000, 3656 and 4000. This data interestingly appears to rule out power-of-two row-to-row aliasing effects. The similarity between these effects on the Phenom and Core 2 Duo – but not the Xeon – would also make this explanation unlikely. We offer no further explanation but propose a future profiling experiment to isolate the causes of this degraded performance.

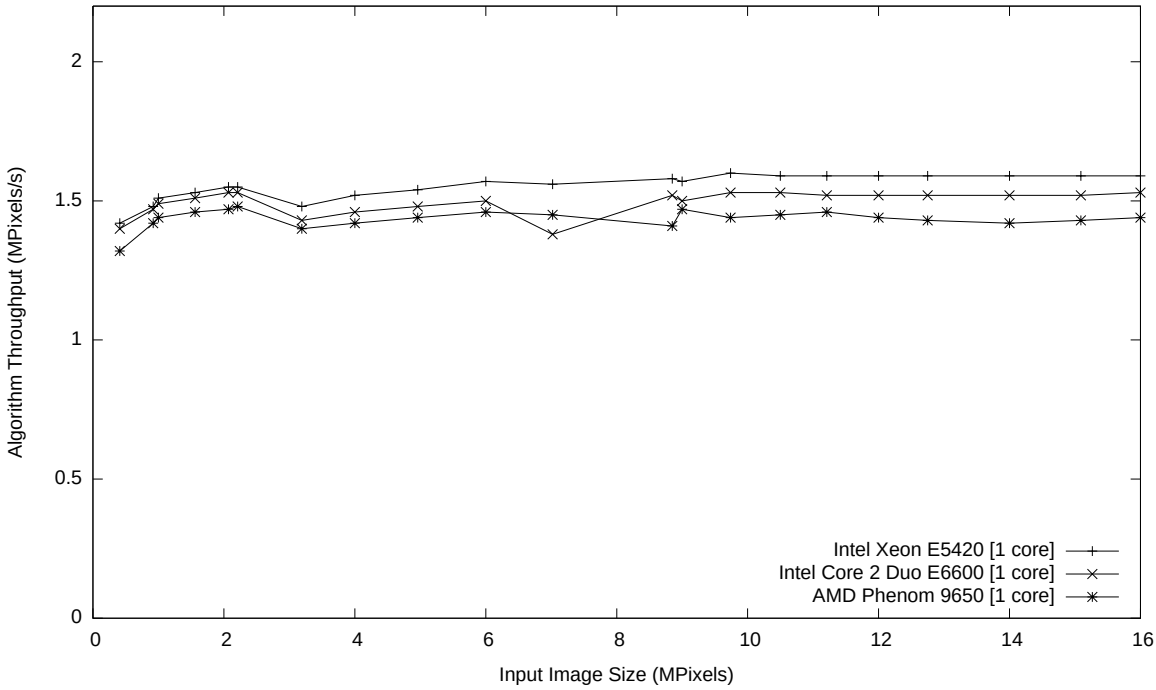
A comparison of throughput before and after code generation and static specialisation is presented in Figure 3.8. The Y axis measures the speed-up (throughput after ÷ throughput before) for each both effect on all three benchmarking platforms. We fix the image size to an unremarkable data point from the previous four graphs, at 12 MPixels. The gains made from code flattening and static specialisation range between 1.4x and 2.2x, with an average of 1.8x. We have introduced no performance degradation and made substantial gains through a series of processes which our vendor compiler does not exploit; using only static syntactic translation and information from runtime delayed evaluation. In subsequent chapters we harness metadata to achieve even larger performance gains on top of those which we have attained so far.

We now take our best optimised implementations and reintroduce multicore scalability to assess their parallel throughput on each benchmarking system. Figures 3.9a and 3.9b graph the speed-up (throughput after ÷ throughput before) over a single core on the Y axis against the number of CPU cores used on the X axis. As stated earlier in this section, we confirmed through experimentation that OpenMP makes use of the subset of cores with minimal resource contention before adding those which contend more heavily with the set. We again make use of the 12 MPixel image size to isolate outlying performance effects identified in previous graphs.

Optimum scalability follows the line Y=X on each graph. Neither effect attains this ideal, with



(a) Wavelet-based degraining



(b) Diffusion filtering

Figure 3.7: Throughput of the wavelet-based degraining and diffusion filtering visual effects in execution of flattened, statically-specialised generated code (from the object-oriented representation) on a *single core* of each of three benchmarking platforms. Y-axis throughput measures the number of output pixels generated per second with variation of the input image size on the X-axis.

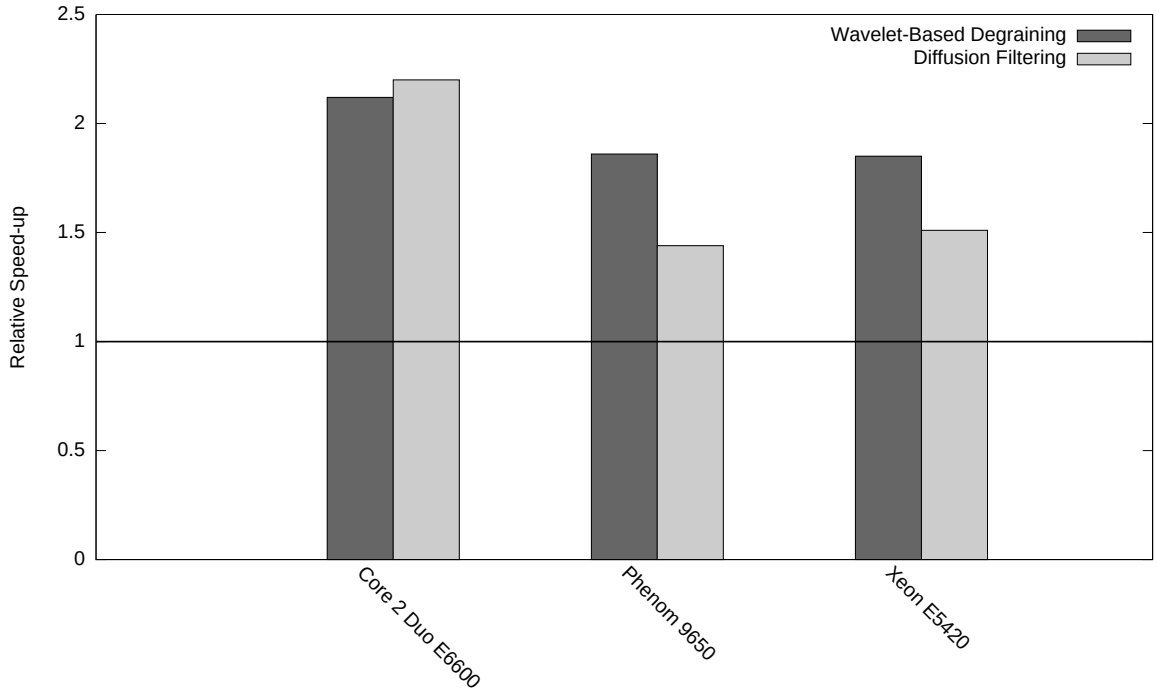
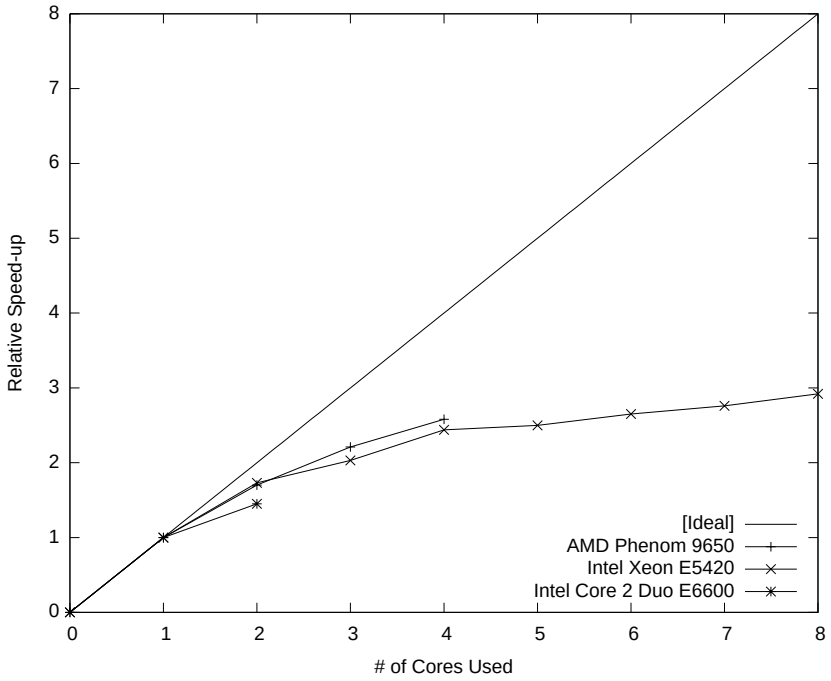


Figure 3.8: Relative speed-ups (1x = no speed-up) of the wavelet-based degraining and diffusion filtering effects on a *single core* of each benchmarking platforms for a fixed input image size of 12 MPixels, following code generation and static specialisation (Section 3.6) from the object-oriented representation. Speed-ups are attributable to a combination of simplified code analysis and optimisations enabled by the presence of specialised values, such as loop unrolling.

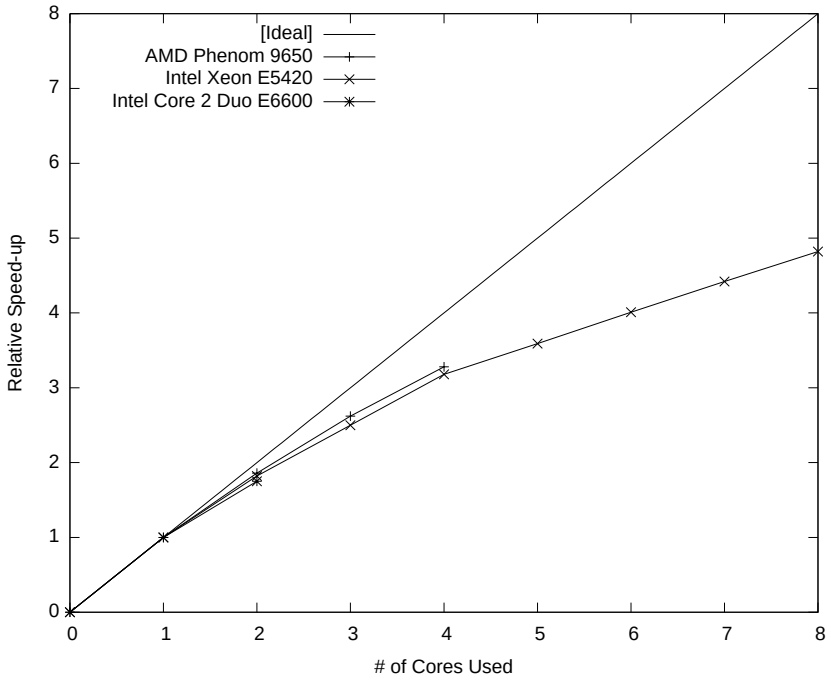
all systems experiencing reductions in improvement as successive cores are added. On our most scalable platform, the 8-core Xeon E5420, speed-ups of only 2.9x for wavelet-based degraining and 4.8x for diffusion filtering are achieved, compared to an ideal speed-up of 8x. As we will demonstrate in Chapter 4, and describe optimisations to improve it, this poor scalability is mainly the result of contention for cache and memory bus resources. Notice in particular how the Xeon's scalability experiences a visible slowdown after 4 cores, where the 5th, 6th, 7th and 8th cores contend with the 1st, 2nd, 3rd and 4th cores respectively for the shared L2 cache.

We observe that scalability is worse in wavelet-based degraining than in diffusion filtering. There is no conclusive explanation for this, but it is likely that the more heavily networked primitives in wavelet-based degraining are bottlenecked by receiving and sending images from/to other primitives. The diffusion filtering effect, on the other hand, has a smaller number of primitives and is dominated by one with significant computational and cache intensity. This argument is further supported by the failure of schedule optimisation in Chapter 4 on this effect.

To conclude our performance study for this chapter we present the per-primitive throughput breakdown of each visual effect. Figures 3.10a and 3.10b take our best optimised versions so far – with code generation, static specialisation and full multicore parallelism – and measure the fine-grained execution time for each primitive constituting the effect. We do this to illustrate the different constructions of each effect and to form a basis for optimisation in the next chapter. Wavelet-based degraining is formed from 37 primitives of 4 different types: Listing 3.3 showed

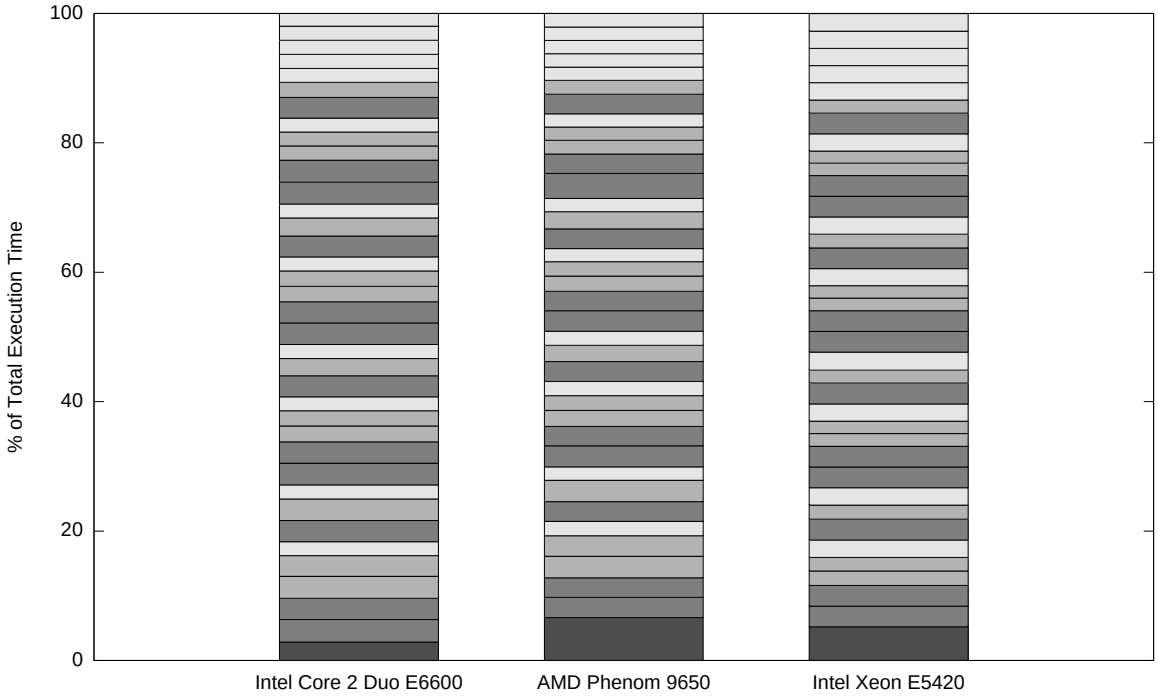


(a) Wavelet-based degraining

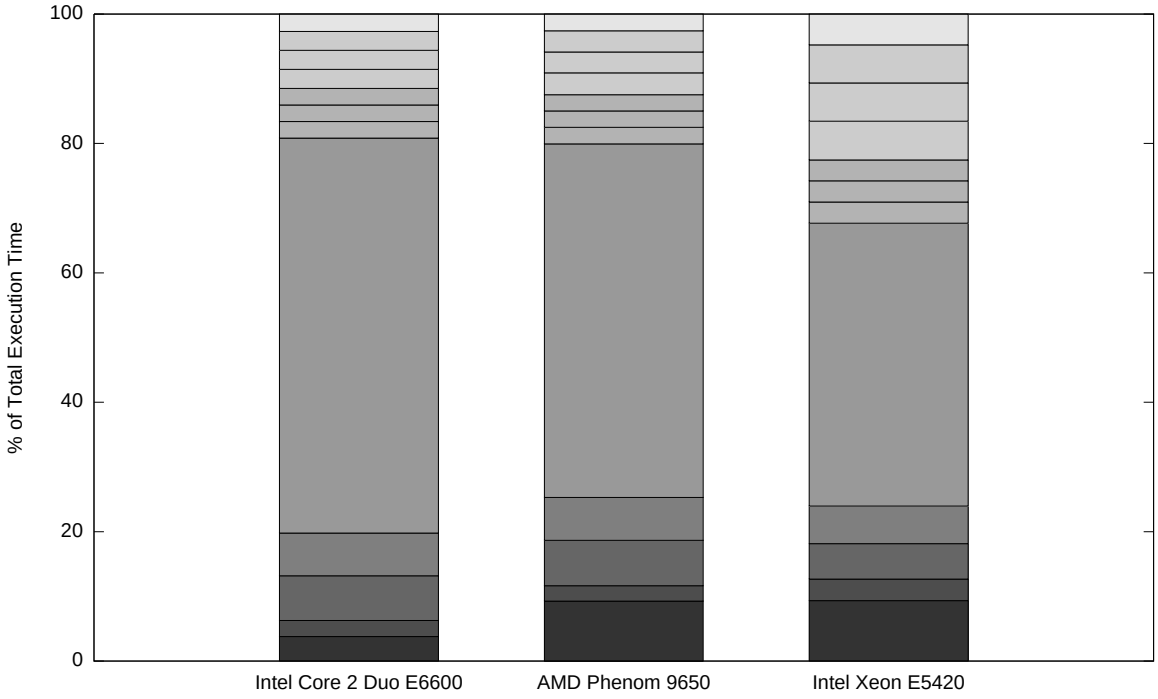


(b) Diffusion filtering

Figure 3.9: Relative speed-ups (1x = no-speed-up) over single-core throughput of the wavelet-based degraining and diffusion filtering visual effects for a fixed input image size of 12 MPixels, in execution of the statically-specialised generated code as the number of active cores is scaled to the processor limit. Cache and memory resource contention leads to sublinear scalability.



(a) Wavelet-based degraining



(b) Diffusion filtering

Figure 3.10: Breakdown of the per-primitive contribution to execution time of the wavelet-based degraining and diffusion filtering visual effects on all cores of the statically-specialised generated code for a fixed input image size of 12 MPixels. These serialisations are space-optimal and derived through exhaustive search.

earlier the complete construction, with the fourth primitive growing input image data as described in Section 3.5.2. Diffusion filtering is built from 12 primitives of 8 different types. The specific construction cannot be disclosed here but the effect is dominated by a large 2D spatial filter.

Two very different optimisation challenges are apparent here. Wavelet-based degraining is not dominated by any one primitive and needs a inter-primitive optimisation, which we describe in Chapter 4, and intra-primitive optimisations, discussed in Chapter 5, to maximise its throughput. The performance of diffusion filtering could be improved substantially by optimising the dominating primitive, which accounts for 45–60% of the total execution time. We demonstrate a different optimisation approach which increases the hardware resources available to the algorithm, using a GPU, in Chapter 6.

3.8 Concluding Remarks

In this chapter, we described a domain-specific C++ framework for developing VFX applications from reusable metadata-annotated primitives. Our key innovation is to abstract information about data dependence and memory access patterns from the algorithm and expose this information as explicit annotations in the primitive representation. Our source-to-source compiler collects these annotations and, as we will demonstrate in the coming chapters, exploits the information contained within as a substitute for complex program analyses. We described a simple parallel scheduling strategy for multicore CPUs which leverages data parallelism in the primitives constituting a visual effect. A space-optimal schedule is desirable but we were only able to offer an intractable solution of exponential complexity; albeit sufficiently fast for the 37 primitive effect described. We further described a scalar CPU code generation method with a simple static specialisation optimisation to improve the availability of program information to the compiler.

We observed 1.4–2.2x speed-ups from code generation and static specialisation over the object-oriented form. The scalability of each effect on multicore CPUs was analysed and determined to be reasonable up to 2–4 cores depending on the effect. Scaling beyond 4 cores gave comparatively poor improvements in throughput due to saturation of parts of the memory system.

Chapter 4

Space and Schedule Optimisation

4.1 Introduction

In Chapter 3 we laid the groundwork for a metadata-supported program optimisation framework. The performance gains documented in Section 3.7 were obtained without the use of metadata. In this chapter we present our first set of code optimisations which make use of the metadata encapsulated in the visual effect expression in order to simplify program analysis. We leverage the isolation of iteration control from data access, implicit in our framework design, to study two connected performance transformations: schedule and space optimisation. The former is implemented through a combination of established transformations [BGS94]: loop shifting, loop peeling, loop fusion and loop interchange. The latter is an advancement of the array contraction (also known as memory reduction [SXWL01] or buffering) transformation. The material presented in this chapter is based upon a development of previously published work [CKPN07].

The optimisations discussed in this chapter primarily target the wavelet-based degraining effect. Recall from Figures 3.2 and 3.10a in the preceding chapter that the degraining effect is constructed from a network of 37 smaller primitives. Each primitive consists of a nested set of loops iterating over the pixels of input images, some of which are shared with other primitives, concurrently producing new pixels in output images. As data is passed from one primitive to the next it is unlikely to reside in any cache level, as most images are substantially larger than contemporary CPU caches. Thus, there is a performance cost in the loop control logic itself and in the bottlenecked main memory bus. Schedule optimisation tackles the former while space optimisation tackles the latter. The diffusion filtering effect, on the other hand, is built from only 12 primitives and dominated by one, as can be seen in Figure 3.10b. We do not expect to make significant gains on this effect in this chapter. The techniques we develop here are suited to large networks of low-cost primitives: a common pattern in VFX.

4.2 Schedule Optimisation

Consider the loop nests constituting a connected series of visual primitives. The default loop schedule executes all iterations of the first primitive before the second, and then those in the

second before the third, and so on. By manipulating the schedule of the collective loop nests in a visual effect, we might hope to:

- **Reduce loop instruction control costs** by sharing a single loop fragment among many primitives that are iterating over a common, large image subregion. Most primitives in a visual effect will have a large region of iteration space in common.
- **Improve temporal data locality** by rescheduling statements which use data elements to be closer to those that produce them. The default schedule separates producers and consumers by at least one completion iteration across a large image, by which point the produced data will have been flushed from the cache.

Furthermore, in Section 4.2.5 we explain how specific schedule manipulations enable the space optimisation discussed in Section 4.3.

Simple manipulations of loop schedules are traditionally carried out in a high-level form: the GCC compiler, for example, builds a structured representation of single-latch natural loops [ASU86, GCC09] and manipulates them through a series of independent transformations. This leads to a phase-ordering problem and limits the kinds of optimisations which can be achieved, leading to conservative schedule optimisation. Recent developments have focused upon the polyhedral loop representation [BCG+03, CGT04, CSG+05] as a tool to rephrase the schedule optimisation problem as a single, composite transformation guided by performance heuristics. Even with the ability to manipulate the loop schedule in this way, constructing accurate performance heuristics and making data dependence guarantees about unknown code is very difficult and constrains the optimisation capabilities of modern compilers. For example, both GCC and the Intel C/C++ compilers fail to perform beneficial loop fusions [BGS94] between series of loop nests in the generated code from Chapter 3.

Our contribution is an improvement upon polyhedral loop manipulation by leveraging metadata for dependence and performance information. We implement loop transformations in our source-to-source code generation framework and supply source code with pre-optimised loop schedules to the Intel C/C++ compiler for low-level optimisation. We first demonstrate how to construct a polyhedral representation of a visual effect and then discuss a scanning strategy to achieve the outlined performance goals.

4.2.1 Constructing the Constraint Matrices

The CLooG library [Bas04] underpins our loop representation, manipulation and code generation phases. In its finest grain of control, CLooG allows each statement of a program to be assigned a unique iteration space defined by a matrix of *constraints* upon the iteration variables. Since most of the statements in a visual effect can be grouped together within a common iteration space – i.e. those from the same visual primitive – we instead treat whole kernel bodies as CLooG statements. Thus our atomic program unit is the kernel. We do not perform loop transformations which would break a kernel into smaller pieces; e.g. loop distribution [BGS94]. Furthermore, any

```
for(int y = y1; y <= y2; ++ y)
  for(int x = x1 - r1; x <= x2 + r1; ++ x)
    Produce data at (x,y)

for(int c = 0; c <= 2; ++ c)
  for(int y = y1; y <= y2; ++ y)
    for(int x = x1; x <= x2; ++ x)
      Consume data in (x - r1...+r1, y) locality
```

Listing 4.1: Example loop nests for a visual effect consisting of two primitives, the latter of which contains a horizontal 1D spatial filter indexer. The first loop nest produces additional data in the radial region ($x1 - r1 \rightarrow x2 + r2$) for the second to consume.

loops contained within the kernel are neither identified nor modified by our schedule optimisation process. Since there is no metadata associated with these loops we would need to duplicate the analyses of a traditional compiler in order to manipulate them. There would be no advantage in doing this at a pre-compiler stage.

To illustrate the construction of constraint matrices, consider a visual effect consisting of the two primitives whose iteration spaces are outlined in Listing 4.1. The second primitive has a spatial filter indexer to make the two loop nests distinct: the first primitive must produce additional data in the radial region for the second primitive to consume. Furthermore, the first primitive consumes whole pixels at a time (its indexers have ePixel metadata) while the second accesses each plane independently (eComponent metadata). x and y are variable parameters to the constraint matrix: they define the iteration domain. $x1$, $x2$, $y1$, $y2$ and $r1$ are constant parameters to the constraint matrix: they allow the constraint matrix to adapt to externally variable – but constant during iteration – parameters, such as image dimensions and spatial filter radii.

Table 4.1 shows the constraint matrices which our framework would construct for this effect. The loop nests of each primitive are defined by a pair of bounds upon the iteration variables for each dimension of the loop nest. These bounds define a convex polytope in a common space with the other loop nests of the visual effect. Notice that we have used a loop interchange [BGS94] transformation to place the component loop at the deepest nesting level. This is necessary to align the common dimensions of each polytope and the safety of the transformation is guaranteed by eComponent metadata, which asserts the lack of data dependence between statement instances processing different image planes. Superposition of these polytopes within this common space is the process by which loop control overheads can be reduced and temporal data locality improved, as we will discuss in detail in Section 4.2.2.

Each line of the constraint matrices forms the coefficients to an equation of the form $ax + by + cz + ... + n \geq 0$. Equality with zero is indicated by a zero in the first column, but in this example we only use inequality. Algebraic equivalents of each matrix row are provided in the adjacent table for convenience. There is a direct correspondence between rows of the constraint matrix and bounds of the two loop nests. The second matrix has an additional column and two additional rows for the component loop, as required by eComponent metadata, giving its corresponding polytope an extra dimension.

$=/\geq$	y	x	$x1$	$x2$	$y1$	$y2$	$r1$	1	constraint
1	1	0	0	0	-1	0	0	0	$y \geq y1$
1	-1	0	0	0	0	1	0	0	$y \leq y2$
1	0	1	-1	0	0	0	1	0	$x \geq x1 - r1$
1	0	-1	0	1	0	0	1	0	$x \leq x2 + r1$

$=/\geq$	y	x	c	$x1$	$x2$	$y1$	$y2$	$r1$	1	constraint
1	1	0	0	0	0	-1	0	0	0	$y \geq y1$
1	-1	0	0	0	0	0	1	0	0	$y \leq ytex2$
1	0	1	0	-1	0	0	0	0	0	$x \geq x1$
1	0	-1	0	0	1	0	0	0	0	$x \leq x2$
1	0	0	1	0	0	0	0	0	0	$c \geq 0$
1	0	0	-1	0	0	0	0	0	2	$c \leq 2$

Table 4.1: Constraint matrices for the loop nests shown in Listing 4.1. Each loop nest has one constraint matrix, bounding its multidimensional iteration space. Columns representing the coefficients of loop control variables are separated from the coefficients of parameters to the polytope.

Constructing these matrices programmatically requires an additional step which we have not yet discussed. While it would be trivial interrogate a primitive at runtime for its iteration bounds, we would not be able to derive the construction of this value from the parameters shared by all loop nests. One solution might be to assign a unique parameter to each loop bound, for which we know the corresponding value at runtime, but this quickly leads to an intractable code generation problem for all but the simplest loop nests. Instead, we record the algebraic construction of iteration bounds from image dimensions ($x1...x2$, $y1...y2$), spatial filter radii ($r1$) and integers (not shown here but required by moving averages) during the DOD propagation algorithm discussed in Section 3.5.2. Loop bound expressions are progressively constructed as the ROIs of each primitive contribute radii and integers to the iteration space, in a modified version of the propagation algorithm from Listing 3.5.

From these matrices CLooG is able to reconstruct an abstract syntax tree representing the original loop nests, through a process called polyhedral scanning. We then translate the AST directly into code. Before attempting to generate an optimised loop schedule from these matrices, we demonstrate some methods for reducing code generation complexity which, as we will show, are key to making the polyhedral scanning problem tractable.

4.2.2 Minimising Polyhedral Scanning Complexity

CLooG accepts a second matrix, called the *context matrix*, which is common to all of the constraint matrices. Its sole purpose is to reduce the complexity of generated code by restricting the values of variable parameters to the constraint matrices; or, more accurately, communicating existing restrictions on their values to the polyhedral scanner. Table 4.2 shows the context matrix which we would generate for the example shown in Listing 4.1.

The first row of the matrix asserts that the image is substantially larger in size than the radial parameters to the effect, and that the rightmost bound is greater than the left. The second row

$=/\geq$	$x1$	$x2$	$y1$	$y2$	$r1$	1	context
1	-1	1	0	0	-1	0	$x1+r1 \leq x2$
1	0	0	-1	1	-1	0	$y1 \leq y2$
1	0	0	0	0	1	-1	$r1 \geq 1$

Table 4.2: Global context matrix used in all loop nest generations. Each matrix row defines a relation between parameters to the constraint matrices and integers.

similarly constrains the vertical edge parameters. This allows the polyhedral scanner to omit many fragmented, conditional paths which handle radii that are larger than the image dimensions and dimensionally inverted images (which are disallowed in our framework). When the former constraint is not met we do not use the optimised code path. Since the images are likely to be very small if the radii are large in comparison, the performance impact is negligible. The third row disallows zero-sized and negative spatial filter radii, which are of no use to us, to omit further conditional paths.

We can further reduce polyhedral scanning complexity by exploiting redundancy in the spatial filter radii of commonly used visual primitives. For example, the horizontal and vertical phases of a separable 2D spatial filter may share the same radii if it is square in shape. Another example is the series of box filters that make up a Gaussian filter approximation, which share the same radii between different primitives. Consider the loop nests of the 2D Gaussian approximation shown in Figure 4.1a. The first three loop nests belong to horizontal box filters while the last three form the vertical passes. The DOD propagation algorithm (Section 3.5.2) grows the bounds of each loop nest, starting from the end of the effect and working backwards, to feed the spatial filter indexers of each primitive with additional border region data.

Each of the constraint matrix parameters $r1$, $r2$, $r3$, $r4$, $r5$ and $r6$ in this example share the same value. With this knowledge in hand we would be able to generate the much simpler loop bound expressions shown in Figure 4.1b. Thus, our framework provides a simple mechanism through which radial *objects*, which wrap the different radial values, can be constructed and passed as parameters to different visual primitives. During construction of the constraint and context matrices we can associate a single matrix variable parameter with multiple loop nests.

The final complexity optimisation we employ was discovered by accident as a side-effect of the symmetricity constraint we placed upon spatial filter indexers. One of the largest sources of combinatorial loop fragment explosion in optimised schedule generation is the number of unique loop bound expressions involved. For example, consider the superposition of the four successive primitives with spatial filter indexers shown in Figure 4.2a. The outer loop nests produce extra data for the inner loop nests to consume in their radial regions. There are a total of 16 different expressions (only 8 are shown for opposite corners) defining the bounds of these loops.

As we will explain in more detail in Section 4.2.3, direct superposition of the polytopes constituting these loop nests results in invalid generated code. The loop shifting [BGS94] transformation is needed to ensure that data dependence is not violated. Figure 4.2b shows the same superposition once loop shifting has been applied. All loop nests, except those of the first, outermost primitive,

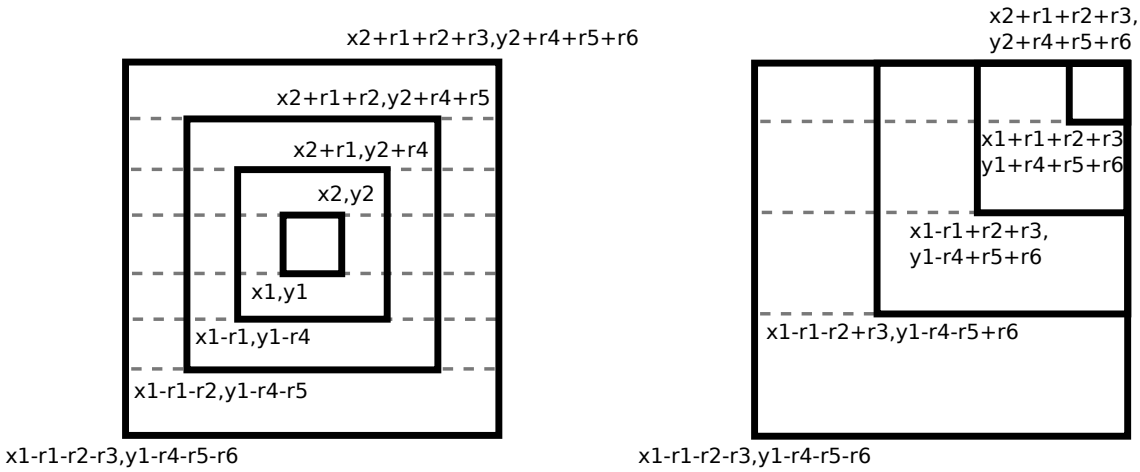
```
// [Producing primitive]
for(y = y1−r1−r2−r3; y <= y2+r1+r2+r3; ++ y)
  for(x = x1−r4−r5−r6; x <= x2+r4+r5+r6; ++ x)
    ...

// Vertical 1D iteration #1.
for(y = y1−r2−r3; y <= y2+r2+r3; ++ y)
  for(x = x1−r4−r5−r6; x <= x2+r4+r5+r6; ++ x)
    ...

// Vertical 1D iteration #2.
for(y = y1−r3; y <= y2+r3; ++ y)
  for(x = x1−r4−r5−r6; x <= x2+r4+r5+r6; ++ x)
    ...

// Vertical 1D iteration #3.
for(y = y1; y <= y2; ++ y)
  for(x = x1−r4−r5−r6; x <= x2+r4+r5+r6; ++ x)
    ...

// Horizontal 1D iteration #1.
for(y = y1; y <= y2; ++ y)
  for(x = x1−r5−r6; x <= x2+r5+r6; ++ x)
    ...

// Horizontal 1D iteration #2.
for(y = y1; y <= y2; ++ y)
  for(x = x1−r6; x <= x2+r6; ++ x)
    ...

// Horizontal 1D iteration #3.
for(y = y1; y <= y2; ++ y)
  for(x = x1; x <= x2; ++ x)
    ...
```

(a) Before radial sharing.

```
// [Producing primitive]
for(y = y1−3∗R; y <= y2+3∗R; ++ y)
  for(x = x1−3∗R; x <= x2+3∗R; ++ x)
    ...

// Vertical 1D iteration #1.
for(y = y1−2∗R; y <= y2+2∗R; ++ y)
  for(x = x1−3∗R; x <= x2+3∗R; ++ x)
    ...

// Vertical 1D iteration #2.
for(y = y1−R; y <= y2+R; ++ y)
  for(x = x1−3∗R; x <= x2+3∗R; ++ x)
    ...

// Vertical 1D iteration #3.
for(y = y1; y <= y2; ++ y)
  for(x = x1−3∗R; x <= x2+3∗R; ++ x)
    ...

// Horizontal 1D iteration #1.
for(y = y1; y <= y2; ++ y)
  for(x = x1−2∗R; x <= x2+2∗R; ++ x)
    ...

// Horizontal 1D iteration #2.
for(y = y1; y <= y2; ++ y)
  for(x = x1−R; x <= x2+R; ++ x)
    ...

// Horizontal 1D iteration #3.
for(y = y1; y <= y2; ++ y)
  for(x = x1; x <= x2; ++ x)
    ...
```

(b) After radial sharing.

Figure 4.1: Reducing loop bound complexity by merging filter radii variables that are known to be identical. This is a critical polyhedral math optimisation in loop fusion.



(a) Without loop shifting.

(b) With loop shifting.

Figure 4.2: Reduction in loop fragment generation following loop shifting of a superposition of spatial filter primitives. Dotted lines indicate necessary breaks in the x and y loops and the enclosed regions represent generated loop fragments. Loop shifting reduces fragmentation from 25 loops to just 10.

$=/\geq$	$p1$	y	x	c	$x1$	$x2$	$y1$	$y2$	$r1$	1
0	0	0	0	0	0	0	0	0	0	0

$=/\geq$	$p1$	y	x	c	$x1$	$x2$	$y1$	$y2$	$r1$	1
0	1	0	0	0	0	0	0	0	0	0

Table 4.3: Scattering matrices for generating an unoptimised loop schedule for the loop nests shown in Listing 4.1.

$=/\geq$	$p1$	y	x	c	$x1$	$x2$	$y1$	$y2$	$r1$	1
0	0	0	0	0	0	0	0	0	0	0

$=/\geq$	$p1$	y	x	c	$x1$	$x2$	$y1$	$y2$	$r1$	1
0	0	0	0	0	0	0	0	0	0	0

Table 4.4: Scattering matrices for generating a fused loop schedule for the loop nests shown in Listing 4.2.

have their bounds shifted by an expression of radial parameters. The second nest has been shifted by $(r3, r6)$, the third by $(r2+r3, r5+r6)$ and the last by $(r1+r2+r3, r4+r5+r6)$. By nature of the symmetricity of our spatial filter design – i.e. the left radial region is the same size as the right, and correspondingly the bottom and top – all loop nests now share a common corner. We have reduced the number of unique loop bound expressions from 16 to 13.

This reduction may not seem significant at first, but consider the number of number of loop fragments this leads to. The dotted lines in Figures 4.2a and 4.2b indicate necessary breaks in the x and y loops. Each enclosed region gives rise to a single loop fragment. We have reduced the number of generated loop fragments from 25 to just 10. In the context of combinatorial explosion this is a valuable tool in maintaining tractability.

4.2.3 Constructing the Scattering Matrices

The final set of matrices accepted by CLooG control its polyhedral scanning facility. The *scattering matrix* defines a temporal ordering on a statement whose iteration space is bounded by one of the constraint matrices. This can be used for a number of purposes [Bas04], and constructed from complex expressions, but we limit our use to explicitly generating unoptimised or fused loop schedules. A scattering matrix for an unoptimised schedule of our example loop nests from Listing 4.1 is shown in Table 4.3. We set the statement of the second loop nest to execute at a later time step (1 vs 0) than the first. This is sufficient to reproduce the loop nest in Listing 4.1 from the complete set of matrices.

However, consider a different loop schedule for the same code shown in Listing 4.2. This is produced by the scattering matrix given in Table 4.4. Both statements now execute at the same time step and the two loop nests of different sizes are fused into a single, fragmented nest. The resulting program is incorrect: clearly it makes no sense to be producing data in the third nested loop that will not be consumed. By performing loop fusion with this schedule we have violated

```
for(int y = y1; y <= y2; ++ y)
  for(int x = x1 - r1; x < x1; ++ x)
    Produce data at (x,y)

  for(int x = x1; x <= x2; ++ x) {
    Produce data at (x,y)
    Consume data in (x -r1...+r1, y) locality
  }

  for(int x = x2 + 1; x <= x2 + r1; ++ x)
    Produce data at (x,y)
}
```

Listing 4.2: A fused schedule for the loop nests originally defined in Listing 4.1. The program is invalid because data dependence was violated during loop fusion.

a data dependence between the two statements and generated an invalid program. This is a well-known problem in schedule optimisation and can be corrected by a loop shifting transformation prior to fusion. Loop shifting adds an offset to the bounds of the second loop so that it executes a few iterations later, when enough data is available for it to consume.

To minimise the temporal reuse distance of produced and consumed data we would like the loop shift factor to be as small as possible. For primitives with no spatial filter indexers this distance is precisely 0: the consumer can collect its consumed data value(s) in a given iteration immediately after they are produced. No shifting is required in this case. For those with spatial filter primitives the optimal shift distance is a function of the radii and ROIs of their spatial filter indexers. The shift for each visual primitive is computed by a dominator traversal graph algorithm. This algorithm is outlined in Listing 4.3. Because the incoming edges to a primitive with multiple inputs may have accumulated different shifts up until that point, the shift for that primitive becomes the union of all shifts required by its indexer objects. This may lead to suboptimal reuse distances for some indexers but, without loop fission, we cannot improve upon this.

Listing 4.4 shows the optimised schedule with loop shifting applied. Because the loop indices have been offset, any use of them in the kernel body must be corrected to compensate. We apply this offset during the code generation process (Section 3.6) when replacing accesses through spatially localised indexer objects with globally addressed array accesses. It is in fact possible to express the loop shifting transformation as a structure in the scattering matrix. We found it easier to use the simple scattering matrix from Table 4.4 and to reformulate the loop shifting transformation as a modification of loop bounds during construction of the constraint matrix.

We further use a combination of the scattering matrices from Tables 4.3 and 4.4 to produce partially optimised schedules in which some loop nests are fused but others are not. The utility of this will become evident in the schedule optimisation discussion in the next section.

4.2.4 Impact of Schedule Optimisation on Parallelisation

Recall from Section 3.5.4 that our parallelisation strategy consists of dividing the DOD of each visual primitive into row-striding strips and generating the data within each strip on a different

```
Map[Node:Functor] LoopShifts

Dominator Walk:
  Evaluate(Node)

function Evaluate(Node) {
  foreach((Edge, InputNode) in Node.InputEdges) {
    if(IsSource(InputNode))
      // No shift needed, all data is available.
      continue

    // Compute the shift required by this indexer.
    Indexer ConsumingIdxr = Node.IndexerFromEdge(Edge)
    Vector LocalShift;

    if((ConsumingIdxr.Freedom == e1D && ConsumingIdxr.Axis == eHorizontal)
      || ConsumingIdxr.Freedom == e2D)
    {
      LocalShift.X = ConsumingIdxr.RadiusX
    }
    if((ConsumingIdxr.Freedom == e1D && ConsumingIdxr.Axis == eVertical)
      || ConsumingIdxr.Freedom == e2D)
    {
      LocalShift.Y = ConsumingIdxr.RadiusY
    }

    // Accumulate the shift along this branch.
    Vector AbsoluteShift = LoopShifts[InputNode] + LocalShift

    // Our shift is the largest shift of all our indexers.
    LoopShifts[Node].Union(AbsoluteShift)
  }
}
```

Listing 4.3: Psuedocode for an automatic loop shift computation algorithm for maximal fusion. Loop shifts accumulate forwards from the source node as spatial filters contribute their radii.

```
for(int y = y1; y <= y2; ++ y)
  for(int x = x1 - r1; x < x1 + r1; ++ x)
    Produce data at (x,y)

  for(int x = x1 + r1; x <= x2 + r1; ++ x) {
    Produce data at (x,y)
    Consume data in ((x - r1) -r1...+r1, y) locality
  }

  for(int x = x2 + r1; x <= x2 + (2 * r1); ++ x)
    Consume data in ((x - r1) -r1...+r1, y) locality
}
```

Listing 4.4: A fused, shifted schedule for the loop nests shown in Listing 4.1. Data dependence is preserved by shifting the second loop with an offset of *r1* before fusing the common fragments.

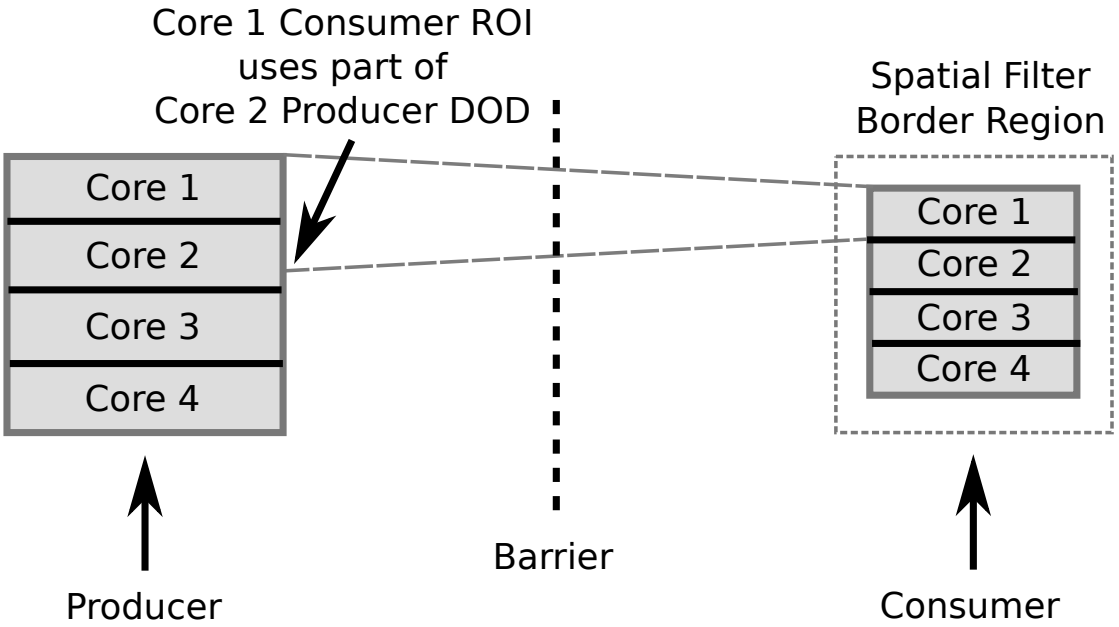


Figure 4.3: A parallelised spatial filter primitive requires partial data produced by a different core in the preceding primitive. Barrier synchronisation ensures that this data is available to each thread. In the optimised schedule it is not possible to use a barrier in this way.

CPU core. This stripwise parallelism strategy is implemented by parallelising the *y* loop of each visual primitive and placing a thread barrier between each loop nest. Two things change following schedule optimisation: the *y* loops may become fragmented and there may no longer be an obvious place to insert a synchronisation barrier.

Fragmentation of the *y* loop necessitates individual parallelisation of each fragment. The overhead of parallelisation becomes proportionally larger as the trip count of the parallelised loops decreases. Figure 4.3 illustrates the barrier synchronisation problem. By rescheduling the statements of the consuming spatial filter kernel to execute closely with those of the producing kernel, threads may try to consume data which has not been produced by other threads yet; in the unoptimised schedule a barrier between the producer and consumer loop nests guarantees that this data is available.

Our solution is to parallelise the composite fused primitive as a whole. The DOD of this fused primitive is divided into strips as per our normal parallelisation strategy. To overcome the barrier synchronisation problem, we allow each thread to do additional work in preceding primitives in order to construct the extra border region data they need in the consuming primitive. This leads to redundant computation between threads but greatly simplifies the parallelisation problem. The level of redundancy is proportional to the number of fused spatial filter primitives with a vertical border region (i.e. 1D vertical or 2D) and inversely proportional to the height of the image. Since the fusion of 1D vertical and 2D spatial filter primitives leads to a large temporal reuse distance, our parallelisation strategy further reduces the gains that can be made by doing so.

4.2.5 Optimising the Polyhedral Schedule

Sections 4.2.1, 4.2.2 and 4.2.3 outlined the mechanics of our schedule optimisation strategy. We now consider the application of this technique to whole visual effects and its implications for their computational performance. Figure 4.4 illustrates the loop fusion and loop shifting process for the wavelet-based degraining [SCW05] visual effect. We take the 37 primitives constituting the effect and choose a space-optimal serialisation, as described in Section 3.5.1. Figure 4.4a graphs the X and Y iteration spaces (C is not shown nor involved here) of each primitive, beginning with the start of the serialisation at the front. Variations in the size of the iteration space are caused by the larger ROIs of spatial filter indexers and by subtrees of the DAG having different DOD requirements. The scales of the image bounds and spatial filter radii have been exaggerated for illustration by using an image size of 50x50 pixels. In practice the image dimensions are much larger than the spatial filter radii.

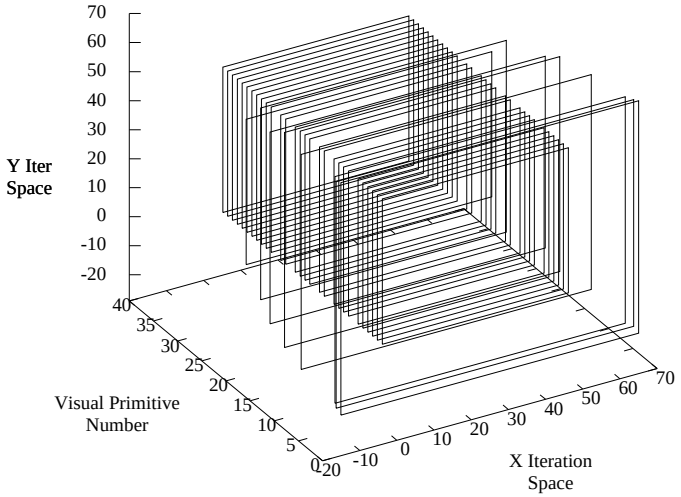
Figure 4.4b illustrates complete schedule fusion without the loop shifting transformation. It is a projection of Figure 4.4a onto the frontmost plane. Different regions of the projected iteration space, which are not fully bounded in this diagram (each horizontal line must be extended to the edges of the plane to see the loop segmentation), correspond to different sets of kernels executing together. The centremost region contains all of the kernels within the effect executing in order of their serialisation. Of course, without loop shifting this schedule leads to an invalid program. Figure 4.4c completes the illustration by showing the fused and shifted iteration spaces for this effect. Observe the tendency for iteration spaces to share the upper-right hand corner, as we discussed in Section 4.2.2. Not all primitives share this corner because some paths within the visual effect DAG have smaller DODs than the others.

The fully fused effect leads to 5303 lines of condition-free generated code and takes approximately 5 seconds for the polyhedral scanning process to complete. Without the context matrix optimisation described in Section 4.2.2 the process generates 43006 lines of heavily conditional code in 40 seconds. A partial fusion, in which we fuse pairs of iterations of the algorithm shown in Listing 3.3, and make use of in our performance analysis, takes 2 seconds and produces 2025 lines of code. We have now completed the description of our schedule optimisation process.

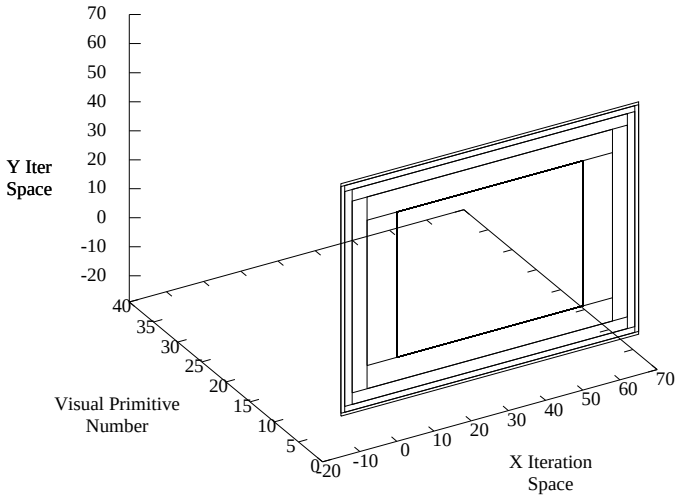
Returning to the goals of schedule optimisation outlined in Section 4.2, we have made progress towards both:

- **Reduce loop instruction control costs**. The constituent loop control functionality of each primitive is now merged into a single loop nest, or few nests with partial fusion. Thus we have reduced the costs of executing similar loop control instructions by visiting them only once. The complexity of our loop bounds may increase slightly, from loop shifting, but code hoisting [BGS94] of these constant bounds in the backend compiler will offset this cost.

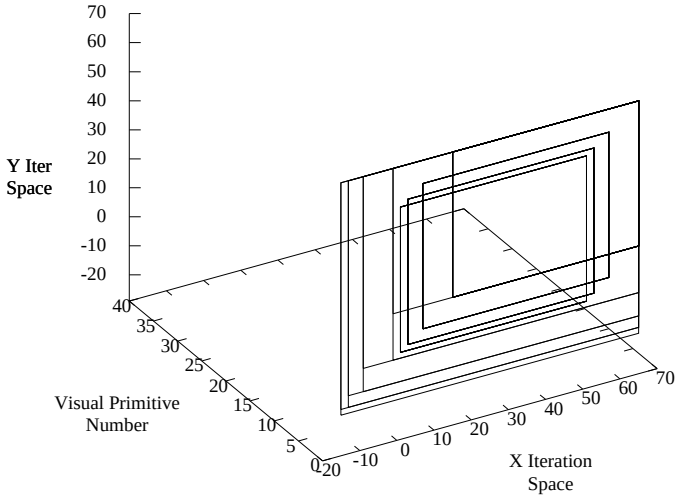
- **Improve temporal data locality**. The producers and consumers of data within the optimised schedule now execute as close in time as data dependence permits. The temporal reuse distance is equivalent to the amount of relative loop shift between a pair of producer/-consumer kernels.



(a) Before loop fusion.



(b) Loop fusion with no shifting.



(c) Loop fusion with shifting.

Figure 4.4: A graphical representation of the 2D iteration spaces constituting a serialisation of the wavelet-based degraining [SCW05] visual effect.

In the next section we build upon these successes with a performance-critical optimisation.

4.3 Space Optimisation

Improving the temporal reuse distance of intermediate data sets is an important optimisation to improve cache utilisation. In practice, as we will show in Section 4.4, we found that other factors were dominating the performance gains from this process. An unfortunate side effect of schedule optimisation is that a larger number of data sets are accessed together in the optimised loop nests, which substantially increases the working set size and consequently cache pressure. We presented the schedule optimisation material in Section 4.2 in full knowledge of this flaw, because the transformations discussed in that section enable a key space optimisation which overcomes these dominating cache effects. This optimisation [SXWL01] – called array contraction, memory reduction or buffering in various literature – aims to reduce the size of memory regions touched by the working set. By reducing this amount to magnitudes close to the cache capacities it is possible to minimise cache spilling, leading to greatly improved utilisation of the cache interconnects and main memory bus.

Listing 4.5 illustrates the utility of space optimisation as an enabling transformation for array contraction. The first set of loop nests produce data into array *a[]*, consume it and then produce output into array *b[]*. *b[]* is considered an output array from this simple operation; *a[]* is not required and is considered a *transient array*. Schedule optimisation, as described in Section 4.2, leads to the second set of loop nests. The temporal reuse distance of the produced/consumed transient array *a[]* is reduced from 100 iterations to just 2. Although this improves cache utilisation, the contents of array *a[]* are still stored and written to main memory despite not being used afterwards. Furthermore, since array *a[]* is large it will occupy multiple cache lines for some time before being evicted, thus reducing cache availability for the working set.

The third set of loop nests in Listing 4.5 illustrate our space optimisation transformation. The transient array *a[]* has been replaced by a new, much smaller array *c[]*. This array is just large enough to carry data across the full reuse distance – two elements – plus a third to carry the current intra-iteration element. Index expressions used in accesses to array *a[]* have been replaced by the same expressions modulus the size of array *c[]*. As the iteration variable *i* counts from 0 to 99, writes into array *c[]* follow a pattern of 0, 1, 2, 0, 1, 2, 0, 1, 2, etc. Each write to the array overwrites a previously produced and consumed element that will not be used again. This process of overwriting unused data corresponds precisely to the temporal reuse distance: array *c[]* is the minimum size required for correctness in this example.

By applying this space optimisation we are able to keep reads and writes of the elements of array *c[]* in the cache; in this example, quite possibly at the L1 level. Since we continuously reuse the same set of cache lines, they are not evicted and do not contend for the main memory bus with accesses to array *b[]*. *b[]* itself could be contracted to a smaller array if a subsequent primitive consumed its elements to produce new data sets. In general, the only data sets which cannot be contracted are those that form inputs to a visual effect – unless, perhaps, the data came from

```
float a[100], b[100];

// Before schedule optimisation.
for(int i = 0; i < 100; ++ i)
  a[i] = i;
for(int i = 1; i < 99; ++ i)
  b[i] = (a[i-1] + a[i] + a[i+1]) / 3.0f;

// After schedule optimisation.
for(int i = 0; i < 2; ++ i)
  a[i] = i;
for(int i = 2; i < 100; ++ i) {
  a[i] = i;
  b[i-1] = (a[i-2] + a[i-1] + a[i]) / 3.0f;

// After space/schedule optimisation.
float c[3];

for(int i = 0; i < 2; ++ i)
  c[i%3] = i;
for(int i = 2; i < 100; ++ i) {
  c[i%3] = i;
  b[i-1] = (c[(i-2)%3] + c[(i-1)%3] + c[i%3]) / 3.0f;
}
```

Listing 4.5: Progressive space and schedule optimisation of the transient array in a loop-fused point primitive and 3-tap horizontal 1D spatial filter.

disk and we could include the disk loading kernel in schedule optimisation – and outputs from it. In some cases the temporal reuse distance will be too large to maintain the contracted array in cache. There is a balance between the gains made from space optimisation with the increased cache pressure from schedule optimisation. Pairs of primitives with small reuse distances (e.g. non-spatial filters and horizontal 1D spatial filters) lead to larger gains than those with large reuse distances (e.g. vertical 1D and 2D spatial filters).

Our space optimisation implementation goes a step further to reduce the extra computational costs incurred in the modulus expressions used in array indexing. Integer modulus is an expensive operation and can be substituted by a much cheaper bitwise AND operator when the contracted array size is a power-of-two. The expression (*i % 4*), for example, is equivalent to (*i & (4-1)*). By padding the transient array to a power-of-two we can make use of this cheaper arithmetic form. Of course, this leads to increased cache pressure and becomes particularly costly with large contracted arrays, where the next power-of-two can be quite far from the optimal reuse distance. Listing 4.6 presents the previous loop nest example with this method.

4.4 Performance Analysis

We now return to the two visual effects whose base performance was established in Chapter 3 to explore the effects of applying schedule and space optimisation to their constituent primitives. As noted at the beginning of this chapter, we expect to see the largest gains in the wavelet-based

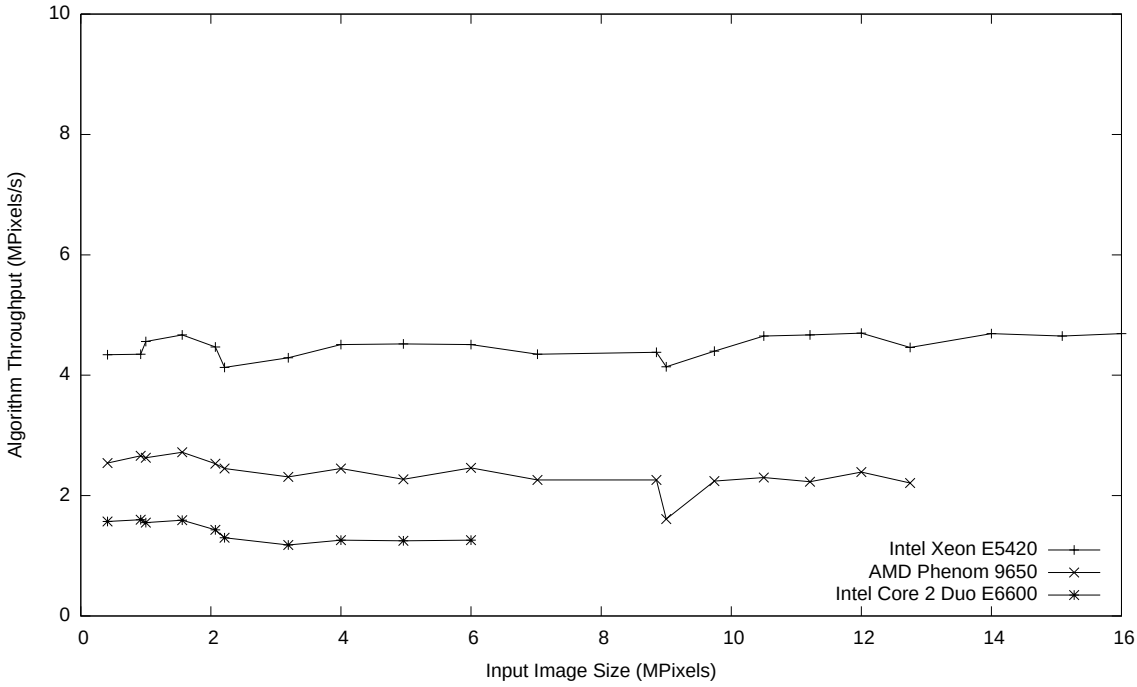
```
float b[100];

// The contracted 'c' array is padded to 4 elements.
// This allows cheap i&(4-1) bitwise index wrapping.
float c[4];

for(int i = 0; i < 2; ++ i)
  c[i&3] = i;
for(int i = 2; i < 100; ++ i) {
  c[i&3] = i;
  b[i-1] = (c[(i-2)&3] + c[(i-1)&3] + c[i&3]) / 3.0f;
}
```

Listing 4.6: Space optimisation with a contracted array padded to a power-of-two to enable cheap bitwise arithmetic in index expressions.

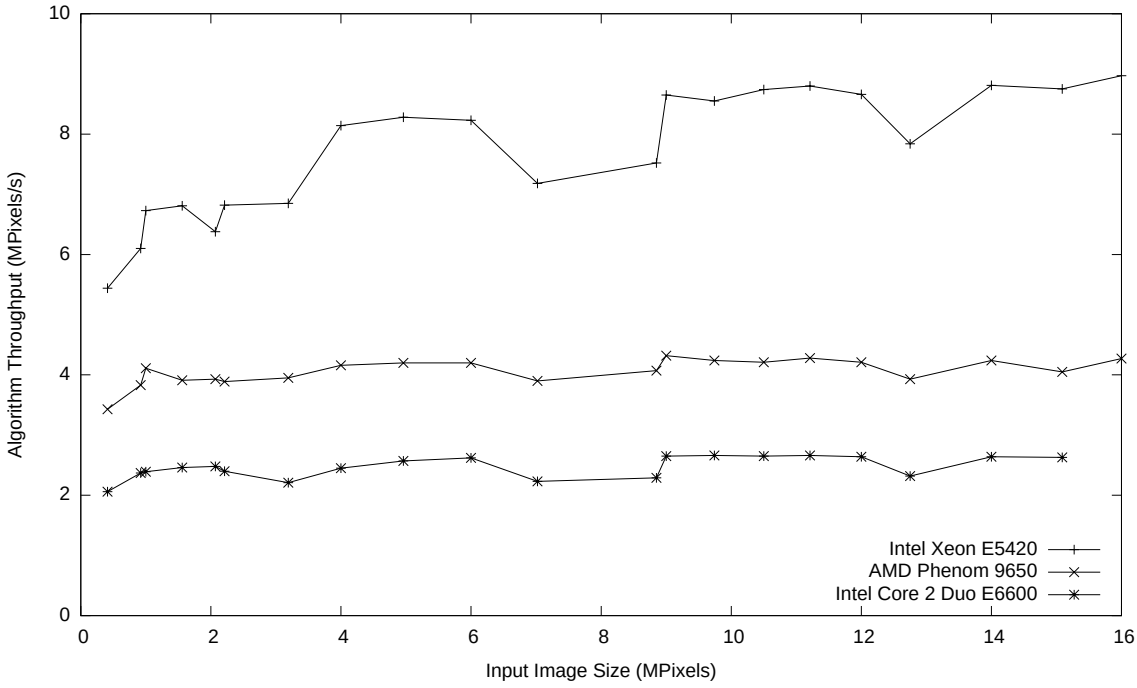
degraining effect, which is built from a large number of simple primitives. The diffusion filtering effect, whose composite performance is broken down in Figure 3.10b, is not dominated by inter-primitive traffic and will not benefit to the same degree.

Figure 4.5a graphs the throughput of the schedule- (but not space-) optimised wavelet-based degraining effect as the input image size is varied. This effect is fully fused: all 37 primitives execute together with optimal data reuse distances. Some results do not extend for the full range of input image sizes. Fusion substantially increases the working set size and this leads to exhaustion of the physical memory (see Table 3.1) available to those benchmarking platforms. We do not graph the effect of disk paging systems as this is beyond the scope of our study and largely irrelevant in practical applications, where cheap additional memory will simply be installed or the problem tiled appropriately in a higher level application. The graph has few effects of significance aside from slightly increased instability over Figure 3.7a as the image size is varied. There is a stable, repeatable drop in throughput for the 3000x3000 pixel image on the Xeon and Phenom platforms. This correlates with a similar but smaller drop in performance in the data without schedule optimisation (Figure 3.7a) on the Phenom platform. We are unable to suggest a cause for this effect and propose inspection with a profiling tool.

Figure 4.5b similarly graphs the throughput of the fully schedule- (but not space-) optimised diffusion filtering effect as the input image size is varied. Most of the data extends for the full range of image sizes because of the smaller number of primitives in this effect, compared to wavelet-based degraining, which leads to a smaller working set inside the fused loop nests. Space/schedule optimisation does not change the contribution of data-dependent performance in this effect because the amount of computation carried out is not affected. We observe an interesting rise in throughput as the image size is varied on the Xeon platform, which is not present on the other two platforms or in the same data without schedule optimisation (Figure 3.7b). We speculate that this is due to the overheads incurred by our schedule-optimised parallelisation strategy, as discussed in Section 4.2.4: the redundant computation in a 12-primitive chain (of which 4 have vertical border regions) with 8 cores is constant with image size, so we see proportionally more "useful" work done in larger images. There is a curious correlation between the small peaks and troughs of



(a) Wavelet-based degraining



(b) Diffusion filtering

Figure 4.5: Throughput of the wavelet-based degraining and diffusion filtering visual effects with maximal fusion (but no space optimisation) on all cores of each of three benchmarking platforms. Y-axis throughput measures the number of output pixels generated per second with variation of the input image size on the X-axis.

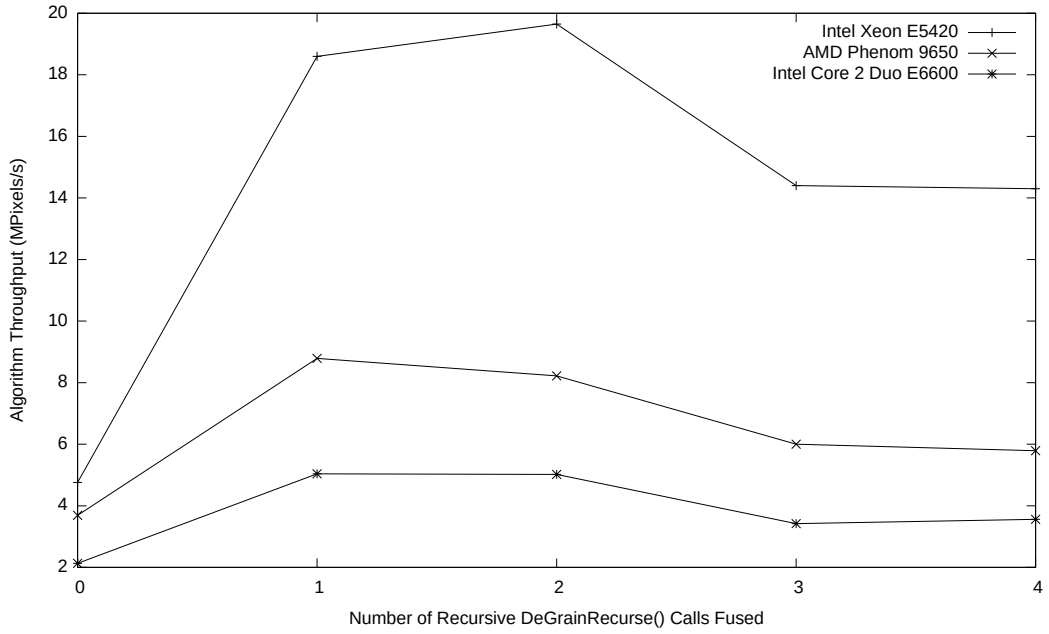
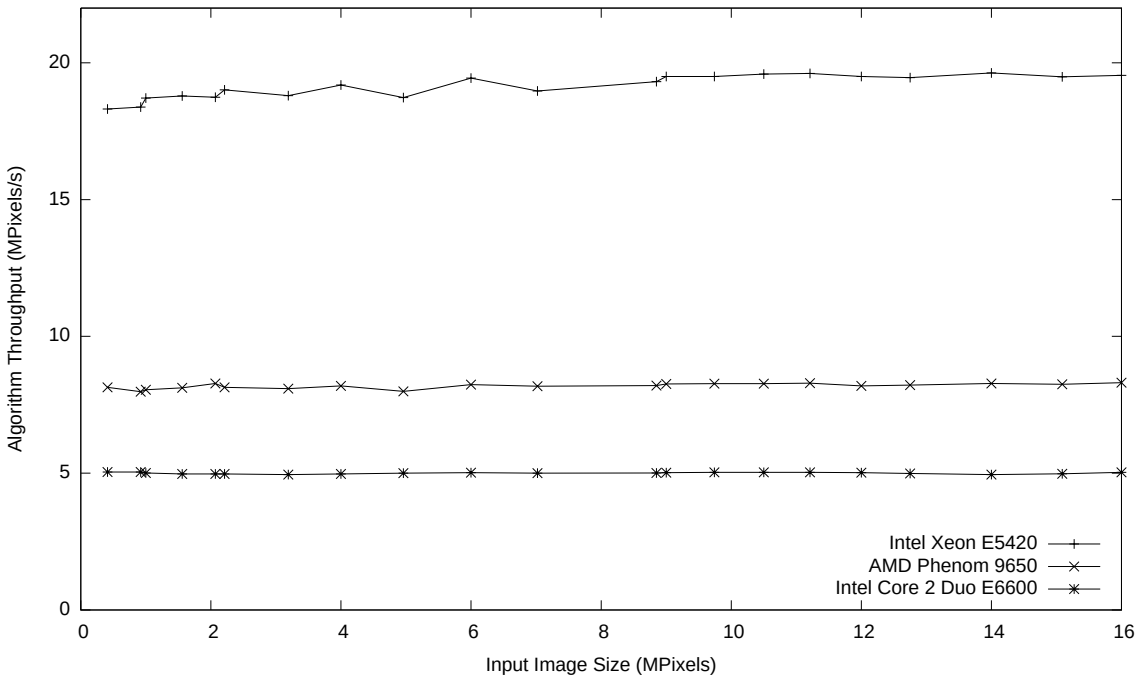


Figure 4.6: Throughput of the wavelet-based degraining visual effect with varying levels of schedule optimisation on all cores of each of three benchmarking platforms for a 12 MPixel image. Y-axis throughput measures the number of output pixels generated per second. Referring back to Listing 3.3, the X-axis records the number of recursive calls to the DeGrainRecursive function which space/schedule optimisation takes place across. 0 indicates no fusion. 1 indicates fusion within the function. 2 indicates two fusions across two levels of recursion. 3 indicates fusion across three levels and fusion within the last. 4 indicates maximal fusion.

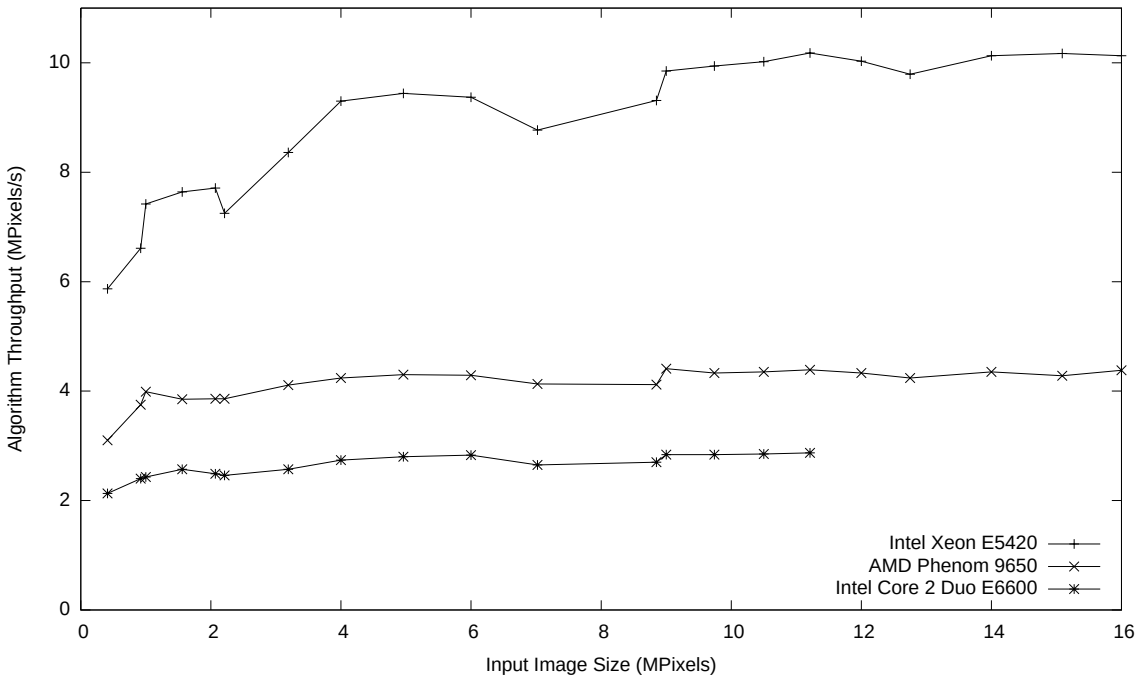
the Phenom and Core 2 Duo data sets (and a smaller but still significant correlation with the Xeon data set). This is particularly striking since the two data sets were generated by very different CPU architectures; the implementation details of a specific CPU can normally explain the instabilities in a pattern unique to that architecture.

Before considering the space-/schedule-optimised results in detail, we explore an intricacy of schedule optimisation in the wavelet-based degraining effect. Figure 4.6 presents the throughput for the space- and schedule-optimised effect on a 12 MPixel image as the level of schedule optimisation is varied. Recall from Listing 3.3 that the wavelet-based degraining effect is constructed from 4 iterations of a subset of primitives. Between each iteration lies a 1D vertical spatial filter primitive which substantially increases the costs of fusion. The graph (see caption for detail) measures the algorithm throughput as fusion is applied within the iterations but not between, and within/between subsets of iterations as well. This graph aims to show that there is an optimal level of fusion that is not complete fusion for this effect: the costs of increased working set size begin to dominate the gains from space-/schedule- optimisation after a certain degree of fusion. The optimal point is 1 or 2 iterations depending on the benchmarking platform: we choose 2, which gives an extra 6% throughput on the Xeon platform over 1. We defer examination of the speed-up over 0 fused/contracted iterations until Figure 4.8.

Figure 4.7a graphs the throughput of the space-/schedule- optimised wavelet-based degrain-



(a) Wavelet-based degraining



(b) Diffusion filtering

Figure 4.7: Throughput of the wavelet-based degraining and diffusion filtering visual effects with optimal fusion and space optimisation on all cores of each of three benchmarking platforms. Y-axis throughput measures the number of output pixels generated per second with variation of the input image size on the X-axis.

ing effect with fusion and contraction within and between pairs of iterations, as discussed in the preceding paragraph. Each of the data sets extend the full range of the X-axis as contraction has immensely reduced the working set size over the schedule- but not space-optimised case in Figure 4.5a. These results are also much more stable, which is likely the result of less cache spilling. The dips in throughput observed in the schedule- but not space-optimised graph do not appear in this data set, which suggests that they are a cache or memory effect. The 8-core Xeon is approximately four times faster than the 2-core C2D, as we would expect. The 4-core Phenom does not perform quite as well as expected and, as we will show in Chapter 5, this is a result of being more heavily computationally bound.

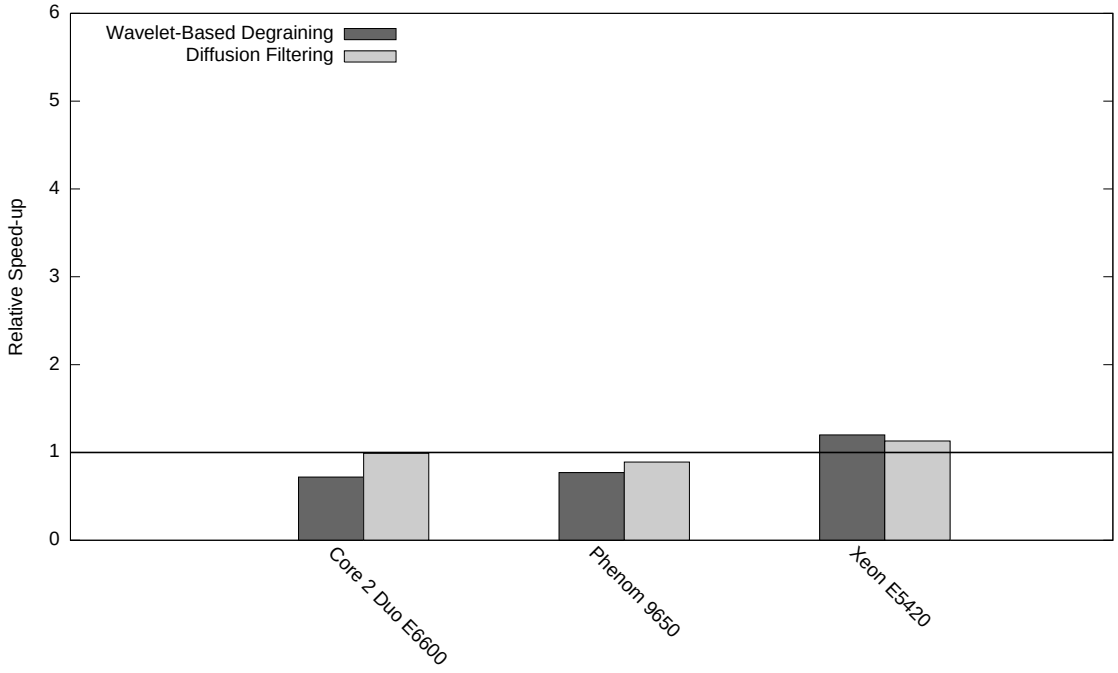
Figure 4.7b shows a similar graph for the space-/schedule- optimised diffusion filtering effect with maximal fusion and contraction. A comparison of this graph with the schedule- but not space-optimised case in Figure 4.5b shows that little has changed. As we expected, the performance of this effect is not bounded by inter-primitive communication. We have improved the memory utilisation of this effect through array contraction but not sufficiently so to fully extend the Core 2 Duo results.

Finally, we consider the improvements made from the optimisations in this chapter. Figure 4.8a summarises the performance gains and losses from schedule optimisation only in the wavelet-based degraining and diffusion filtering visual effects. This data indicates that schedule optimisation alone is not generally beneficial to performance. Changes in performance range from 0.7x to 1.2x. The gains from improved temporal reuse distance and reduced loop control overhead are in strong contention with, or dominated by, increased cache pressure and memory traffic arising from the larger working set and by inefficiencies in our parallelisation strategy.

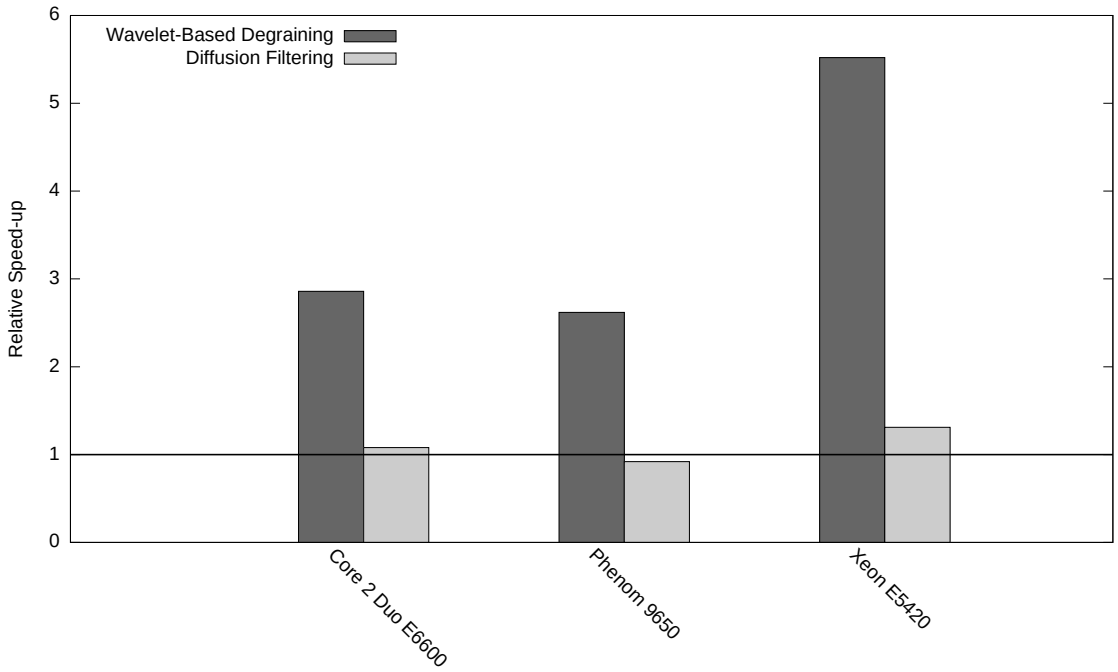
However, Figure 4.8b vindicates our efforts by demonstrating that large performance are made possible by schedule optimisation as an enabling transformation for space optimisation. We observe speed-ups between 2.6x–5.5x on the wavelet-based degraining effect and 0.9x–1.3x on the diffusion filtering effect. By reducing the working set size to an absolute minimum our performance gains are able to dominate the negative effects of schedule optimisation.

We speculated in Section 3.7 that the scalability of our generated code was limited to some degree by inter-primitive communication. The final two graphs of our performance analysis in this chapter support this assertion. Figures 4.9a and 4.9b graph the throughput of the wavelet-based degraining and diffusion filtering effects following space and schedule optimisation as the number of CPU cores in use is scaled. Compared with the same generated code without space and schedule optimisation in Figures 3.9a and 3.9b, we have improved scalability in both cases. Wavelet-based degraining, in particular, now achieves near-linear scalability for all benchmarked CPUs. We observe a 7.3x speed-up with 8 cores in this effect compared with 2.9x before schedule- and space-optimisation. Diffusion filtering experiences a smaller, but still significant, increase from 4.8x speed-up with 8 cores to 6.3x.

The improvement is scalability is particularly evident as the number of cores in use grows. Note the downward curve of scalability in Figures 3.9a and 3.9b compared with the new results in Figures 4.9a and 4.9b. As the number of cores in use increases the performance of both effects

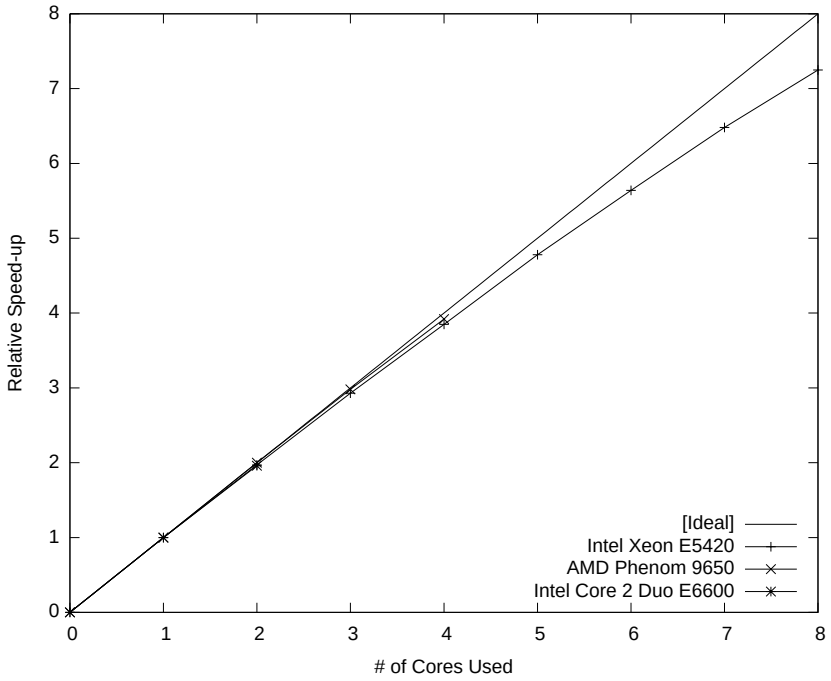


(a) Schedule optimisation only

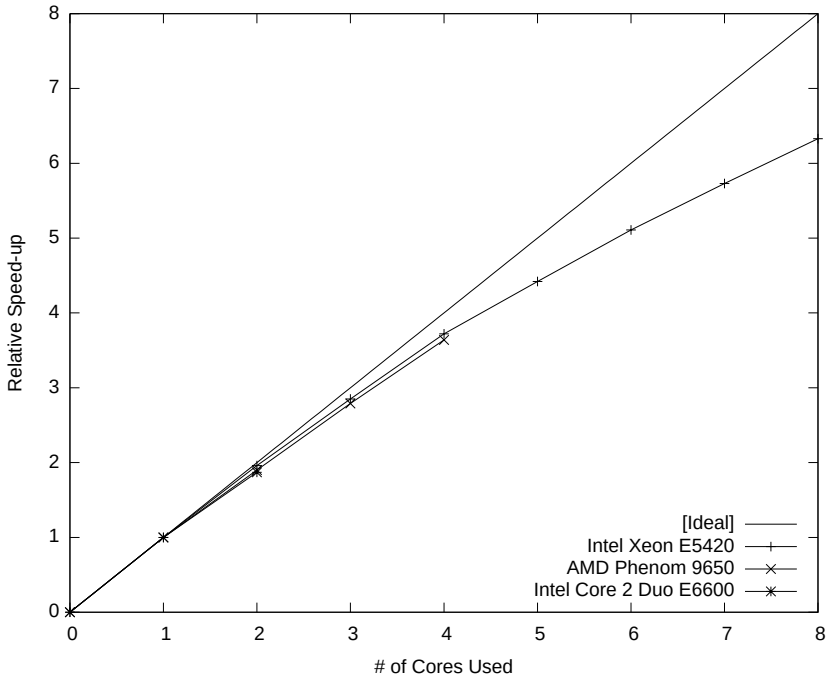


(b) Space and schedule optimisation

Figure 4.8: Relative speed-ups (1x = no speed-up) of the wavelet-based degraining and diffusion filtering effects on all cores of each benchmarking platform for a fixed input image size of 12 MPixels, following schedule- and space-/schedule- optimisation. Speed-ups are attributable to reduced temporal data reuse distance and loop control overhead (Section 4.2). Slow-downs arise from increased cache pressure and memory bus traffic, due to the increased working set size, and from higher overheads in parallelisation (Section 4.2.4).



(a) Wavelet-based degraining



(b) Diffusion filtering

Figure 4.9: Relative speed-ups (1x = no-speed-up) over single-core throughput of the wavelet-based degraining and diffusion filtering visual effects for a fixed input image size of 12 MPixels, following space and schedule optimisation. Memory optimisation improves scalability over the previous experimental results in Figure 3.9.

experiences a reduction in computational bound and increase in memory bound, as the computational resources increase while memory resources – such as the shared main memory bus and partially shared cache – become increasingly contended. Thus the effects of schedule and space optimisation become particularly significant when large numbers of cores are in use. As this is the expected use for such a system in practice we have a strong interest in improving memory resource utilisation. It is still possible to observe a small downward turn in Figures 4.9a and 4.9b after 4 cores as the Xeon begins to share partitions of its cache between pairs of cores.

Having optimised the memory performance of these effects we are now in a good position to tackle their computational performance in Chapter 5. We will return to the issue of memory optimisation in Chapter 6 from a very different perspective by automating the generation of code for architectures with memory resources of much higher capacity and bandwidth.

4.5 Concluding Remarks

In this chapter, we described an application of program metadata to a cross-component optimisation on the graph of primitives constituting a visual effect. The goals of this optimisation are to reduce the size of the working set, allowing it to fit into lower levels of the memory hierarchy, and to reduce the computational overhead of loop control structures. This optimisation is a composite of well-known program transformations, including loop fusion and array contraction. Static metadata facilitates a purely syntactic approach to transformation, by communicating high-level information about data dependence and visual effect structure directly to the optimisation engine in order to sidestep complex program analyses. Dynamic metadata further simplifies this optimisation by capturing the visual effect construction and spatial filter parameters at runtime in order to identify beneficial cross-component optimisations and tune the fused loop structures. Our space and schedule optimisation adapts dynamically to the image sizes and spatial filter radii by representing and manipulating the iteration space through variable parameters to the polyhedra.

This optimisation is best suited to effects constructed from a wide network of small interconnected primitives. Wavelet-based degraining is a good example of this class of effect, owing to its recursive construction from a short chain of 1D spatial filter and point-based primitives. Strong performance gains are observed on all benchmarking platforms with this effect. The diffusion filtering effect, built from a short chain of primitives in which a single primitive dominates performance, provides a counterexample to the optimisation in being dominated by intra-primitive, rather than inter-primitive, memory hierarchy throughput. Much smaller gains, and even slight performance degradation on one platform, are observed with this effect.

Chapter 5

SIMD Code Generation and Optimisation

In Chapter 4 we presented an application of our framework to optimising source-to-source code transformation by making use of high-level program metadata encoded in the visual primitive representation. This metadata can also be used to implement syntactic and semantic transformations between different languages. In this chapter we leverage this capability to augment our framework with a second code generation backend which targets SIMD ISAs. Chapter 3 established a basic scalar CPU backend with code flattening and static specialisation. We build upon this work by introducing vector-scale parallelisation and intrinsic instruction substitution to target the Intel SSE vector instruction set. The ideas presented in this chapter should scale well to other SIMD-oriented ISAs, such as the Intel Larrabee [SCS+08] graphics processor's ISA, although we focus on the constraints of non-scatter/gather data processing. Chapter 6 presents a third code generation backend, for SIMT ISAs, with an emphasis on large-scale parallelism and the capabilities of scatter/gather instructions.

SIMD is of particular interest to our study by virtue of its pervasiveness in modern Intel and AMD multicore CPUs and its severe underutilisation by contemporary vendor compilers. Autovectorisation is often considered a solved problem and is easy to enable with a compiler flag. We found the Intel C/C++ and GCC compilers' vectorisation capabilities to be underwhelming and, as we will demonstrate in our performance evaluation, ineffective even in the highly simplified form of code produced by the framework of Chapter 3. Throughout our research we speculated about a number of reasons for this: ambiguous data dependence, data alignment difficulties, conservative optimisation to avoid performance degradation, etc. None of our experiments were conclusive on these points but empirical evidence clearly demonstrates that compilers are failing to exploit the SIMD computational resources of CPUs effectively.

The work in this chapter was borne out of a need to exploit these resources to maximise our application performance on multicore CPUs. As we will demonstrate, the metadata captured by our framework provides an easy path to sidestep the numerous problems a vectorising compiler would face. All but one of our transformation phases are analysis-free: a single phase requires a

simple scope analysis which is made trivial by the ROSE [SQ03] source-to-source transformation tool. The work presented in this chapter is based on a substantial development of our published work on this topic [CKPN07]. In particular, we transitioned our framework from packed to planar layout images primarily to improve the ease and efficiency of SIMD parallelisation. Interaction with the space and schedule optimisations, discussed in Chapter 4, is another post-publication development which delivers improved performance and provides new perspective on the computation and memory bounds of VFX algorithms, and the ways in which we can tackle them.

5.1 Transformation Phases

Our SIMD code generation engine processes the C++ metadata-annotated kernel ASTs and polyhedral loop schedule to produce vector instruction-augmented source code for a vendor compiler, as summarised earlier in Figure 3.4. Our SSE implementation communicates vectorised instructions to the compiler in the form of intrinsic functions, which relate closely to the machine-level instructions. The compiler – Intel C/C++ 11.0 in our experimental backend – manages the task of register allocation by allowing intrinsics to operate on variables. We observed that the compiler frequently prefers in-memory operands for complex architectural performance reasons, anyway. The intrinsic code generation process is divided into five phases, characterised by independent traversals of the visual primitive kernel ASTs. These are summarised below and explained in detail in the five subsections which follow.

- **Strip Mining**. The loop schedule is modified by a classic strip mining [BGS94] transformation to group work inside the kernel into vector-sized chunks. We leverage dependence metadata to assert the safety of this transformation. Data sets are also padded to avoid the need for scalar fragments where vectorised work extends beyond the loop bound, which would otherwise pose an unacceptable cost in schedule optimisation.

- **Scalar Promotion**. We exploit knowledge of data dependence and parallelism in the statements of the kernel, provided to us by metadata, to implement a simple iterative syntactic translation algorithm which promotes the types of variables carrying image data, and data derived from it, from scalars to vectors. Upon completion of this iterative process, any statements operating on promoted vector types are themselves promoted to their vector counterparts. This process is entirely analysis-free.

- **Divergent Conditional Predication**. Scalar promotion may promote the condition variables of conditional branches within the kernel. Since different control paths may be taken for each scalar within the vector condition, we use a branch predication transformation to execute both branches and mask the results of each branch into vectors. Determining the variables that are live out from each branch – and hence those that must be masked – requires a simple variable scope analysis. We exploit the syntactic analysis capabilities of the ROSE library to provide this information, which our metadata does not.

- **Memory Access Realignment**. A correct SSE program may be constructed by using misaligned load and store instructions to move vector-sized data between main memory and registers. We describe a metadata-supported algorithm which identifies accesses that begin on aligned address boundaries to substitute faster aligned access instructions. Furthermore, we modify the input and output data sets to ensure that alignment is guaranteed within the largest loop fragments, where aligned instructions have the largest effect on performance.

- **Contracted Load/Store Rescheduling**. A consequence of the space optimisation transformations described in Chapter 4 is that vector accesses to contracted arrays must be wrapped through modulo addressing. A non-scatter/gather SIMD architecture can only address the first scalar of a vector. Thus, contracted arrays, in their post-space optimisation form, cannot be correctly addressed in a SIMD kernel implementation. We describe an extension to contracted array addressing which uses a second array to ensure that wrapping is not required during vector load/stores in at least one of the two arrays.

Our SIMD code generation process differs from established research in two ways. First, parallelisation is not the primary challenge: metadata establishes clear data dependence relationships between the statements of different kernels and between those within a kernel. Most autovectorisation literature research focuses on data dependence analysis and on the transformations required to enable parallelisation. Secondly, our code generation process operates at the C source level. Established work takes place mainly in the SSA form at a lower level of AST abstraction.

5.1.1 Strip Mining

Our first AST transformation phase is common to all SIMD code generation strategies. One or more loops in a visual primitive loop nest must be strip mined [BGS94] in order to group the nested computation and memory access statements into vector-sized chunks. A non-scatter / gather architecture requires these chunks to be contiguous in memory and in registers. Thus, it is typical to select only the x loop for strip mining, in which horizontally successive (and hence contiguous) data members are processed. There are two cases in which this strategy fails:

- **Packed image layouts**. Recall from Chapter 3 that our framework operates in the planar image layout domain. This layout forms a horizontally contiguous ordering of successive component intensities from a common plane, which satisfies the contiguity requirement of a non-scatter/gather SIMD architecture as described above. In an earlier prototype framework we experimented with packed image layouts. The contiguity requirement is not broken in this case: although intensities are no longer spatially contiguous, the data layout matches the computation ordering of the loop nests (i.e. R, G, B, R, G, B, etc.). However, since the component loop is nested innermost in this ordering, and because its trip count is frequently smaller than the vector width (an RGB image leads to a trip count of only 3), we are unable to strip mine it. Interchanging the x and component loops would lead to a mismatch between data layout and computation. Thus, our only remaining options are to use

an unroll-and-jam transformation [BGS94] or to pad images to RGBA sizes. We explored the latter in [CKPN07] and identified overheads in memory access and limitation to architectures with a vector width no larger than 4.

- **Horizontal moving averages**. Recall from Definition 2.11 that a moving average primitive has a loop-carried data dependence in one axis. In a horizontal moving average this data dependence lies between successive iterations of the x loop. In order to strip mine the x loop we require a guarantee that no data dependence exists between iterations within a vector-sized group. From Metadata 3.3.2 we know that this is not the case for a horizontal moving average. We experimented with block-transposed loads and stores (see Chapter 6) to reformulate the primitive as a vertical moving average, which does not pose a similar problem, but found the overheads of transposition to exceed the gains from exploiting SIMD instructions. Thus, our SIMD code generation process leaves horizontal moving average kernels in their scalar form.

The granularity of the strip mining transformation is determined by the vector width of the target architecture. This process is scalable to any vector size (V_{size}) that is substantially smaller than the trip count of the strip mined loop. As V_{size} increases towards the trip count, the percentage of work lost due to the last iteration overhanging the data set tends towards 50%, with exceptions near vector widths at which the trip count is divisible by the vector width. The SSE ISA has a vector width of 4, which is much smaller than any image we work with in practice.

Listing 5.1 shows simplified generated code for the grain reduction visual primitive of the wavelet-based degraining algorithm [SCW05] with strip mining for a 4-wide vector architecture. We use an established notation from [BGS94] to indicate that a statement must be evaluated for a set of values dependent upon the strip-mined loop's iterator. In doing so, we are asserting that there exist no data dependencies which would be violated in a parallel execution of this statement for all values. Contrary to the established literature, we choose not to introduce a scalar cleanup loop to handle the case where $width$ is not exactly divisible by the vector width. Instead, we introduce a padding factor into the row strides of the input and output images to ensure that vector writes at the edge fall within an allocated but unused part of the data set. This trades a small computational and memory access expense – at most 3 iterations per image row for SSE – for a simpler loop structure which avoids combinatorial explosion in schedule optimisation (see Chapter 4).

The transformation is implemented as a modification of the polyhedral schedule (Section 4.2). For example, the constraint matrix for the loop nests in Listing 5.1 is shown in Table 5.1. Strip mining introduces a new, unused iteration variable into the matrix for each loop that is modified. An extra constraint defines a linear relationship between the loop's iteration variable and the extra unused variable. The coefficient of the unused variable in this constraint is set to the granularity of the strip-mining transformation: i.e. the vector width, which is 4 in the case of SSE. Because the polyhedral scanning process will only visit integral points in the polyhedron, this is sufficient to discard iterations 1...3, 5...7, 9...11 etc. of the strip-mined loop where x' would have a fractional value. The net effect is to set the step of the strip-mined loop to the vector width.

```
for(int c = 0; c <= 2; ++ c) {
  for(int y = 0; y <= height - 1; ++ y) {
    for(int x = 0; x <= width - 1; x += 4) {
      const float val = Input[c*pStride + y*rStride + x[0:3]];
      const float absVal = fabsf(val);

      float newVal = val;
      if(absVal < alpha1) {
        newVal = 0.0f;
      } else if(absVal <= alpha3) {
        newVal = copysign(alpha3 * (absVal - alpha1) * beta, val);
      }

      Output[c*pStride + y*rStride + x[0:3]] = newVal;
    }
  }
}
```

Listing 5.1: A visual primitive implementation of the grain reduction stage in the wavelet-based degraining algorithm described in [SCW05]. The x loop has been strip-mined for a SIMD architecture with 4-element vectors. The notation x[0:3] indicates a set of successive x values, beginning at x, for which the statement must be executed and may be computed in parallel.

$=/\geq$	c	y	x	x'	*width*	*height*	1	constraint
1	1	0	0	0	0	0	0	$c \geq 0$
1	-1	0	0	0	0	0	2	$c \leq 2$
1	0	1	0	0	0	0	0	$y \geq 0$
1	0	-1	0	0	0	1	-1	$y \leq height - 1$
1	0	0	1	0	0	0	0	$x \geq 0$
1	0	0	-1	0	1	0	-1	$x \leq width - 1$
0	0	0	1	-4	0	0	0	$x = 4 \times x'$

Table 5.1: Constraint matrix for the strip-mined loop nest shown in Listing 5.1. A new iteration variable x' is introduced and an additional constraint links x to a multiple of x'. Since the statement will only execute for integral configurations of the iteration variables, the unused variable x' spaces values of x apart by its multiple.

In this intermediate form it is non-trivial to generate a valid C implementation of the visual primitive. Statements operating upon grouped values of x can be expanded with ease, but their outputs scale to multiple values which necessitates further expansion of any other statements which consume them. We do not consider the scalar code generation problem here – essentially a form of loop unrolling [BGS94] – and instead leverage the grouped form to transform the kernel statements directly into vector intrinsic equivalents.

5.1.2 Scalar Promotion

Our second AST transformation phase implements the key vector intrinsic substitution process which replaces grouped statements from the strip-mining phase with vector instructions. The substitution algorithm, which we call scalar promotion, leverages data dependence guarantees from Metadata 3.3.2 to assert the absence of loop-carried data dependence between statements of

the kernel for all iterations within the group. Note that a vertical moving average, for example, has a loop-carried dependence between successive iterations of the non-strip-mined y loop. This does not inhibit scalar promotion because the amount of parallelism required for SIMD is small and constrained to the parallel x loop.

We developed an iterative algorithm which terminates when no further changes are made to the kernel within an iteration. This iterative process allows promoted vectors to propagate from array accesses, where grouped statements begin, through variables to other statements of the kernel. Throughout this process the kernel will be type-unsafe; only once the iteration process has completed is type correctness restored, with the exception of vector conditionals that are disallowed by the language and tackled in Section 5.1.3. We make no claims about the complexity or efficiency of this algorithm and, in particular, expect further research to uncover a more efficient formulation of this problem and its solution. Performance of the scalar promotion algorithm has not been a concern for the kernels we have studied in this thesis.

Listing 5.2 outlines pseudocode for the scalar promotion algorithm. The algorithm is divided into two iterative stages: promotion of variables to vector types followed by promotion of the expressions operating upon those variables. This division is desirable because the promotion of a scalar variable can have propagative effects which reach other statements, whereas promotion of a scalar expression has an influence that is limited to the statement in which the expression is contained. Both stages are applied repeatedly until no more changes are made to the kernel by either stage. Square-bracketed pseudocode indicates simple constructs and tests which we trivially implemented in the ROSE compiler infrastructure through traversals of the AST. A key advantage of this algorithm is the absence of program analysis: only type-sensitive syntax-directed translation is used.

The first stage of this algorithm promotes the types of scalar variable declarations to their vector counterparts. Constraint 3.2.1 narrowed the domain focus to images of floating-point data type. In SSE intrinsic terminology, the $float$ type is promoted to a vector of 4 floats: $__m128$. Any assignment with a scalar variable of $float$ type on the left-hand side is examined for an expression on the right-hand side of vector type. If this is found, the type of the LHS variable is promoted to $__m128$ by modifying its declaration statement. This process is iterated in case the promotion of a variable gives rise to an assignment in which the RHS is a variable that we just promoted, which may lead to promotion of another variable, etc. Promoting the declarations first has the most wide-reaching effect on the kernel, and thus this is separated from subexpression promotion in the second stage.

After the first stage of the algorithm terminates, our example kernel from Listing 5.2 is transformed into the type-invalid intermediate form shown in Listing 5.3. Two variables are promoted from $float$ to the $__m128$ vector type. *val* is the first variable to be promoted: the group of scalars received from the Input[] array, due to the strip-mined iteration variable x, necessitates promotion to receive all four values in a single iteration. Depending on the layout of the AST and order of traversal, the next variable will either be promoted in the same iteration or in the next. *newVal* is promoted through direct assignment from the promoted variable *val*. Notice that *absVal* has not

```
do {
  CycleChanged = false

  // Promote variables to vector type.
  do {
    IterChanged = false
    foreach(ScalVar in [Kernel.Assignments.LHS]) {
      if [RHS is group of scalars or vector type] {
        [Promote ScalVar.Declaration.Type to vector type]
        IterChanged = CycleChanged = true
      }
    }
  } while(IterChanged)

  // Promote vector:scalar operations to vector:vector.
  do {
    IterChanged = false
    foreach(ScalExp in [Kernel.Expressions]) {
      foreach(SubExpr in [ScalExp.SubExpressions]) {
        if [SubExpr is group of scalars or vector type] {
          [Replace ScalExp with vector equivalent]
          IterChanged = CycleChanged = true
        }
      }
    }
  } while(IterChanged)
} while(CycleChanged)
```

Listing 5.2: Pseudocode for the two-stage iterative scalar promotion algorithm.

yet been promoted: the type-validity of $fabsf(val)$ is not be resolved until the second stage. This is the reasoning behind the outer iterative loop surrounding both stages which does not terminate until both stages make no further changes.

Listing 5.4 shows the kernel once the scalar promotion algorithm has terminated. *absVal* has been promoted to $_m128$ type following the promotion of the RHS of its initialiser to a vector intrinsic expression equivalent to $fabsf(val)$. Promotion of expressions to their vector counterparts propagates from those involving promoted variables to expressions consisting of promoted expressions. For each target ISA we would have a set of syntactic rules for translating the scalar variables and expressions into equivalent SIMD instructions. Some of the promotion cases we have shown for SSE are quite complicated: for example, $fabsf()$ and $copysign()$ are both implemented with bitwise mask constructs. It is important to note that this kernel is still not quite type-safe: the language does not support conditional tests with vector expressions, which the $_mm_cmp*$ intrinsic family return. We will return to this problem in Section 5.1.3.

Brief consideration must be given to kernel expressions involving the strip-mined iteration variable. The normal form of scalar variable promotion produces a vector with the value replicated in all of its elements. A strip-mined iteration variable, such as x from Listing 5.4, however, must be promoted to a vector consisting of $[x, x+1, x+2, x+3, ...]$ for the semantics of the kernel to remain unchanged. The example we show here does not make use of the iteration variable, but other kernels, such as the image border grow kernel – which must detect regions of output that

```
for(int c = 0; c <= 2; ++ c) {
  for(int y = 0; y <= height - 1; ++ y) {
    for(int x = 0; x <= width - 1; x += 4) {
      const __m128 val = Input[c*pStride + y*rStride + x[0:3]];
      const float absVal = fabsf(val);

      __m128 newVal = val;
      if(absVal < alpha1) {
        newVal = 0.0f;
      } else if(absVal <= alpha3) {
        newVal = copysign(alpha3 * (absVal - alpha1) * beta, val);
      }

      Output[c*pStride + y*rStride + x[0:3]] = newVal;
    }
  }
}
```

Listing 5.3: The grain reduction primitive from Listing 5.1 after the first stage of the scalar promotion algorithm shown in Listing 5.2. Changes are highlighted in bold.

```
for(int c = 0; c <= 2; ++ c) {
  for(int y = 0; y <= height - 1; ++ y) {
    for(int x = 0; x <= width - 1; x += 4) {
      const __m128 val = _mm_load_ps(&Input[c*pStride + y*rStride + x]);
      const __m128 absVal = _mm_and_ps(val, _mm_castsi128_ps(
                                          _mm_set1_epi32(0x7fffffff)));
      __m128 newVal = val;
      if(_mm_cmplt_ps(absVal, _mm_set_ps1(alpha1))) {
        newVal = _mm_setzero_ps();
      } else if(_mm_cmple_ps(absVal, _mm_set_ps1(alpha3))) {
        newVal = _mm_or_ps(_mm_and_ps(_mm_mul_ps(_mm_mul_ps(_mm_set_ps1(alpha3),
          _mm_sub_ps(absVal, _mm_set_ps1(alpha1)), _mm_set_ps1(beta))),
          _mm_castsi128_ps(_mm_set1_epi32(0x80000000))), val);
      }

      _mm_store_ps(&Output[c*pStride + y*rStride + x], newVal);
    }
  }
}
```

Listing 5.4: The grain reduction primitive from Listing 5.1 after termination of the complete scalar promotion algorithm shown in Listing 5.2. Changes are highlighted in bold.

correspond to real data – do make use of this side case.

5.1.3 Divergent Conditional Predication

Upon completion of the scalar promotion phase described in the preceding section, the AST has been transformed into a SIMD intrinsic-augmented form that is almost compilable. The remaining correctness problem lies in the type incompatibility of the conditional test (*boolean*) and the SSE vector-wide comparison intrinsic ($__m128$), which contains 4 boolean results from a pairwise comparison of scalars between two vectors. Figure 5.1 illustrates why this incompatibility can-

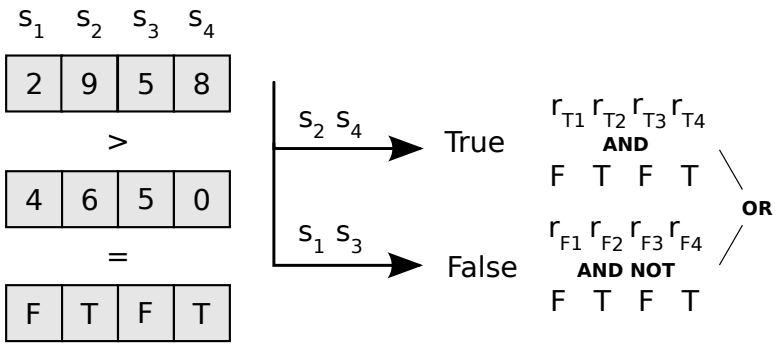


Figure 5.1: Illustration of divergence in a vector-conditional branch, resolved with scalar-wise conditional predication. The boolean result vector consists of true and false elements. In predication, both branches are executed and the live-out variables of each branch are masked with this vector and merged.

not be resolved with a simple transformation. The booleans in the mask can be, and often are, a mix of true and false values. Taking either branch would be incorrect as we may be executing statements for which the conditional test says we should not. One could isolate cases where the condition vector is constructed from scalars with identical boolean values and execute the conditional branch as normal, but in practice it is more expensive to test for this case than to treat all vector conditional cases uniformly.

This is a classic problem in autovectorisation and has even been studied as a software and hardware optimisation [TF96] for scalar code, where the costs of branch control logic and misprediction can be substantial in comparison to the size of the branch bodies. The solution is *branch predication*: executing both paths of the conditional branch and masking the live-out variables of each branch to form the same outputs that would have been produced in a conditional, scalar execution. In vectorisation, branch predication is often more of a correctness requirement than an optimisation unless the ISA provides support for instruction predication in hardware.

Our implementation of branch predication consists of a single traversal of the AST. Listing 5.5 outlines pseudocode for this algorithm. The algorithm first identifies a set of variables that are possibly live-out from either branch of the condition. Each of these variables is replaced with a shadow copy of the variable and an assignment of the live-in value is made to each copy prior to the branch bodies. The condition is removed and the true and false branches are both executed in sequence. Finally, the copied live-out values from both branches are bitwise combined with the vector boolean condition and assigned to the corresponding original variables to complete the predication transformation.

The algorithm uses a function $ScopeOf$ to determine the scope in which a variable is defined. Since we disallow pointers and references in the kernel, which serve no useful function given the data access methods permitted, an implementation of this function can be a pure syntactic analysis. We use the ROSE compiler's built-in scope analysis to derive this information.

Returning to the grain reduction primitive discussed earlier in this chapter, the kernel takes the form shown in Listing 5.6 following the conditional predication phase. Two levels of conditionals

```
foreach(IfStmt in AST.IfStatements) {
  if TypeOf(IfStmt.Condition) == Vector {
    foreach(BranchBody in [IfStmt.TrueBody, IfStmt.FalseBody]) {
      LiveOutVariables = FindLiveOutVariables(BranchBody)

      foreach(LiveOutVariable in LiveOutVariables) {
        TmpLiveOutVariable = TmpVariable(LiveOutVariable)
        BranchBody.Replace(LiveOutVariable, TmpVariable(TmpLiveOutVariable))
        BranchBody.Prepend(Assignment(TmpLiveOutVariable, LiveOutVariable))
      }
    }

    MergeBlock = []
    foreach(TrueLiveOutVariable, FindLiveOutVariables(IfStmt.TrueBody)) {
      MergeBlock.Append(Assignment(TrueLiveOutVariable,
        TmpVariable(TrueLiveOutVariable) AND IfStmt.Condition))
    }
    foreach(FalseLiveOutVariable, FindLiveOutVariables(IfStmt.FalseBody)) {
      MergeBlock.Append(Assignment(FalseLiveOutVariable,
        TmpVariable(FalseLiveOutVariable) AND NOT IfStmt.Condition))
    }

    Replace(IfStmt with [IfStmt.TrueBody, IfStmt.FalseBody, MergeBlock])
  }
}

function FindLiveOutVariables(Body) {
  LiveOutVariables = []
  foreach(Assignment in Body) {
    if ScopeOf(Assignment.LHS) != Body
      LiveOutVariables += Assignment.LHS
  }
  Return LiveOutVariables
}
```

Listing 5.5: Pseudocode for our divergent branch predication algorithm.

are predicated in turn; it does not matter in which order this occurs. A pair of vector copies is created for the live-out vector *newVal* in both predicated loop nests and combined after the true and false branches have been executed, in the manner shown in Figure 5.1, to place the correct vector value back into *newVal*. The kernel is now correct, given the data alignment criteria set out in Section 5.1.4.

5.1.4 Memory Access Realignment

The non-scatter/gather SIMD architecture, which we focus on in this chapter, is only capable of addressing the first scalar in a vector load/store. Furthermore, for architectural reasons it is considerably more efficient if the address of the first scalar is a multiple of V_{size} [SJV06]. A program-level choice of aligned or misaligned vector load/store intrinsics determines whether the faster or slower method is used. Using aligned intrinsics to access misaligned data leads to incorrect data being read or written. Using misaligned intrinsics to access aligned data is acceptable, but will not achieve the performance of the equivalent aligned intrinsic.

```
for(int c = 0; c <= 2; ++ c) {
  for(int y = 0; y <= height - 1; ++ y) {
    for(int x = 0; x <= width - 1; x += 4) {
      const __m128 val = _mm_load_ps(&Input[c*pStride + y*rStride + x]);
      const __m128 absVal = _mm_and_ps(val, _mm_castsi128_ps(
                                          _mm_set1_epi32(0x7fffffff)));
      __m128 newVal = val;
      __m128 mask_0 = _mm_cmplt_ps(absVal, _mm_set_ps1(alpha1));
      __m128 newVal_True_0 = newVal;
      __m128 newVal_False_0 = newVal;
      {
        newVal_True_0 = _mm_setzero_ps();
      }
      {
        __m128 mask_1 = _mm_cmple_ps(absVal, _mm_set_ps1(alpha3));
        __m128 newVal_True_1 = newVal_False_0;
        __m128 newVal_False_1 = newVal_False_0;
        {
          newVal_True_1 = _mm_or_ps(_mm_and_ps(_mm_mul_ps(_mm_mul_ps(
            _mm_set_ps1(alpha3),_mm_sub_ps(absVal, _mm_set_ps1(alpha1)),
            _mm_set_ps1(beta))),_mm_castsi128_ps(_mm_set1_epi32(0x80000000))),
            val);
        }
        newVal_False_0 = _mm_or_ps(_mm_and_ps(newVal_True_1, mask_1),
                                   _mm_andnot_ps(newVal_False_1, mask_1));
      }
      newVal = _mm_or_ps(_mm_and_ps(newVal_True_0, mask_0),
                         _mm_andnot_ps(newVal_False_0, mask_0));

      _mm_store_ps(&Output[c*pStride + y*rStride + x], newVal);
    }
  }
}
```

Listing 5.6: The grain reduction primitive from Listing 5.4 following application of the conditional predication algorithm shown in Listing 5.5. The kernel is now valid SSE-augmented C code. Changes are highlighted in bold.

In this section we show how Metadata 3.3.1 (DAG) and Metadata 3.3.3 (memory access) can be used to determine the alignment of loads/stores in a generated kernel, and then to correct misalignment to enable high throughput aligned instructions. The kernel shown in Listing 5.6 makes use of aligned instructions – *_mm_load_ps* and *_mm_store_ps* instead of the misaligned *_mm_loadu_ps* and *_mm_storeu_ps* – and relies on alignment techniques discussed in this section for correctness. This is the easiest case in which to achieve alignment. Factors which complicate alignment include:

- **1D/2D spatial access**. Horizontal freedom in image access, as identified in the metadata associated with a kernel, prevents us from guaranteeing alignment by allowing the vector address to move in increments that are not a multiple of V_{size}. Vertical freedom does not similarly block alignment because the strides of each image are padded to a multiple of V_{size}.
- **Subregion access**. If the ROI of an indexer is smaller than the size of the associated image

in the horizontal axis, accesses may or may not be misaligned in the kernel. The possibility of misalignment can be derived from Metadata 3.3.1 (DAG). The degree of misalignment can be computed at runtime and, if equal to zero, an aligned code variant can be used. This is particularly beneficial on architectures with small vectors as the probability of alignment decreases with V_{size}.

- **Fragmented loops**. Horizontal fragmentation in visual primitive loop nests, following the schedule optimisation described in Chapter 4, may restrict alignment guarantees to only the first fragment in a horizontal set. We deliberately misalign access in all but the last loop fragment, where we guarantee alignment, which is likely to have the largest trip count and hence the biggest impact on performance.

The first problem, involving horizontal spatial freedom in vector loads/stores, is simply solved by using misaligned intrinsics when Metadata 3.3.3 (memory access) states, statically, that an indexer may have horizontal spatial freedom ($e1D$ or $e2D$) and, dynamically, that the axis of spatial freedom for $e1D$ indexers is horizontal ($eHorizontal$). There is little we can do to make use of aligned access here because scalar-separated accesses are a common pattern in spatial filter primitives. We have considered, but not implemented, a scheme (illustrated in Figure 5.2) which could try to buffer all of the data within a spatial filter window into registers using aligned reads. Alignment is guaranteed at the start of the window because the leftmost column of the image is aligned and iteration increments in V_{size} steps. Misaligned reads could then be simulated by permuting and combining the segmented vectors. Similarly, for writes one could buffer multiple vector stores locally and combine them in registers to form a smaller set of aligned writes. However, our current framework does not permit spatial filter metadata on output indexers.

The second problem, involving access to potentially misaligned subregions of image data, cannot be corrected in the general case. There is no freedom to adjust the alignment of image data because, by accessing a subregion, there is an implication that another primitive is consuming the entire DOD (else the image wouldn't have been that large to begin with) and any adjustment is likely to induce misalignment – $(V_{size} - 1)$ out of V_{size} times – in the other primitive. However, we can detect the possibility of misalignment by searching for images which are used by multiple primitives in Metadata 3.3.1 (DAG). With this possibility confirmed one can generate different code variants for aligned and misaligned runtime cases. It is beneficial to detect this statically else we observe a combinatorial code explosion as the number of different images processed by a primitive increases. At runtime the state of alignment can be determined as follows and used to select the appropriate aligned or misaligned code variant:

$$misaligned = ((roi.x1 \bmod vectorwidth) \neq 0) \tag{5.1}$$

The third problem, involving horizontal loop fragmentation following schedule optimisation, can be corrected to a very good degree. This is important because the cost of misalignment may reduce or overshadow any performance gains from space optimisation. An observation key to solving this problem is that, in the majority of visual effects, spatial filter dimensions are much

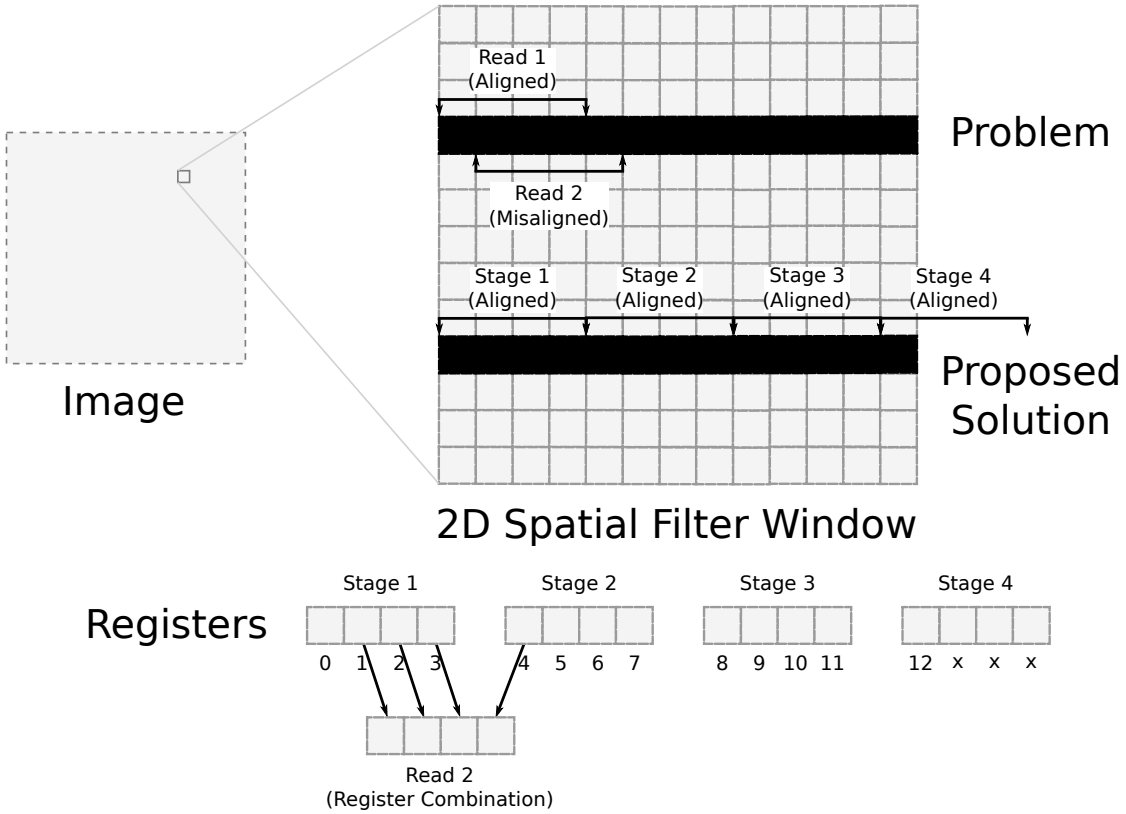


Figure 5.2: A proposed, but as yet unimplemented, solution to the problem of vector load alignment for an indexer with horizontal (in this case 2D) spatial freedom. Aligned loads are made to stage the whole spatial filter window into registers and individual indexer loads are reconstructed by combining the staged registers.

smaller than the images being processed. As a consequence, the loop shifts required in schedule optimisation are much smaller than the whole iteration space. Fragmentation gives rise to a number of very low trip count loops followed by a single, much larger loop fragment (in which fused kernels all execute together).

We can exploit this observation by focusing our alignment efforts on the largest loop fragment. Since the majority of execution time will be spent here, misalignment in the other loop fragments will have little impact on overall performance. Thus, we exploit the ability to shift the base address of image data (or, equivalently, contracted intermediate arrays) so that it is definitely aligned in the largest loop fragment and assumed to be misaligned everywhere else. Shifting the base address requires that we always allocate an extra vector at the end of the image but this cost is insignificant. The amount by which the image or contracted array must be shifted can be similarly computed:

$$misalignment = vectorwidth - ((roi.x1 + shift) \bmod vectorwidth) \tag{5.2}$$

A caveat of shifting the image base address to align loads is that it causes misalignment in stores in the producing kernel. Only loads or stores may be forcibly aligned for any given image. Misaligned stores are considerably more expensive than misaligned loads because two loads must

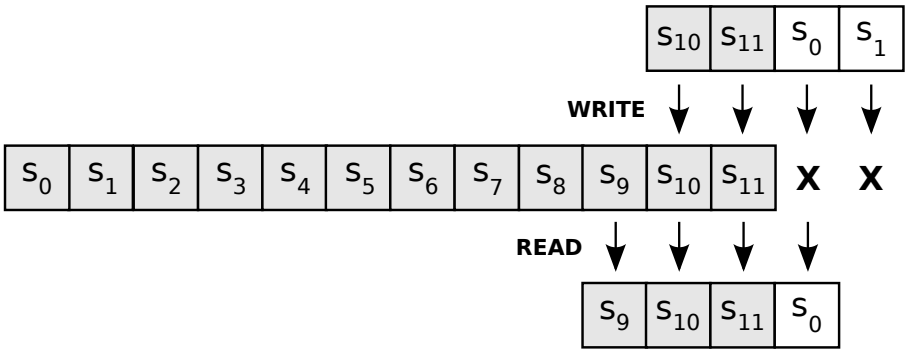


Figure 5.3: Array contraction on a non-scatterable SIMD device. Some data is lost and misread off the array edge because only the first vector element is addressable.

first be issued, the new value then permuted into the two vectors, and two aligned stores issued to write the value back to memory. Thus, our framework attempts to align stores at the expense of misaligned loads. Prior to realignment, both are considered to be misaligned and thus we still see a significant improvement.

5.1.5 Contracted Load/Store Rescheduling

The final transformation phase corrects a subtle problem which arises when contracted arrays, formed during space optimisation, are accessed in V_{size} blocks. Contraction requires all indices used to access the contracted array to be wrapped inside the allocated region with a modulus calculation. When vector loads and stores are made to the contracted array, only the first scalar can be addressed and thus wrapped. Figure 5.3 illustrates how array contraction becomes an invalid transformation under these circumstances. Vectors written to or read from the last $(vectorwidth - 1)$ elements of the array lose data or read invalid data off the end of the array respectively.

We explored two methods to resolve this and settled on one as the superior solution. Our rejected solution involved simulating misaligned reads from the contracted array – i.e. those which might fall outside the array, which is padded to a power-of-two $\geq vectorwidth$ – by issuing two aligned reads, both wrapped inside the array, and piecing together elements from the two vectors. Similarly, misaligned writes were simulated by issuing two aligned loads, shuffling new data into the two vectors, and issuing two aligned writes to the array. This proved to be considerably more expensive than using real misaligned intrinsics (which of course would not perform this wrapping) and slower than our chosen method.

Figure 5.4 illustrates our chosen solution. The contracted array is made $vectorwidth - 1$ elements larger to permit writes off the edge. We allocate a second contracted array of the same size. This has the unfortunate side effect of doubling the contracted working set size, but it is an acceptable cost for the benefits of space optimisation. Each time a store is issued to the array, a second store is issued to the second array in a wrapped location at 180° to the first. This ensures that the data is correctly written in at least one of the two arrays, if both are at least $3 \times V_{size}$ in size. The first and last vectors of both arrays are considered unsafe because loads from either may not see correctly wrapped data.

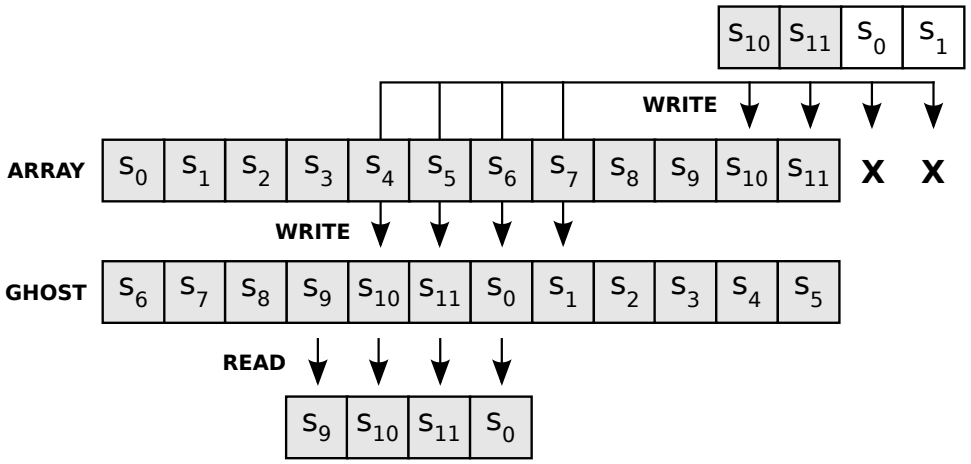


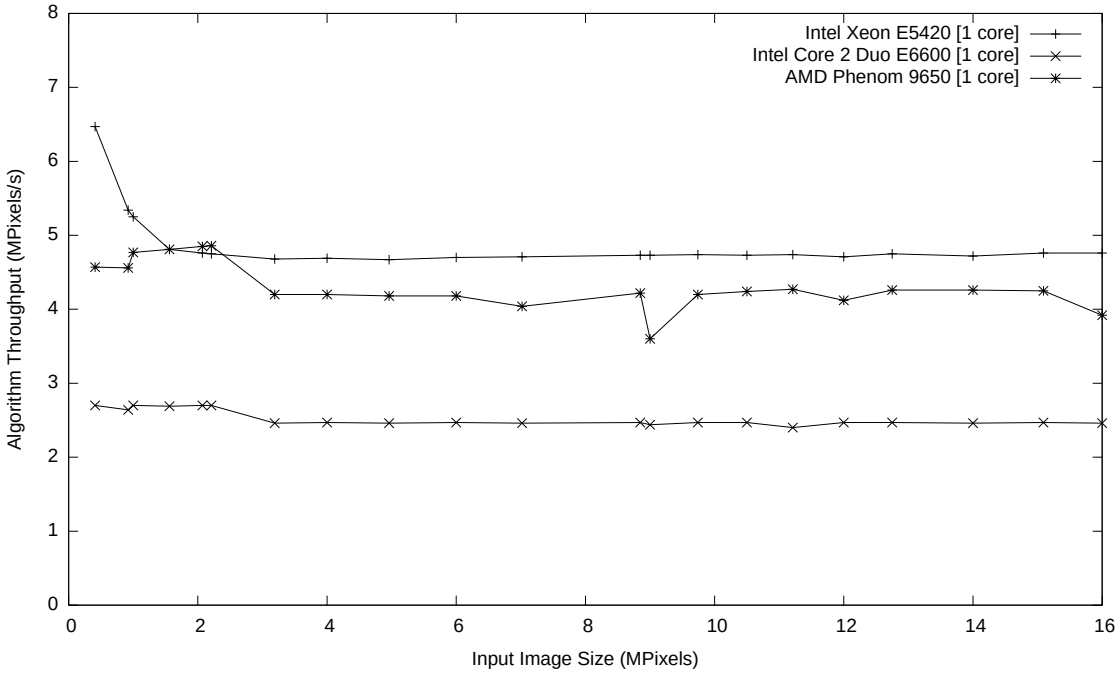
Figure 5.4: A scheme for correct SIMD array contraction. Stores are made to two arrays, one shifted by 180°, so data is not lost. Loads choose a safe array to read from.

The trick here is to select which array to read from during a load. For any given scalar offset one or both of the arrays will contain a safe, contiguous region to read the data from. We use an optimised bitwise expression to choose a base offset (first or second array) and a wrapped shift (0 degrees or 180 degrees) by examining the top two bits of the scalar offset. Since the array will always be padded to a power of two in size we can conservatively assume that the first and last quarters of the array are unsafe and that the other two are safe. Thus whenever an offset falls within the first or last quarter of the first array it is redirected to a corresponding offset in the second or third quarters of the second array.

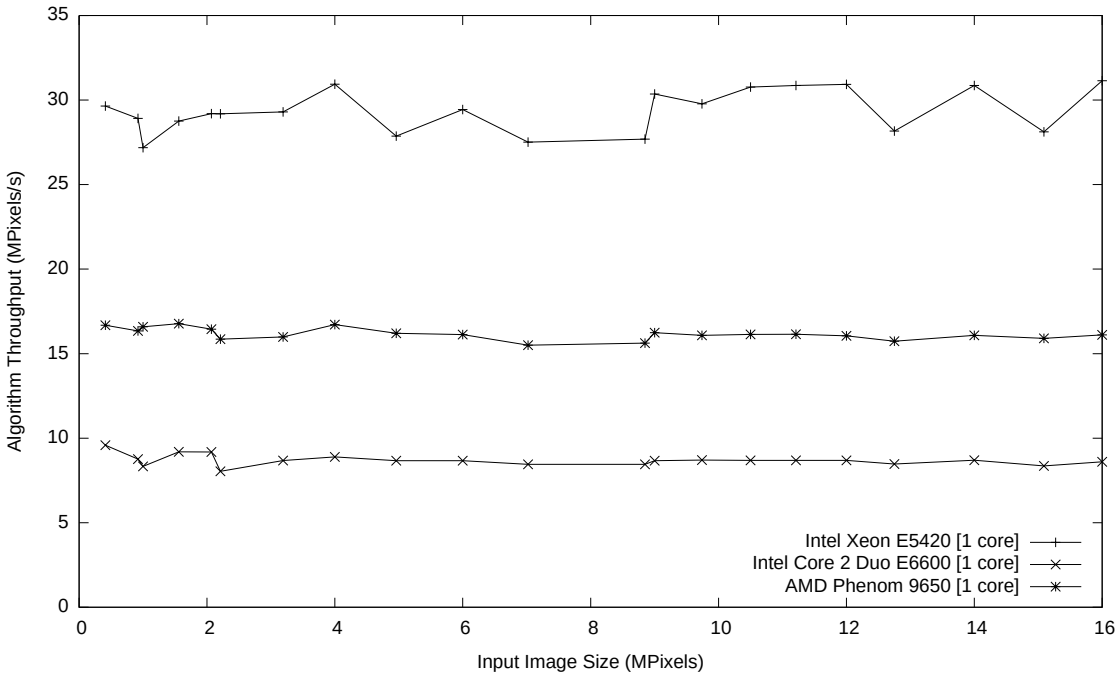
5.2 Performance Analysis

Before we present experimental results it is worth considering the effects of combining the techniques discussed in this chapter with the space and schedule optimisations described in Chapter 4. The performance limiting factor in the experimental results presented in Chapter 3 was the throughput of the memory hierarchy. Space optimisation improved this considerably in the wavelet-based degraining effect, although the diffusion filtering effect remained bounded in other ways. The ordering of these chapters is significant: the application of SIMD ISAs, whose sole purpose is to improve computational throughput, to a visual effect which is largely memory bound is ineffective. The experimental results in Chapter 4 rebalanced the computational and memory bounds in favour of computational acceleration, as we will now demonstrate.

Figure 5.5a presents the throughput of SSE intrinsic-augmented generated code for the wavelet-based degraining effect as image size is varied. Space and schedule optimisation is not applied in this case. The range of throughputs across each platform is quite narrow. Performance is completely dominated by the memory hierarchy, which does not vary between each platform as much as their computational capabilities. We use a large Y axis scale to permit direct comparison with the space/schedule optimised, SSE intrinsic-augmented implementation shown in Figure 5.5b.



(a) No space/schedule optimisation



(b) Space/schedule optimised

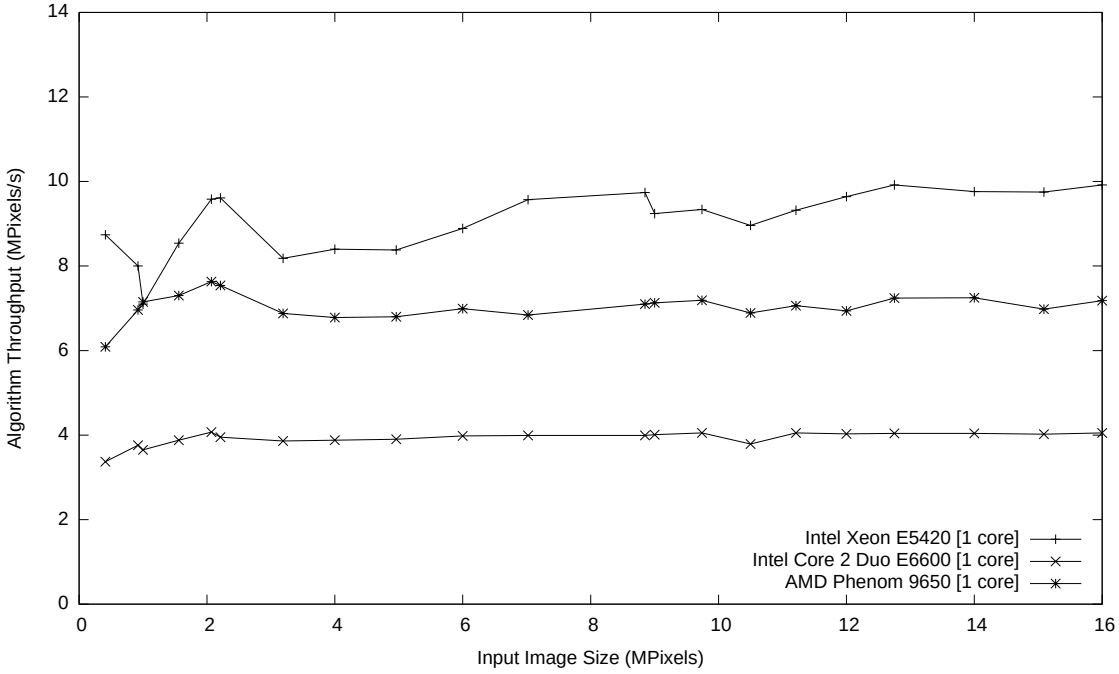
Figure 5.5: Throughput of the wavelet-based degraining visual effect with an SSE intrinsic implementation for all constituent visual primitives, both with and without space/schedule optimisation, on all cores of each of three benchmarking platforms. Y-axis throughput measures the number of output pixels generated per second with variation of the input image size on the X-axis.

Our goal here is not to show the speed-up from using SSE, but instead to emphasise the changing performance bound from memory to computational as space/schedule optimisation is applied. Differences in throughput between each platform increase from 1.7x and 1.1x (two to four cores, four to eight cores respectively) to 1.9x and 1.9x respectively. This is almost perfectly in line with the differences in computational capability of each device, architectural differences aside.

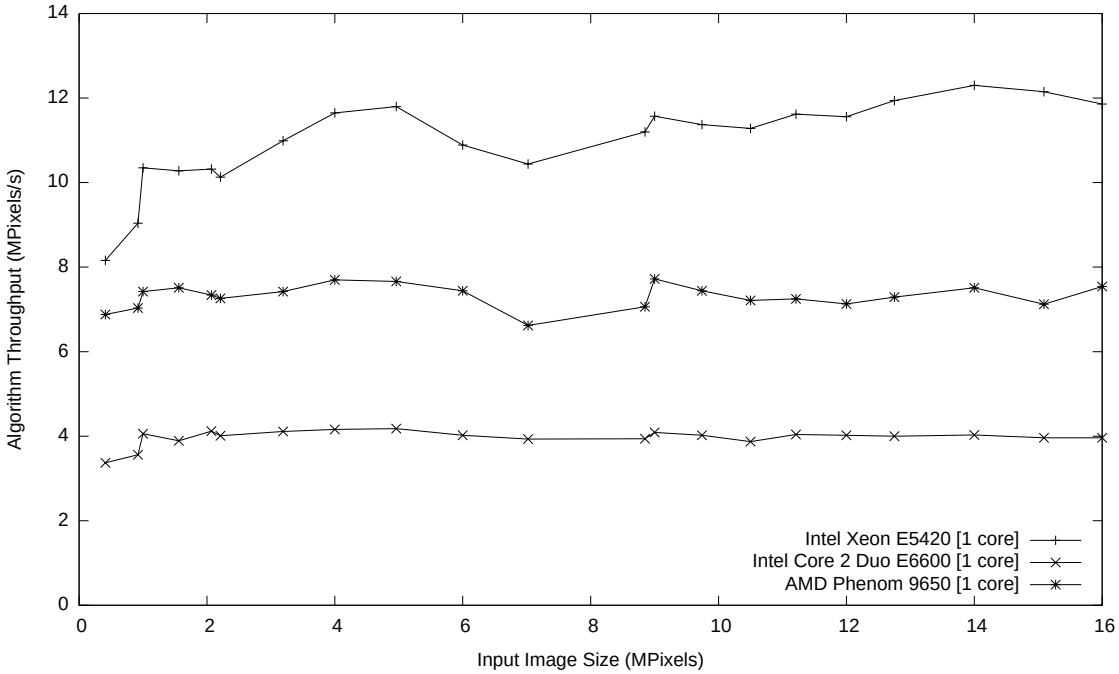
Figures 5.6a and 5.6b present a similar progression of throughput for an SSE intrinsic augmented implementation of the diffusion filtering effect. In this case only 9 out of the 12 constituent primitives have a generated SSE kernel due to the presence of three horizontal moving averages, which block the strip mining transformation as discussed in Section 5.1.1. Recall from Chapter 4 that the speed-up from space/schedule optimisation in the scalar implementation was only 0.9x–1.3x. This effect remains largely memory bound and the application of SSE does not yield significant growth in throughput or differentiation between the three platforms. Since SSE offers only increased computational throughput, and no improvement in memory bandwidth or latency, this is the expected result. SSE vectorisation reduces the contribution of data-dependent performance in this effect (not graphed) because divergent branch predication forces the execution of branches with data-dependent conditions which might otherwise be bypassed.

Speed-ups from SSE vectorisation, in combination with the space/schedule optimisation results from Chapter 4, are presented in Figure 5.7a. The segments of each bar are sized proportionally to the speed-up (throughput after $\div$ throughput before) delivered by the corresponding optimisation, beginning at 1x for no optimisation. The total bar size indicates the overall speed-up attained from both space/schedule optimisation and SSE vectorisation in composition. A comparison with Figure 4.8b reveals the space/schedule performance results that are carried over to this new graph. In wavelet-based degraining SSE improves the speed-up over the generated code results in Chapter 3 from 2.9x to 5.0x on the Core 2 Duo, 2.6x to 5.1x on the Phenom and 5.5x to 8.1x on the Xeon. Space and schedule optimisation tackled the memory bound of this algorithm, whilst SSE vectorisation increases the computational throughput. Both work in concert to deliver a large overall speed-up over the unoptimised implementation. Diffusion filtering sees a smaller improvement from 1.1x to 1.5x (C2D), 0.9x to 1.4x (Phenom) and 1.3x to 1.5x (Xeon). We speculate that this algorithm remains memory bound in ways our space and schedule optimisation have not tackled: i.e. intra-primitive bandwidth.

Figure 5.7b presents an alternative visualisation of the speed-up from SSE vectorisation. Here, we reverse the application of space/schedule optimisation and SSE vectorisation to increase the computational throughput before tackling the memory throughput. The wavelet-based degraining effect shows a large difference in speed-up contributions from the first phase ordering in Figure 5.7a. Using the SSE ISA now gives a much smaller contribution to the overall performance, before space/schedule optimisation is able to relieve the memory bound and saturate the SSE vector units. The overall speed-up remains unchanged: both whole bars represent the composition of space/schedule optimisation followed by SSE vectorisation in the same phase ordering. This graph is mainly intended to emphasise the need for memory hierarchy and computational throughput optimisations to work in harmony in order to improve overall performance.

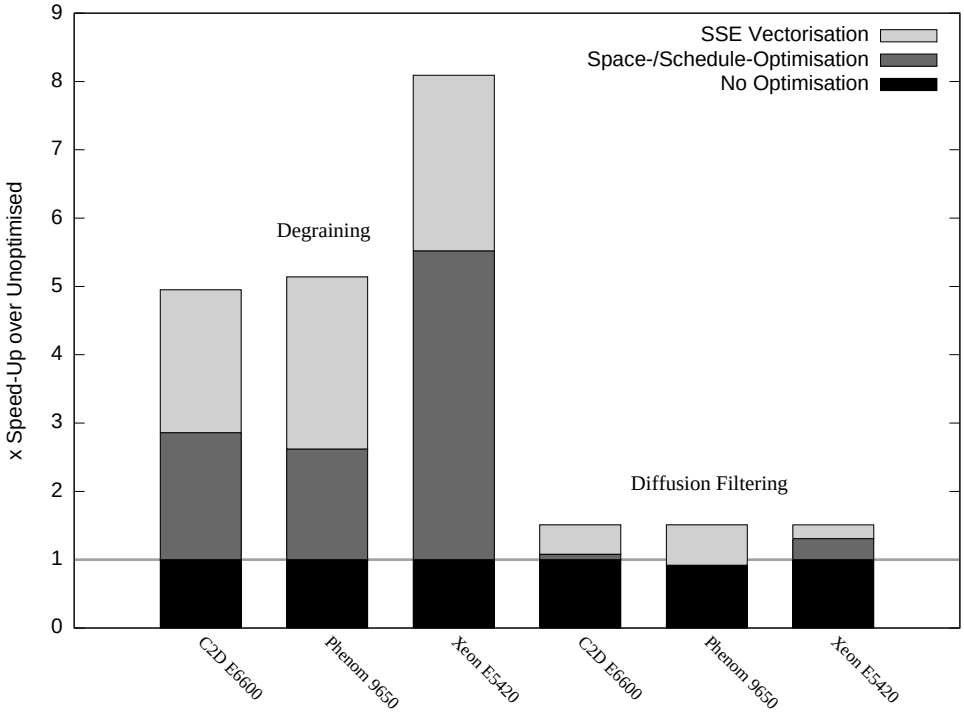


(a) No space/schedule optimisation

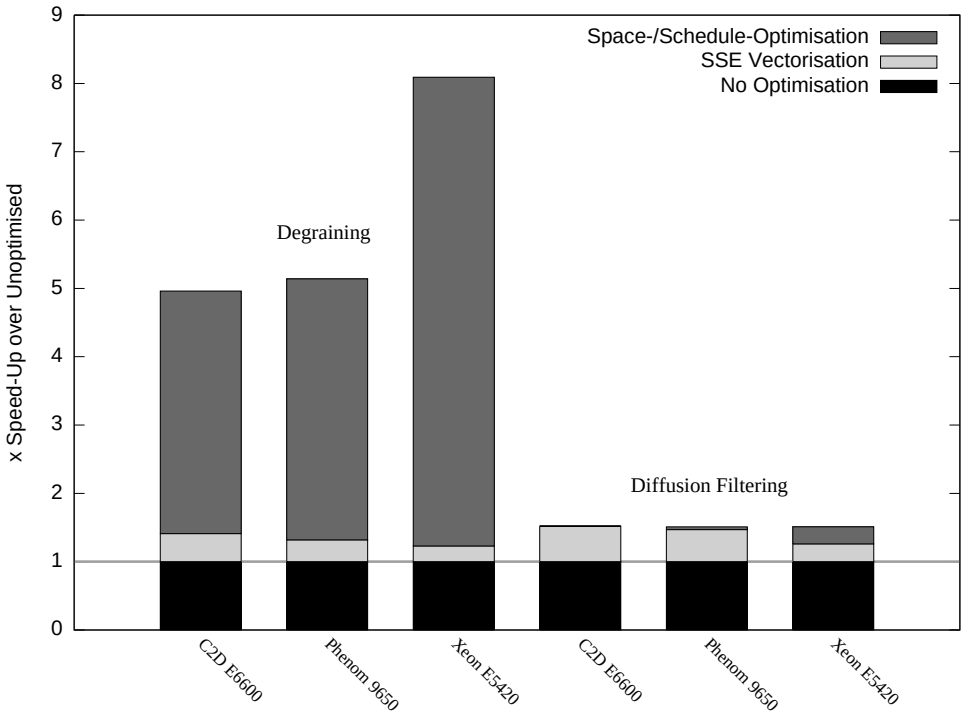


(b) Space/schedule optimised

Figure 5.6: Throughput of the diffusion filtering visual effect with an SSE intrinsic implementation for 9 out of 12 constituent visual primitives, both with and without space/schedule optimisation, on all cores of each of three benchmarking platforms. Y-axis throughput measures the number of output pixels generated per second with variation of the input image size on the X-axis.



(a) Space/schedule optimisation, then vectorisation



(b) Vectorisation, then space/schedule optimisation

Figure 5.7: Relative speed-ups (1x = no speed-up) of the wavelet-based degraining and diffusion filtering effects on all cores of each benchmarking platform for a fixed input image size of 12 MPixels, following space/schedule optimisation and SSE vectorisation. In (a) we apply space/schedule optimisation first. In (b) we apply SSE vectorisation first. The phase ordering of the optimisation composition is fixed in both cases: space/schedule optimisation first followed by SSE vectorisation. Speed-ups from SSE are attributable to the increased computational throughput of the ISA over scalar paths and are particularly effective in rebalancing the computational/memory bound once space and schedule optimisation has been applied.

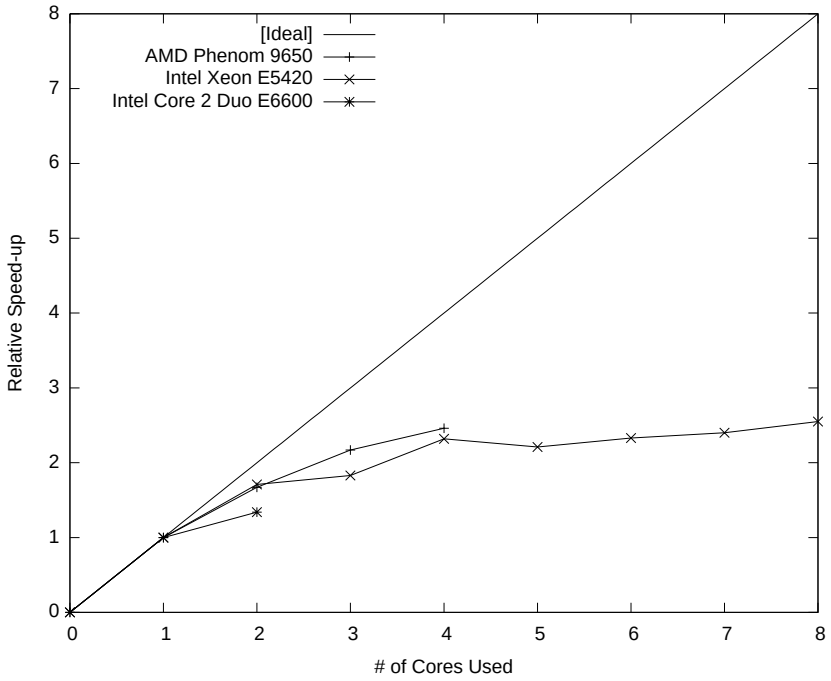
To conclude our performance study of the SSE code generation process, we demonstrate the impact on multicore scalability of, first, SSE vectorisation and, then, space/schedule optimisation. Figure 5.8a summarises the scalability of the SSE intrinsic-augmented wavelet-based degraining implementation without space or schedule optimisation. Scalability is slightly worse than the scalar implementation from Figure 3.9a due to the increased memory bandwidth requirements of each core, without a corresponding increase in memory bandwidth to supply it with data. By using space/schedule optimisation as a bandwidth enhancement, as shown in Figure 5.8b, scalability returns to a more favourable state and is comparable to, although not quite as linear as, the corresponding scalar version from Figure 4.9a. Thus, we are able to deduce that the SSE ISA requires a little more memory bandwidth than we are able to free up with space/schedule optimisation.

Figures 5.9a and 5.9b present the corresponding scalability graphs for the diffusion filtering effect. Scalability of the SSE vectorised, but not space/schedule optimised, implementation is again slightly worse than the corresponding scalar implementation, whose scalability is graphed in Figure 3.9b. Space/schedule optimisation provides a small degree of recovery but scalability remains poor after 4 cores. The memory bound of this effect is primarily intra-primitive and we have not yet developed a CPU optimisation to target this.

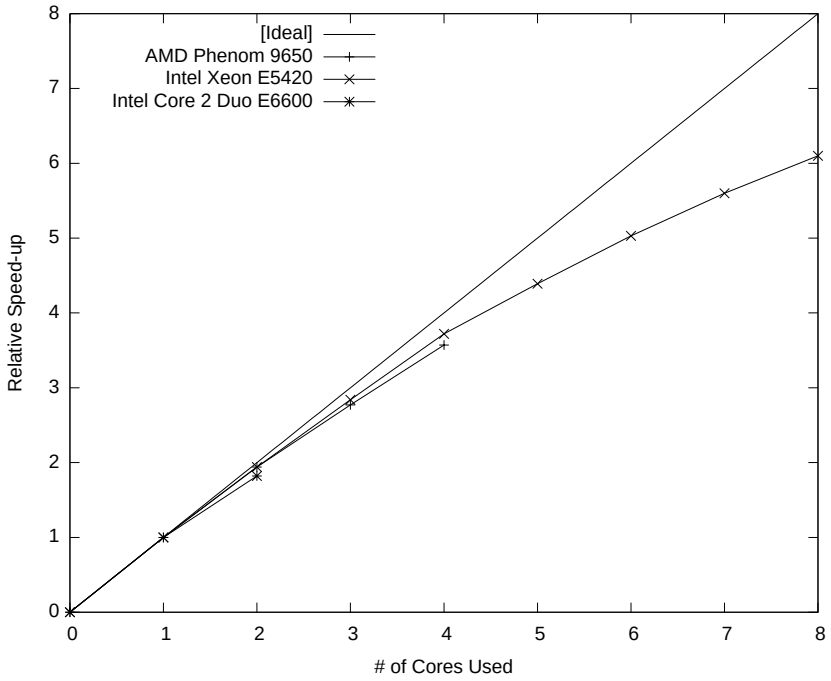
5.3 Concluding Remarks

In this chapter, we described an application of program metadata to automatic code generation for SIMD ISAs, such as Intel's SSE, from the visual primitive representation. Leveraging these ISAs enables algorithms to take advantage of high computational throughput paths, which are otherwise unused in the scalar form, by exploiting parallelism in the computations comprising a visual effect. Parallelism is made explicit by static metadata describing the data dependence structure of each visual primitive, thus sidestepping the complex analyses underlying the autovectorisation problem. Furthermore, we make use of dynamic metadata to tune realignment transformations, which enable the use of fast aligned memory access instructions, at runtime to the configurable primitive parameters which change the fused loop structures created by schedule optimisation. Our focus on non-scatter/gather SIMD architectures, which are common in short-vector computational paths, led to refinements of the space optimisation algorithm to operate at vector granularity.

SIMD code generation is best suited to effects which are dominated by computation, rather than memory access. The SSE ISA, for example, offers a theoretical improvement in computational throughput of 4x; in practice, 2-3x is typically observed. Memory access relies on the same hierarchy as scalar pathways and can therefore offer only small theoretical gains, by bypassing certain cache levels for programmer-marked transient loads/stores; but we did not explore this feature. We observe that the performance of the wavelet-based degraining effect does not respond well to SIMD code generation unless space optimisation is first applied to alleviate the memory bound, after which there are substantial gains to be made. In the diffusion filtering effect we observe that the intra-primitive memory bound, which cannot be resolved by our space optimisation, leads to smaller gains from its SIMD implementation.

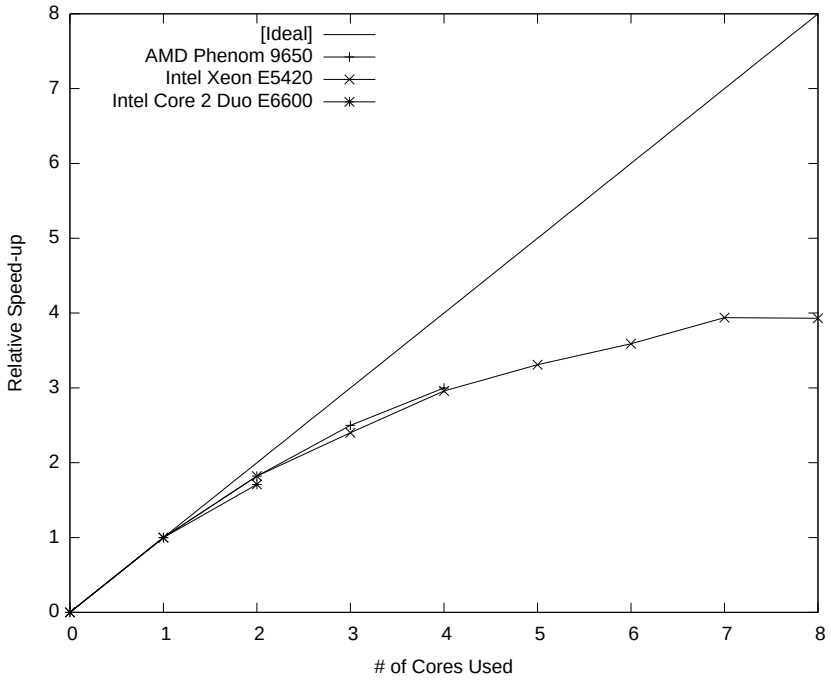


(a) No space/schedule optimisation

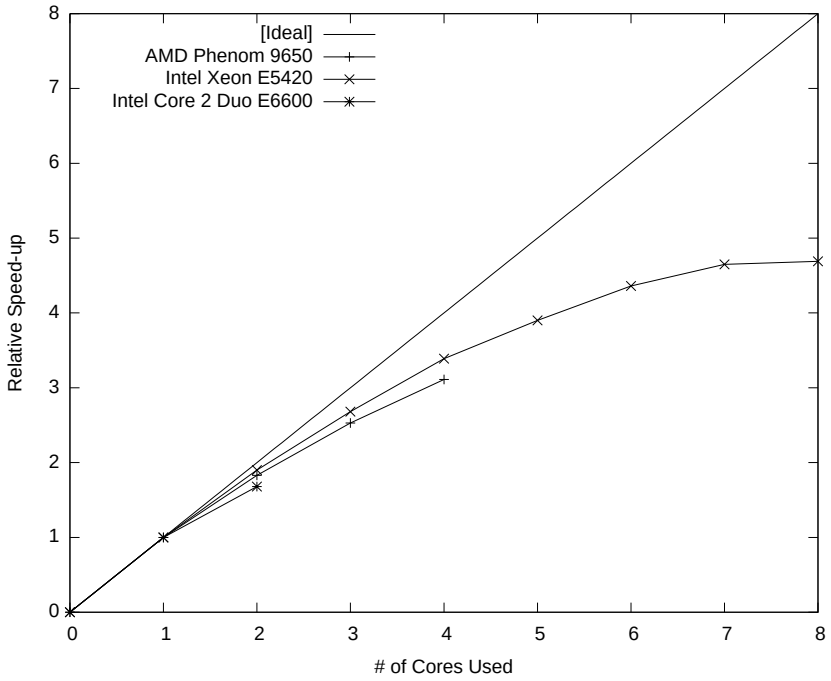


(b) Space/schedule optimised

Figure 5.8: Relative speed-ups (1x = no-speed-up) over single-core throughput of the wavelet-based degraining visual effect for a fixed input image size of 12 MPixels, following SSE vectorisation both with and without space/schedule optimisation. Scalability has worsened slightly over the scalar code from Figure 3.9a as SSE instructions saturate memory bandwidth more quickly.



(a) No space/schedule optimisation



(b) Space/schedule optimised

Figure 5.9: Relative speed-ups (1x = no-speed-up) over single-core throughput of the diffusion filtering visual effect for a fixed input image size of 12 MPixels, following SSE vectorisation both with and without space/schedule optimisation. Scalability has worsened slightly over the scalar code from Figure 3.9b as SSE instructions saturate memory bandwidth more quickly.

Chapter 6

SIMT Code Generation and Optimisation

In Chapter 5 we outlined an application of high-level program metadata to efficient SIMD code generation from the visual primitive representation described in Chapter 3. We will now demonstrate another code generation process, which shares some of the aspects of SIMD code generation, for SIMT (Single Instruction, Multiple Thread) [NBGS08] architectures. This application of metadata is a departure from the CPU optimisation-oriented goals of Chapters 4 and 5. Here, our aim is to improve algorithm throughput by mapping kernels to an entirely different class of hardware: GPUs, introduced in Section 2.4.3. SIMT devices have very high computational and memory throughput owing to their massively parallel computational units fed by high latency, high bandwidth memory. It is a brute force approach to optimising the performance of visual effects which, as we will demonstrate, gives a very high performance and cost effective solution in the VFX domain.

There are two key challenges in generating high performance code for a SIMT architecture:

- **Large scale parallelism**. In Chapter 5 we tackled short-vector SIMD architectures with parallelism requirements of only 4 threads. In contrast, a SIMT device must keep tens of thousands of threads in flight in order to hide the latency of uncached memory access. Launching fewer threads leads to underutilisation of computational and memory resources.
- **Memory hierarchy optimisation**. Because the memory hierarchy of a SIMT architecture has only limited caching facilities – e.g. NVIDIA devices provide a small, programmer-managed region of low latency memory in addition to small hardware-managed caches – it is essential to optimise memory access by issuing superword transactions (coalescing) and making explicit use of low latency regions to share data between threads.

The focus of our code generation process, which is outlined in Section 6.1, is on tackling these challenges. As will demonstrate in Section 6.1.1, the process of generating correct – but not necessarily fast – SIMT code is much simpler than the vectorisation and intrinsic substitution problems in SIMD code generation. This is a direct consequence of the SIMT model, which allows each

thread to run a scalar kernel independently of other threads. In addition to the code transformations required to support efficient use of the hardware, Section 6.2 outlines our process for automatically selecting parameters to the configurable parallelism offered by SIMT devices. The material presented in this chapter is based upon a development of our published work on this topic [CHK+09]. In particular, we offer a deeper discussion of the interactions between space/schedule optimisation and SIMT code generation. This is a potentially valuable combination, but is complicated by the synchronisation requirements discussed in Section 4.2.4.

6.1 Transformation Phases

Our SIMT code generation engine processes the C++ metadata-annotated kernel ASTs to produce SIMT-optimised kernels in NVIDIA's CUDA language, which is based upon an extension of C, as summarised earlier in Figure 3.4. The polyhedral loop representation is not used during this process and the reasons for this are discussed in Section 6.3. Upon completion of all transformation phases, the kernel is passed to NVIDIA's CUDA compiler and C++ stub code is generated to interface between the framework and the CUDA runtime. The SIMT code generation process is divided into six phases, characterised by independent traversals of the visual primitive kernel ASTs. These are summarised below and explained in detail in the six subsections which follow.

- **Syntax-Directed Translation**. The kernel is first passed through a syntactic translation process to augment it with CUDA syntax and to map mathematical functions to corresponding built-in CUDA intrinsics. By virtue of CUDA's grounding in the C language, which resembles the C++ abstract representation, this phase is a trivial syntactic transformation.

- **Constant and Shared Memory Staging**. Metadata 3.3.3 (memory access) is used to identify dynamically bounded regions of image data which will be accessed by multiple threads in the kernel, due to spatial filter access patterns in the primitive representation. These regions are first copied into the explicitly-managed cache through cooperation between groups of threads, from which they can be accessed independently by each thread with lower latency and higher bandwidth.

- **Memory Access Transposition**. Coalescing, a memory transaction optimisation made available on SIMT architectures to reduce memory control traffic, can only take place if successive threads issue loads to a dense, contiguous set of data. Because a horizontally serialised primitive, as identified by Metadata 3.3.2 (data dependence), leads to discontiguous reads between threads, we leverage a graph-level transposition optimisation which modifies Metadata 3.3.1 (graph) by inserting extra transpose primitives before and after the operation. Thus, a vertically serialised variant of the primitive – which satisfies the constraints for coalescing – can be generated and executed instead.

- **Memory Access Realignment**. In addition to the memory access transposition optimisation, coalescing opportunities are further enhanced by identifying cases in which the memory address accessed by the first thread is not aligned to a particular boundary – alignment

being a requirement for coalescing to take place – and performing a realigning transformation, not unlike the one used for SIMD architectures in Section 5.1.4. Metadata 3.3.1 and 3.3.3 (DAG and memory access) are used to identify realignment opportunities and to derive dynamic shift amounts for the transformation.

- **Maximising Parallelism**. Metadata 3.3.2 (data dependence) identifies kernels with serialisation in a single axis. In cases where parallelism in the other axis is insufficient to meet the multithreaded demands of a SIMT device, we exploit the separation of roll-up and incremental update kernels in the frontend of a moving average visual primitive to divide the serial axis into a small, parallel set of serialised iterations. This doubles, triples, etc. the amount of parallelism available at a small additional cost of roll-up computation.

- **Scheduling Overhead Reduction**. Compositions of simple pointwise kernels may consume so little execution time individually that the cost of thread scheduling dominates the computation. From Metadata 3.3.3 we identify kernels with a 1:1 mapping of input to output data elements. An unroll-and-jam [BGS94] transformation is used to increase the amount of per-thread work, reducing the number of threads needed and hence the scheduling overhead.

6.1.1 Syntax-Directed Translation

The first phase of SIMT code generation translates the C++ kernel into the CUDA language syntax. CUDA is very similar to the C language, with some additional features taken from C++, so this process is quite straightforward. Once syntax-directed translation has taken place, the kernel is correct and sufficient for execution on a GPU. The remaining transformation phases described in this chapter serve only to optimise parallelism and memory access in the kernel. The key kernel features which must be translated are as follows:

- **Indexer Accesses**. Reads and writes to image data through C++ indexer objects are flattened to array accesses, with base addresses and strides marshalled as parameters to the CUDA kernel.

- **Non-Mathematical Functions**. CUDA threads do not have a stack and, as such, cannot execute function calls. Function calls are manually inlined with a source-level transformation from the ROSE software.

- **Mathematical Functions**. Some functions have optimised CUDA built-in implementations, with various trade-offs of speed and accuracy. We maintain a list of key functions which should not be inlined but, instead, replaced with calls to these intrinsic functions.

- **Constant Class Member Variables**. Parameters to the kernel which were previously stored as constant members of the kernel class are marshalled into constant memory. These parameters are shared between all threads of the computation and cannot be changed.

```
__global__ void BoxBlur_Horizontal_CompKernel(float *Input,float *Output,const
float MultBy,const int _radius,int __dyStride,int __dpStride,int __iyStride,
int __xElems,int __yElems)
{
  float MovingSum;

  // Kill threads with off-edge workloads.
  int __thrOffset = __mul24(blockIdx.y,blockDim.y) + threadIdx.y;
  if (__thrOffset >= __yElems)
    return ;

  // Roll-Up
  int __i = __thrOffset * __iyStride - 1;
  MovingSum = 0.0f;
  for (int i = -_radius; i <= _radius; ++i) {
    MovingSum += Input[__i + i];
  }

  // Kernel
  int __j = __thrOffset * __dyStride;
  int __endj = __j + __xElems;
  for (; __j < __endj; ++__j) {
    ++__i;
    MovingSum -= Input[__i + (-_radius) - 1];
    MovingSum += Input[__i + _radius];
    Output[__j] = (MovingSum * MultBy);
  }
}
```

Listing 6.1: A horizontal implementation of the one-dimensional box blur visual primitive from Listing 3.2 following syntax-directed translation for a CUDA architecture.

- **Moving Averages**. A moving average is constructed by joining the roll-up and kernel functions, and moving non-constant class member variables – which carry state along the serialised axis – into per-thread local scope.

To illustrate this transformation phase, and the optimising phases in the coming sections, we return to the box blur visual primitive from Listing 3.2 as an example. Following syntax-directed translation the kernel appears as shown in Listing 6.1. This is a vertically-specialised implementation of the primitive; a corresponding horizontal implementation is also generated but is not shown here. Notice the absence of an (x,y) iteration domain: this implicitly surrounds a CUDA kernel and is configured through the parameters discussed in Section 6.2. The component part of an (x,y,c) iteration, for a componentwise kernel, is implemented by repeating the entire execution once per plane, moving the *Input* and *Output* pointer parameters to their corresponding planes. This avoids the need for a third iteration dimension per thread block, which would negate the benefits of componentwise iteration by processing all planes simultaneously. Most of the apparent complexity of this kernel comes from the mapping of thread/block ID to locations in the input and output images containing and receiving data involved in a thread's computation.

The *RollUp* and *Kernel* functions of Listing 3.2 have been joined together to form a single, serialised moving average per thread in the vertical axis. Thus, a single thread is responsible

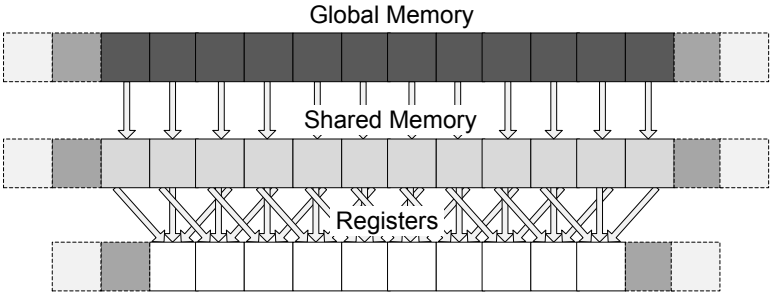


Figure 6.1: A simulated cache for primitives with spatial filter access. Threads cooperate to stage the elements of a region of memory into local, manually managed memory, from which overlapping accesses between multiple threads can be served with lower latency and higher bandwidth.

for computing an entire column of output data (the `__j` loop iterates along this column). The constant member variable *MultBy* is marshalled as a constant parameter to the kernel and the state variable *MovingSum* is moved to thread scope, so that each thread maintains unique state along its associated column. This phase constitutes well-known boilerplate work for constructing a CUDA program and does not require sophisticated analysis or metadata. We refer the reader to [NBGS08] for additional information.

6.1.2 Constant and Shared Memory Staging

A typical GPU memory system consists of a high bandwidth, high latency, large global memory space and very small, very high bandwidth, low latency shared memory and constant memory spaces. Large images are stored in global memory, small amounts of constant parameters (in our case constant class members) go into constant memory and shared memory is used as a working space. A typical functor with pointwise indexers has little need for shared memory, apart from the realignment optimisation discussed in Section 6.1.4. There is no benefit to loading pointwise-accessed data through shared memory and, indeed, the additional copy would reduce throughput.

However, with 1D and 2D spatial filter indexers there is substantial overlap between the data read by each thread. A 3x1 spatial filter region, for example, would carry information needed by three different threads. It would be very wasteful for each thread to issue a private global memory load; worse, the minimum transaction size is typically four data elements of which only one would be used. Instead, as illustrated in Figure 6.1, threads can coordinate to read common data elements exactly once from global memory into shared memory. From there the cost of redundant, overlapped loads into registers is much lower due to the higher bandwidth and lower latency of this region. Opportunities to improve performance through staging are clearly identifiable from Metadata 3.3.3. Static metadata defines the number of dimensions of a spatial filter region. Runtime metadata captures the configured size of these regions to adapt the staging process dynamically.

A one-dimensional CUDA shared memory staging function is given in Listing 6.2. The function takes a source address in global memory, a destination address in shared memory, the number of elements to be copied and a guard to test if the thread's logical workload lies off the edge of the image data. Each thread copies one or more elements, with the loop trying to spread the work

```
__device__ inline void Stage1D(const float *const from,
                               float *const to,
                               const int nElems,
                               const int guardX)
{
  for(int i = threadIdx.x;
          i < nElems && i < guardX;
          i += blockDim.x)
  {
    to[i] = from[i];
  }
  __syncthreads();
}
```

Listing 6.2: A 1D global-to-shared memory staging CUDA function. The *i* loop makes thread block-sized strides across the data to allow more data elements to be staged than there are threads.

evenly, and all threads reach a barrier to ensure that the whole copy has completed before any threads read from shared memory. This function does not typically carry the barrier instruction – it has been left here for clarity – so that several regions of memory may be staged together into different partitions with a single barrier in the calling code.

Shared memory (and constant memory) is, of course, a limited resource. When there are multiple regions of memory to be staged, the code generator must decide which to place into the available memory. Dynamic metadata establishes the precise amount of memory required per indexer. There is no guarantee that a kernel will use all of the staged data; legitimate algorithms, such as the discrete wavelet transform shown earlier in Listing 3.1, touch only a subset of an N-element region. While it is true that all of the staged data will be read by at least one thread, this may not be sufficient to overcome the additional cost of staging. This is really a flaw in our approximation of the DWT access pattern, and other sparse patterns, as spatial filters: it actually consists of just three scalar reads per iteration with left, centre and right offsets.

6.1.3 Memory Access Transposition

The coalesced memory transaction optimisation is predicated on having a contiguous, sequential mapping of data elements to thread IDs. As illustrated in Figure 6.2, memory accesses in a horizontal moving average primitive are physically disjoint. Those in the vertical moving average are contiguous and thus meet the requirements for coalescing. Metadata 3.3.2 (data dependence) allows the two cases to be distinguished statically. In practice, in the horizontal case it is often faster to simply spawn one thread per pixel, each initialising its state by reading the entire filter region beneath it, and to treat the kernel as if it had no dependence or state at all than to suffer the performance penalty of uncoalesced access. We use this approach in the unoptimised case because the higher thread count helps to hide the latency of uncoalesced access.

It is in the interests of performance to prefer vertical moving averages over their horizontal counterparts. A simple way to achieve this is to transpose the input data sets to a horizontal moving average functor, replacing it with an equivalent but faster vertical implementation, and

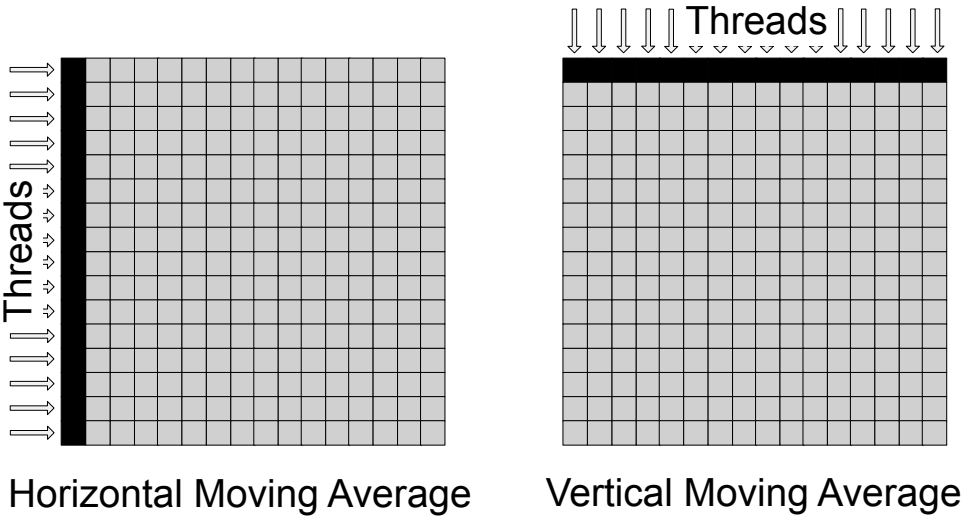


Figure 6.2: SIMT threads in a moving average execution. Memory accesses made in the horizontal moving average do not satisfy coalescing rules due to discontinuity.

then transpose the output data sets back. This would pay off if the total runtime of the three stages – transpose, execute, transpose – is less than the horizontal implementation's runtime. There is an opportunity for further optimisation here if the context of the kernel is known: Metadata 3.3.1 (DAG) provides this information dynamically. Sequences of horizontal moving averages, e.g. a series of box blurs making up the horizontal part of a Gaussian blur approximation, can be optimised by noting the redundant double transposition nodes at the boundaries between them. By eliminating pairs of transpositions at the graph level the effect's performance can be increased.

A similar problem arises in fully parallel primitives with vertical 1D spatial filter access patterns, identifiable from Metadata 3.3.1 and 3.3.3 (DAG and memory access). There is a trade-off between using a vertical thread block configuration (see Section 6.2) to stage the overlapped region into shared memory, suffering discontiguous column-wise global memory reads, or using a horizontal thread:work mapping to achieve coalescing but sharing no data at all. In practice we found that the latter was faster. However, transposing the entire primitive is a better option as it enables both coalescing and shared memory staging.

This optimisation tackles an instance of the data alignment problem [LC91]. The relative distribution of each input data set to a visual primitive affects the overall amenability of memory accesses to the coalescing optimisation. The impact of these distributions may extend beyond the primitive: each data set can be reused in different contexts, and the distribution of new data sets may depend upon the distribution of input data sets. An optimum distribution, minimising transpositions throughout the DAG, is a global optimisation problem inside the DAG, and potentially beyond it to the producers and consumers of its input and output data sets. The functional problem formulation may have advantages in this domain [DGTY95], but we have not yet explored this.

We explored an alternative, potentially faster solution to transposition which did not require explicit transpose nodes in the graph. This is illustrated in Figure 6.3. Transposition was carried out on 16x(16+2R) element blocks – where R is the spatial filter radius – with appropriately

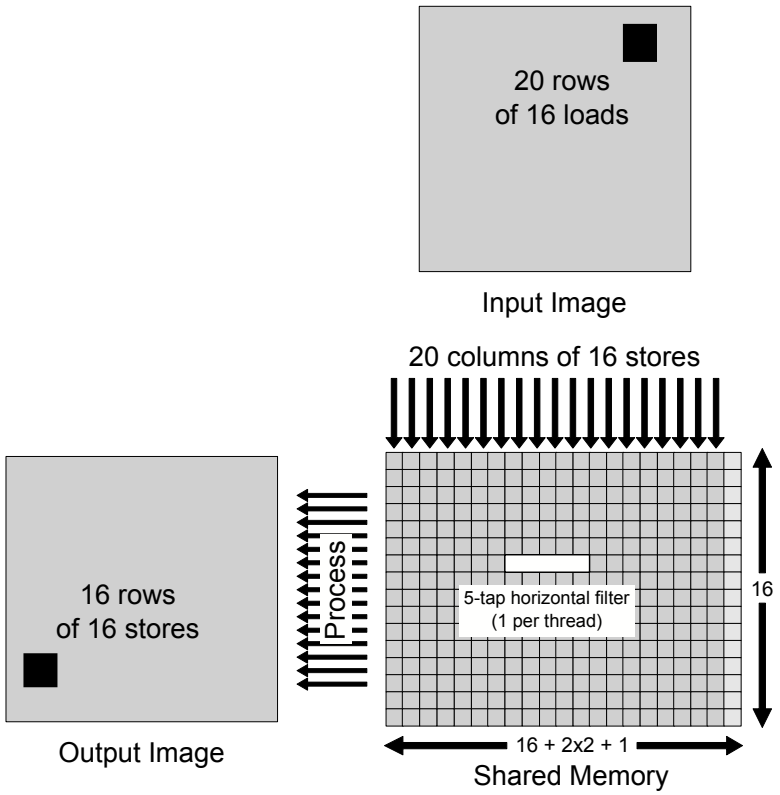


Figure 6.3: A scheme for memory access transposition without explicit transpose operations. Small blocks are transposed on the fly into shared memory, where coalescing is a non-issue, and processed by a horizontal filter implementation.

transposed indices, staging rows of a block into columns of shared memory. By padding the shared memory row stride to 17 elements, bank conflicts were avoided in both memory spaces. Furthermore, coalescing requirements were satisfied by the row-wise global memory loads. For reasons which were not entirely clear we were unable to improve upon the performance of the graph-level transpose solution with this method. We speculate that the additional shared memory and register requirements of this approach reduced parallelism to a degree which negated the gains from improved use of the memory system. Profiling tools, which were not available at the time of this experiment, could shed more light on this.

6.1.4 Memory Access Realignment

A second requirement for the coalescing optimisation is that the address accessed by thread 0 must be aligned to a specific multiple: typically 64 bytes in current implementations. CUDA implementations guarantee that the base address of allocated memory regions satisfy this alignment. However, there are two key sources of misalignment in our framework:

- **ROI smaller than producer DOD**. The DOD of a primitive is guaranteed to be equal in size to the primitive's output data sets. Thus, if these data sets satisfy the alignment requirement then the DOD will also meet this requirement. The ROI of an indexer in a consuming

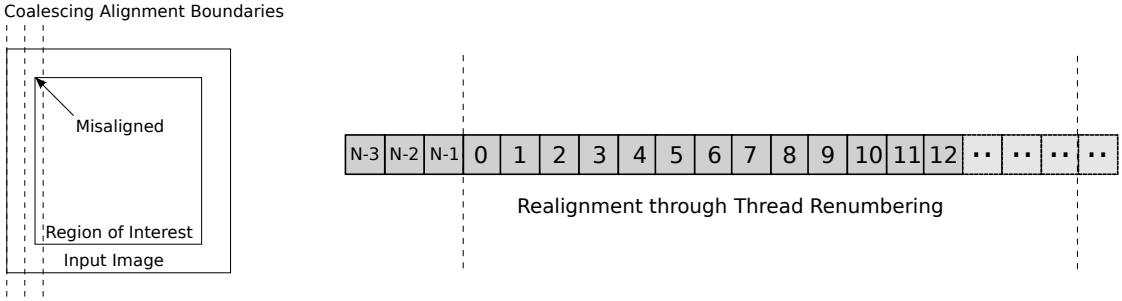


Figure 6.4: Global memory load/store misalignment occurs when the region of interest is smaller than the input image. Coalescing can be mostly restored by redistributing the thread:data mapping so that thread 0 is aligned on a boundary.

```
__device__ inline void Stage1DMisaligned(const float *const from,
                                         float *const to,
                                         const int nElems,
                                         const int guardX,
                                         const int alignBy)
{
  for(int i = (threadIdx.x + alignBy) & (blockIdx.x - 1);
          i < nElems && i < guardX;
          i += blockDim.x)
  {
    toBase[i] = fromBase[i];
  }
  __syncthreads();
}
```

Listing 6.3: A 1D global-to-shared memory, realigning staging CUDA function. This function implements the thread:data remapping scheme illustrated in Figure 6.2.

primitive, however, may be smaller than the data set it is reading from. Depending upon its precise size, the base address of the ROI may be (and likely will be) misaligned.

- **Horizontal 1D or 2D access**. Alignment can be ensured in vertical spatial filters by introducing small row-to-row padding factors when data sets are allocated. Alignment cannot be ensured with horizontal access freedom, however, because a kernel is free to request accesses with 4 byte offsets.

(DAG and memory access) Both of these cases can be identified from Metadata 3.3.1 and 3.3.3 (DAG and memory access). The latter case is already resolved through the staging optimisation discussed in Section 6.1.2, which always passes an aligned base address to the *Stage1D* function from Listing 6.2. Alignment can be partially restored in the former case through a staging transformation similar to that described in Section 6.1.2, but with a change in the thread:data mapping to ensure that thread 0's work is aligned. This process is illustrated in Figure 6.4 and a realigning staging function is shown in Listing 6.3.

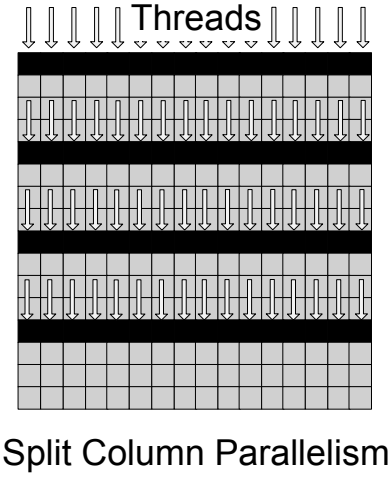


Figure 6.5: Serialisation in moving average (Definition 2.11) kernels leads to underutilisation of SIMT parallel resources. Parallelism can be enhanced by splitting serialised rows/columns into parallel groups of serialised sub-rows/columns, exploiting the explicit roll-up function in the visual primitive front-end.

6.1.5 Maximising Parallelism

A typical modern GPU keeps tens of thousands of threads in flight in order to tolerate latencies of hundreds of cycles in the high bandwidth, high latency global memory system. Physical computational parallelism is measurable in hundreds of threads. Satisfying the parallelism demands of these devices in visual effects should be easy: each image typically consists of millions of pixels. For fully parallel algorithms this is as simple as associating each thread with a single output pixel.

However, consider the moving average algorithm shown in Figure 6.2. The loop-carried dependence (in either axis) serialises entire rows or columns of pixel computations. Parallelism is reduced from the order of millions of threads to only a few thousand for large images. This is insufficient to satiate the demands of a modern GPU and will lead to underused memory load/store and computational issue slots.

There is a way to create more parallelism. A moving average consists of a state initialisation function and a kernel, as illustrated by the box blur example from Listing 3.2. The initialisation function can be called from any location in the image to build up state for the next run of pixels. By splitting the serialised row or column into several pieces, as illustrated in Figure 6.5, we can double, triple, etc. the amount of parallelism available to us. As long as the initialisation function is run once in each thread the results will be correct. There is a small overhead in recomputation at the split row/column boundaries but this is dominated by improvements in processor thread occupancy.

By looking at Metadata 3.3.2 (data dependence) we can locate primitives with moving average dependence and the dynamically serial axis to identify functors with poor parallelism. Using the dependence guarantees provided by Metadata 3.3.2, the split row/column transformation can be applied with no modifications to the kernel whatsoever. By simply redistributing work:thread mappings and increasing the number of threads the transformation is completed.

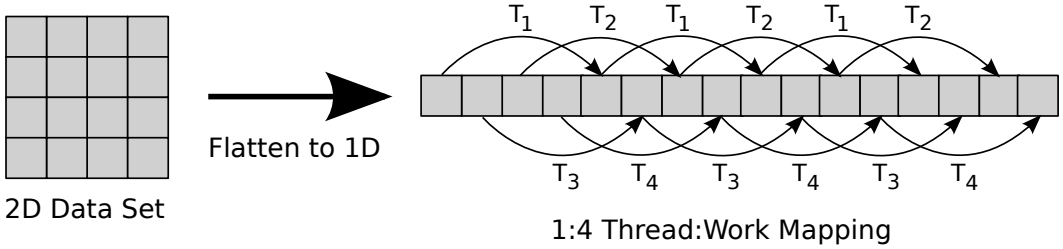


Figure 6.6: A coalescing-safe work distribution method to assign multiple work items to each thread. This is useful in simple kernels where the scheduling costs of high thread counts can dominate the computational work being done. Increased register usage to manage the mapping operation typically has no impact on parallelism due to the low register requirements of the kernel.

6.1.6 Scheduling Overhead Reduction

A 1:1 mapping of output work items to threads leads to considerable overhead in the hardware scheduling system. This is especially significant in simpler primitives in which the runtime of the kernel is of comparable magnitude to this overhead. A simple way to alleviate this overhead is by using an N:1 mapping of output work items to threads. Fewer threads can be spawned with this method and, as a result, the cost of SIMT scheduling is reduced. Any mapping of work to threads can be used but we chose the mapping illustrated in Figure 6.6 because it preserves the requirements for coalescing to take place. Threads form an interleaved work pattern, as in a 1:1 mapping of work:threads, but continue to take additional elements of data by striding over the work being carried out by in-flight threads. Implementing this mapping carries a small computational cost and requires additional registers; however, the kernels typically targeted for this optimisation are already underutilising computational and register resources.

Listing 6.4 shows a sample of generated code for a two-operand summation kernel, following a transformation to implement an N:1 thread:work mapping. The value of N is dynamic and selected by the parallelism configuration process discussed in Section 6.2. We identify primitives which might benefit from this transformation by looking for primitives with only pointwise indexer access patterns, from static Metadata 3.3.3 (memory access). Our rationale is that we cannot know true runtime costs without substantial static analysis or a dynamic feedback mechanism. However, long runtimes typically arise from iteration across 1D or 2D regions of local data. It is important to avoid applying this transformation indiscriminately because the increase in register requirements can lead to reduced parallelism in kernels with high register counts, which is likely to dominate any gains from scheduling optimisation.

6.2 Thread Block Size, Shape and Count Selection

Once an appropriately optimised SIMT kernel has been generated, marshalling code is also generated to manage data movement and computation. An important decision made by this code is the configuration of parallelism for each kernel. The configuration of a SIMT kernel is parameterised by thread block and grid configuration [NBGS08]. Choosing a thread block and grid configuration

```
__global__ void Add(float *Input1, float *Input2, float *Output, int nElems)
{
  for(int _off = __mul24(blockIdx.x, blockDim.x) + threadIdx.x;
          _off < _nElems;
          _off += __mul24(gridDim.x, blockDim.x))
  {
    Output[_off] = Input1[_off] + Input2[_off];
  }
}
```

Listing 6.4: A summation kernel following the N:1 work:thread mapping optimisation described in Section 6.1.6. The kernel strides over the data set in grid-sized steps.

can drastically alter performance. These factors can even determine whether a program can run at all under the resource constraints of a given device. It is worth beginning by noting the hard limits constraining our choices of thread block size, shape and count. CUDA specifies five relevant parameters in each version of its Compute Capability (CC). T_{max} is the maximum number of threads per thread block. T_{mpm} is the maximum number of threads that can be managed concurrently on a multiprocessor. R_{max} is the total number of registers available to a multiprocessor. S_{max} is the number of bytes of shared memory per multiprocessor. Our study focuses on CC 1.0, but note that our selection techniques scale to newer versions by simply using different parameterisations of these factors. For CC 1.0 the parameters are:

$$T_{max} = 512, T_{mpm} = 768, R_{max} = 8192, S_{max} = 16000 \tag{6.1}$$

For CC 1.3 the parameters are:

$$T_{max} = 512, T_{mpm} = 1024, R_{max} = 16384, S_{max} = 16000 \tag{6.2}$$

T_{max} is a hard limit on the number of threads per block, but we are additionally constrained by the availability of registers (R_{max}) for those threads to use and by the shared memory requirements of a particular kernel. Focusing just on the register requirement for now, the maximum number of threads per block T_{pb} for a given kernel is related to the number of registers the kernel uses R_{pt}. The latter value can be extracted from the compiled kernel metadata. Equation 6.3 relates these two factors. The floor notation $\lfloor a,b \rfloor$ denotes a rounded down to the next integer multiple of b. This equation is undocumented but used in the CUDA Occupancy Calculator spreadsheet. It has value in choosing optimal thread block configurations and thus we have listed it. T_{phw} is the number of threads per half-warp, 16 for all current versions of CC.

$$T_{pb} = min(T_{max}, \left\lfloor \frac{R_{max}}{T_{phw} \times R_{pt}}, 4 \right\rfloor \times T_{phw}) \tag{6.3}$$

Per-kernel shared memory requirements are a little harder to formalise. Each kernel uses a fixed amount of shared memory for parameters and internal use S_{pk}, advertised in the compiled kernel metadata. The remaining shared memory consumption arises from the staging optimisations

discussed in Section 6.1.2. These can be broadly classified as shared memory per-thread S_{pt} and a constant amount per-block independent of the block size S_{cpb}. Thus the maximum number of threads per block for a given kernel is additionally constrained by Equation 6.4.

$$S_{pk} + S_{cpb} + S_{pt} \times T_{pb} \leq S_{max} \tag{6.4}$$

The factors discussed so far are hard limits. In fact, performance can often be increased by substantially undercutting them. By reducing T_{pb} the shared memory S_{max} can be divided amongst multiple blocks. Additionally, CC 1.0 specifies a maximum number of threads per multiprocessor T_{mpm} of 768. A T_{pb} of 512 could not fully saturate a multiprocessor on this architecture because there is an insufficient number of free threads to process a second block on the multiprocessor. In this case reducing T_{pb} to 256 may permit three blocks to run on the multiprocessor – if shared memory and register constraints are satisfied – potentially increasing performance further than the 512 thread block could have through its more efficient intra-block communication.

Once T_{pb} has been chosen it must be virtualised into a two-dimensional thread block $T_{pbx} \times T_{pby}$. This could be a horizontal or vertical line, a square or something in-between. The shape of a thread block primarily affects the ratio of information shared amongst threads to the incommunicable but logically shared information at the boundaries of thread blocks. Maximising this ratio requires different shapes in different cases. Additionally, the shape of a thread block may be constrained by dependence in the horizontal or vertical axis.

$T_{pbx} = T_{pb}$, $T_{pby} = 1$ (a horizontal line) is required by vertical moving average functors and the scheduling overhead reduced functors described in Section 6.1.6. It is also used to maximise shared information in horizontal 1D spatial filter indexers: the shared information is T_{pb} elements and incommunicable information is only (*2 x radius*) elements. On the other hand, $T_{pbx} = 1$, $T_{pby} = T_{pb}$ (a vertical line) is required by horizontal moving average functors. Note that we expect vertical one-dimensional filter indexers to have been transposed by this point, thus they would also use a horizontal thread block.

$T_{pbx} = \lfloor\sqrt{T_{pb}}\rfloor$, $T_{pby} = \lfloor\sqrt{T_{pb}}\rfloor$ is a compromise of the two approaches. It is used for two-dimensional spatial filters and for mixes of 1D horizontal and vertical filters. It is suboptimal in both axes but maximises the shared data ratio in the 2D filter. Rectangular variations can be used when the horizontal and vertical filter radii differ.

Once a thread block size has been chosen, the grid size $B_{pgx} \times B_{pgy}$ is fixed. In most cases this is given by $B_{pgx} = \left\lceil \frac{dodWidth}{T_{pbx}} \right\rceil$, $B_{pgy} = \left\lceil \frac{dodHeight}{T_{pby}} \right\rceil$. Scheduling overhead reduced kernels are a special case where we set $B_{pgx} = 160$, $B_{pgy} = 1$. This block count is tuned experimentally: it is small to minimise block management overheads but large enough to keep all multiprocessors occupied at reasonable thread block sizes.

6.3 Challenges in SIMT Space/Schedule Optimisation

Chapter 4 outlined a cross-component optimisation which reduces data reuse distance and working set size in a visual effect. The optimisation was designed for scalar CPU ISAs but in Section 5.1.5

we described a simple adaptation to vector CPU ISAs. In both cases, large performance gains were observed on an effect consisting of a large DAG of simple pointwise and 1D spatial filter primitives. In principle, the benefits of this optimisation carry across to SIMT devices as well. A number of challenges, however, make this transition harder than for SIMD devices:

- **Limited cache space**. Current generation SIMT architectures define a small piece of manually managed, high bandwidth, low latency shared memory. This is typically 16KB per multiprocessor on an NVIDIA GPU. Compared with caches exceeding 10MB per multiprocessor on modern multicore CPUs, the space available to fit contracted working sets on a SIMT device is much smaller. Indeed, this is by design: SIMT devices are intended to hide latency through massive multithreading. However, the bandwidth to shared memory is considerably higher than to global memory and, as a result, array contraction remains a potentially useful memory optimisation.

- **No inter-block communication**. Shared memory space is local to each multiprocessor and there are no explicit communication language constructs to pass data between them. The contracted working sets of fused primitives with 1D and 2D access patterns cannot be segmented without synchronisation or communication because threads must read subregions of the working set – computed by multiple threads from different multiprocessors – which span segmentation boundaries.

- **Resource/parallelism trade-off**. The performance of a SIMT kernel is closely linked to the level of parallelism that can be achieved. Parallelism is constrained by the availability of registers and shared memory, which must be divided between all of the in-flight threads of a multiprocessor. Fusion increases the shared memory and register requirements of a kernel, which may lead to degraded performance when these additional resources are unavailable. This situation differs from a CPU, which has a greater availability of cache space to spill registers into and to store the contracted working set in.

In spite of these limitations, space/schedule optimisation remains potentially useful on SIMT devices for fusing pointwise primitives – i.e. those with no 1D or 2D access patterns (Metadata 3.3.3). For such primitives, contraction can take place entirely within registers local to each thread, since each parallel working set remains associated with a single thread. We do not explore this option in our performance analysis, since we were seeking a uniform solution which incorporated primitives with 1D and 2D access patterns to support more interesting classes of effects. In unpublished work we identified large performance gains with a specialisation of space/schedule optimisation to pointwise primitives only, but do not make use of this in our evaluation.

Tackling contraction in primitives with 1D and 2D access patterns is not inconceivable and may yet yield attractive performance gains. We propose two potential solutions to this problem:

- **Partial fusion**. The lack of a facility to communicate contracted data at borders of thread blocks is the primary barrier to fusion. This problem may not affect the entirety of the

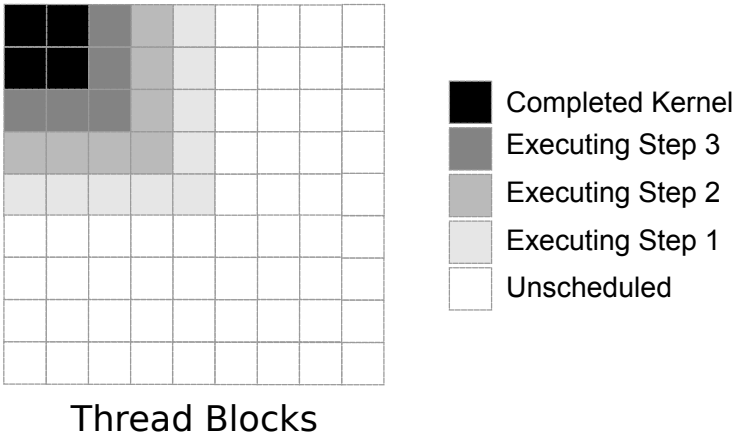


Figure 6.7: A proposed wavefront scheduling method for executing space/schedule optimised effects on a SIMT architecture with hardware support for inter-block shared memory communication. The first step of the fused kernel executes in parallel across all thread blocks. The second step does not begin in each block until its neighbouring blocks have completed the first step and synchronised to exchange computed data.

thread block, however. With a large block. comparatively small spatial filter radii and short fusion chains, a substantial percentage of each thread block does not need to communicate with its neighbours. It might be beneficial to compute these regions with fused/contracted code and then complete the remaining elements with an unfused pass.

- **Wavefront parallelism**. If a mechanism for inter-block thread communication were to be made available, the need for synchronisation would still limit the application of this optimisation. It would be necessary to barrier the entire computation between fusion boundaries within the fused kernel to exchange computed border data between thread blocks. The strategy we used for CPU multicore parallelism – recomputing the missing border data on each core – would be impractical given the scale of parallelism required by SIMT. One option would be to execute thread blocks in a wavefront pattern, as illustrated in Figure 6.7. The first step of the fused kernel is executed by all thread blocks in parallel. The second step in each thread block does not begin until the first step of all the blocks which provide computed data to that block has been completed. This requires synchronisation with neighbouring blocks only or a larger set depending upon the spatial filter radius. If the computation is scheduled from the bottom-left of the data set this creates a logical *wavefront* along which the computation progresses.

6.4 Performance Analysis

The SIMT research in this chapter was targeted at NVIDIA's CUDA architecture with Compute Capability (CC) 1.0. This limited our experimentation to the GeForce 8800 GTX graphics card throughout most of this work. In the final stages of research NVIDIA announced new GPUs with different performance characteristics and new Compute Capability. Although we did not focus our

Name	RAM (MB)	Cores	Mem Bandwidth (GB/s)	Compute Capability
GeForce 8800 GTX	768	128	86	1.0
GeForce GTX 260	896	216	112	1.3
Tesla C1060	4096	240	102	1.3

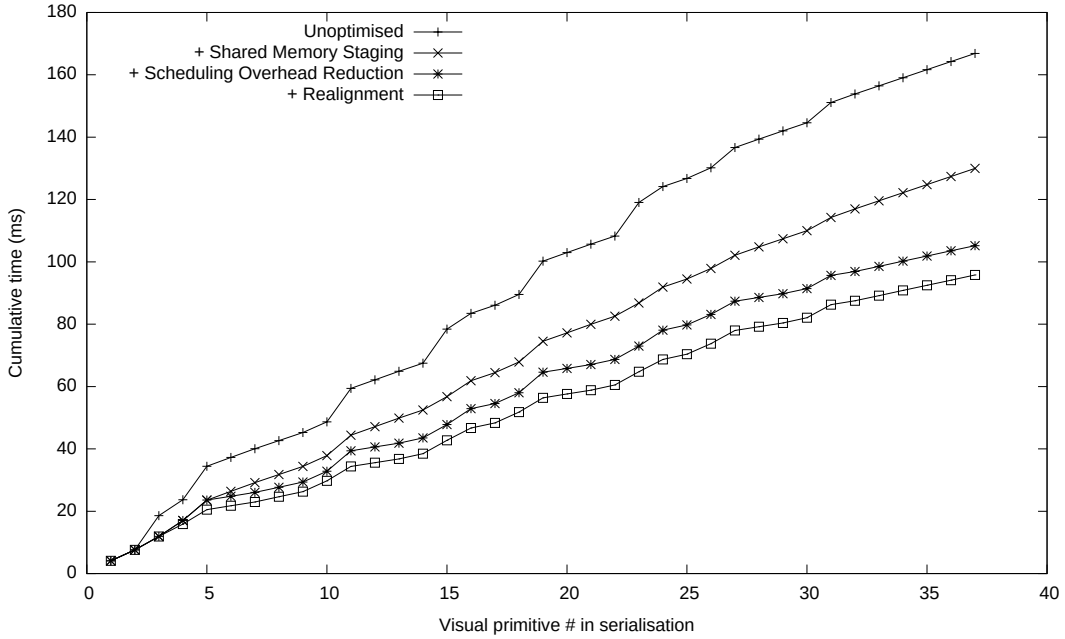
Table 6.1: Three NVIDIA GPUs based upon the CUDA architecture used in our performance evaluation. Our optimisations were developed for the GeForce 8800 GTX but we evaluate them on two newer devices as well.

optimisations on these devices, in this section we will evaluate their effectiveness on these newer generation devices. Our evaluation focuses on the three NVIDIA GPUs listed in Table 6.1. The two newer devices have CC 1.3, a substantial increase in processor cores and small improvements in memory bandwidth. Their larger memory capacities do not impact our evaluation.

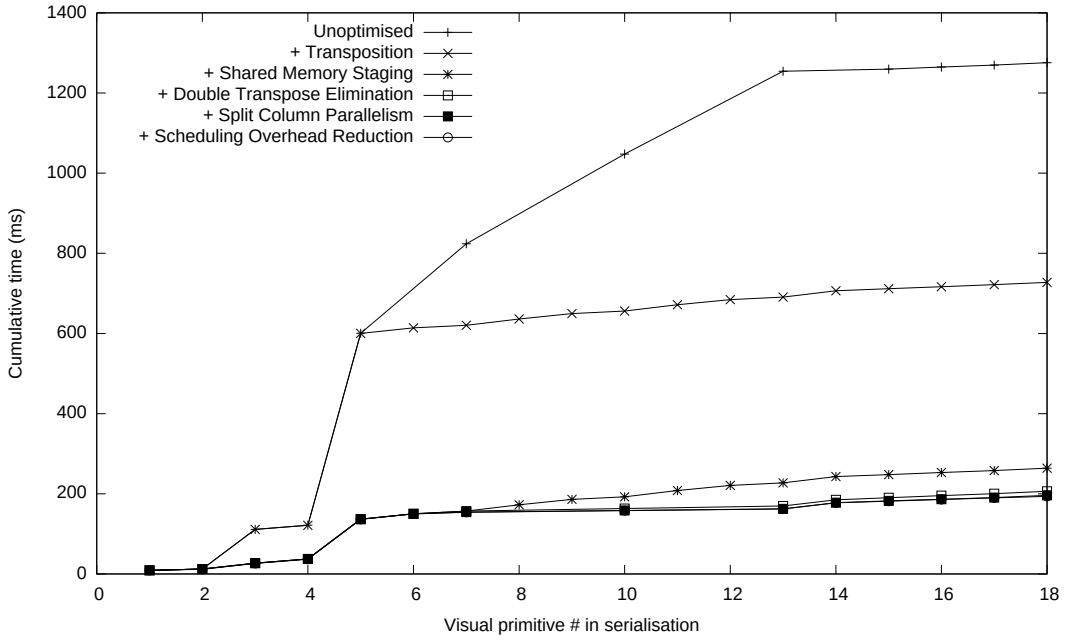
Figure 6.8a graphs the cumulative execution time of the automatically generated wavelet-based degraining effect on a GeForce 8800 GTX for a 2063x1545x3 (prime stride) 32-bit floating-point image. The effect DAG is serialised per the discussion in Section 3.5.1 and the cumulative execution time at each primitive in the serialisation is plotted. The uppermost line represents our best generated implementation without the optimisations described in Section 6.1. Optimisations are applied cumulatively to form successively lower (and faster) plots. The primitives constituting the degraining effect have largely uniform execution times, with the exception of vertical 1D filters causing small steps in the graph. The shared memory staging optimisation (Section 6.1.2) flattens these steps by localising thread-shared data in spatial filters and by improving coalescing in the vertical case. Setting up an N:1 work:thread mapping (Section 6.1.6) reduces scheduling overhead in sets of pointwise primitives (seen here in groups of three). Finally, the realignment optimisation (Section 6.1.4) corrects image subregion ROI access in the complex DAG to improve coalescing. Neither memory access transposition (Section 6.1.3) nor split-row/column parallelism (Section 6.1.5) were necessary because the effect does not contain moving average primitives. We observe an overall performance improvement of 1.7x on this hardware.

Figure 6.8b presents a similar graph for the diffusion filtering effect on a GeForce 8800 GTX for a 3072x2304x3 32-bit floating-point image. Here, the effect's performance is dominated by primitives 5–8 and this manifests itself as a large rise in cumulative execution time in the graph. There are fewer primitives in total, compared with the degraining effect, because the diffusion filtering DAG (see Figure 3.2) is smaller. The most significant optimisation for this effect is memory access transposition (Section 6.1.3), which works to coalesce the memory accesses in vertical moving average primitives 6–8. Shared memory staging (Section 6.1.2) localises thread-shared data in 2D spatial filter primitive 5 and improves coalescing. Other optimisations had a less significant impact on this effect: DAG-level multiple transpose node elimination (Section 6.1.3) helped to reduce the costs of memory access transposition a little. Overall, we achieved a 6.6x speed-up on this hardware.

Figure 6.9a graphs the performance of degraining on a previously untested device: the GeForce GTX 260. As part of an architectural improvement over the previous generation of GPUs, the costs of uncoalesced access are significantly reduced on this device. We would expect to see a reduc-



(a) Wavelet-based degraining



(b) Diffusion filtering

Figure 6.8: Cumulative execution time of the wavelet-based degraining and diffusion filtering effects on a fixed-size single-precision floating-point images in CUDA on an 8800 GTX (CC 1.0). Kernel execution times are cumulative across the DAG serialisation and optimisations are applied incrementally down the graph to improve performance. Gaps indicate transpose primitives that are not present in a particular composition of optimisations.

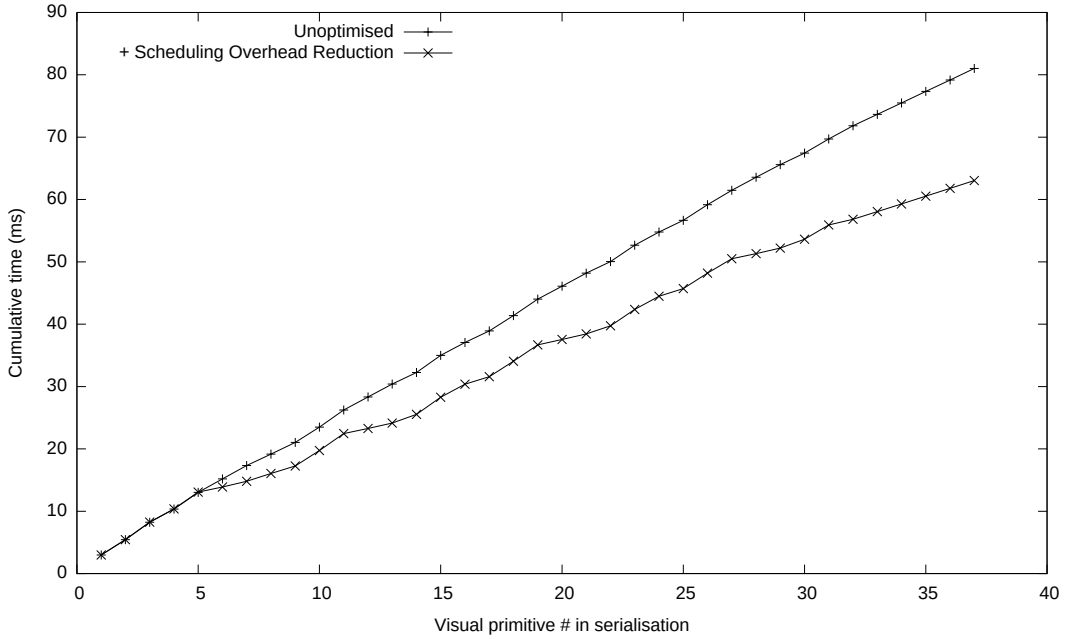
tion in performance gains from memory-oriented optimisations. Two such optimisations: shared memory staging and realignment showed no improvement at all and are omitted from the graph. The only optimisation to have any effect in this example was scheduling overhead reduction (Section 6.1.6), which gave a modest 1.3x speed-up. In contrast, the diffusion filtering effect on the GeForce GTX 260, shown in Figure 6.9b, continued to benefit from the same optimisation set as the older architecture. The gains were markedly reduced owing to the lower cost of uncoalesced access on this architecture. We observe a 1.9x speed-up with this effect.

From these results we can deduce that the architectural changes which modern GPUs are undergoing will have a significant impact on attempts to develop useful optimisations for them. We have not yet had the opportunity to carry out detailed profiling analyses on the code generated by our compiler for GPUs newer than the 8800 GTX. Our framework is well-placed to implement new metadata-supported optimisations that may prove important on these devices.

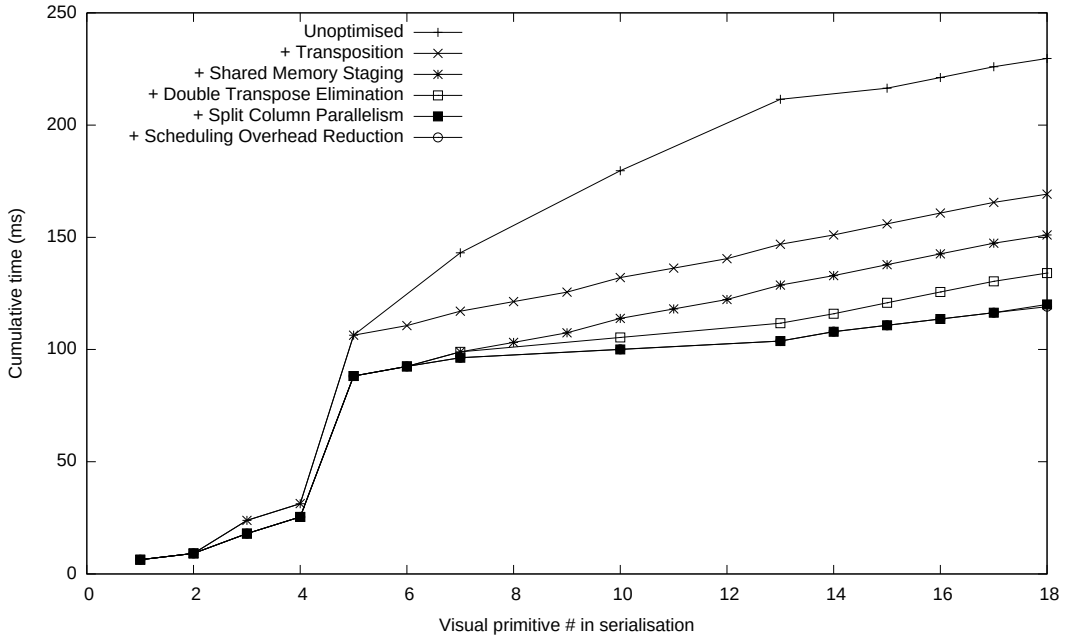
Before comparing performance of the SIMT code with the CPU-based SIMD generated code described in Chapter 5, we present a summary of the contribution of each optimisation described in this chapter in Figure 6.10. An additional GPU – the workstation-oriented Tesla C1060 – has been evaluated and added to this data set. As discussed in the preceding paragraphs, our optimisations had only a modest effect on the wavelet-based degraining effect (left three bars). Scheduling reduction overhead (Section 6.1.6) had the most significant effect across all three GPUs. Shared memory staging (Section 6.1.2) and memory access realignment (Section 6.1.4) worked well on the 8800 GTX but had limited impact on the newer cards, which exhibit less costly uncoalesced memory access. As such, our optimisations had limited effect on the two newer GPUs.

Most of the optimisations discussed in this chapter proved beneficial on the diffusion filtering effect, as shown by the right three bars of Figure 6.10. Memory access transposition (Section 6.1.3) and shared memory staging (Section 6.1.2) had the largest effects, particularly on the first generation 8800 GTX GPU. Split row/column parallelism (Section 6.1.5) had a useful effect while the remaining optimisations made smaller contributions on all devices. Interestingly, the workstation class Tesla C1060 benefitted substantially more than the similar generation GTX 260. The data offers no clear explanation for this effect but we speculate that the weaker computation:memory access ratio (see Table 6.1) of this device lends itself better to memory optimisation.

Finally, we present a comparison of the GPU implementation with space/schedule optimised SIMD implementations from Chapter 5. Figure 6.11a shows the now familiar throughput graph as input image size is varied on the X axis. We have omitted price:performance data as these variables change rapidly over time. However, for a reference comparison the most powerful CPU configuration (2x Xeon E5420) was similarly priced to the Tesla C1060 at the time of writing. The two higher end GPUs exhibit an interesting rise in throughput as the image size grows to 2 MPixels – images below this size are unable to saturate the computational units of these two models – but are otherwise fairly stable. A comparison between the upper three GPU plots and the lower three CPU plots shows perhaps less distinction than one might hope for. In fact, the CPU implementations have benefitted greatly from the space/schedule optimisation (Chapter 4) which we have not yet ported to the GPU, for the reasons outlined in Section 6.3. The amount by



(a) Wavelet-based degraining



(b) Diffusion filtering

Figure 6.9: Cumulative execution time of the wavelet-based degraining and diffusion filtering effects on a fixed-size single-precision floating-point image in CUDA on a GTX 260 (CC 1.3). Kernel execution times are cumulative across the DAG serialisation and optimisations are applied incrementally down the graph to improve performance. Gaps indicate transpose primitives that are not present in a particular composition of optimisations.

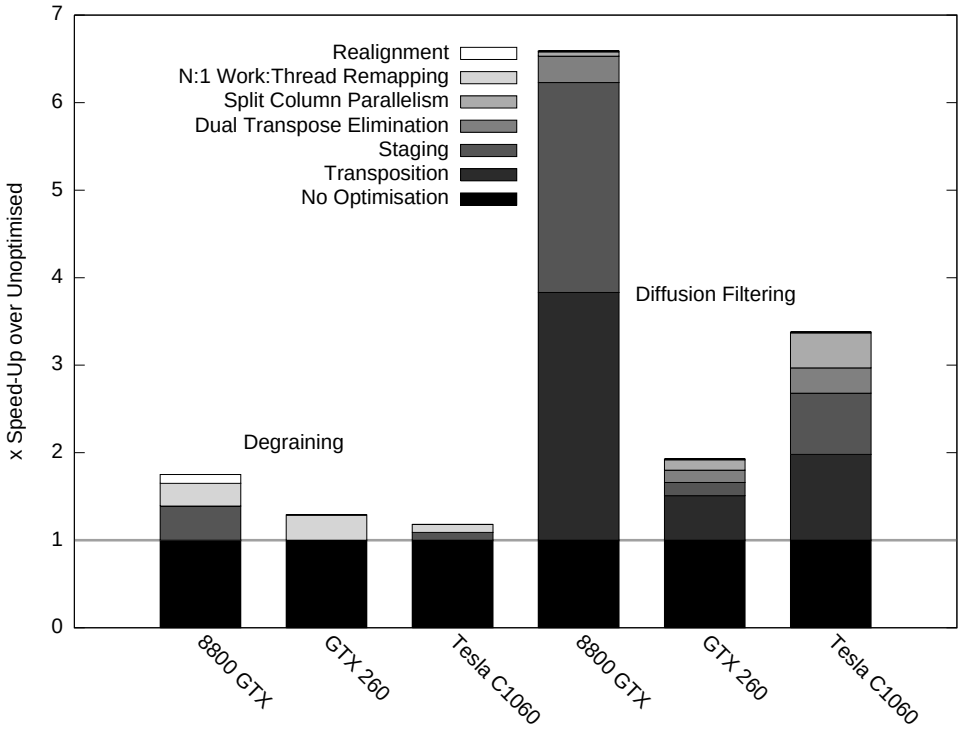


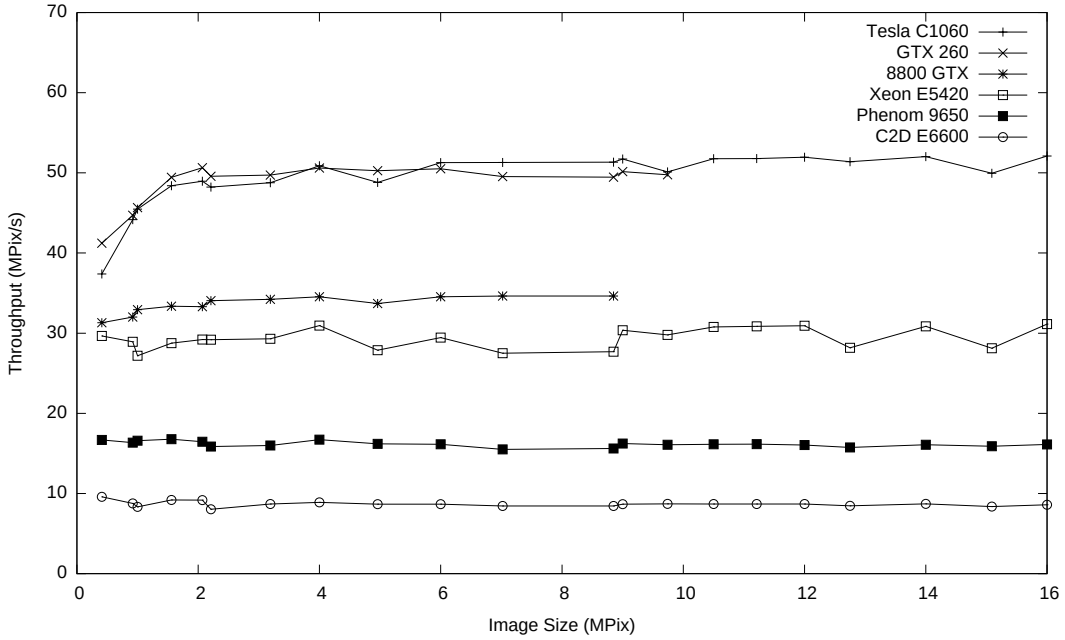
Figure 6.10: Relative speed-ups (1x = no speed-up) of the wavelet-based degraining and diffusion filtering effects on each GPU benchmarking platform for a fixed input image size of 12 MPixels, with a fractional breakdown of the contributions of each optimisation phase outlined in Section 6.1

which the two high-end GPUs are bounded by memory is evident in their similar plots; despite a 15% difference in computational capacity. In terms of cost-effectiveness the highest end GPU still maintains a $\sim$3x benefit over the highest end CPU.

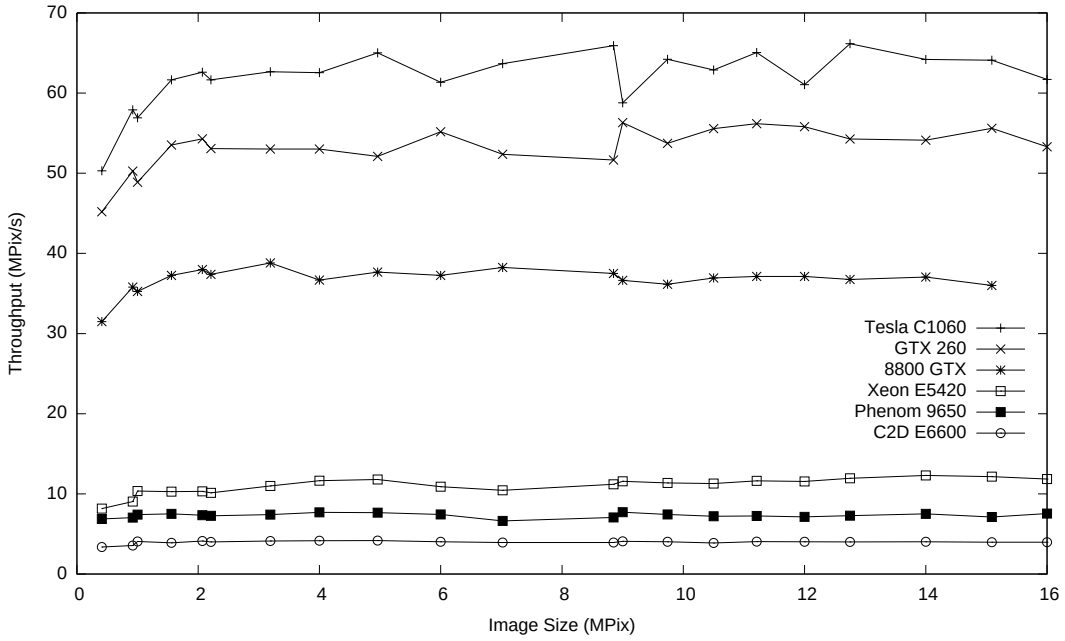
The CPU/GPU performance divide is more substantial in the diffusion filtering effect, as shown in Figure 6.11b. Here, the GPU's high memory bandwidth proves to be an effective brute force solution to the slow 2D spatial filter primitive contained within the effect. This leads to a 5.3x throughput divide between the highest end GPU and CPU. The plots of each GPU are more widely distributed than in the wavelet-based degraining effect, indicating that the devices are not strongly memory bound with this effect. Greater availability of registers in CC 1.3 helps to saturate all of the compute cores in the two 1.3 devices for the complex, register-intensive 2D spatial filter primitive. There is a curious mirror effect in the plots of the two 1.3 devices for image sizes between 4 and 11 MPixels. This effect was stable and repeatable but we are unable to offer a reasonable explanation for it and propose further investigation with profiling tools.

6.5 Concluding Remarks

In this chapter, we described an application of program metadata to automatic code generation for SIMT ISAs, such as NVIDIA's CUDA, from the visual primitive representation. The future of high throughput computational accelerators lies in massively multithreaded architectures,



(a) Wavelet-based degraining



(b) Diffusion filtering

Figure 6.11: Throughput of the wavelet-based degraining and diffusion filtering visual effects with a SIMT implementation of all constituent visual primitives, with all optimisations described in Section 6.1, on each of three GPU benchmarking platforms. Y-axis throughput measures the number of output pixels generated per second with variation of the input image size on the X-axis.

which we are able to exploit through the exposure of large scale parallelism in our visual primitive representation. We are able to sidestep complex data dependence analyses, as in Chapter 5, by exploiting static metadata describing inter-iteration dependence. High-level transformations are leveraged to fragment serialised algorithms when the primitive exposes insufficient parallelism to run effectively on a SIMT device. Several optimisations are employed to improve efficiency in memory access, by adding explicit statements to stage shared information into a small cache and by rescheduling global memory accesses to respect the requirements for cross-thread transaction merging (coalescing), making strong use of memory access metadata. We identified several challenges in exploiting the space/schedule optimisation on SIMT architectures and made proposals to address this situation in the future.

SIMT code generation worked best when the effect was not dominated by inter-primitive communication. Our optimisations achieved only 1.3–1.7x speed-ups on the wavelet-based degraining effect, which requires space/schedule optimisation to use the memory system efficiently. These optimisations were more effective on the diffusion filtering effect, which is bounded by a slow 2D spatial filter primitive, achieving 1.9–6.6x speed-ups through better use of the memory system and exploitation of additional parallelism. The divide between the CC 1.0 generation of GPUs, which this research focused upon, and the newer CC 1.3 GPUs proved to be quite substantial. This was mainly due to improvements in the memory system which reduced the cost of uncoalesced memory access, making some of our optimisations less useful. Further research is needed to identify the performance limiting factors of these newer devices in order to better target optimisations for them. Finally, a comparison between space/schedule optimised SIMD throughput and SIMT throughput identified 3.2–5.3x improved performance with the highest end GPU over the highest end CPU. Both were priced similarly at the time of writing.

Chapter 7

Conclusions and Further Work

This chapter concludes our study with a review of the research contributions and the successes and lessons learned throughout the work described in this thesis. A discussion of proposals for future work follows, outlining key threads of research to tackle new challenges identified throughout this thesis. Finally, we conclude the thesis with an analysis of the tangible outcomes of our continued partnership with commercial VFX developers The Foundry.

7.1 Review of Objectives

From the objectives outlined in Section 1.2:

- *To devise a single-source modular representation of a visual effect which closely resembles the algorithm structure and integrates seamlessly with the development workflow.*

  Sections 3.4.1 and 3.4.2 describe a reusable component-based representation which exploits the decomposable structure of visual effects into graphs of kernels within skeleton abstractions. Section 3.4 describes a novel hybrid static/dynamic code generation toolchain which exploits the active library paradigm to minimise workflow interference.

- *To develop a methodology for automatically deriving SIMD and SIMT parallel implementations from the scalar single-source representation.*

  Section 3.3 outlines three key metadata augmentations to the effect representation which enable analysis-free parallelisation and related memory optimisations. Sections 5.1 and 6.1 describe in detail compiler-driven methods for exploiting this metadata to construct optimised SIMD and SIMT implementations.

- *To identify performance-critical optimisations for these implementations and to devise a strategy for automatic optimisation of known and previously untested visual effects.*

  Chapter 4 discusses a compiler-driven performance-critical memory optimisation for CPUs

and Sections 5.1.5 and 6.3 describe its application to SIMD ISAs and to (potentially) GPUs. Sections 5.1 and 6.1 describe several compiler-driven device-specific optimisation phases. Each leverages metadata to optimise any primitive which fits into the skeleton abstraction.

- *To evaluate the efficacy of these ideas on industrial VFX software.*

  Sections 3.7, 4.4, 5.2 and 6.4 document empirical analyses of each code generation process and optimisation for two industrial visual effects on a set of CPUs and GPUs.

7.2 Technical Achievements

- We presented an adaptation of space and schedule optimisation to the dynamic working sets and schedules of a runtime-parameterisable visual effect in Chapter 4. By fusing polyhedral schedule transformation with static metadata encoded in the visual effect representation and dynamic metadata captured from the effect parameterisation at runtime, we were able to create an optimal analysis-free space and schedule optimisation engine. We exploited three properties to maintain the tractability of the polyhedral scanning process: symmetricity in spatial filters, parameter constraints for the common case and redundancy in spatial filter sizes both between and within primitives. By isolating fused schedule generation to an offline code generation process with minimal runtime parameterisation, to tune the schedule to a given effect configuration, we were able to eliminate the cost of schedule generation completely. This came only at the expense of unspecialised code: a minor concern given the large ranges of parameters that would need to be generated by runtime static specialisation.

- We presented a set of analysis-free transformations on a scalar data-parallel kernel to produce optimised non-scatter/gather SIMD vector-parallel implementations in Chapter 5. A combination of static and dynamic metadata provided sufficient information about data dependence and memory access patterns to guide SIMD vectorisation and optimisation. Our vectorisation techniques succeeded on many kernels on which the Intel C/C++ vendor compiler failed. We found that SIMD code generation was most effective when coupled with the memory bandwidth-enhancing space/schedule optimisation. The composite of these optimisations required a novel modification of the array contraction transformation to support the coarser granularity of vector addressing in a non-scatter/gather architecture.

- We presented a set of analysis-free transformations on a scalar data-parallel kernel to produce optimised SIMT massively parallel implementations in Chapter 6. Static and dynamic metadata were again used as a substitute for program analysis in the SIMT code generation process, demonstrating its scalability from SIMD to the very different challenges of a SIMT architecture. An analysis of our optimisations on two newer generation devices, which we had not previously studied, showed that some of our optimisations were no longer necessary; others continued to deliver strong speed-ups. The architectures of SIMT devices are evolving rapidly at this time. The parallel structure of a SIMT architecture proved to be a

formidable challenge in composing the space/schedule optimisation: the principles still apply, but the cache space is much smaller, synchronisation is more complex and the infancy of the CUDA compiler led to parallelism-reducing rises in per-thread register consumption. We were unable to reconcile these two optimisations by the time our research concluded.

- We presented an experimental evaluation of the first three contributions on two industrially developed visual effects algorithms in the penultimate sections of Chapters 4, 5 and 6. The two effects presented unique performance challenges and consequently responded in different ways to our optimisations. The sprawling DAG of simple primitives constituting wavelet-based degraining responded well to space/schedule optimisation, SIMD code generation and, to some degree, SIMT code generation. The smaller DAG of diffusion filtering, bounded primarily by a single complex primitive, saw much smaller gains from CPU-based optimisations. A SIMT implementation, however, showed large improvements by making much more bandwidth available to the effect. Our speed-ups ranged from 1.5–8.1x (mean: 3.8x) on CPUs and 1.2–6.6x (mean: 2.7x) on GPUs.

7.3 Critical Analysis

With the benefit of hindsight, we can now address the relative merits and weaknesses of the research documented in this thesis. As with all research, there are aspects which we would liked to have tackled differently. Similarly, there are elements of the work which were spontaneous and worked out better than we could have hoped. Here are some of the key aspects for criticism:

- **Optimisation efficacy**. Establishing a set of important optimisations in the VFX domain was a key goal throughout our research. Strong performance gains proved to be elusive in the early phases, even once the theory for each optimisation had been established. However, perseverance in experimentation eventually led to a step change in performance: the speeds established are capable of making previously non-realtime effects run in realtime. We were unfortunate to begin our SIMT research at a turbulent point in GPU architectural development, which eroded some of our gains on the newer GPUs available towards the end of the research period.

- **Experimental scope**. One of the strong elements of our experimental analyses was realism: the two effects we studied were genuine industrial applications. However, this worked against our goals in several ways. A significant amount of time was invested in the recovery and verification of algorithms from dusty-deck code. This severely limited the scope of our experiments to just two effects, since we had little time to construct more examples. In lieu of this limitation, we chose two very different effects which would illustrate all of the aspects of our research. In retrospect, a stronger focus on academic and open source image processing libraries would have lent more weight to our experimental analyses.

- **Practicality**. The practical applications of this research surpassed our expectations. By working closely with an industrial partner we were able to address many of the issues which

hinder the uptake of academic research in commercial software. The scope and design of the framework proved aptly scalable to new problems following the research period. Many of the changes made during the transition to an industrial setting were purely aesthetic, while the core ideas of component dependence and memory access metadata were carried across as they are presented in this thesis.

- **Parallelisation study**. Parallelisation was a key concern in the work on SIMT optimisation. However, our emphasis on this aspect was weaker in the settings of space/schedule optimisation and SIMD vectorisation. We chose a simple parallelisation strategy with wasteful recomputation at parallelisation boundaries, on the assumption that this recomputation would be small. Our use of the polyhedral model presented an interesting opportunity to implement a more optimal parallelisation strategy, but we did not attempt this. Furthermore, we did not fully explore the failings of SIMD autovectorisation in contemporary compilers. We cannot deduce whether the vectorisations were too complicated for the compiler, or whether they were not attempted because it was not clear whether they would improve performance.

- **Flexibility**. A key strength of our work proved to be its adaptability. Despite the simplicity of the metadata to which we had constrained our study, we were able to apply the information it held to a variety of complex optimisations. This allowed us to focus on new problems while continuing to treat each algorithm as a black box. The metadata has only marginally evolved in the industrial setting to accommodate new domain-specific information and opportunities for optimisation.

- **Formality**. An informal link between this work and that of the seminal THEMIS [KBFB01] paper was well known from an early stage. However, a formal link to the metadata definitions set out in that paper was not established until much later into the research period and was not documented in this thesis. The informal optimisation algorithms described in Chapters 4, 5 and 6 would also have benefited from stronger links to established work. In particular, we did not describe some circumstances under which those optimisations would not be applicable within the existing algorithm formulations.

7.4 Further Work

In light of the research pathways we have explored in this thesis, we now present ideas for promising new threads of research to build upon our work. Some of these ideas are being explored in a commercial context (see Section 7.5) while others are more speculative in nature and would require further academic study before yielding potentially useful results.

7.4.1 Compiler-Assistive Metadata

The set of metadata outlined in Section 3.3 is a domain-specific instance of generic program information of use to compilers. Indeed, many of the optimisations we described in this thesis

are supported by existing vendor compilers which simply lack a mechanism to obtain this information; e.g. through ambiguities in the results of their program analyses. Our decision to rely on domain-specific metadata came about for two reasons. Firstly, the granularity of information recorded in metadata must be brought to an appropriate level for the VFX developer to express. We chose a much more constrained structure of dependence metadata, for example, than advocated by [KBFB01] in order to keep the frontend representation concise. Secondly, we were unable to find a compiler with a mechanism to receive metadata from our framework. We chose instead to build a source-to-source preoptimisation phase in front of the vendor compiler.

There is potential value in exploring a direct route for program metadata, such as data dependence and patterns of memory access, to the optimisation phases of a compiler. Striking a balance between the granularity of the frontend expression and the needs of low-level optimisations would be crucial to generalising the approach we have demonstrated. Substantial gains stand to be made, however. We have shown the effectiveness of leveraging a small number of compiler transformations – including loop fusion, array contraction, vectorisation and parallelisation – in situations where vendor compilers were unable to apply these themselves. However, similar limitations may apply to much finer grained optimisations (e.g. instruction pipelining) which are much harder to tackle without direct integration with the compiler.

7.4.2 Dynamic Runtime Optimisation

Our hybrid offline runtime code generation process, described in Section 3.4, was designed to mitigate the performance impact of runtime code generation while retaining some of the benefits of greatly simplified analysis, e.g. to construct a component graph. A disadvantage of this approach is that information which can change at runtime – e.g. the user-tunable parameterisation of a visual effect – cannot be used in an offline optimisation process without specialising generated code to all possible values, or at least to common values. True runtime code generation requires a careful balance of specialisation against code generation costs to maintain a net performance benefit.

One opportunity we identified is to specialise to non-static graph structures which are constructed by the user at runtime. We observed in a compositing environment that graphs change infrequently with respect to parameter changes on those graphs once constructed. Furthermore, we identified an opportunity for an even higher level of cross-component optimisation. It is common for users to connect multiple effects of the same type, or effects of a similar class (e.g. colour corrections), which can be combined together and replaced with a composite node – or even the same node with a composite parameterisation – at an abstract level. This avoids the costly process of runtime code generation altogether while leveraging runtime context to implement cross-component optimisations. A component-level mechanism through which these interactions might be managed is left open to future research.

7.4.3 Heterogeneous Multiprocessing

The device-specific code generation procedures and optimisations discussed in Chapters 5 and 6 are, in our current framework, essentially incompatible. We choose a homogeneous subset of devices – e.g. a set of SMP multicore CPUs, or two GPUs – on which to evaluate a computation and maintained that same set throughout. In doing so, we are leaving other resources in the system unused or underutilised to avoid the complication of heterogeneous scheduling and data placement management. Clearly, there is scope to extend our framework with a scheduling subsystem to distribute the workload – either through data- or task-parallelism – across all of the computational resources of a given system. Our metadata abstractions encapsulate ample degrees of parallelism to support heterogeneous multiprocessing on even the highest end workstations. There is, perhaps, scope to extend metadata to record – through analysis, modelling, runtime feedback or otherwise – the affinity of individual algorithms to the capabilities and resources of different devices.

7.5 Conclusions

In this thesis we presented a novel approach to software optimisation which discards program analysis in favour of programmer annotations to support performance-critical high-level optimisations. The ease with which we were able to compose complex transformations and optimisations – albeit in a domain-specific context – under the guidance of metadata convinced our researchers and industrial partners that this was a powerful approach to delivering high performance software. Far from being a burden upon the programmer, we found that our metadata abstractions were succinctly interwoven with the algorithm expressions to the point of being useful program documentation in addition to guiding our optimising compiler.

Testament to the accomplishment our objectives, as set out in Section 7.1, is the successful transfer of this technology to an industrial environment. Throughout the three years of research culminating in this thesis, we met regularly with our research partners at The Foundry to discuss our ideas, results and the technical requirements that would need to be satisfied in order to deploy our optimisations in commercial software. I have personally continued this relationship with the company during the write-up period of this thesis to ensure that we have met both our academic and industrial goals. The formation of a High Performance Computing team at The Foundry came as a direct result of our interactions and many of the technologies we have constructed together are now being adapted for commercial deployment.

Some of the technical advances we made after the research period had completed include:

- An extension to the algorithmic patterns described in this thesis to support new visual effects. We defined a class of globally random access indexers with locally coherent access patterns to support projections in motion estimation algorithms. We permitted per-indexer local shifts so that the DWT algorithm, from Listing 3.1, could be efficiently implemented with three pointwise indexers. We also added support for parallel reductions to our dependence metadata suite to support statistical calculations on image data.

- A backend for the cross-device OpenCL parallel compute language – similar in many ways to CUDA – with true runtime code generation for cross-component fusion. The low cost of compilation with the Clang/LLVM compiler removed the main barrier to runtime optimisation and we observed 10x speed-ups when fusing graphs of visual primitives in a compositing prototype; caching and restricted parameter specialisation ensured that parameters could be tuned in realtime without recompilation.

- Metadata mappings from iteration space to indexer accesses. We found that the direct spatial correspondence of iteration space to data source/sinks in input and output images was insufficient to represent many algorithms. For example, a filtered scaling algorithm needs a fractional correlation between ROI and DOD with locally contiguous access to read the filter taps. We introduced a set of well-defined metadata mappings, as well as an arbitrary homogeneous transform mapping, which had value in producing fast SIMD and SIMT implementations.

- Metadata filters on indexer accesses. To support the application of common reconstruction filters – such as bilinear, Gaussian, Bell, etc. – we defined a set of well-known functions that take an indexer object and a floating-point coordinate and return a filtered result. This is especially useful on SIMT devices which often support a subset of filtering modes in texturing hardware. Work is ongoing to identify an elegant extension to indexer metadata to eliminate filtering functions entirely and provide explicit floating-point access on the indexers themselves; this would greatly simplify the SIMT code generation process and reduce clutter in the frontend.

Our research has begun to evolve in two directions. Firstly, we are beginning to generalise the framework by introducing metadata at a lower level of abstraction. This exhibits elements of the approach taken by Æcute and similarly allows our framework to encompass a wider domain of applications. Secondly, we are introducing metadata with deeper domain-specific meaning. In doing so, we can exploit common elements of a given domain to target optimisations to the specific performance challenges of applications within that domain. This shares ideals with the field of domain-specific languages. What we have constructed is a compromise of these two research pathways to tackle the broad but individually challenging domain of visual effects applications.

We continue to develop this framework in a commercial context and are working hard to find opportunities to collaborate on future topics of performance research. There is much crossover in the work of academia and industry and our research is strongest and most valuable when working together to assist each others' goals. This thesis concludes as a reminder of what can be achieved.

Appendix A

Framework Ontology

An accompaniment to the research in this thesis is a software implementation of the ideas from which the experimental results were derived. This software is informally outlined in Chapter 3. In this appendix we formally define the language constructs used to synthesise visual effects in the active library front-end.

A.1 The Functor Class

The `Functor` class, whose interface is defined in Listing A.1, is a base class from which all visual primitives are derived. The key elements of the interface are:

- `Axis`: An enumeration categorising the axis of serialisation in a moving average, or the axis of freedom in a 1D spatial filter indexer. `eHorizontal` indicates the horizontal, or X, axis. `eVertical` indicates the vertical, or Y, axis.
- `Dependence`: An enumeration categorising the pattern of data dependence in a visual primitive. `eParallel` indicates that there is no loop-carried dependence between any iterations of the X and Y loops: i.e. all iterations can execute in parallel. `eMoving` indicates the presence of a loop-carried dependence, and hence serialisation, in the X or Y axis.
- `Image`: An opaque handle linking the outputs of a visual primitive to the inputs of another.
- `DERIVED`: A template parameter to the `Functor` class identifying the type of the derived visual primitive class. This parameter is exploited to implement the Curiously Recurring Template Pattern (CRTP) [Cop96], a form of static polymorphism, for efficient invocations of methods in the derived class.
- `DEPENDENCE`: A template parameter to the `Functor` class identifying the data dependence pattern used by the derived visual primitive class.
- `dependenceAxis`: A runtime parameter to the `Functor` class constructor specifying the axis of serialisation when `DEPENDENCE` is `eMoving`. This value is ignored when `DEPENDENCE` is `eParallel`.

```
enum Axis
{
  eHorizontal,
  eVertical
};

enum Dependence
{
  eParallel,
  eMoving
};

class Image {};

template <class DERIVED,
          Dependence DEPENDENCE = eParallel>
class Functor
{
public:
  Functor(Axis dependenceAxis = eHorizontal);

  void RollUp() {}

  void Kernel() {}

  void operator()(Image image1, ...);
};
```

Listing A.1: The C++ interface to the Functor class.

- `RollUp`: A default implementation of a method which should be overridden in the derived class when `DEPENDENCE` is `eMoving`. This function is invoked once at the beginning of the serialised row or column (as specified by `dependenceAxis`) to provide the opportunity to initialise state member variables in the derived class.

- `Kernel`: A default implementation of a method which should be overridden in the derived class. This function is invoked at every point in the (X, Y) iteration space and should be used to compute and write elements of output data from input data. When all `Indexer` instances (see Section A.2) have `eChannel` granularity this function is invoked once per channel at each point in the iteration space. When all have `ePixel` granularity the function is invoked only once at each point. Other configurations are rejected by our compiler.

- `void operator()(Image image1, ...)`: A delayed evaluation function which constructs a visual effect DAG by connecting the inputs (defined by the `Indexer` member objects with `DIRECTION` of `eInput`) in the derived visual primitive class to the outputs (those with `DIRECTION` of `eOutput`) of another. The order in which parameters are passed corresponds to the order of `Indexer` member objects in the derived class to which they are connected. Each `Image` may be connected to more than one visual primitive but must only be connected to a single `Indexer` with `DIRECTION` of `eOutput`.

```
enum Direction
{
  eInput,
  eOutput
};

enum Granularity
{
  eChannel,
  ePixel
};

enum Freedom
{
  e0D,
  e1D,
  e2D
};

template <Direction DIRECTION,
          Granularity GRANULARITY,
          Freedom FREEDOM>
class Indexer
{
public:
  Indexer(Axis freedomAxis = eHorizontal,
          int freedomRadiusX = 0,
          int freedomRadiusY = 0);

  float &operator *();                              // GRANULAIRTY = eChannel

  float &operator[](int channel);                   // GRANULARITY = ePixel

  float &operator()(int offX);                      // GRANULARITY = eChannel

  float &operator()(int offX, int channel);         // GRANULARITY = ePixel

  float &operator()(int offX, int offY);            // GRANULARITY = eChannel

  float &operator()(int offX, offY, int channel); // GRANULARITY = ePixel
};
```

Listing A.2: The C++ interface to the Indexer class.

A.2 The Indexer Class

The `Indexer` class, whose interface is defined in Listing A.2, encapsulates access to image data. Instances of the class are declared as members of the visual primitive class, which derives from `Functor`, and bound to the primitive with the `mFunctorIndexers` macro. Each instance manages access to a single input or output image. The key elements of this interface are:

- `Direction`: An enumeration categorising the form of data access allowed on an `Indexer` instance. `eInput` indicates read-only access to the underlying data. `eOutput` indicates write-only access to the data.

- `Granularity`: An enumeration categorising the granularity of data access allowed on an `Indexer` instance. `eChannel` indicates that each channel can be processed independently and that only a single channel will be available at each invocation of the `Kernel` method. `ePixel` indicates that channels must be processed together and that all channels will be available together at each point in the iteration space.

- `Freedom`: An enumeration categorising the dimensionality of axial freedom (see Section 3.3.3) of data access allowed on an `Indexer` instance. `e0D` indicates that only a single component or pixel centred on the (X, Y) iteration position will be accessible. `e1D` indicates that a row or column of components or pixels will be accessible. `e2D` indicates that a rectangular region of components or pixels will be accessible.

- `DIRECTION`: A template parameter to the `Indexer` class identifying the form of data access permitted.

- `GRANULARITY`: A template parameter to the `Indexer` class identifying the granularity of data access permitted.

- `FREEDOM`: A template parameter to the `Indexer` class identifying the dimensionality of axial freedom of data access permitted.

- `freedomAxis`: A runtime parameter to the `Indexer` class constructor specifying the axis of freedom when `FREEDOM` is `e1D`. This value is ignored when `FREEDOM` is `e0D` or `e2D`.

- `freedomRadiusX`: A runtime parameter to the `Indexer` class constructor specifying the radius of access freedom permitted in `freedomAxis` when `FREEDOM` is `e1D`. This specifies the horizontal radius when `FREEDOM` is `e2D`. This value is ignored when `FREEDOM` is `e0D`.

- `freedomRadiusY`: A runtime parameter to the `Indexer` class constructor specifying the vertical radius of access freedom permitted when `FREEDOM` is `e2D`. This value is ignored when `FREEDOM` is `e0D` or `e1D`.

- `float &operator *()`: A component access method available when `GRANULARITY` is `eChannel`. A reference to a single component of image data, centred at the (X, Y) iteration position, is returned. This value can be read when `DIRECTION` is `eInput` and written to when `DIRECTION` is `eOutput`.

- `float &operator [](int channel)`: A component access method available when `GRANULARITY` is `ePixel`. A reference to a single component of image data, centred at the (X, Y) iteration position and selected by index `channel` from the RGBA set, is returned. This value can be read when `DIRECTION` is `eInput` and written to when `DIRECTION` is `eOutput`.

- `float &operator ()(int offX)`: A component access method available when `GRANULARITY` is `eChannel` and `FREEDOM` is `e1D`. A reference to a single component of image data, centred at the (X + `offX`, Y) iteration position, is returned. This value can be read when `DIRECTION` is `eInput` and written to when `DIRECTION` is `eOutput`.

- `float &operator ()(int offX, int channel)`: A component access method available when `GRANULARITY` is `ePixel` and `FREEDOM` is `e1D`. A reference to a single component of image data, centred at the (X + `offX`, Y) iteration position and selected by index `channel` from the RGBA set, is returned. This value can be read when `DIRECTION` is `eInput` and written to when `DIRECTION` is `eOutput`.

- `float &operator ()(int offX, int offY)`: A component access method available when `GRANULARITY` is `eChannel` and `FREEDOM` is `e2D`. A reference to a single component of image data, centred at the (X + `offX`, Y + `offY`) iteration position, is returned. This value can be read when `DIRECTION` is `eInput` and written to when `DIRECTION` is `eOutput`.

- `float &operator ()(int offX, int offY, int channel)`: A component access method available when `GRANULARITY` is `ePixel` and `FREEDOM` is `e2D`. A reference to a single component of image data, centred at the (X + `offX`, Y + `offY`) iteration position and selected by index `channel` from the RGBA set, is returned. This value can be read when `DIRECTION` is `eInput` and written to when `DIRECTION` is `eOutput`.

Bibliography

[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. In *The 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Lecture Notes in Computer Science, pages 195–210. Springer, 2003.

[AKO04] Thomas J. Ashby, Anthony D. Kennedy, and Michael F. P. O'Boyle. Cross component optimisation in a high level category-based language. In *Euro-Par*, pages 654–661, 2004.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[Bas04] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, 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. A compiler framework for optimization of affine loop nests for GPGPUs. In *ICS '08: Proceedings of the 22nd Annual International Conference on Supercomputing*, pages 225–234, New York, NY, USA, 2008. ACM.

[BCG+03] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. Putting polyhedral 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.

[BCI+08] Sergio Barrachina, Maribel Castillo, Francisco D. Igual, Rafael Mayo, and Enrique S. Quintana-Ortí. Evaluation and tuning of the level 3 CUBLAS for graphics processors. In *IPDPS*, pages 1–8, 2008.

[BFG+04] Olav Beckmann, Anthony J. Field, Gerard Gorman, Andrew Huff, Marc Hull, and Paul H. J. Kelly. Overcoming barriers to restructuring in a modular visualisation

environment. In *LCR '04: Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems*, pages 1–7, New York, NY, USA, 2004. ACM.

[BGGT01] Aart Bik, Milind Girkar, Paul Grey, and Xinmin Tian. Efficient exploitation of parallelism on Pentium III and Pentium 4 processor-based systems. In *Intel Technology Journal Q1*, pages 1–9, March 2001.

[BGGT02] Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Automatic intra-register vectorization for the Intel architecture. *Int. J. Parallel Program.*, 30(2):65–98, 2002.

[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.

[BHKM03] Olav Beckmann, Alastair Houghton, Paul H. J. Kelly, and Michael Mellor. Run-time code generation in C++ as a foundation for domain-specific optimisation. In *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 23–28. Springer, 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.

[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.

[BJK03] Olav Beckmann, Thiyagalingam Jeyarajan, and Paul Kelly. An exhaustive evaluation of row-major, column-major and Morton layouts for large two-dimensional arrays. In *UK performance engineering workshop (UKPEW 2003), Warwick, UK, July 2003*, 2003.

[Bri08] Ron Brinkmann. *The art and science of digital compositing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, second edition, 2008.

[CBK06] Jay L. T. Cornwall, Olav Beckmann, and Paul H. J. Kelly. Automatically translating a general purpose C++ image processing library for GPUs. In *Proceedings of the Workshop on Performance Optimisation for High-Level Languages and Libraries (POHLL)*, pages 381–388, April 2006.

[CCJ05] Patrick Carribault, Albert Cohen, and William Jalby. Deep jam: Conversion of coarse-grain parallelism to instruction-level and vector parallelism for irregular applications. In *PACT '05: Proceedings of the 14th International Conference on Par-*

allel Architectures and Compilation Techniques, pages 291–302, Washington, DC, USA, 2005. IEEE Computer Society.

[CDG+06] Albert Cohen, Sébastien Donadio, Maria-Jesus Garzaran, Christoph Herrmann, Oleg Kiselyov, and David Padua. In search of a program generator to implement generic transformations for high-performance computing. *Sci. Comput. Program.*, 62(1):25–46, 2006.

[CDK+01] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[CEG+00] Krzysztof Czarnecki, Ulrich W. Eisenecker, Robert Glück, David Vandevoorde, and Todd L. Veldhuizen. Generative programming and active libraries. In *Selected Papers from the International Seminar on Generic Programming*, pages 25–39, London, UK, 2000. Springer-Verlag.

[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 *Proceedings of the Conference on Computing Frontiers*, May 2009.

[Chu94] Charles K Chui. *An Introduction to Wavelets*. Academic Press, 1994.

[CKPN07] Jay L. T. Cornwall, Paul H. J. Kelly, Phil Parsonage, and Bruno Nicoletti. Explicit dependence metadata in an active visual effects library. In Vikram S. Adve, María Jess Garzarán, and Paul Petersen, editors, *Languages and Compilers for Parallel Computing (LCPC)*, volume 5234 of *Lecture Notes in Computer Science*, pages 172–186. Springer, October 2007.

[CN96] Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 145–156, New York, NY, USA, 1996. ACM.

[Col91] Murray Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, Cambridge, MA, USA, 1991.

[Cop96] James O. Coplien. Curiously recurring template patterns. pages 135–144, 1996.

[Cor05] Jay L. T. Cornwall. Efficient multiple pass, multiple output algorithms on the GPU. In *The 2nd European Conference on Visual Media Production (CVMP 2005)*, pages 253–262, December 2005.

[CRM07] Michael Chu, Rajiv Ravindran, and Scott Mahlke. Data access partitioning for fine-grain parallelism on multicore architectures. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 369–380, Washington, DC, USA, 2007. IEEE Computer Society.

[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.

[DGTY95] John Darlington, Yike Guo, Hing Wing To, and Jin Yang. Functional skeletons for parallel coordination. In *Euro-Par '95: Proceedings of the First International Euro-Par Conference on Parallel Processing*, pages 55–66, London, UK, 1995. Springer-Verlag.

[DM98] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *Computing in Science and Engineering*, 5(1):46–55, 1998.

[DRLC08] Alastair Donaldson, Colin Riley, Anton Lokhmotov, and Andrew Cook. Auto-parallelisation of Sieve C++ programs. In *Proceedings of the Workshop on Highly Parallel Processing on a Chip (HPPC)*, volume 4854 of *Lecture Notes in Computer Science*, pages 18–27. Springer, 2008.

[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 *In Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2005.

[EOO+06] A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using advanced compiler technology to exploit the performance of the Cell Broadband Engine^{TM} architecture. *IBM Syst. J.*, 45(1):59–84, 2006.

[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.

[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.

[FLPR99] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 285, Washington, DC, USA, 1999. IEEE Computer Society.

[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.

[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.

[GCC09] GCC Developers. Loop representation. `http://gcc.gnu.org/onlinedocs/gccint/Loop-representation.html`, 2009.

[GG02] Kazushige Goto and Robert van de Geijn. On reducing TLB misses in matrix multiplication. Technical report, University of Texas at Austin, Department of Computer Sciences, November 2002.

[GSW+07] A. Ghuloum, T. Smith, G. Wu, X. Zhou, J. Fang, P. Guo, B. So, M. Rajagopalan, Y. Chen, and Chen B. Future-proof data parallel algorithms and software on Intel® multi-core architecture. *Intel Technology Journal*, (11):333–348, November 2007.

[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.

[HAA+96] Mary W. Hall, Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *Computer*, 29(12):84–89, 1996.

[HLDK09] Lee W. Howes, Anton Lokhmotov, Alastair F. Donaldson, and Paul H.J. Kelly. Deriving efficient data movement from decoupled access/execute specifications. In *4th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC)*, volume 5409 of *LNCS*, pages 168–182. Springer, 2009.

[HLKF08] Lee Howes, Anton Lokhmotov, Paul Kelly, and A. J. Field. Optimising component composition using indexed dependence metadata. In *First International Workshop on New Frontiers in High-performance and Hardware-aware Computing (HipHaC)*, November 2008.

[Hof05] H. Peter Hofstee. Power efficient processor architecture and the Cell processor. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance*

Computer Architecture, pages 258–262, Washington, DC, USA, 2005. IEEE Computer Society.

[HOM08] M. Hassaballah, Saleh Omran, and Youssef B. Mahdy. A review of SIMD multimedia extensions and their usage in scientific and engineering applications. *Comput. J.*, 51(6):630–649, 2008.

[HOS+07] M. Harris, J. Owens, S. Sengupta, Y. Zhang, and A. Davidson. CUDPP: CUDA data parallel primitives library, 2007.

[HS86] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986.

[Hud96] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, page 196, 1996.

[IBM05] IBM. PowerPC microprocessor family: Vector/SIMD multimedia extension technology programming environments manual, August 2005.

[JDW+92] Leah H. Jamieson, Edward J. Delp, Chao-Chun Wang, Juan Li, and Frank J. Weil. A software environment for parallel computer vision. *Computer*, 25(2):73–77, 1992.

[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.

[KFA08] Vahid Kazempour, Alexandra Fedorova, and Pouya Alagheband. Performance implications of cache affinity on multicore processors. In *Euro-Par '08: Proceedings of the 14th international Euro-Par conference on Parallel Processing*, pages 151–161, Berlin, Heidelberg, 2008. Springer-Verlag.

[KKO02] P. M. W. Knijnenburg, T. Kisuki, and M. F. P. O'Boyle. Iterative compilation. *Embedded processor design challenges: systems, architectures, modeling, and simulation-SAMOS*, pages 171–187, 2002.

[KLM+98] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean marc Loingtier, and John Irwin. Aspect-oriented programming. In *Lecture Notes in Computer Science 1357*, pages 48–3. Springer Verlag, 1998.

[Koe02] D. Koelma. Horus C++ reference. Technical report, Intelligent Sensory Information Systems, Faculty of Science, University of Amsterdam, The Netherlands, January 2002.

[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 *Proceedings of the 2007 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2007.

[LA00] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. *SIGPLAN Not.*, 35(5):145–156, 2000.

[LC91] Jingke Li and Marina Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, 13(2):213 – 221, 1991.

[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.

[LHKK79] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, 1979.

[LNOM08] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[MCB+99] P. J. Morrow, D. Crookes, J. Brown, G. McAllese, D. Roantree, and I. Spence. Efficient implementation of a portable parallel programming model for image processing. In *Concurrency: Practice and Experience*, volume 11, pages 671–685, 1999.

[McC06] Michael D. McCool. Data-parallel programming on the Cell BE and the GPU using the RapidMind development platform. In *GSPx Multicore Applications Conference*, 2006.

[Moo08] S.K. Moore. Multicore is bad news for supercomputers. *Spectrum, IEEE*, 45(11):15–15, November 2008.

[Mun09] A. Munshi. The OpenCL specification. Khronos OpenCL Working Group, 2009.

[Nai04] Dorit Naishlos. Autovectorization in GCC. In *GCC Developer's Summit*, pages 105–118, June 2004.

[NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[NH06] Dorit Nuzman and Richard Henderson. Multi-platform auto-vectorization. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 281–294, Washington, DC, USA, 2006. IEEE Computer Society.

[NVI08] NVIDIA. NVIDIA CUDA: Compute Unified Device Architecture programming guide, 2008.

[PCB+06] Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Pierre Jouvelot, Georges-Andre Silber, and Nicolas Vasilache. GRAPHITE: Loop optimizations based on the polyhedral model for GCC. In *Proceedings of the 4th GCC Developers' Summit*, June 2006.

[PGI09] PGI. PGI Fortran and C accelerator programming model white paper, June 2009.

[PHP03] Neungsoo Park, Bo Hong, and Viktor K. Prasanna. Tiling, block data layout, and memory hierarchy performance. *IEEE Trans. Parallel Distrib. Syst.*, 14(7):640–654, 2003.

[PMJ+05] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[Prz90] Steven A. Przybylski. *Cache and memory hierarchy design: a performance-directed approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[QRW00] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *Int. J. Parallel Program.*, 28(5):469–498, 2000.

[Rei07] James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.

[RMKB08] Francis P. Russell, Michael R. Mellor, Paul H.J. Kelly, and Olav Beckmann. DESOLA: An active linear algebra library using delayed evaluation and runtime code generation. *Science of Computer Programming*, 2008.

[Row08] Robin Rowe. Linux powers: The Spiderwick Chronicles. *Linux J.*, 2008(166):5, 2008.

[RW96] Gerhard X. Ritter and Joseph N. Wilson. *Handbook of Computer Vision Algorithms in Image Algebra*. CRC Press, Boca Raton, FL, 1996.

[Sag94] Hans Sagan. *Space Filling Curves*. Springer-Verlag, New York, 1994.

[SCS+08] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[SCW05] Antonio De Stefano, Bill Collis, and Paul White. Synthesising and reducing film grain. *Journal of Visual Communication and Image Representation*, 17(1):163–182, 2005.

[SG02] Jocelyn Sérot and Dominique Ginhac. Skeletons for parallel image processing: an overview of the SKIPPER project. *Parallel Comput.*, 28(12):1685–1708, 2002.

[She92] M.J. Shensa. The discrete wavelet transform: wedding the à trous and Mallat algorithms. In *IEEE Transactions on Signal Processing*, volume 40, pages 2464–2482, 1992.

[SJV06] A. Shahbahrami, B.H.H. Juurlink, and S. Vassiliadis. Performance impact of misaligned accesses in SIMD extensions. In *Proceedings of the 17th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC 2006)*, pages 334–342, November 2006.

[SK04] F. J. Seinstra and D. Koelma. User transparency: a fully sequential programming model for efficient data parallel image processing: Research articles. *Concurr. Comput. : Pract. Exper.*, 16(6):611–644, 2004.

[SPN96] Ashley Saulsbury, Fong Pong, and Andreas Nowatzyk. Missing the memory wall: the case for processor/memory integration. *SIGARCH Comput. Archit. News*, 24(2):90–101, 1996.

[SQ03] M. Schordan and D. Quinlan. A source-to-source architecture for user-defined optimizations. In *Proceedings of the Joint Modular Languages Conference (JMLC'03), Lecture Notes in Computer Science*, volume 2789, pages 214–223, Aug 2003.

[Ste04] E. Stewart. *Intel Integrated Performance Primitives: How to Optimize Software Applications Using Intel IPP*. Intel Press, 2004.

[SXWL01] Yonghong Song, Rong Xu, Cheng Wang, and Zhiyuan Li. Data locality enhancement by memory reduction. In *ICS '01: Proceedings of the 15th International Conference on Supercomputing*, pages 50–64. ACM Press, 2001.

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[TaMS+08] Christian Terboven, Dieter an Mey, Dirk Schmidl, Henry Jin, and Thomas Reichstein. Data and thread affinity in openmp programs. In *MAW '08: Proceedings of the 2008 workshop on Memory access on future processors*, pages 377–384, New York, NY, USA, 2008. ACM.

[TEL98] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 533–544, New York, NY, USA, 1998. ACM.

[TF96] Gary Tyson and Matthew Farrens. Evaluating the effects of predicated execution on branch prediction. *Int. J. Parallel Program.*, 24(2):159–186, 1996.

[TH99] Shreekant (Ticky) Thakkar and Tom Huff. Internet streaming SIMD extensions. *Computer*, 32(12):26–34, 1999.

[TWFO09] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael F.P. O'Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 177–187, New York, NY, USA, 2009. ACM.

[ULBH08] Sain-Zee Ueng, Melvin Lathara, Sara S. Baghsorkhi, and Wen-Mei W. Hwu. CUDA-lite: Reducing GPU programming complexity. *Languages and Compilers for Parallel Computing: 21th International Workshop, LCPC 2008, Edmonton, Canada, July 31 - August 2, 2008, Revised Selected Papers*, pages 1–15, 2008.

[VBC06] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. Polyhedral code generation in the real world. In *Proceedings of the International Conference on Compiler Construction (ETAPS CC'06)*, LNCS, pages 185–201, Vienna, Austria, March 2006. Springer-Verlag.

[VG98] Todd L. Veldhuizen and Dennis Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*, 1998.

[WD98] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[WFW+94] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: an infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Not.*, 29(12):31–37, 1994.

[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.

[Wol95] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[YLR+05] K. Yotov, X. Li, G. Ren, M.J.S. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE*, 93(2):358–386, February 2005.

[YQ04] Qing Yi and Dan Quinlan. Applying loop optimizations to object-oriented abstractions through general classification of array semantics. In *The 17th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2004.