Performance Optimization of Stencil Computations on Modern SIMD Architectures

DISSERTATION

Presented in Partial Fulfillment of the Requirements
for the Degree Doctor of Philosophy
in the Graduate School of The Ohio State University

By

Thomas Steel Henretty, M.Sci.

Graduate Program in Computer Science and Engineering

The Ohio State University

2014

Dissertation Committee:

P. Sadayappan, Advisor

Atanas Rountev

Radu Teodorescu

ABSTRACT

Performance of scientific computing codes on modern high-performance computing (HPC) systems has, in some cases, not achieved a significant percentage of the system's peak performance. Three of the fundamental causes of this lack of efficiency are (1) less than optimal utilization of the short-vector SIMD units found in nearly all modern HPC systems, (2) less than optimal utilization of the memory hierarchy and (3) less than optimal utilization of all computing cores available in a system. Codes that are able to overcome one or more of these limitations are generally very complex and their implementation requires both an expert programmer and a substantial amount of time.

In this work, a class of scientific computing codes known stencil computations is examined and shown to exhibit a fundamental algorithmic limitation that interferes with the generation of optimal SIMD code. A data layout transformation (DLT) to overcome this limitation is described and comprehensive results for cache-resident problem sizes are presented. It is shown that this DLT can significantly increase the performance of stencil computations on modern SIMD architectures.

While substantial performance gains can be realized using the DLT for small problem sizes, larger problem sizes require the application of spatial and temporal loop tiling techniques to relieve pressure on the memory subsystem and exploit all available multicore parallelism. Two closely related tiling techniques, nested and hybrid split tiling, are developed and shown to exhibit high performance across a variety of modern multicore SIMD architectures and stencil benchmarks.

Combining SIMD, memory hierarchy, and parallelism optimizations for stencil computations leads to code that is very complex and difficult for scientists and even seasoned programmers to implement. Further, these optimizations are difficult to integrate into a general purpose compiler as there is no existing framework for reliably identifying and representing stencil computations in a general purpose language such as C. These problems are resolved with the creation of the Stencil Domain Specific Language (SDSL). This language uses data structures and concepts specific to stencil computations to enable the retention of fundamental information about the stencil throughout the compilation process. Preserving the details of a stencil computation enables the automated generation of complex, highly optimized code for multiple parallel vector architectures from a simple specification in SDSL.

For my family...

ACKNOWLEDGMENTS

Over the years a number of people have made a profound difference in my life and have led me to the point I am at today. This space is to thank them and show my sincere appreciation for their guidance, wisdom, companionship, and service.

First, I would like to thank my advisor Saday. Through the many twists and turns of graduate school he consistently challenged me to expand my abilities and showed me exactly what it means to be a dedicated and outstanding professional.

I would also like to thank the other members of my committee, Prof. Nasko Rountev and Prof. Radu Teodorescu, for their guidance and advice throughout the course of my studies. Nasko constantly pushed me to think about ideas critically and comprehensively; Radu kept us on the absolute leading edge of architecture research. Both of them were especially great at both knowing the right way to do things and doing things the right way.

Prof. J. "Ram" Ramanujam, Prof. Louis-Nöel Pouchet, and Prof. Franz Franchetti were consistently helpful throughout graduate school and helped develop many of the ideas contained in this work. They consistently had me in awe of their remarkable ability to immediately grasp and extend the most challenging concepts, a skill I can only aspire to have.

I am also grateful to Prof. Jason Cong and everyone involved with the Center for Domain Specific Computing at the university of California, Los Angeles. Visiting was always a pleasure and it was great to see so many people from so many different backgrounds working to achieve a common goal.

The day-to-day experience of going through grad school was made much easier thanks to a great group of fellow students and labmates over the years. Bruce Adcock, Muthu Baskaran, Uday Bondhugula, Jim Dinan, Albert Hartono, William Harvey, Brian Larkins, Josh Levine, Qingda Lu, and Tim Miller welcomed me to the graduate program and did a great job introducing me to the pragmatics of graduate studies.

Md. Humayun Arafat, Naznin Fauzia, Justin Holewinski, Martin Kong, Pai-Wei Lai, Mahesh Ravishankar, Naser Sedaghati, Kevin Stock, and Sanket Tavarageri were a constant presence in the lab and it was a pleasure going through the program with them. Samyam Rajbhandari and Prasanth Rawat are the next generation of HPCRL students; I hope I have helped you in the way that so many have helped me.

The technical and office staff of the CSE department has been a great help in so many different ways over the years. Thank you to Catrena Collins, Don Havard, Aaron Jenkins, Dave Kneisly, Lynn Lyons, Kathryn Reeves, Shaun Rowland, Carrie Stein, and Kat Wenger for all the work you put in every day for me and everybody else in the department. Surely the place would come to a halt without your considerable expertise.

My deepest thanks are reserved for the people outside of academia that made my time in graduate school so rewarding. Thank you to Brian Galensky, Ken Kushner, Amanda Burkholder-Kushner, Siew Loke, Steven Maravich, Cindy Maravich, Elizabeth Miller, Judy Warner and the late Serge Warner for their friendship and support. I don't think I would have made it without them.

The people at Reservoir Labs have made my transition to industry a pleasure; Thank you to Robert Budesa, Patrick Clancy, Ann Johnson, Dr. Richard Lethin, Nicole Leung, Dr. Benoît Meister, Melanie Peters, and Trevor Serfrass.

My old friends Justin Casey, Jon Darkow, Kevin Mohler and Clint Smith: those long summer nights spent on a hill in Fairlawn, Ohio debating anything and everything really were the beginning of this pursuit. Thank you for all of the experiences over the years.

Of course none of this would have been possible without my family. Mom, thank you for the years of support and putting up with me. Suzi and Gary Kraus, thank you for being my Los Angeles area tour guides off and on for the better part of four years. Bob, thank you for the years of Scrabble battles. Finally, thank you to my brother Sean for giving me the perspective that sometimes the best computer is the one that is unplugged.

Thomas S. Henretty
Brooklyn, New York
August 25, 2014

VITA

1976	Born: Akron, OH, USA
1999—2001	Programmer-Analyst, Cendant Corporation, Columbus, OH, USA
2001—2006	Owner, Steel Administration, Akron, OH, USA
2008	B.S.C.S.E., The Ohio State University, Columbus, OH, USA
2008—2014	Graduate Associate, The Ohio State University, Columbus, OH, USA
2012	M.S., The Ohio State University, Columbus, OH, USA
2013—Present	Senior Engineer, Reservoir Labs, Inc., New York, NY, USA

PUBLICATIONS

Research Publications

Tom Henretty, Justin Holewinski, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, P. Sadayappan, A Domain-Specific Language and Compiler for Stencil Computations on Short-Vector SIMD and GPU Architectures. In *Proceedings of the 17th Workshop on Compilers for Parallel Computing (CPC)*, Lyon, France, USA. July 3–5, 2013.

Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, P. Sadayappan, A Stencil Compiler for Short-Vector SIMD Architectures. In *Proceedings of the 27th International Conference on Supercomputing (ICS)*, Eugene, Oregon, USA. June 10–14, 2013.

Tom Henretty, Justin Holewinski, Naser Sedaghati, Louis-Noël Pouchet, Atanas Rountev, P. Sadayappan Stencil Domain Specific Language (SDSL) User Guide. *Ohio State University Technical Report OSU-CISRC-4/13-TR09*, April 2013.

Kevin Stock, Tom Henretty, Iyyappa Murugandi, P. Sadayappan, Robert Harrison, Model-Driven SIMD Code Generation for a Multi-Resolution Tensor Kernel. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Anchorage, Alaska, USA. May 16–20, 2012.

Tom Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J. Ramanujam, P. Sadayappan, A Stencil Compiler for Short-Vector SIMD Architectures. In *Proceedings of the 20th International Conference on Compiler Construction (CC)*, Saarbrücken, Germany. March 26–April 3, 2011.

Muthu Baskaran, Albert Hartono, Sanket Tavarageri, Tom Henretty, J. Ramanujam, P. Sadayappan, Paramaterized Tiling Revisited. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Toronto, Ontario, Canada. April 24–28, 2010.

Qingda Lu, Christophe Alias, Uday Bondhugula, Tom Henretty, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, P. Sadayappan, Yongjian Chen, Haibo Lin, Tin-Fook Ngai, Data Layout Transformation for Enhancing Data Locality on NUCA Chip Multiprocessors. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Raleigh, North Carolina, USA. September 12–16, 2009.

Albert Hartono, Qingda Lu, Tom Henretty, Sriram Krishnamoorthy, Huaijian Zhang, Gerald Baumgartner, David E. Bernholdt, Marcel Nooijen, Russell Pitzer, J. Ramanujam, Atanas Rountev, P. Sadayappan, Performance Optimization of Tensor Contraction Expressions for Many-Body Methods in Quantum Chemistry. In *The Journal of Physical Chemistry A Vol. 113.* September 28, 2009.

FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in High Performance Computing: Prof. P. Sadayappan

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF LISTINGS

CHAPTER 1

Introduction

Stencil computations represent an important class of scientific computations, occurring in many image processing applications, computational electromagnetics and solution of PDEs using finite difference or finite volume discretizations. This work develops a high performance code generation backend for a DSL developed for stencil computations. We seek to accelerate stencil computations on modern short-vector Single Instruction Multiple Data (SIMD) multicore architectures by exploiting all available SIMD and multicore parallelism. A technique is developed to increase SIMD performance using a dimension-lift-and-transpose (DLT) data layout transformation. Further techniques are developed using split tiling, a "time tiling" technique, to (1) achieve coarse-grained parallelism across cores and (2) substantially reduce memory bandwidth consumption of stencil codes on multicore system. These optimizations are automatically generated for stencils developed in the Stencil Domain Specific Language (SDSL), a stencil programming language described in this work.

1.1 Stream Alignment Conflicts

Stencil computations are readily expressible in a form with vectorizable innermost loops where arrays are accessed at unit stride. There is, however, a fundamental performance limiting factor with all current short-vector SIMD instruction sets. We formalize the problem through the abstraction of stream alignment conflicts. We note that the alignment conflict issue we formulate and solve in this dissertation pertains to algorithmic alignment constraints and is distinctly different from the previously studied topic of efficient code generation on SIMD architectures with hardware alignment constraints [17, 18, 35, 60]. We address the problem of resolving stream alignment conflicts through the novel use of a nonlinear DLT data layout transformation and develop a compiler framework to identify and suitably transform the computations.

```
     for (t = 0; t < T; ++t) {
       for (i = 0; i < N; ++i)
         for (j = 1; j < N+1; ++j)
S1:        C[i][j] = A[i][j] + A[i][j-1];
       for (i = 0; i < N; ++i)
         for (j = 1; j < N+1; ++j)
S2:        A[i][j] = C[i][j] + C[i][j-1];
     }
```

Perfor-mance:	AMD Phenom	1.2 GFlop/s
	Core2	3.5 GFlop/s
	Core i7	4.1 GFlop/s

(a) Stencil code

```
     for (t = 0; t < T; ++t) {
       for (i = 0; i < N; ++i)
         for (j = 0; j < N; ++j)
S3:        C[i][j] = A[i][j] + B[i][j];
       for (i = 0; i < N; ++i)
         for (j = 0; j < N; ++j)
S4:        A[i][j] = B[i][j] + C[i][j];
     }
```

Perfor-mance:	AMD Phenom	**1.9** GFlop/s
	Core2	**6.0** GFlop/s
	Core i7	**6.7** GFlop/s

(b) Non-Stencil code

Figure 1.1: Example to illustrate addressed problem: The stencil code (a) has much lower performance than the non-stencil code (b) despite accessing 50% fewer data elements.

We use a simple example to illustrate the problem addressed. Consider the two sets of loop computations in Figure 1.1(a) and Figure 1.1(b), respectively. With the code shown in Figure 1.1(a), for each of the two statements, $2 \times N^2 + N$ distinct data elements are referenced, with N^2 elements of C and $N^2 + N$ elements of A being referenced in S1, and N^2 elements of A and $N^2 + N$ elements of C being referenced in S2. With the code shown in Figure 1.1(b), each of S3 and S4 access $3 \times N^2$ distinct data elements, so that the code accesses around 1.5 times as many data elements as the code in Figure 1.1(a). Both codes compute exactly the same number ($2 \times N^2$) of floating point operations. Figure 1.1 shows the achieved single-core performance of the two codes on three different architectures, compiled using the latest version of ICC with auto-vectorization enabled. It may be seen that on all systems, the code in Figure 1.1(b) achieves significantly higher performance although it requires the access of roughly 50% more data elements.

The reason for the lower performance of the stencil code in Figure 1.1(a) is that adjacent data elements (stored as adjacent words in memory) from arrays A and C must be added together, while the data elements that are added together in the code of Figure 1.1(b) come from independent arrays. In the latter case, we can view the inner loop as representing the addition of corresponding elements from two independent streams `B[i][0:N-1]` and `C[i][0:N-1]`, but for the former, we are adding shifted versions of data streams: `A[i][0:N-1]`, `A[i][1:N]`, `C[i][0:N-1]`, and `C[i][1:N]`. Loading vector registers in this case requires use of either (a) redundant loads, where a data element is moved with a different load for each distinct vector register position it needs to be used in, or (b) load operations followed by additional inter- and intra-register movement operations to get each data element into the different vector register slots where it is used. Thus the issue we address is distinctly different from the problem of hardware alignment that has been addressed in a number of previous works. The problem we address manifests itself even on architectures where hardware alignment is not necessary and imposes no significant penalty (for example recent architectures).

1.2 Memory Hierarchy Optimizations

Stencil computations, when coded naively, demand more performance than modern, cache-based memory hierarchies can deliver. The reason for this is that there are relatively few arithmetic instructions performed for each memory operation.

In addition to optimizing stencil computations for SIMD microarchitectures, we extend these optimizations to large problem sizes using a technique known as split tiling, a time tiling strategy where multiple timesteps of a stencil are executed in succession for predefined subsets of the stencil computation. This increases the number of arithmetic operations performed per memory operation thereby reducing pressure on the memory subsystem.

1.3 Domain Specific Programming Language

We integrate DLT and split tiling optimizations into the code generation backend of a domain-specific programming language created for describing stencil computations. We have developed a stencil domain specific language (SDSL) to provide information specific to stencil computations. This extra information makes program analysis, loop transformations, and code generation easier than in it is in the context of a general purpose compiler and leads to highly optimized stencil codes. An implementation of SDSL has been made available for download at `http://hpcrl.cse.ohio-state.edu`.

1.4 Outline

A high-level overview of our approach is shown in Figure 1.2. A stencil code starts out as an SDSL program embedded in a C/C++ program. The SDSL code is parsed and high-performance code is generated by a platform-specific backend. Currently, backends exist for generating (1) affine C99 code intended for further optimization by polyhedral compiler tools including PolyOpt/C [44] and PoCC [43], (2) CUDA code with overlapped tiling optimizations as described by Holewinski, Pouchet, and Sadayappan [27], and (3) short-vector SIMD code for SSE2, SSE4, AVX, and NEON vector ISA with the split tiling and DLT optimizations described in this dissertation.

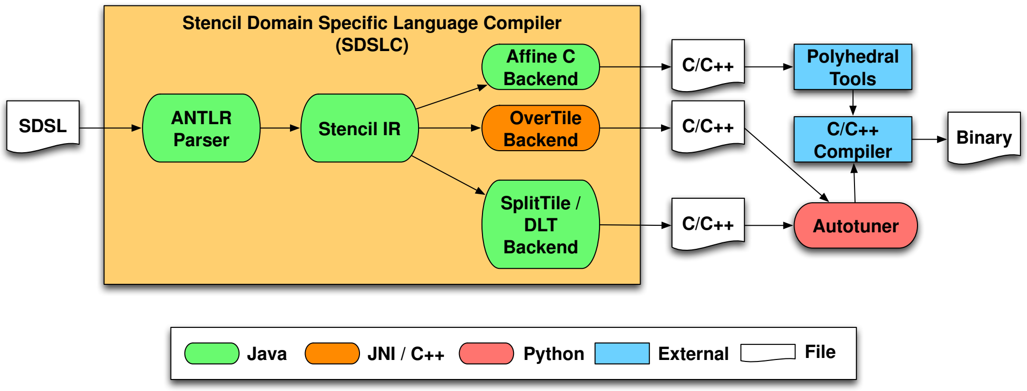


Figure 1.2: Integration of components

The rest of the dissertation is organized as follows. Chapter 2 contains relevant background background material. Formal definitions of key concepts used throughout the work are provided in Chapter 3. In Chapter 4 we formalize stream alignment conflicts and present a compile-time approach to identifying vectorizable computations that can benefit from a data layout transformation. Next, in Chapter 5, we develop a novel approach to overcoming the problem for cache-resident problem sizes via data layout transformation. In Chapter 6 we introduce split tiling as a technique for extending cache-resident layout transformation performance to larger problem sizes. Chapter 7 provides a definition of the key concepts in the SDSL domain specific programming language for stencil computations. Related work is explored in Chapter 8, and future research directions are explored in Chapter 9. Finally, a summary of the dissertation is presented in Chapter 10.

CHAPTER 2

Background

2.1 The Evolution of Short-Vector SIMD Architectures

Short-vector SIMD instructions have their roots in the vector supercomputers of the 1970s and 1980s. These machines used the Single Instruction Multiple Data paradigm to give programmers the ability to perform operations such as addition and subtraction on entire arrays.

With the MMX, SSE, and 3DNow instruction set extensions to the x86 architecture Intel and AMD introduced SIMD instructions to mainstream processors. While the idea of using one instruction to enable the execution of multiple identical instructions was taken from vector supercomputers, these new instructions operated on 64-bit fixed length vectors (two single-precision floating point values). Most instructions operating on 64-bit vectors were basic arithmetic ops.

The SSE2 (x86) and VMX (Power) instruction sets expanded the vector length to 128-bits. This allowed operations to be performed on four single-precision or two double-precision floating point values. Longer vectors required the introduction of specialized permutation instructions for intra- and inter- vector data movement.

Later versions of SSE including SSE3, SSSE3, and SSE4.x added more permute operations and some application specific instructions while the vector length remained 128 bits. The AVX instruction set expanded vector length to 256 bits on the x86 architecture, while Intel's MIC accelerator architecture uses 512 bit vectors.

Short vector SIMD extensions are now included in all major high-performance CPU. While ubiquitous, the ISA and performance characteristics vary from vendor to vendor and across hardware generations. For instance, Intel has introduced SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, and LRBni ISA extensions over the years, all of which have different performance characteristics. With every processor, the latency and throughput numbers of instructions in these extensions change. IBM, Freescale, and Motorola have introduced AltiVec, VMX, VMX128, VSX, Cell SPU, PowerXCell 8i SPU SIMD implementations. In some instances (RoadRunner, BlueGene/L), custom ISA extensions were designed since the supercomputing installation was big enough to warrant such an investment. These extensions provide from 2-way adds and multiplies up to 16-way fused multiply-add operations, promising significant speed-up. It is therefore important to optimize for these extensions.

2.2 Short-Vector SIMD Programming

Short-vector SIMD instructions can be placed in a program in a number of different ways. The following list gives the most common methods, from least to most programmer effort.

- Compiler autovectorization
- Programmer generated pragmas

- Vector intrinsics
- Inline assembly code

Compiler autovectorization is a feature of nearly all modern compilers, including the GNU compiler collection, Intel compilers, IBM compilers, and Portland Group compilers. While each compiler uses different algorithms to vectorize codes, they all proceed through essentially the same steps. Vectorization is accomplished by the compiler performing a dependency analysis on the innermost loops of a program. If an innermost loop is found to be parallel and there are no complicating factors (e.g. functions with side effects, potentially aliased arrays, etc.) the compiler will generate vectorized machine code.

Programmer generated pragmas are inserted by a programmer to give a hint to the compiler that a loop is vectorizable. The `#pragma ivdep` and `#pragma vector always` constructs used with the Intel C Compiler instruct the compiler to ignore the results of its dependency analysis (`ivdep`) and vectorize the loop marked by the `vector always` pragma.

Vector intrinsics provide the programmer nearly transparent access to the vector operations available for a given architecture. Vector intrinsics consist of data types for vectors and functions to perform on these data types. Using intrinsics is similar to programming with library code, although intrinsics, in general, directly map to a very small number of machine code instructions (usually one). Vector intrinsics provide more fine-grained control over how a program is vectorized when compared to autovectorization and pragmas.

Inline assembly code gives the programmer direct access to vector instructions and provides more control over the resulting machine code than any other programming model. While the programmer can write very high performance code using assembly language there is a steep learning curve, and in some cases inline assembly may inhibit the ability of the compiler to perform more advanced optimizations.

2.3 Stencil Computations

Stencil computations, also known as structured grid computations, commonly occur in computational science and image processing codes. Typically they are used to implement solvers for partial differential equations using the finite difference and finite volume methods or in image processing codes. There has been considerable recent interest in optimization of stencil computations [4, 10–13, 15, 16, 31, 32, 34, 36, 40–42, 47, 53, 55–57, 59].

Stencil computations combine neighboring points in one or more arrays using basic arithmetic operations to produce a result in an element of one or more arrays. The spatial loops that sweep arrays are typically surrounded by an outer iterative loop, and there is frequently a convergence test after each outer loop iteration that determines if the solver has finished.

Codes containing stencil computations generally do not achieve peak performance on a given architecture. One reason, previously identified in the literature, is that stencil computations are prone to saturate the memory subsystem. Stencil computations tend to have a large number of memory operations and a small number of arithmetic operations. The ratio of inner loop arithmetic operations to memory operations is known as *arithmetic intensity*.

Programs with a low arithmetic intensity may spend a large amount of time waiting for operands to be fetched from and stored to memory. As problem sizes scale up through the memory hierarchy this effect becomes more pronounced. Large stencil computations typically require one or more arrays to stream into and out of cache for every outer loop iteration. This induces a large number of compulsory cache misses and causes the processor to idle for long periods and significantly impacts performance in a negative way. Tiling the outer iterative loop ("time-tiling") allows for the reuse of operands in cache. Many recent publications use some form of time-tiling to ease pressure on the memory subsystem and enable higher performance.

2.4 Domain Specific Languages

Domain specific languages (DSL) are programming languages tailored to a specific application domain. They have custom syntax and semantics that allow an individual familiar with the application domain to produce programs that give a compiler more information about the intent of the programmer and thus allow the compiler to perform more aggressive optimizations than possible with a general purpose programming language. In the stencil domain typical optimizations are loop tiling, vectorization, and parallelization.

Domain specific languages can be standalone or embedded into a general purpose programming language. PATUS [7] is an example of a standalone DSL for stencil computations. Programs are written in the PATUS language and the PATUS compiler converts them into C code. The generated C code has had a number of optimizations applied, including vectorization, tiling, and parallelization. Further, PATUS generates codes for multiple targets, including a number of SIMD instruction sets and GPU.

An example of an embedded DSL for stencil computations is Pochoir [52]. Pochoir code is embedded into vanilla C++ code. The Pochoir compiler then converts hybrid C++ / Pochoir programs into C++ code optimized for the Intel C Compiler and x86 targets. Optimizations include parallelization and loop tiling.

CHAPTER 3

Formal Definitions

In this chapter a number of formal definitions are presented. These definitions provide the mathematical basis for the ideas discussed in future sections.

3.1 Grids and Subgrids

A *grid* is a rectangular region in $\mathbb{N}^N$. Each dimension of the grid is defined on the interval $[0, S_i)$ for some $S_i \in \mathbb{N}$. All grids can be formally and completely defined by an N-dimensional vector of sizes.

$$\vec{S} \in \mathbb{N}^N = (S_0, ..., S_{N-1}) \tag{3.1}$$

The grid represented by the size vector can be constructed by taking the Cartesian product of all intervals defined by the vector.

$$\mathbb{G}_{\vec{S}} = [0, S_0) \times [0, S_1) \times ... \times [0, S_{N-1}) \tag{3.2}$$

A stencil computation (see Section 3.4) is defined for every point of a grid, as are all data structures used in the stencil computation. Portions of a stencil computation may, however, operate on different *subgrids* of the grid. We define a subgrid to be a rectangular subregion of an N-dimensional grid. Each dimension of the subgrid is defined on the interval $[L_i, U_i]$ where $L_i, U_i \in \mathbb{N}^N$ and $0 \leq L_i, U_i \leq S_i$. A subgrid can be formally and completely defined by an N-dimensional vector of boundaries.

$$\vec{B} \in (\mathbb{N}, \mathbb{N})^N = ((L_0, U_0), ..., (L_{N-1}, U_{N-1})) \tag{3.3}$$

As with a grid, a subgrid may be constructed by taking the Cartesian product of all intervals defined in the vector.

3.2 Data Grids

We define the rectangular regions of data associated with a specific stencil computation as *data grids*. A data grid $D_{\mathbb{G}}$ maps a value to every point in a grid $\mathbb{G}$. This value may be an integer, a real number, or more complex recursive structures. The grammar below specifies allowable data types.

$$\begin{aligned} \mathit{Type} &\longrightarrow \text{real} \mid \text{integer} \mid \text{vector of } \mathit{Type} \mid \mathit{StructType} \\ \mathit{StructType} &\longrightarrow \{\mathit{field}_1 : \mathit{Type}_1, \mathit{field}_2 : \mathit{Type}_2, ...\} \end{aligned}$$

Stencils computations make use of one or more data grids, and these data grids contain input, output, and intermediate results of the computation.

3.3 Stencil and Reduction Functions

A *stencil function* is a computation applied to one or more data grid points that produces a result value in some data grid. All data grid points used to compute the result are located at a fixed offset from the grid point where the computation is taking place.

Similar to a stencil function, a *reduction function* uses neighboring points to compute a value. While a stencil function maps points to a domain, a reduction function maps points to a single value.

We define stencil function s formally as a relation where neighbor points are mapped to a domain. Similarly, we define reduction function r as a relation where neighbor points are mapped to a single *reduction value*.

$$s \;:\; T_A^M \times T_B^N \times \ldots \times T_C^P \rightarrow T_D \tag{3.4}$$

$$r \;:\; T_A^M \times T_B^N \times \ldots \times T_C^P \rightarrow T_{reduction} \tag{3.5}$$

In Equations 3.4 and 3.5, T denotes a data type, A, B, C and D are data grids containing neighboring points, $M, N, P \in \mathbb{N}$ are the number of points from each data grid used in the computation. The domain of each relation consists of an arbitrary rectangular region of the computation's grid, $\mathbb{B}_D$.

A *sweep* is formally defined as a finite, ordered sequence of one or more stencil and reduction functions operating on a common subgrid $\mathbb{B}$.

$$S_{\mathbb{B}} = (f_1, f_2, \ldots, f_k) \tag{3.6}$$

For each point in subgrid $\mathbb{B}$, functions $f_1, f_2, \ldots, f_k$ are evaluated and the results are written back in the order defined by the sweep. While the order that the functions are applied is dictated by the sweep definition, the order in which points of the domain are traversed is arbitrary.

3.4 Stencil Computation

A *stencil computation* is a complete program. It consists of a grid over which the computation occurs, zero or more subgrids, a sequence of sweeps and over those subgrids. The sequence of sweeps is performed for a set number of iterations, or until a convergence condition is reached.

We define a convergence condition to be a relation between one or more reduction values r_i and a boolean value B:

$$c : r_1 \times r_2 \times ... \to B \tag{3.7}$$

If B evaluates to 'TRUE' at the end of any iteration of the stencil computation the stencil computation terminates.

Formally, a stencil computation SC is completely defined by a maximum number of iterations, $I_{max} > 0$, a grid $\mathbb{G}$ divided into k subgrids. Each subgrid $\mathbb{B}_i$ has at least one corresponding sweep S_i^j. Finally, a stencil computation is organized as an ordered finite sequence P of one or more sweeps S and an optional convergence condition c.

$$SC = \begin{cases} \mathbb{G} = \bigcup_{i=0}^{k-1} \mathbb{B}_i \\ I_{max} \in \mathbb{N} \\ S_0^0, S_0^1, ..., S_1^0, S_1^1, ..., S_i^j : \forall \mathbb{B}_i \; \exists S_i^j \\ P = (S, r...) \\ c \end{cases}$$

CHAPTER 4

Stream Alignment Conflict

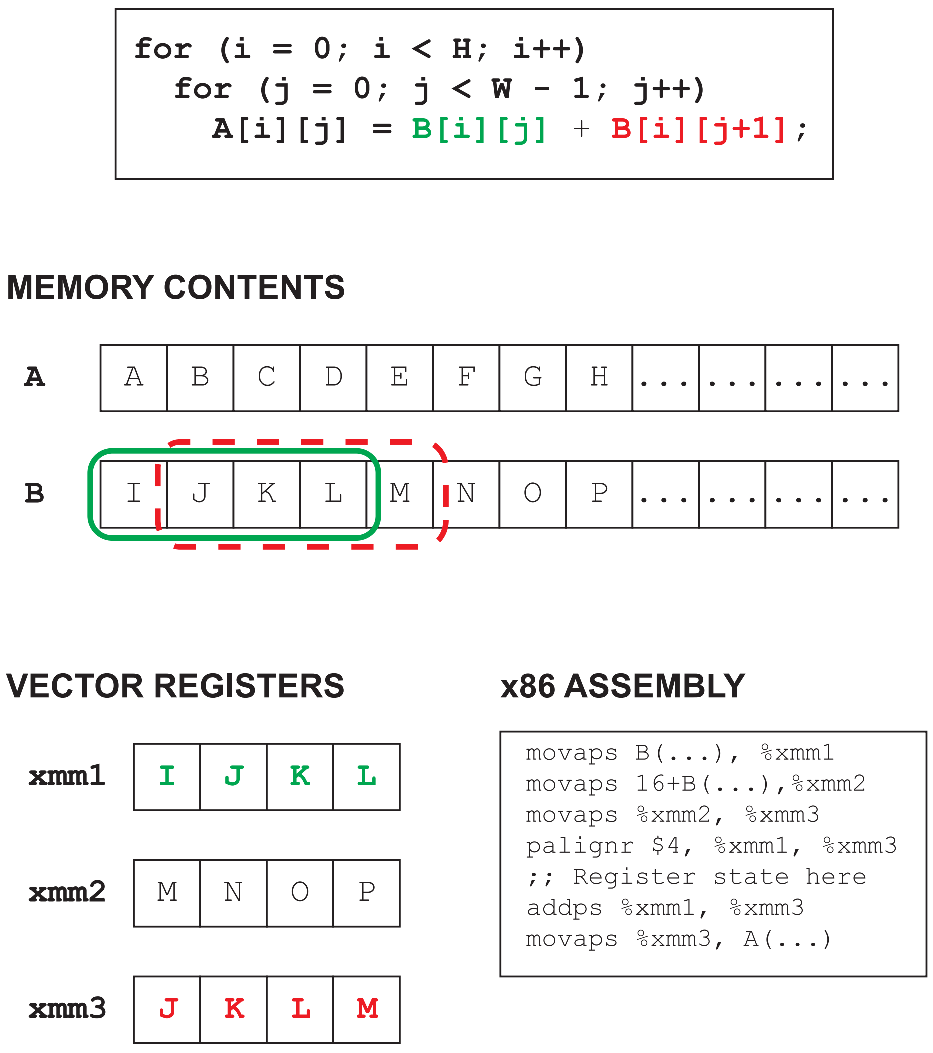
This chapter provides a more formal definition of a stream alignment conflict and provides an algorithm to detect stream alignment conflicts and the arrays affected by them.

4.1 Motivation

The reason for the significant performance difference between the two codes shown in Chapter 1, is that one of them (Figure 1.1(a)) exhibits what may be called *stream alignment conflict*, while the other (Figure 1.1(b)) is free of such conflicts. When stream alignment conflicts exist, the compiler must generate additional loads or inter-register data movement instructions (such as shuffles) in order to get interacting data elements into the same vector register positions before performing vector arithmetic operations. These additional data movement operations cause the performance degradation seen in Figure 1.1.

4.2 Illustrative Example

Figure 4.1 shows the x86 assembly instructions generated by the Intel ICC compiler on a Core 2 Quad system for code similar to that in Figure 1.1(a), along with


```
for (i = 0; i < H; i++)
  for (j = 0; j < W - 1; j++)
    A[i][j] = B[i][j] + B[i][j+1];
```

```
movaps B(...), %xmm1
movaps 16+B(...),%xmm2
movaps %xmm2, %xmm3
palignr $4, %xmm1, %xmm3
;; Register state here
addps %xmm1, %xmm3
movaps %xmm3, A(...)
```

Figure 4.1: Illustration of additional data movement for stencil operations

an illustration of the grouping of elements into vectors. The first four iterations of inner loop j perform addition on the pairs of elements `(B[0],B[1])`, `(B[1],B[2])`, `(B[2],B[3])`, and `(B[3],B[4])`. Four single-precision floating-point additions can be performed by a single SSE SIMD vector instruction (`addps`), but the corresponding data elements to be added must be located at the same position in two vector registers. To accomplish this, ICC first loads vectors `B[0:3]` and `B[4:7]` into registers xmm1 and xmm2. Next, a copy of vector `B[4:7]` is placed into register xmm3. Registers xmm1 and xmm3 are then combined with the `palignr` instruction to produce the unaligned vector `B[1:4]` in register xmm3. This value is used by the `addps` instruction to produce an updated value for `A[0:3]` in xmm3. Finally, an aligned store updates `A[0:3]`. The vector `B[4:7]` in register xmm2 is ready to be used for the update of `A[4:7]`.

In this example there is a stream alignment conflict between the two references to array `B`. ICC solves the problem on this architecture by using the `palignr` shuffle operation. This problem could also have been solved by using an unaligned load of `B[1:5]`. Both solutions, however, result in an expensive non-computational instruction in the innermost loop of the program.

4.3 Stream Alignment Conflict

We now use a number of examples to explain the issue of stream alignment conflicts. Before we proceed, we reiterate that the issue we address is a more fundamental algorithmic data access constraint and is not directly related to the hardware alignment restrictions and penalties on some SIMD instruction set architectures. For example, on IBM Power series architectures using the VMX SIMD ISA, vector loads and stores must be aligned to quadword boundaries. On x86 architectures, unaligned vector loads/stores are permitted but may have a significant performance penalty, as on the Core 2 architecture. On the more recent Core i7 x86 architecture, there is very little penalty for unaligned loads versus aligned loads. The performance difference on the Core i7 shown in Figure 1.1, however, provides evidence that the problem we address is a more fundamental algorithmic alignment issue. While hardware alignment restrictions/penalties on a target platform may exacerbate the problem with stream alignment conflicts, the problem exists even if an architecture has absolutely no restrictions/penalties for unaligned loads/stores. The work we present thus addresses a distinctly different problem than that addressed by many other works on optimizing code generation for SIMD vector architectures with hardware alignment constraints.

Vectorizable computations in innermost loops may be viewed as operations on streams corresponding to contiguous data elements in memory. The computation in the inner loop with statement S1 in Figure 1.1(b) performs the computation `C[i][0:N-1] = A[i][0:N-1]+ B[i][0:N-1]`, i.e. the stream of N contiguous data elements `A[i][0:N-1]` is added to the stream of N contiguous elements `B[i][0:N-1]` and the resulting stream is stored in `C[0][0:N-1]`. In contrast, the inner loop with statement S1 in Figure 1.1(a) adds `A[i][1:N]` and `A[i][0:N-1]`, where these two streams of length N are subsets of the same stream `A[i][0:N]` of length N+1, but one is shifted with respect to the other by one element. When such shifted streams are computed upon using current short-vector SIMD architectures, although only N+1 distinct elements are accessed, essentially $2 \times N$ data moves are required (either through additional explicit loads or inter-register data movement operations like shuffles). The extra moves are required because of the inherent characteristic of short-vector SIMD instructions that only elements in corresponding slots of vector registers can be operated upon.

While in the example of Figure 1.1(a) the stream alignment conflict is apparent because the misaligned streams arise from two uses of the same array in the same statement, the same underlying conflict may arise more indirectly, as illustrated in Figure 4.2.

```
      for (i = 0; i < N+1; ++i)
        for (j = 0; j < N+1; ++j) {
S3:       A[i][j] = B[i][j] + C[i][j];
S4:       D[i][j] = A[i][j] + C[i][j];
      }
```

(a) No stream alignment conflict

```
      for (i = 0; i < N+1; ++i)
        for (j = 0; j < N+1; ++j) {
S5:       A[i][j] = B[i][j] + C[i][j];
S6:       D[i][j] = A[i][j] + C[i][j+1];
      }
```

(b) Stream alignment conflict exists

```
      for (i = 0; i < N+1; ++i)
        for (j = 0; j < N+1; ++j) {
S7:       A[i][j] = B[i][j] + C[i][j];
S8:       C[i][j] = A[i][j] + D[i][j+1];
      }
```

(c) No stream alignment conflict

Figure 4.2: Illustration of indirect occurrence of stream alignment conflict

In Figure 4.2(a), the stream computations corresponding to the inner loop j are `A[i][0:N] = B[i][0:N]+C[i][0:N]` for S3 and `D[i][0:N] = A[i][0:N]+C[i][0:N]` for S4. Streams `A[i][0:N]` and `C[i][0:N]` are used twice, but all accesses are mutually aligned. With the example shown in Figure 4.2(b), however, the stream computations corresponding to the inner j loop are `A[i][0:N] = B[i][0:N]+C[i][0:N]` for S5 and `D[i][0:N] = A[i][0:N]+C[i][1:N+1]` for S6. Here we have a fundamental stream alignment conflict. If `A[i][0:N]` and `C[i][0:N]` are aligned together for S5, there is a misalignment between `A[i][0:N]` and `C[i][1:N+1]` in S6. Finally, there is no stream alignment conflict in Figure 4.2(c) since the same alignment of `A[i][0:N]` with `C[i][0:N]` is needed for both S7 and S8.

The abstraction of stream alignment conflicts is applicable with more general array indexing expressions, as illustrated by the examples in Figure 4.3. In Figure 4.3(a), there is no stream alignment conflict. In S9, streams `B[i][i:i+N]` and `B[i+1][i:i+N]` (the second and third operands) need to be aligned. Exactly the same alignment is needed between these streams in S10 (the first operand `B[i][i+1:i+N+1]` and the second operand `B[i+1][i+1:i+N+1]`).

```
  for (i = 1; i < N+1; ++i)
    for (j = 0; j < N+1; ++j) {
S9:    A[i][j]   = B[i-1][i+j] +
                   B[ i ][i+j] +
                   B[i+1][i+j];
S10:   A[i+1][j] = B[ i ][i+j+1] +
                   B[i+1][i+j+1] +
                   B[i+2][i+j+1];
    }
```

(a) No stream alignment conflict

```
  for (i = 1; i < N+1; ++i)
    for (j = 0; j < N+1; ++j) {
S11:   A[i][j]   = B[i-1][i+j+1] +
                   B[ i ][ i+j ] +
                   B[i+1][i+j+1];
S12:   A[i+1][j] = B[ i ][i+j+2] +
                   B[i+1][i+j+1] +
                   B[i+2][i+j+2];
    }
```

(b) Stream alignment conflict exists

Figure 4.3: Illustration of stream alignment conflict with general array indexing expressions

In contrast, in Figure 4.3(b) a fundamental algorithmic stream alignment conflict exists with the reused streams in S11 and S12. In S11, the stream `B[i][i:i+N]` (second operand) must be aligned with `B[i+1][i+1:i+N+1]` (third operand), but in S12 stream `B[i][i+2:i+N+2]` (first operand) must be aligned with `B[i+1][i+1:i+N+1]` (second operand). Thus the required alignments between the reused streams in S11 and S12 are inconsistent — a +1 shift of `B[i][x:y]` relative to `B[i+1][x:y]` is needed for S11 but a -1 shift of `B[i][x:y]` relative to `B[i+1][x:y]` is needed for S12 i.e., a stream alignment conflict exists.

Stream alignment conflicts often occur when vectorizing stencil computations, as shown in the examples of Section 1.1. Our goal in this chapter is to develop compiler algorithms for detecting stream alignment conflicts. Note that stream alignment conflict is not limited to stencil computations; therefore, dimension-lifted transposition is useful for a more general class of programs. In this work, we limit the discussion to vectorizable innermost loops. We assume that high-level transformations to expose parallel innermost loops that are good vectorization candidates have already been applied to the code [3]. The first task is to collect all vectorizable innermost loops. Some conditions must hold for our analysis to be performed: (1) all memory references are either scalars or accesses to (multidimensional) arrays; (2) all dependences have been identified; and (3) the innermost loop variable has a unit increment. Note that our framework handles unrolled loops as well; for ease of presentation, we will assume a non-unrolled representation of unrolled loops.

4.4 Program Representation

We first define our representation of candidate loops to be analyzed. The objective is to detect (a) if a loop is vectorizable, and (b) if it has a stream alignment conflict that can be resolved using dimension-lifted transposition of accessed arrays. We operate on an abstract program representation that is independent of the programming language and analyze the memory references of candidate innermost loops.

Innermost loops Innermost loops are the only candidates we consider for vectorization after dimension-lifted transposition of arrays. As stated earlier, we assume that such loops have a single induction variable that is incremented by one in each iteration of the loop. This is required to ensure the correctness of the analysis we use to characterize the data elements accessed by an innermost loop. We also require the innermost loops to have a single entry and exit point, so that we can precisely determine where to insert the data layout transformation (i.e., dimension-lifted transposition) code.

Memory references A loop contains a collection of memory references. All data elements accessed by any given iteration of the loop must be statically characterized by an access function f of the enclosing loop indices and other parameters. For a given execution of the loop whose iterator is `i`, all function parameters beside `i` must be constant.

f can be multidimensional for the case of an array reference, with one dimension in f for each dimension of the array. For instance, for the reference `B[i][alpha+M*j]` with innermost loop iterator j, the access function is $f_B(j) = (i, alpha + M.j)$.

4.5 Candidate Vector Loops

Dimension-lifted transposition may be useful only for arrays referenced in vectorizable loops. We now provide a simple compiler test to determine which inner loops are candidates for vectorization, based on data dependence analysis of the program. Specifically, we require a candidate innermost loop to meet the following constraints[1]: (1) the loop does not have any loop-carried dependence; and (2) two consecutive iterations of the loop access either the same memory location or two locations that are contiguous in memory, i.e., the loop has either stride-zero or stride-one accesses.

[1]The case of vectorizing reductions is not addressed here.

4.5.1 Loop-carried dependences

To determine if a loop is parallel, we rely on the standard concept of dependence distance vectors [2]. There is a dependence between two iterations of a loop if they both access the same memory location and one of these accesses is a write. If there is no loop-carried dependence, then the loop is parallel.

We now define the concept of a *array-distance vector* between two references, restricted to the inner loop (all other variables are assumed constant).

Definition 1 (Array-Distance Vector between two references). *Consider two access functions f_A^1 and f_A^2 to the same array* `A` *of dimension n. Let $\imath$ and $\imath'$ be two iterations of the innermost loop. The array-distance vector is defined as the n-dimensional vector $\delta(\imath,\imath')_{f_A^1,f_A^2} = f_A^1(\imath) - f_A^2(\imath')$.*

Let us illustrate Definition 1 with two references `A[i][j]` and `A[i-1][j+1]`, enclosed by loop indices i and j with j as the innermost loop iterator. We have $f_A^1(j) = (i, j)$ and $f_A^2(j) = (i-1, j+1)$. The array-distance vector is

$$\delta(j,j')_{f_A^1,f_A^2} = (1, j - j' - 1)$$

A necessary condition for the existence of a dependence carried by the innermost loop is stated below.

Definition 2 (Loop-carried dependence). *There exists a dependence carried by the innermost loop between two references f_A^1 and f_A^2, at least one of which is a write to memory, if there exists $\imath \neq \imath'$ such that $\delta(\imath,\imath')_{f_A^1,f_A^2} = \vec{0}$.*

Note that $\delta(\imath,\imath')_{f_A^1,f_A^2}$ requires the values of loop iterators other than the innermost loop to be fixed. Also, when $\delta(\imath,\imath')_{f_A^1,f_A^2}$, the difference between the access functions, still contains symbols beyond the inner-loop iterator (e.g., $\delta(\imath, \imath')_{f_A^1, f_A^2} = (k, alpha)$) we conservatively assume a loop-carried dependence unless the value of the symbols can be determined statically.

4.5.2 Stride-one memory accesses

The second condition for a loop to be a candidate for SIMD vectorization is that all memory accesses are such that two consecutive iterations access either consecutive memory locations or an identical location, for all iterations of the innermost loop.

To test for this property, we operate individually on each access function. Memory references that always refer to the same location for all values of the innermost loop are discarded: they are the equivalent to referencing a scalar variable. For each of the remaining references, we form the array-distance vector between two consecutive iterations, and ensure the distance between the two locations is equal to 1. For the case of multidimensional arrays, for the distance in memory to be 1, the array-distance vector must have a zero value in all but the last dimension where it must be one. This is formalized as follows:

Definition 3 (Stride-one memory access for an access function)**.** *Consider an access function f_A surrounded by an innermost loop. It has stride-one access if $\forall \imath$, $\delta(\imath, \imath + 1)_{f_A, f_A} = (0, \dots, 0, 1)$.*

For example, consider `A[i][j]` surrounded by the innermost j loop. $f_A = (i, j)$ and $\delta(j, j + 1)_{f_A, f_A} = (0, 1)$. Hence the j loop has stride-one access for f_A as above. Let us suppose that for the same f_A, loop i is the innermost loop; in this case, we would have $\delta(i, i+1)_{f_A, f_A} = (1, 0)$ which does not satisfy the condition in Definition 3. Therefore loop i does not have stride-one access with respect to f_A.

4.6 Detection of Stream Alignment Conflict

A stream alignment conflict occurs when a given memory location is required to be mapped to two different vector slots. So, it follows that if a memory element is not used twice or more during the execution of the innermost loop, then a stream alignment conflict cannot occur. If some memory element is reused, one of two cases apply: either the element is reused only during the same iteration of the innermost loop, or it is used by another iteration of the innermost loop. Stream alignment conflict can occur only for the latter; if the element is reused during the same iteration, it is mapped to the same vector slot and there is no stream alignment conflict.

4.7 Cross-iteration reuse due to distinct references

Our principle to detect if a given memory location is being referenced by two different iterations follows the concept of loop-carried dependence, but we extend now to *all pairs* of references and not limit to those containing a write operation.

For a given innermost loop, we thus compute the array-distance vector for all pairs of distinct references f^1 and f^2 to the same array, and see if there exists two distinct iterations such that

$$\delta(\imath, \imath')_{f^1, f^2} = \vec{0}$$

Consider the example of Figure 4.4(a). For the case of $f_B^1 = (i, j+1)$ and $f_B^2 = (i, j)$, we can determine if there is cross-iteration reuse by forming the constraint

$\delta(j,j')_{f_B^1,f_b^2} = (0, j+1-j')$ and solving the system of constraints

$$\mathcal{S} : \begin{cases} j \neq j' \\ 0 = 0 \\ j+1-j' = 0 \end{cases}$$

Since S has a solution, there is cross-iteration reuse in the innermost loop.

Technically, the presence of a solution to $\mathcal{S}$ is only a necessary condition for cross-iteration reuse. Some cases of cross-iteration reuse may not be detected, typically when distinct variables used in the access functions have the same value at runtime: the *evaluation* of the expression of δ may be 0, but simply doing the difference of the symbols may not give 0. Addressing this issue requires classical data-flow analysis problem for arrays [2] and can be solved with a more complete analysis and under stronger assumptions on the form of the input code. In our framework, the only consequence of not detecting dynamic cross-iteration reuse is that we may not have applied dimension-lifted transposition at places where it could have been useful. Our analysis is conservative in that sense but it requires only minimal assumptions on the input code.

```
for (i = lbi; i < ubi; ++i)
  for (j = lbj; j < ubj; ++j)
    A[i][j] = B[ i ][j+1] +
              B[ i ][ j ] +
              B[i-1][ j ];
```

(a)

```
 for (i = lbi; i < ubi; ++i)
   for (j = lbj; j < ubj; ++j)
   {
R:   A[i][j] += B[i][j-1];
S:   C[i][j] += B[i][ j ];
   }
```

(b)

```
for (i = lbi; i < ubi; ++i)
{
  A[i][lbj] += B[i][lbj-1];
  for (j = lbj+1; j < ubj; ++j)
  {
    A[i][ j ] += B[i][j-1];
    C[i][j-1] += B[i][j-1];
  }
  C[i][ubj] += B[i][ubj];
}
```

(c)

Figure 4.4: Code examples

4.8 Stream offset

The presence of a cross-iteration reuse does not necessarily imply the presence of a stream alignment conflict. In particular, all cases where cross-iteration reuse can be removed by iteration shifting (that is, a simple offset of the stream) do not correspond to a stream alignment conflict. Thus, in order to detect only the set of arrays to be dimension-lifted and transposed, we need to prune the set of references which do not generate a stream alignment conflict from the set of all references with cross-iteration reuse.

Consider the example of Figure 4.4(b) that has cross-iteration reuse. It is possible to modify the innermost loop such that the reuse disappears via *iteration shifting* [2]. Shifting is the iteration space transformation view of manipulating the stream offset, as it influences which specific *instance* of the statements are being executed under the same iteration of loop j. Consider shifting the second statement by 1, the transformed code is shown in Figure 4.4(c).

Definition 4 (Stream offset via shifting)**.** *Consider two statements R and S. Changing the stream offset is performed by shifting the iterations of R with respect to the iterations of S by a constant, scalar factor σ_R. All streams used by R have the same offset σ_R.*

There are two important properties of shifting for stream offset. First, as shown above, shifting can make cross-iteration reuse disappear in some cases. It can also introduce new cross-iteration reuse by assigning different offsets to streams associated with the same array.

Second, shifting is not always a legal reordering transformation. It may reorder the statements inside the loop, and change the semantics. This translates to a strong condition on the legality of iteration shifting.

Definition 5 (Legality of iteration shifting)**.** *Consider a pair of statements R and S such that they are surrounded by a common vectorizable innermost loop. Let σ_R be (resp. σ_S) the shift factor used to offset the streams of R (resp. S). If there is a dependence between R and S, then to preserve the semantics it is sufficient to set $\sigma_R = \sigma_S$.*

To determine if iteration shifting can remove cross-iteration reuse, we first observe that it changes the access functions of a statement R by substituting j with $j - \sigma_R$, given j as the innermost loop iterator. The variable j can be used only in the last dimension of the access function, since the loop is vectorizable with stride-one access. We then formulate a problem similar to that of cross-iteration reuse analysis, with the important difference being that we seek values of σ that make the problem infeasible; indeed if the problem has no solution, then there is no cross-iteration reuse. We also restrict it to the pairs of references such that all but the last dimension of the access functions are equal, as all cases of reuse require this property. To ease the solution process, we thus reformulate it into a feasibility problem by looking for a solution where $\imath = \imath'$ that is independent of the value of $\imath$.

Returning to the example in Figure 4.4(b), we have for `B`, $f_B^1(j) = (i, j-1)$ and $f_B^2(j) = (i, j)$. We first integrate the offset factors σ into the access function. As the two statements are not dependent, we have one factor per statement that can be independently computed. The access functions to consider are now $f_B^1(j) =$

$(i, j - \sigma_R - 1)$ and $f_B^2(j) = (i, j - \sigma_S)$. We consider the problem

$$\mathcal{T} : \begin{cases} j = j' \\ j - \sigma_R - 1 - j' + \sigma_S = 0 \end{cases}$$

If $\mathcal{T}$ has a solution for all values of j, that is, a solution independent of j then there is no cross-iteration reuse. For $\mathcal{T}$, $\sigma_R = 0$ and $\sigma_S = 1$ is a valid solution and this iteration shifting removes all cross-iteration reuse.

In order to find a valid solution for the whole inner-loop, it is necessary to combine all reuse equalities in a single problem: the σ variables are shared for multiple references and must have a unique value. Hence, for code in Figure 4.4(a), the full system to solve integrates $\delta(j, j')_{f_A^1, f_A^2}$, is augmented with σ_R and σ_S, and is shown below.

$$\mathcal{T} : \begin{cases} j = j' \\ j - \sigma_R - 1 - j' + \sigma_S = 0 & \text{Conditions for } \mathtt{B} \\ j - \sigma_R - j' - \sigma_S = 0 & \text{Conditions for } \mathtt{A} \end{cases}$$

$\mathcal{T}$ has no solution, showing that there is no possible iteration shifting that can remove all cross-iteration reuse in Figure 4.4(a). Dimension-lifted transposition is thus required.

4.9 Putting it all together

We now address the general problem of determining if a given innermost loop suffers from a stream alignment conflict that can be solved only via dimension-lifted transposition. That is, we look for cross-iteration reuse that cannot be eliminated via iteration shifting.

First, let us step back and precisely define in which cases dimension-lifted transposition can be used to solve a stream alignment conflict. Dimension-lifted transposition in essence spreads out the memory locations of references involved in a stream alignment conflict. In order to ensure there is no conflict remaining, one must precisely know, at compile-time, the distance in memory required to separate all elements. This distance must be a constant along the execution of the innermost loop. This translates to an additional constraint on the cross-iteration reuse that is responsible for the stream alignment conflict: the reuse distance must be a constant. We define the scope of applicability of dimension-lifted transposition as follows.

Definition 6 (Applicability of dimension-lifted transposition)**.** *Consider a collection of statements surrounded by a common vectorizable inner loop. If there exists cross-iteration reuse of a constant distance that cannot be eliminated by iteration shifting, then the stream alignment conflict can be solved with dimension-lifted transposition.*

If an array accessed in a candidate vector inner loop is dimension-lifted-and-transposed, all arrays in the inner loop are also dimension-lifted-and-transposed. The data layout transformation implies significant changes in the loop control and in the order the data elements are being accessed. All arrays must be dimension-lifted unless some computations simply could not be vectorized anymore. We present in Algorithm 4.1 a complete algorithm to detect which arrays are to be transformed by dimension-lifted transposition in a program.

Procedure `createIterationShiftingProblem` creates a system of equalities that integrates the shift factors σ as shown in Section 4.8. If this system has no solution, then at least one cross-iteration reuse remains even after iteration shifting. Since we have prevented the cases where the reuse is not a scalar constant, then the conflict can be solved with dimension lifting. We thus place all arrays of the loop into the list of arrays to be dimension-lifted-and-transposed.

Algorithm 4.1: GetDLTArrays()

```
Input: $P$: input program
Output: $Arrays$: the set of arrays to be dimension-lifted
$\mathcal{L} \leftarrow \emptyset$
forall the innermost loops $l$ in $P$ do
    forall the arrays $A$ referenced in $l$ do
        // Check loop-carried dependence
        forall the write references $f_A^1$ to $A$ do
            forall the references $f_A^2$ to $A$ do
                $\mathcal{S} \leftarrow \{\exists(\imath, \imath\prime), \delta(\imath, \imath\prime)_{f_A^1, f_A^2} = \vec{0}\}$
                if $\mathcal{S} \neq \emptyset$ then
                    goto next loop $l$
                end
            end
        end
        // Check stride-one
        forall the references $f_A$ to $A$ do
            $\mathcal{S} \leftarrow \{\forall \imath, \delta(\imath, \imath + 1)_{f_A, f_A} = (0, \ldots, 0, 1)\}$
            if $\mathcal{S} = \emptyset$ then
                goto next loop $l$
            end
        end
        // Check scalar reuse distance
        forall the pairs of references $f_A^1, f_A^2$ to $A$ do
            $\mathcal{S} \leftarrow \{\exists(\imath, \imath\prime)\delta(\imath, \imath\prime)_{f_A^1, f_A^2} = \vec{0}\}$
            if $\mathcal{S} \neq \emptyset$ then
                $\mathcal{S} \leftarrow \{\alpha \in \mathbb{Z}, \delta(\imath, \imath)_{f_A^1, f_A^2} = (0, \ldots, 0, \alpha)\}$
                if $\mathcal{S} = \emptyset$ then
                    goto next loop $l$
                end
            end
        end
        $\mathcal{L} \leftarrow \mathcal{L} \cup l$
    end
end
forall the $l \in \mathcal{L}$ do
    // Check conflict after iteration shifting
    $\mathcal{T} \leftarrow$createIterationShiftingProblem($l$)
    if $\mathcal{T} = \emptyset$ then
        $Arrays(l) \leftarrow$all arrays in $l$
    end
end
return $Arrays$
```

While solving $\mathcal{T}$, we compute values for σ to remove cross-iteration reuse. When it is not possible to remove all cross-iteration reuses with iteration shifting, we compute values for σ that minimize the distance between two iterations reusing the same element. The largest among all reuse distances in the iteration-shifted program is kept and used during code generation to determine the boundary conditions.

CHAPTER 5

Data Layout Transformation for Stencil Computations

In this chapter we present a novel approach through data layout transformation to avoid performance degradation due to stream alignment conflicts. We show how the poor vectorization resulting from stream alignment conflicts can be overcome through data layout transformation. As explained using Figure 4.1, the main problem is that adjacent elements in memory map to adjacent slots in vector registers, so that vector operations upon such adjacent elements cannot possibly be performed without either performing another load operation from memory to vector register or performing some inter-register data movement. The key idea behind the proposed data layout transformation is for potentially interacting data elements to be relocated so as to map to the same vector register slot.

5.1 Dimension-Lifted Transposition

We explain the data layout transformation using the example in Figure 5.1. Consider a one-dimensional array Y with 24 single-precision floating point data elements, shown in Figure 5.1(a), used in a computation with a stream alignment conflict, such as `Z[i] = Y[i-1] + Y[i] + Y[i+1]`.

Figure 5.1(b) shows the same set of data elements from a different logical view, as a two-dimensional 4×6 matrix. With row-major ordering used by C, such a 2D matrix will have exactly the same memory layout as the 1D matrix Y. Figure 5.1(c) shows a 2D matrix that is the transpose of the 2D matrix in Figure 5.1(b), i.e., a dimension-lifted transpose of the original 1D matrix in Figure 5.1(a). Finally, Figure 5.1(d) shows a 1D view of the 2D matrix in Figure 5.1(c).

It can be seen that the data elements A and B, originally located in adjacent memory locations Y[0] and Y[1], are now spaced farther apart in memory, both being in column zero but in different rows of the transposed matrix shown in Figure 5.1(c). Similarly, data elements G and H that were adjacent to each other in the original layout are now in the same column but different rows of the dimension-lifted transposed layout. The layout in Figure 5.1(c) shows how elements would map to slots in vector registers that hold four elements each. The computation of A+B, G+H, M+N, and S+T can be performed using a vector operation after loading contiguous elements [A,G,M,S] and [B,H,N,T] into vector registers.

5.2 Stencil Computations on Layout Transformed Data

Figure 5.2 provides greater detail on the computation using the transformed layout after a dimension-lifted transposition. Again, consider the following computation:

```
for (i = 1; i < 23; ++i)
Sb: Z[i] = Y[i-1] + Y[i] + Y[i+1];
```

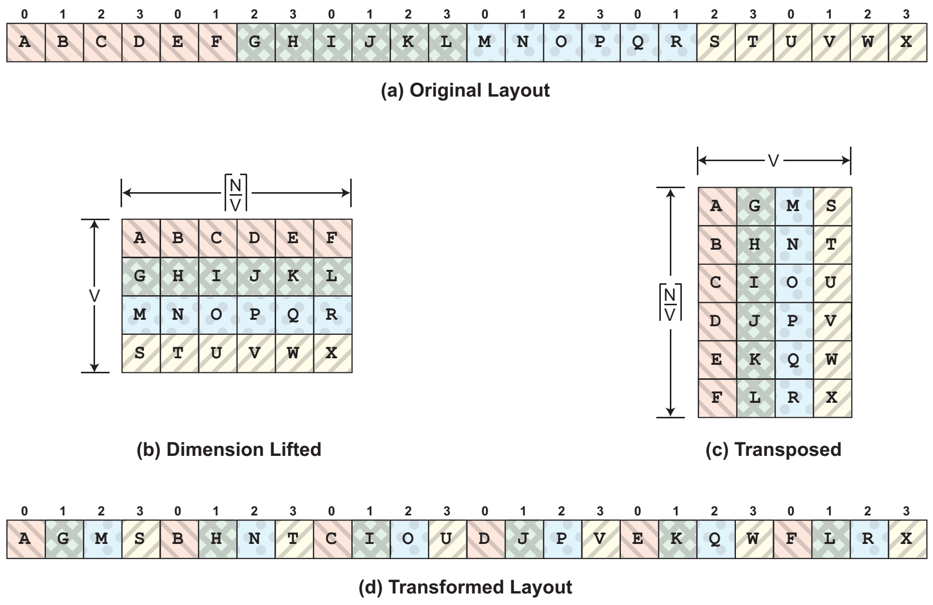


Figure 5.1: Data layout transformation for SIMD vector length of 4

Sixteen of the twenty two instances of statement Sb can be computed using full vector operations: [A,G,M,S] + [B,H,N,T] + [C,I,O,U]; [B,H,N,T] + [C,I,O,U] + [D,J,P,V]; [C,I,O,U] + [D,J,P,V] + [E,K,Q,W]; and [D,J,P,V] + [E,K,Q,W] + [F,L,R,X]. In Figure 5.2, these fully vectorized computations are referred to as the "steady state". Six statement instances (computing E+F+G, F+G+H, K+L+M, L+M+N, Q+R+S, and R+S+T) represent "boundary" cases since the operand sets do not constitute full vectors of size four. One possibility is to perform the operations for these boundary cases as scalar operations, however, it is possible to use masked vector operations to perform them more efficiently. Figure 5.2 illustrates the boundary cases corresponding to elements at the top and bottom rows in the transposed layout. At the top boundary, a "ghost cell" is created by performing a right-shift to the bottom row vector [F,L,R,X] by one position to align [X,F,L,R] above [A,G,M,S]. The

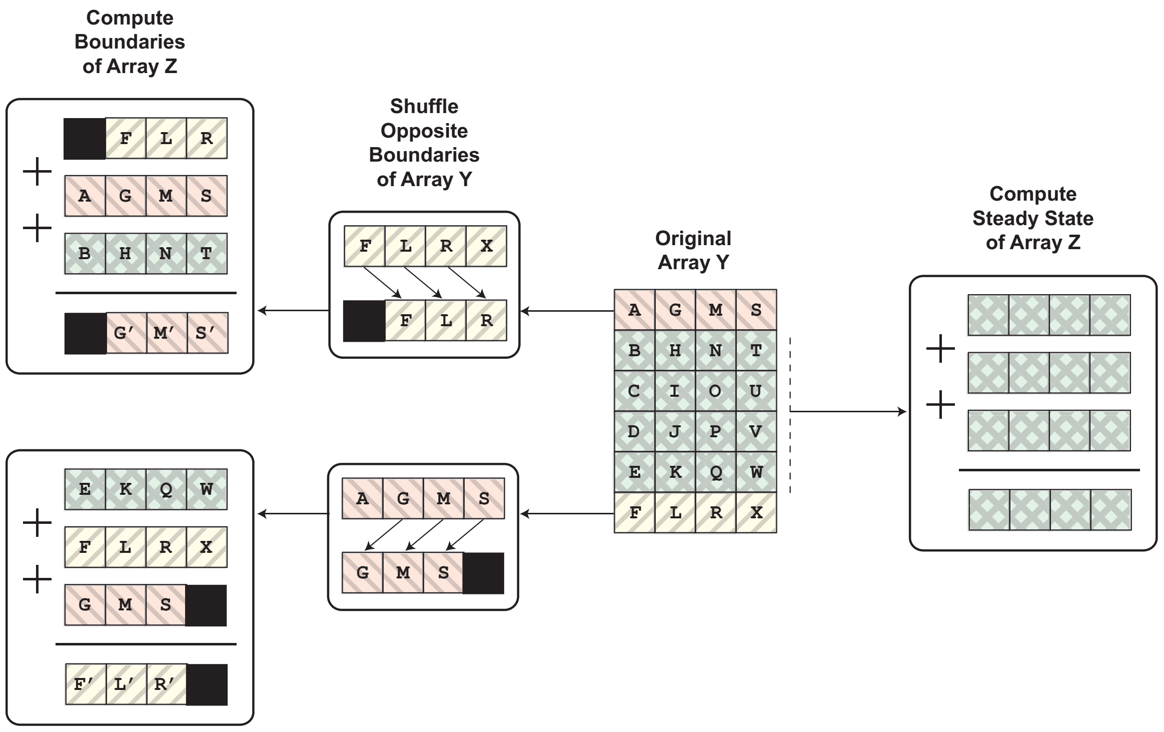


Figure 5.2: Illustration of steady state and boundary computation

vector operation [X,F,L,R]+[A,G,M,S]+[B,H,N,T] is then performed and a masked write is performed so that only the last three components of the result vector get written. The bottom boundary is handled similarly, as illustrated in Figure 5.2. Higher order stencils simply result in a larger boundary area. Further, the dimension lifted transpose layout transformation can be applied in the same manner for multi-dimensional arrays, as illustrated in Figure 5.3. The fastest varying dimension (rightmost dimension for C arrays) of a multi-dimensional array is subjected to dimension-lifting and transposition (with padding if the size of that dimension is not a perfect multiple of vector length). Figure 5.3 highlights a set of array locations accessed by a 2D 5-point stencil computation on a four wide vector architecture.

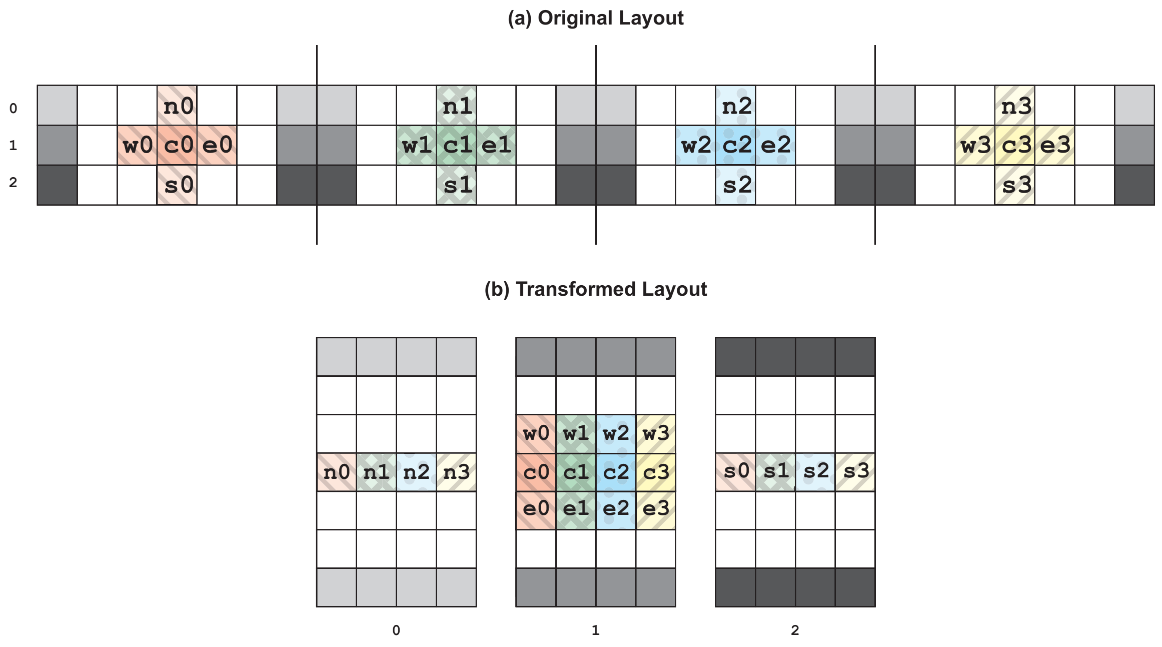


Figure 5.3: Illustration for 2D 5-point stencil

In the following section, we present a compiler framework to automatically detect where dimension lifting can be used to overcome the performance loss due to stream alignment conflicts.

5.3 Code Generation

In this section we present an algorithm for generating vectorized DLT code from an original code with a known stream alignment conflict. We also apply this algorithm, step-by-step, to a simple example.

Algorithm 5.1 describes the steps necessary to create correct vectorized DLT code. As stated above, this algorithm takes, as input, a segment of code known to have a stream alignment conflict. On line 1, the maximum reuse distance, q

of the stencil is determined as described in Secs. 4.7 and 4.9. This value will be used to (1) calculate the size of ghost cells to place at the end of each DLT array, (2) generate ghost cell copy code at the end of each iteration, and (3) adjust inner loop boundaries to account for ghost cells at the beginning of DLT arrays.

Line 2 uses Algorithm 4.1 to retrieve the set A of all arrays that will have the data layout transformation applied. Lines 3 and 4 generate code to allocate / free arrays that store the DLT data and copy the existing data into and out of the DLT form described in Section 5.1. Line 5 gets the set L of innermost loops in the stencil computation and line 6 creates an empty set L_{dlt} to place transformed loops in.

Lines 7–14 transform each inner loop of the stencil computation to a vectorized, DLT version. Line 8 generates end boundary code to copy data into the ghost cells at the end of each spatial sweep of the stencil. Line 9 vectorizes the statements located inside the inner loop. The algorithm used to vectorize statements has one key restriction: *all statements must be arithmetic expressions using **only** the assignment operator and operators that have corresponding SIMD instructions in the ISA we are generating code for, e.g. $+$ and `addps` for x86*.

Line 10 adjusts the loop boundaries of the inner loop to account for (1) the ghost cells at the beginning of the DLT arrays and (2) the reduction of the inner loop iteration count from N to $N/VLENGTH$ caused by vectorization. Line 11 combines the results of lines 8-10 into a vector loop operating over DLT arrays and line 12 places this loop into the set of transformed loops.

The algorithm concludes on line 14 with the creation of the full vectorized, DLT stencil code. Procedure `GenSCDLT` inserts the copy-in and copy-out code before and

Algorithm 5.1: CodegenDLT()

```
Input: $SC$: A stencil computation
Output: $SC_{dlt}$: A layout transformed version of $SC$
1  $q$ ←GetMaximumReuseDistance($SC$)
2  $A$ ←GetDLTArrays($SC$)
3  $C_i$ ←GenCopyIn($A$, $q$)
4  $C_o$ ←GenCopyOut($A$, $q$)
5  $L$ ←GetInnermostLoops($SC$)
6  $L_{dlt}$ ←$\emptyset$
7  forall the innermost loops $l$ in $L$ do
8      $B_s$ ←GenStartBoundary($l$, $q$)
9      $B_e$ ←GenEndBoundary($l$, $q$)
10     $S_v$ ←VectorizeStatements($l$)
11     $l_{adj}$ ←AdjustLoopBoundaries($l$, $q$)
12     $l_{dlt}$ ←GenDLTInnerLoop($l_{adj}$, $B_s$, $S_v$, $B_e$)
13     $L_{dlt}$ ←$L_{dlt} \cup l_{dlt}$
14     $SC_{dlt}$ ←GenSCDLT($SC$, $L_{dlt}$, $C_i$, $C_o$)
15 end
16 return $SC_{dlt}$
```

after the outer iterative loop of the original stencil code and substitutes all inner loops with vectorized DLT versions.

5.4 Codegen Example—1D 3-point Jacobi

In this section we apply Algorithm 5.1 to transform a 3 point, 1 dimensional Jacobi stencil into a vectorized, DLT version of the same code.

```
float k = 1.0/3.0;
float A[N], B[N];
for (t = 0; t < TMAX; ++t) {
  for (i = 1; i < N-1; ++i)
    A[i] = k*(B[i-1] + B[i] + B[i+1]);
  for (i = 1; i < N-1; ++i)
    B[i] = A[i];
}
```

Listing 5.1: Jacobi 1D, 3 point

Listing 5.1 gives the original source of the stencil. An outer time loop surrounds two inner loops, one that computes results for the current time step and another that copies data to an array to be used in the next time step.

We first obtain the maximum reuse distance, which in this case is 3. Next, we obtain all arrays to be layout transformed, giving us arrays `A` and `B`. After this, we generate copy-in and copy-out code for these arrays, shown in Listings 5.2 and 5.3.

```
// Begin Copy-in
int VLENGTH = 4;
int GHOST_SIZE = 3;

// Copy A from linear to ghost padded DLT
__m128 Adlt[N/VLENGTH + 2*GHOST_SIZE];
for (i = 0; i < N/VLENGTH; ++i)
  Adlt[GHOST_SIZE + (i % N/VLENGTH)] =
    insert(A[i], Adlt[GHOST_SIZE + (i % N/VLENGTH)], i/(N/VLENGTH));

// Copy B from linear to ghost padded DLT
__m128 Bdlt[N/VLENGTH + 2*REUSE];
for (i = 0; i < N/VLENGTH; ++i)
  Bdlt[GHOST_SIZE + (i % N/VLENGTH)] =
    insert(B[i], Bdlt[GHOST_SIZE + (i % N/VLENGTH)], i/(N/VLENGTH));

// Copy ghost cell values
for (i = 0; i < GHOST_SIZE; ++i) {
  Adlt[i] = right_shift(Adlt[N/VLENGTH - 1 + i]);
  Adlt[N/VLENGTH - 1 + i] = left_shift(Adlt[i]);
}

for (i = 0; i < GHOST_SIZE; ++i) {
  Bdlt[i] = right_shift(Bdlt[N/VLENGTH - 1 + i]);
  Bdlt[N/VLENGTH - 1 + i] = left_shift(Bdlt[i]);
}
```

Listing 5.2: Copy-in code for Jacobi 1D

We begin the copy-in code by defining the vector length we are using and the size of the ghost cell boundary areas at the start and end of the DLT arrays. For each array, we declare a DLT counterpart with a length equal to the integer division of the original length of the array by the vector length plus ghost cells.

We then loop through each element of the original array and insert it into the DLT array. The vector where each element is placed is determined by the expression `GHOST_SIZE + (i % N/VLENGTH)` while the slot in that vector is `i/(N/VLENGTH)`. The insert function `insert(s,v,pos)` takes as parameters scalar value `s`, destination vector `v`, and a destination vector slot `pos` and returns the vector `v` with `s` inserted into slot `pos`. Finally, initial values are copied into the ghost cells at the beginning and end of each array.

```
// Copy A from DLT to linear
for (i = 0; i < N; ++i)
  A[i] = extract(Adlt[GHOST_SIZE + (i % N/VLENGTH)], i/(N/VLENGTH));

// Copy B from DLT to linear
for (i = 0; i < N; ++i)
  B[i] = extract(Bdlt[GHOST_SIZE + (i % N/VLENGTH)], i/(N/VLENGTH));
```

Listing 5.3: Copy-out code for Jacobi 1D

The copy-out code shown in Listing 5.3 uses similar code to place data back into the linearized arrays. Each position in the original array is filled by a value from the DLT array using an `extract(v,pos)` function. Similar to the `insert` function, the `extract` function takes a vector `v` and slot `pos` and returns the scalar value at that location in the vector. [2]

[2]The exact implementations of the insert and extract functions are dependent upon architecture and can result in one or more intrinsic function calls depending upon the capabilities of the targeted vector instruction set.

After copy code is generated, the two innermost loops are retrieved. For the computation loop we generate the end boundary code to copy ghost cell values to the beginning and end of the array, shown in Listing 5.4.

```
// Copy ghost cell values
for (i = 0; i < GHOST_SIZE; ++i) {
  Adlt[i] = right_shift(Adlt[N/VLENGTH - 1 + i]);
  Adlt[N/VLENGTH - 1 + i] = left_shift(Adlt[i]);
}

for (i = 0; i < GHOST_SIZE; ++i) {
  Bdlt[i] = right_shift(Bdlt[N/VLENGTH - 1 + i]);
  Bdlt[N/VLENGTH - 1 + i] = left_shift(Bdlt[i]);
}
```

Listing 5.4: Ghost cell copy code for Jacobi 1D

Next, we vectorize the statements inside the inner loops. For the computation loop, we replace the `+` and `*` operations with calls to, respectively, `_mm_add_ps()` and `_mm_mul_ps()`. The constant `k` is broadcast to all slots of a vector using the `_mm_set1_ps()` intrinsic. Finally, all array references are replaced with DLT array references whose access functions are padded by the length of the ghost cell boundary. The final statement inside the compute loop is in Listing 5.5.

```
Adlt[GHOST_SIZE + i] =
  _mm_mul_ps(_mm_set1_ps(k),
             _mm_add_ps(Bdlt[GHOST_SIZE + i - 1],
                        _mm_add_ps(Bdlt[GHOST_SIZE + i],
                                   Bdlt[GHOST_SIZE + i + 1])));
```

Listing 5.5: Vectorized compute statement for Jacobi 1D

After vectorizing the statement the inner loop control flow is altered to compensate for the decrease in the size of the data array by a factor of `VLENGTH`. Since we adjust the array access functions when vectorizing the statements we do not have to compensate for ghost cells in the loop lower bound. Only the upper bound is adjusted to N/VLENGTH - 1.

Finally, we generate the transformed inner compute loop using the code generated above. The generated loop appears in Listing 5.6.

```
// Compute loop
for (i = 1; i < N/VLENGTH -1; ++i) {
  Adlt[GHOST_SIZE + i] =
    _mm_mul_ps(_mm_set1_ps(k),
               _mm_add_ps(Bdlt[GHOST_SIZE + i - 1],
                          _mm_add_ps(Bdlt[GHOST_SIZE + i],
                                     Bdlt[GHOST_SIZE + i + 1])));
}

// Copy ghost cell values
for (i = 0; i < GHOST_SIZE; ++i) {
  Adlt[i] = right_shift(Adlt[N/VLENGTH - 1 + i]);
  Adlt[N/VLENGTH - 1 + i] = left_shift(Adlt[i]);
}

for (i = 0; i < GHOST_SIZE; ++i) {
  Bdlt[i] = right_shift(Bdlt[N/VLENGTH - 1 + i]);
  Bdlt[N/VLENGTH - 1 + i] = left_shift(Bdlt[i]);
}
```

Listing 5.6: Vectorized compute loop with ghost cell copies for Jacobi 1D

Algorithm 5.1 repeats this process for the original copy loop. Finally, after the new inner loops have been generated, all components necessary for the final code are assembled into the code seen in Listing 5.7.

```
float k = 1.0f/3.0f; int VLENGTH = 4; int GHOST_SIZE = 3;
// Copy-in
__m128 Adlt[N/VLENGTH + 2*GHOST_SIZE];
for (i = 0; i < N/VLENGTH; ++i)
  Adlt[GHOST_SIZE + (i % N/VLENGTH)] =
    insert(A[i], Adlt[GHOST_SIZE + (i % N/VLENGTH)], i/(N/VLENGTH));
__m128 Bdlt[N/VLENGTH + 2*REUSE];
for (i = 0; i < N/VLENGTH; ++i)
  Bdlt[GHOST_SIZE + (i % N/VLENGTH)] =
    insert(B[i], Bdlt[GHOST_SIZE + (i % N/VLENGTH)], i/(N/VLENGTH));
for (i = 0; i < GHOST_SIZE; ++i) {
  Adlt[i] = right_shift(Adlt[N/VLENGTH - 1 + i]);
  Adlt[N/VLENGTH - 1 + i] = left_shift(Adlt[i]);
}
for (i = 0; i < GHOST_SIZE; ++i) {
  Bdlt[i] = right_shift(Bdlt[N/VLENGTH - 1 + i]);
  Bdlt[N/VLENGTH - 1 + i] = left_shift(Bdlt[i]);
}
// Compute
for (t = 0; t < TMAX; ++t) {
  for (i = 1; i < N/VLENGTH -1; ++i)
    Adlt[GHOST_SIZE + i] =
      _mm_mul_ps(_mm_set1_ps(k),
                 _mm_add_ps(Bdlt[GHOST_SIZE + i - 1],
                            _mm_add_ps(Bdlt[GHOST_SIZE + i],
                                       Bdlt[GHOST_SIZE + i + 1])));
  for (i = 0; i < GHOST_SIZE; ++i) {
    Adlt[i] = right_shift(Adlt[N/VLENGTH - 1 + i]);
    Adlt[N/VLENGTH - 1 + i] = left_shift(Adlt[i]);
  }
  for (i = 0; i < GHOST_SIZE; ++i) {
    Bdlt[i] = right_shift(Bdlt[N/VLENGTH - 1 + i]);
    Bdlt[N/VLENGTH - 1 + i] = left_shift(Bdlt[i]);
  }
  for (i = 1; i < N/VLENGTH -1; ++i)
    Bdlt[GHOST_SIZE + i] = Adlt[GHOST_SIZE + i];
  for (i = 0; i < GHOST_SIZE; ++i) {
    Adlt[i] = right_shift(Adlt[N/VLENGTH - 1 + i]);
    Adlt[N/VLENGTH - 1 + i] = left_shift(Adlt[i]);
  }
  for (i = 0; i < GHOST_SIZE; ++i) {
    Bdlt[i] = right_shift(Bdlt[N/VLENGTH - 1 + i]);
    Bdlt[N/VLENGTH - 1 + i] = left_shift(Bdlt[i]);
  }
}
// Copy-out
for (i = 0; i < N; ++i)
  A[i] = extract(Adlt[GHOST_SIZE + (i % N/VLENGTH)],i/(N/VLENGTH));
for (i = 0; i < N; ++i)
  B[i] = extract(Bdlt[GHOST_SIZE + (i % N/VLENGTH)], i/(N/VLENGTH));
```

Listing 5.7: Final vector DLT code for Jacobi 1D

5.5 Experimental Results

The effectiveness of the dimension-lifting transformation was experimentally evaluated on several hardware platforms using stencil kernels from a variety of application domains. First, we describe the hardware and compiler infrastructure used for experiments. Next, the stencil kernels used in the experiments are described. Finally, experimental results are presented and analyzed.

5.5.1 Hardware

We performed experiments on three hardware platforms: AMD Phenom 9850BE, Intel Core 2 Quad Q6600, and Intel Core i7-920. Although all are x86 architectures, as explained below, there are significant differences in performance characteristics for execution of various vector movement and reordering instructions.

Phenom 9850BE The AMD Phenom 9850BE (*K10h* microarchitecture) is an x86-64 chip clocked at 2.5 GHz. It uses a 128b FP add and 128b FP multiply SIMD units to execute a maximum of 8 single precision FP ops per cycle (20 GFlop/s). The same SIMD units are also used for double precision operations, giving a peak throughput of 10 GFlop/s. Unaligned loads are penalized on this architecture, resulting in half the throughput of aligned loads and an extra cycle of latency. The SSE shuffle instruction `shufps` is used by ICC for single precision inter- and intra-register movement. Double precision stream alignment conflicts are resolved by ICC generating consecutive `movsd` and movhpd SSE instructions to load the low and high elements of a vector register.

Core 2 Quad Q6600 The Intel Core 2 Quad Q6600 (*Kentsfield* microarchitecture) is an x86-64 chip running at 2.4 GHz. Like the Phenom, it can issue instructions to two 128-bit add and multiply SIMD units per cycle to compute at a maximum rate of 19.2 single precision GFlop/s (9.6 double precision GFlop/s). The `movups` and `movupd` unaligned load instructions are heavily penalized on this architecture. Aligned load throughput is 1 load/cycle. Unaligned load throughput drops to 5% of peak when the load splits a cache line and 50% of peak in all other cases. ICC generates the `palignr` SSSE3 instruction for single precision inter- and intra-register movement on Core 2 Quad. Double precision shifts are accomplished with consecutive `movsd-movhpd` sequences as previously described.

Core i7-920 The Intel Core i7-920 (*Nehalem* microarchitecture) is an x86-64 chip running at 2.66 GHz. SIMD execution units are configured in the same manner as the previously described x86-64 processors, leading to peak FP throughput of 21.28 single precision GFlop/s and 10.64 double precision GFlop/s. Unaligned loads on this processor are very efficient. Throughput is equal to that of aligned loads at 1 load/cycle in all cases except cache line splits, where it drops to 1 load per 4.5 cycles. Single precision code generated by ICC auto-vectorization uses unaligned loads exclusively to resolve stream alignment conflicts. Double precision code contains a combination of consecutive `movsd-movhd` sequences and unaligned loads.

5.5.2 Stencil Codes

We evaluated the use of the dimension-lifting layout transformation on seven stencil benchmarks, briefly described below.

Jacobi 1/2/3D

```c
for (t = 0; t < TMAX; ++t) {
  for (i = 1; i < N - 1; ++i) {
    out[i] = 1/3*(in[i-1] + in[i] + in[i+1]);
  }
  tmp = out; out = in; in = tmp;
}
```

Listing 5.8: 3 point Jacobi 1D stencil with pointer swap

The Jacobi stencil is a symmetric stencil that occurs frequently both in image processing applications as well as with explicit time-stepping schemes in PDE solvers. We experimented with one-dimensional, 2D, and 3D variants of the Jacobi stencil, and used the same weight for all neighbor points on the stencil and the central point.

In the table of performance data below, the 1D Jacobi variant is referred as J-1D. For the 2D Jacobi stencil, both a five point "star" stencil (J-2D-5pt) and 9 point "box"(J-2D-9pt) stencil were evaluated.

```
for (t = 0; t < TMAX; ++t) {
  for (i = 1; i < M - 1; ++i) {
    for (j = 1; j < N - 1; ++j) {
      out[i] = 1/9*(in[i-1][j-1] + in[i-1][j] + in[i-1][j+1] +
                    in[i  ][j-1] + in[i  ][j] + in[i  ][j+1] +
                    in[i+1][j-1] + in[i+1][j] + in[i+1][j+1]);
    }
  }
  tmp = out; out = in; in = tmp;
}
```

Listing 5.9: 9 point Jacobi 2D stencil with pointer swap

A seven point "star" stencil (J-3D-9pt) was used to evaluate performance of Jacobi 3D code.

```
for (t = 0; t < TMAX; ++t) {
  for (i = 1; i < M - 1; ++i) {
    for (j = 1; j < N - 1; ++j) {
      for (k = 1; j < P - 1; ++k) {
        out[i] = 1/7*(in[i-1][j][k  ] + in[i][j-1][k] +
                      in[i  ][j][k-1] + in[i][j  ][k] +
                      in[i  ][j][k+1] + in[i][j+1][k] +
                      in[i+1][j][k  ]);
      }
    }
  }
  tmp = out; out = in; in = tmp;
}
```

Listing 5.10: 7 point Jacobi 3D stencil with pointer swap

Heattut 3D This is a kernel from the Berkeley stencil probe and is based on a discretization of the heat equation PDE. [32].

FDTD 2D This kernel is the core computation in the widely used Finite Difference Time Domain method in Computational Electromagnetics. [42]

Rician Denoise 2D This application performs noise removal from MRI images and involves an iterative loop that performs a sequence of stencil operations.

Problem Sizes We assume the original program is tiled such that the footprint of a tile does not exceed the L1 cache size, thus all arrays are sized to fit in the L1 data cache. As is common for stencil codes, for each of the benchmarks, there is an outer loop around the stencil loops, so that any one-time layout transformation cost to copy from an original standard array representation to the transformed representation involves a negligible overhead.

Code versions For each code, three versions were tested:

- Reference, compiler auto-vectorized
- Layout transformed, compiler auto-vectorized
- Layout transformed, explicitly vectorized with intrinsics

5.5.3 Results

Absolute performance and relative improvement for single and double precision experiments across all platforms and codes are given in Figure 5.1. Intel C Compiler icc v11.1 with the '`-fast`' option was used for all machines. Vectorization pragmas were added to the inner loops of reference and layout transformed codes to force ICC auto-vectorization.

Double Precision

Double precision results are shown in columns labeled DP of Figure 5.1. Significant performance gains are achieved across all platforms and on all benchmarks.

		Phenom				Core2 Quad				Core i7			
		SP		DP		SP		DP		SP		DP	
		GF/s	Imp.	GF/s	Imp.	GF/s	Imp.	GF/s	Imp.	GF/s	Imp.	GF/s	Imp.
J-1D	Ref.	4.27	1.00×	3.08	1.00×	3.71	1.00×	2.46	1.00×	8.67	1.00×	3.86	1.00×
	DLT	7.68	1.80×	3.79	1.23×	9.42	2.54×	2.83	1.15×	10.55	1.22×	4.01	1.04×
	DLTi	11.38	2.67×	5.71	1.85×	13.95	3.76×	7.01	2.85×	15.35	1.77×	7.57	1.96×
J-2D-5pt	Ref.	6.96	1.00×	2.71	1.00×	3.33	1.00×	2.94	1.00×	8.98	1.00×	4.54	1.00×
	DLT	9.00	1.29×	3.75	1.38×	8.86	2.66×	4.58	1.56×	10.20	1.14×	5.18	1.14×
	DLTi	11.31	1.63×	5.67	2.09×	11.58	3.48×	5.85	1.99×	13.12	1.46×	6.58	1.45×
J-2D-9pt	Ref.	4.48	1.00×	3.21	1.00×	4.21	1.00×	2.72	1.00×	8.30	1.00×	4.11	1.00×
	DLT	7.71	1.72×	3.81	1.18×	8.04	1.91×	4.08	1.50×	10.23	1.23×	5.23	1.27×
	DLTi	12.26	2.74×	6.11	1.90×	12.01	2.85×	6.03	2.22×	13.62	1.64×	6.80	1.65×
J-3D	Ref.	6.01	1.00×	2.90	1.00×	6.07	1.00×	3.04	1.00×	9.04	1.00×	4.64	1.00×
	DLT	6.84	1.14×	3.73	1.29×	8.07	1.33×	4.25	1.40×	9.46	1.05×	5.02	1.08×
	DLTi	10.08	1.68×	5.36	1.85×	10.36	1.71×	5.31	1.75×	12.02	1.33×	6.04	1.30×
Heatttut-3D	Ref.	6.06	1.00×	3.02	1.00×	6.64	1.00×	3.29	1.00×	8.75	1.00×	4.55	1.00×
	DLT	7.12	1.18×	3.36	1.11×	8.71	1.31×	4.45	1.35×	9.99	1.14×	4.91	1.08×
	DLTi	9.59	1.58×	5.12	1.70×	8.86	1.33×	4.45	1.35×	11.99	1.37×	6.05	1.33×
FDTD-2D	Ref.	5.86	1.00×	3.26	1.00×	6.42	1.00×	3.35	1.00×	8.72	1.00×	4.34	1.00×
	DLT	6.89	1.18×	3.65	1.12×	7.71	1.20×	4.03	1.20×	8.91	1.02×	4.73	1.09×
	DLTi	6.64	1.13×	3.41	1.05×	8.03	1.25×	4.03	1.20×	9.74	1.12×	4.82	1.11×
Rician-2D	Ref.	3.29	1.00×	1.93	1.00×	1.87	1.00×	1.27	1.00×	3.98	1.00×	2.16	1.00×
	DLT	3.46	1.05×	2.40	1.25×	2.59	1.39×	1.27	1.00×	4.13	1.04×	2.23	1.03×
	DLTi	8.09	2.46×	2.56	1.33×	8.50	4.55×	1.27	1.00×	11.31	2.84×	2.23	1.03×

Table 5.1: Summary of DLT experimental results. Ref is the unoptimized, auto-vectorized version. DLT is the layout transformed, auto-vectorized version. DLTi is the layout transformed version implemented with vector intrinsics.

ICC auto-vectorized DLT code equaled or improved upon reference code performance in all cases. The harmonic means of relative improvements across all double precision benchmarks on x86-64 were 1.10× (Core i7), 1.22× (Phenom), and 1.28× (Core 2 Quad). Individual benchmark improvements range from, worst case, 1.00× (2D Rician Denoise on Core 2 Quad) to a best case of 1.56× (5 point 2D Jacobi on Core 2 Quad).

The auto-vectorized layout transformed code was fast but certain areas of it were still very inefficient. While ICC automatically unrolled the inner loop of reference code, no such unrolling was done for the layout transformed code. Further, ICC generated long sequences of scalar code for boundary computations. These deficiencies were addressed in the intrinsic versions of the codes. Scalar boundary code was replaced with much more efficient vector code, and all inner loops were unrolled. Further gains can be also be attributed to register blocking and computation reordering.

Intrinsic codes equaled or improved upon auto-vectorized versions in all cases, with a worst case improvement equal to reference (2D Rician Denoise on Core 2 Quad) and best case of 2.85× (Jacobi 1D on Core 2 Quad). Harmonic means of improvements over reference were 1.35× (Core i7), 1.60× (Phenom), and 1.57× (Core 2 Quad).

Single Precision

While most scientific and engineering codes use double precision for their computations, several image processing stencils use single precision. With the current SSE vector ISA, since only two double precision elements can fit in a vector, acceleration of performance through vectorization is much less than with single precision. However, the increasing vector size of emerging vector ISA such as AVX and LRBni, imply that the performance improvement currently possible with single precision SSE will be similar to what we can expect for double precision AVX, etc. For these reasons we include single precision performance data for all benchmarks.

Significant single precision performance gains are achieved across all platforms and on all stencils. They are reported in Figure 5.1 under the SP columns. Layout transformed code auto-vectorized by ICC ran significantly faster than reference code on all platforms. The harmonic means of relative performance improvements across all benchmarks on x86-64 were 1.11× (Core i7), 1.29× (Phenom), and 1.61× (Core 2 Quad). Individual benchmark improvements range from, worst case, 1.02× (2D FDTD on Core i7) to a best case of 2.66× (5 point 2D Jacobi on Core 2 Quad).

Vector intrinsic code optimizations again further increased the performance gains for auto-vectorized layout transformed code. All intrinsic codes were substantially faster than their corresponding auto-vectorized versions. Minimum relative improvement over reference on x86-64 was 1.12× (2D FDTD on Core i7) while maximum relative improvement was 4.55× (2D Rician Denoise on Core 2 Quad). Harmonic means of improvements over reference were 1.53× (Core i7), 1.81× (Phenom), and 2.15× (Core2 Quad).

Discussion

Performance gains for all x86-64 codes can be attributed to the elimination of costly intra-register movement, shuffle and unaligned load instructions from inner loop code. The performance gains on Core i7, while significant, were consistently the smallest obtained. This is partly explained by the relatively small performance penalty associated with unaligned loads and shuffle on this CPU. Still, the DLT intrinsic versions achieve a 1.53× average performance improvement for single precision and 1.35× for double precision codes on this platform. In contrast, the *Kentsfield* Core 2 Quad demonstrates consistently good performance improvements from layout transformation. This can mainly be attributed to poorly performing vector shuffle hardware.

Generally speaking, 1D Jacobi showed both the largest performance gains, and the fastest absolute performance, while higher dimensional stencils showed smaller, but still significant improvement. Higher dimensional stencils have more operands and more intra-stencil dependences. This leads to higher register occupancy, higher load / store unit utilization, and more pipeline hazards / stalls for these codes. This combination of factors leads to less improvement with respect to the 1D case. General and application-specific optimizations based on the data layout transformation described in this work could likely achieve higher performance through careful instruction scheduling and tuning of register block sizes to address these issues.

CHAPTER 6

Split Tiling

In this chapter we develop a method used to extend the data layout transformation described in Chapter 5 to larger problems that do not fit in the cache of a modern CPU. This is accomplished through the adaptation of the split tiling technique proposed in [34] to vector DLT stencil codes.

6.1 Motivation

We next use a Jacobi 1D stencil example to explain the problem with the use of standard time-tiling in conjunction with DLT layout transformation. Although the input to our stencil compiler uses a special DSL language (briefly described in Section 6.2.1), we use standard loop notation in C to motivate the problem since this lower-level view makes it easier to discuss issues pertaining to loop fusion and time-tiling when compiling general multi-statement stencils for high performance – something that to the best of our knowledge is not addressed by other stencil compilers such as PATUS [7] and Pochoir [52].

Figure 6.1(a) shows code for a 1D Jacobi 3-point stencil with a sequence of two spatial loops within an outer time loop, where S1 performs the stencil computation over the spatial domain and S2 copies the output array into the input array

```
for (t=0; t<T; t++) {
  for (i=1; i<N-1; i++)
    B[i] = 0.33*(A[i-1] + A[i] + A[i+1]); // S1
  for (i=1; i<N-1; i++)
    A[i] = B[i]; // S2
}
```

(a) Unfused

```
for (t=0; t<T; t++) {
  for (i=1; i<N-1; i++)
    B[i] = 0.33*(A[i-1] + A[i] + A[i+1]); // S1
  for (i=1; i<N-1; i++)
    A[i] = B[i]; // S2
}
```

(b) Fused

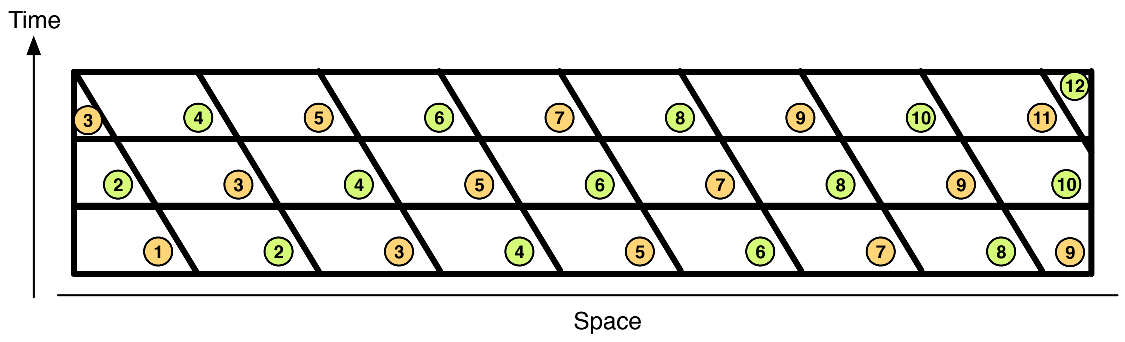
Figure 6.1: Jacobi 1D stencil

for use in the next time step. In order to enhance data locality, time-tiling may be employed, but will first require some transformations in order to create atomic tiles that compute forward for several time steps over a subset of the spatial domain that is small enough to fit within cache. Figure 6.1(b) shows a fused form that creates a unified 2D iteration space with a statement body including both S1 and S2 (along with peeling of an iteration at the boundaries of the i loop).

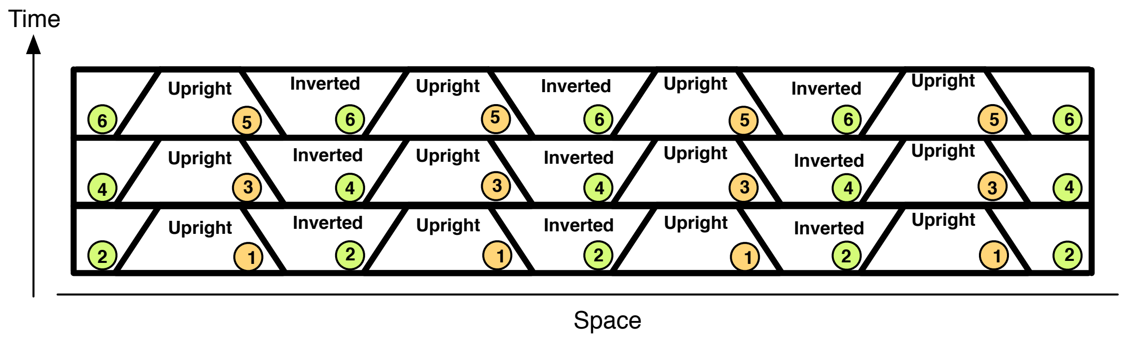
Further skewing of this unified iteration space will be required to create valid "rectangular" tiles, which can equivalently be viewed as parallelogram-shaped tiles in an unskewed iteration space, as shown in Figure 6.2(a). Because of the shape of valid tiles (they cannot be rectangular in an unskewed iteration space due to forward and backward dependences along the spatial dimension), there are inter-tile dependences between adjacent tiles along both the time and spatial dimensions.

This inter-tile dependence along the spatial dimension makes it infeasible to use DLT because DLT causes spatially separated data elements (for example, `B[0]`, `B[6]`, `B[12]`, `B[18]` in Figure 5.1) to be gathered together in a single vector and therefore must be operated upon concurrently. The circled value in each tile of Figure 6.2(a) represents the logical time at which the tile can be executed, such that all tiles it depends on have been previously computed.

When performing the DLT transformation we effectively parallelize a stencil computation. Each vector slot can be thought of as a thread, and each performs computations in parallel. When executing DLT code each vector slot computes results using operands located at substantial distances from each other in the original data layout. This leads to complications when attempting traditional time tiling. Figure 6.2(b) shows a different form of tiling – split tiles. Here, upright and inverted tiles alternate and the inter-tile dependences are only from an upright tile to its two neighboring inverted tiles. As a result, concurrent execution of all upright tiles over a given time range is feasible, followed by concurrent execution of the inverted tiles over the same time range. Again, the circled values within the tiles indicate the sequence of execution of the tiles, where tiles with the same sequence number can be executed concurrently. With such a tiling strategy, it is now feasible to use DLT, as long as all data elements grouped into each vector are all within upright tiles or all within inverted tiles. Further, unlike execution required with standard tiling, the schedule for parallel tile execution with split tiles is fully load balanced and does not have a sequential start and gradual build up of inter-tile parallelism as required for wavefront-parallel standard tiling.



(a) Standard Tiling



(b) Split Tiling

Figure 6.2: Tiled iteration space for 1D space, 1D time

Thus, it has been shown that the time tiling a stencil in the traditional manner will not work with layout-transformed data and a different approach is needed.

6.2 Overview of Approach

Before delving into the details of the algorithms used to split tile code, a brief introduction to the translated Stencil Domain Specific Language (SDSL) is provided. Next, two more advanced split tiling methods used to tile SDSL codes are summarized. Finally, a high-level overview of the approach used to transform an input SDSL program into an semantically equivalent split tiled DLT program is sketched.

6.2.1 Introduction to SDSL

A stencil computation can be summarized as one or more functions being identically applied to points on a regular grid, where the values of some groups of neighboring elements are used for each function, and this process is repeated multiple times. The Stencil Domain Specific Language (SDSL) enables the concise description of stencil computations and is briefly described in the following sections.[3]

Program Description

The computation shown in Listing 6.1 is a standard Jacobi 2D computation, which averages the value of the 5 neighboring points (up, down, left, right, and center) to compute the new value of the center point.

```
1  int dim1;
2  int dim0;
3
4  grid g[dim1][dim0];
5  double griddata a on g at 0,1;
6
7  iterate 100 {
8    stencil five_pt {
9      [1:dim1-2][1:dim0-2] : [1]a[0][0] = 0.2*([0]a[-1][0]+
10                                  [0]a[0][-1]+[0]a[ 0][0]+[0]a[0][1]+
11                                              [0]a[ 1][0]);
12   }
13 }
```

Listing 6.1: A simple Jacobi 2D example in SDSL

Structural Mesh (`grid`) Line 4 of Listing 6.1 defines `g`, the grid where stencil computations may be defined. It is an N-dimensional Cartesian coordinate space (a

[3]A more comprehensive description of SDSL and a detailed end-to-end example of building a complex stencil with SDSL can be found in Section 7.4.

subset of $\mathbb{N}^2$ here), and the computations operate on a subset of this grid. We note that grid size can be a parameter, that is a program constant whose value is not known at compile-time.

Data Elements (`griddata`) Line 5 of Listing 6.1 defines `a`, a double precision data field with the same structure as grid `g`. This field holds data values used in stencil functions, and multiple fields may be defined over a grid. The grid `g` is used to define the size of `a` and sets limits on field indices. The `at` clause specifies that there should be two copies of the field, one associated with the current outermost loop iteration and another at the next outermost loop iteration.

Computation (`iterate` and `stencil`) The last eight lines of Listing 6.1 define a stencil computation. Three key concepts are defined: (1) outer loop trip count, (2) subgrid(s) over which to apply a stencil function and (3) stencil function(s).

The outer loop trip count is defined in the `iterate` construct, and is 100 in the example. The stencil construct is given a unique identifier, `five_pt`, and contains the definition of a subgrid over which to apply the stencil function definition that follows. In the example the subgrid `[1:dim1-2][1:dim0-2]` defines a subset of the grid `g` that contains all elements except a single cell border on all four sides.

A stencil function is defined after the subgrid definition. This function averages the current point and four of its neighbors in `a` at the current timestep and places the results in `a` at the next timestep. References to `griddata` consist of the offset from the current iteration in brackets, followed by the name of the referenced field, followed by offsets from the current point in each spatial dimension in brackets.

General Form of an SDSL Program

In general, an SDSL program contains one `grid`, one or more `griddata`, one `iterate`, and one or more `stencil` definitions, where each `stencil` may define one or more subgrids and the stencil functions that operate upon them.

The abstract form of an SDSL program is represented in Listing 6.2. The program is constrained to be a collection of M K-dimensional grid data and N stencils, with each stencil applying some stencil function f on one or more grid data. Each stencil function is executed on a rectangular subgrid $Z \subset \mathbb{Z}^K | 0 \leq lb_Z^k, ub_Z^k < dim_k \forall k \in \{1..K\}$. While an SDSL program may have multiple subgrids and stencil functions defined inside one `stencil`, the abstract version in Listing 6.2 is semantically equivalent.

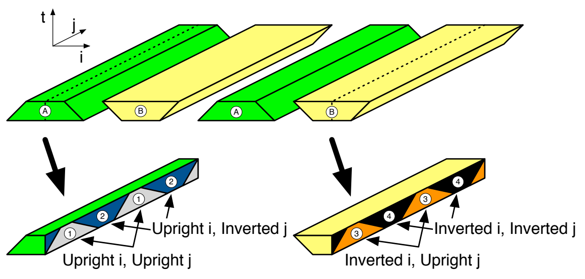
```
grid g[dimK]...[dim1];

griddata g1,g2...,gM on g;

iterate T {
  stencil s1 {
    [lb_s1_K:ub_s1_K]...[lb_s1_1:ub_s1_1] : f1(...);
  }
  stencil s2 {
    [lb_s2_K:ub_s2_K]...[lb_s2_1:ub_s2_1] : f2(...);
  }
  ...
  stencil sN {
    [lb_sN_K:ub_sN_K]...[lb_sN_1:ub_sN_1] : fN(...);
  }
}
```

Listing 6.2: General form of an SDSL program

SDSL programs are parallelized and optimized for data locality using nested and hybrid split tiling, described in the next two sections.



```
for tt
  parfor ii // (A) Upright i
    parfor jj // (1) Upright j
      for t { for i { for j {}}};
    barrier();
    parfor jj // (2) Inverted j
      for t { for i { for j {}}};
    barrier();
  parfor ii // (B) Upright j
    parfor jj // (3) Upright j
      for t { for i { for j {}}};
    barrier();
    parfor jj // (4) Inverted j
      for t { for i { for j {}}};
    barrier();
```

Figure 6.3: 2D nested split tiling.

6.2.2 Nested Split Tiling

In nested split tiling, a d-dimensional loop spatial loop nest is recursively split tiled along each dimension. The outermost spatial loop at level d is split tiled, producing a loop over upright tiles and a loop over inverted tiles. Inside each of these loops, loop level $d - 1$ is split tiled, giving four tile loop nests. Split tiling is performed recursively in each new loop nest until the base loop level 1 is reached and there are 2^d total loop nests corresponding to all possible combinations of upright and inverted tiles on each dimension. Nested split tiling of a 2D code is illustrated in Figure 6.3.

Figure 6.3 depicts, on the left, a series of upright ('A') and inverted ('B') tiles in the i dimension. All upright 'A' tiles may be executed concurrently, followed by all inverted 'B' tiles. Below these tiles are representative cross-sections of an upright and inverted tile showing the nested split tiles in the j dimension. These tiles are labeled such that all tiles with the same number, ('1', '2', '3', or '4') may be executed concurrently, and tiles with a lower number must be executed before tiles with a higher number.

The pseudocode in Figure 6.3 shows the loop nests responsible for producing the diagram. Nested inside the sequential tt loop are two parallel ii loops corresponding to the 'A' and 'B' tiles shown in the diagram. Nested inside the 'A' loop are parallel jj upright ('1') and inverted ('2') tile loops corresponding to the tiles shown in the left cross-section, Similarly, the 'B' loop contains nested parallel '3' and '4' loops corresponding to the right cross-section. A barrier follows each jj tile loop to enforce tile execution order and ensure that no dependences are violated.

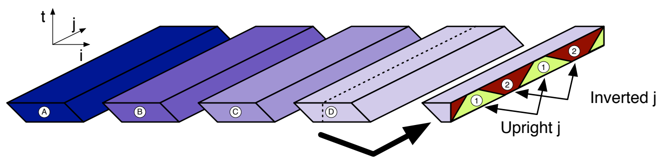
Nested split tiling enables parallelization of all spatial loop nests in a stencil, however (1) it imposes a lower bound on the size of upright tiles for a given time tile size, or equivalently, (2) it imposes an upper bound on the time tile size given an upright tile's size.

In nested split tiling, upright tiles must be sized such that they retain their characteristic trapezoidal shape, as in Figure 6.2(b). If the base of the upright tile is not large enough for a given time tile size, the sloping lines will eventually form a tip. At this point tile execution cannot extend any further in time.

Given an upright tile with a base size of T_U^d, maximum absolute value of slopes in d s_{max}^d, maximum offset of all statements o_{max}^d, and time tile size T_T the following constraint can be stated:

$$T_U^d \geq 2 * T_T * s_{max}^d + 2 * o_{max}^d$$

For higher dimensional problems, the lower bound on upright tile size causes tiles to overflow cache for even small time tile sizes. Consider a 3-dimensional stencil with, for all dimensions, maximum slope $s_{max}^d = 2$, maximum offset $o_{max}^d = 0$,



```
for tt
  for ii // (A) (B) (C) (D) Traditional i
    parfor jj // (1) Upright j
      for t { for i { for j {}}};
    barrier();
    parfor jj // (2) Inverted j
      for t { for i { for j {}}};
    barrier();
```

Figure 6.4: 2D hybrid split tiling.

and $T_T = 8$. This requires $T_U^d \geq 32$. For an upright tile in all dimensions, including the innermost vector dimension, this is at least 32K vector elements, enough to overflow L1 and L2 cache on most modern architectures.

6.2.3 Hybrid Split Tiling

We overcome the tile size constraints of nested split tiling with a hybrid of standard tiling on the outermost space loops and split tiling on the inner loops. Hybrid split tiling for a 2D stencil is illustrated in Figure 6.4. The pseudocode contains a single ii loop nested in the tt loop which corresponds to the four traditional i dimension tiles 'A', 'B', 'C', and 'D' shown in the diagram. These tiles must be executed in sequence from 'A'–'D'. Nested inside the ii loop is the split tiled jj loop which has the same upright / inverted tile structure as the split tiled inner loops described in the previous section.

Standard tiling does not impose any constraint on tile sizes along spatial dimensions as a function of the time tile size. Thus, standard tiles may be compacted to a much smaller size to compensate for the larger tile sizes required by split tiled dimensions. This allows for a substantially reduced multidimensional tile footprint.

Algorithm 6.1: Split tiling / DLT overview

```
   Input: $P$: SDSL program, $dsplit$: number of dimensions to split tile
   Output: $O$: optimized C program
1  $O \leftarrow$PerformDLT($P$)
2  ($\vec{\alpha}$,$\vec{\beta}$,$\vec{o_L}$,$\vec{o_U}$) $\leftarrow$Backslice($P$)
3  $O \leftarrow$PerformSplitTiling($O$, $\vec{\alpha}$, $\vec{\beta}$, $\vec{o_L}$, $\vec{o_U}$, $dsplit$)
4  return $O$
```

In the example at the end of Section 6.2.2, we may reduce the tile size of the outermost dimension to 2, thereby reducing the tile size to 2K elements. Since inner loops are split tiled, we retain adequate parallelism.

6.2.4 Overview of Optimization Algorithm

In order to perform combined data layout transformations for SIMD vectorization with parallel tiling for data locality, we use a multi-stage process to integrate DLT with multi-level split tiling. The overall transformation flow is summarized in Algorithm 6.1.

Function `PerformDLT` applies DLT on all inner-most vectorizable loops, following the method presented in Henretty et al. [26]. We remark that programs that can be modeled in SDSL all have vectorizable inner-loops, so that DLT can be applied for all stencil functions. Details of this function are provided later in Section 6.4.1. Function `Backslice` performs backslicing analysis to compute the exact shape of the split tiles (that is, computing for each stencil function the offsets and slopes of a split tile, to be translated on the entire spatial domain). This is detailed in Section 6.3. Function `PerformSplitTiling` uses the split tile shape information computed to generate split tile code for the d split inner spatial dimensions. This is detailed in Section 6.4. Finally, function `finalizeTiling` completes code generation, by applying standard tiling on the remaining dimensions, if any. The integration as well as the complete algorithm is discussed in Section 6.4.1.

6.3 Backslicing Analysis

Split tiling requires the computation of sets of iteration space points that can be executed atomically – that is a valid tiling – while preserving parallelism between tiles of the same category (i.e., upright tiles have to be parallel with each other). In order to compute the *shape* of the split tile that satisfies those properties, we first highlight the main ideas using a Jacobi 1D stencil, before describing the general algorithm for higher dimensional stencils with an arbitrary number of stencil functions.

6.3.1 Split Tiling Jacobi 1D

We illustrate the main ideas behind split tiling using a Jacobi 1D example. Listing 6.3 shows the corresponding input SDSL program.

```
grid g[1000];
double griddata a on g at 0,1;

iterate 100 {
  stencil f1 {
    [1:998] : [1]a[0] =
            0.33*([0]a[-1]+[0]a[0]+[0]a[1]);
  }
}
```

Listing 6.3: A simple Jacobi 1D example in SDSL

In the SDSL intermediate representation, an explicit copy of the field a1 into the field a0 is added after each time iteration, leading to a program equivalent to the C code shown in Listing 6.4.

```c
for (t = 0; t < 100; t++) {
  for (i = 1; i <= 998; i++) {
    f1: a1[i] = 0.33*(a0[i-1] + a0[i] + a0[i+1]);
  }
  for (i = 0; i <= 999; i++) {
    f2: a0[i] = a1[i];
  }
}
```

Listing 6.4: A simple Jacobi 1D example in C

Statement `f1` performs the actual stencil computation, producing the `a1` field, statement `f2` copies the `a1` field to the `a0` field. This sequence is repeated 100 times.

Examining an Upright Tile

Let us consider the top segment of an upright tile for `f2`, over a span $[P, Q]$, that corresponds to the iterations of `f2` performed at time T. In order to correctly compute those iterations, we need the values $[P-1, Q+1]$ of `a1` that were computed by executing `f1` on the segment $[P-1, Q+1]$ time T, which in turn depends on the values of `a0` over $[P-2, Q+2]$ copied by `f1` at time $T-1$. Based on data dependences between the statements `f1` and `f2`, we can compute precisely which iterations must have been computed at previous time steps for each of the statements in order to compute the segment $[P, Q]$ of `f2` at time step T. This is illustrated in Figure 6.5.

Figure 6.5 shows the set of preceding iterations, for both `f1` and `f2`, that must be computed in order to obtain the segment $[P, Q]$ at time T. We show here a time tile size of 3, that is, we build a split tile that computes over three time steps.

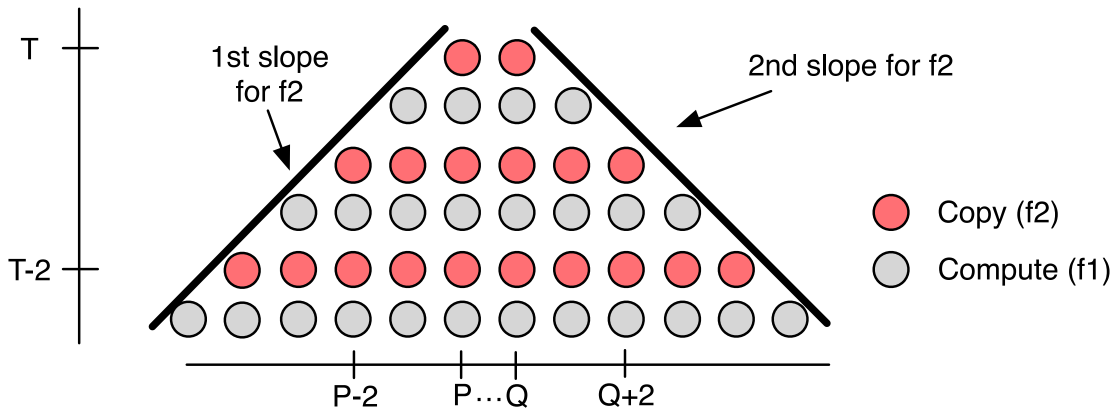


Figure 6.5: Upright split tile for Jacobi 1D

The dependences are analyzed from the SDSL representation. Due to the restriction on stencil shapes to be constant integer offsets (e.g., -1, 2, etc.), the dependences are simple integer relations between time and the access functions. In the next sections we show how data dependences are used to construct a *dependence summary graph* and formulate validity constraints on the split tiles for the slopes and statement offsets.

Building the Dependence Summary Graph

We begin by creating the dependence summary graph (DSG), a multigraph with vertices for each stencil function and edges that summarize flow and anti dependence information between stencil functions. In general, a vector of 2*d+1 components is used to model data dependence in an imperfectly nested loop with maximum loop depth d, with d components representing the distances along the loops, and the other d+1 components being used to mark the relative textual ordering within a loop level. However, the structure of an SDSL program always has the form of an outer time loop surrounding a sequence of perfectly nested loops. For generation of valid split tiled code, the exact textual position of a sequence of statements is not significant, but only whether a dependence is from a textually preceding or succeeding statement within the time loop. Further, when several dependences exist between a pair of statements due to multiple array read references it is only necessary to identify the maximal spatial extent of dependences along the different directions at each time step. Therefore, instead of using the standard general representation of dependence vectors, we separate out the distance vector component along the time (outermost) dimension and the components along the spatial dimensions.

For the Jacobi 1D example, we have the following dependences:

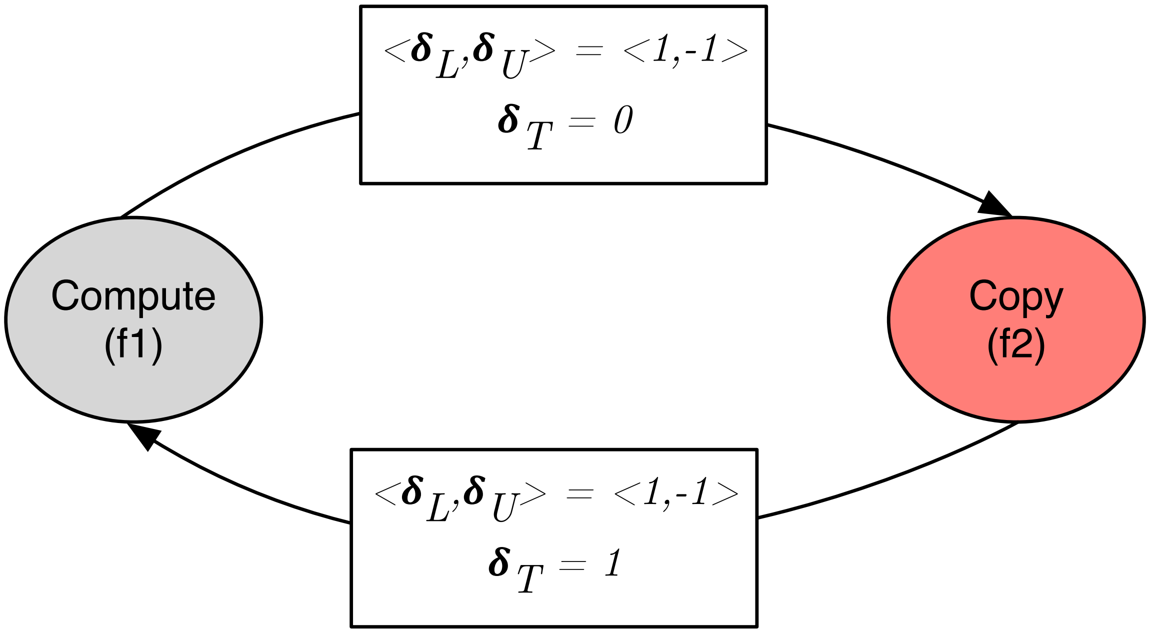
$$\mathcal{D}_{\text{f1}\to\text{f2}} = \begin{cases} \text{flow: f1}(t,i) & \to & \text{f2}(t,i) \\ \text{anti: f1}(t,i) & \to & \text{f2}(t,i-1) \\ \text{anti: f1}(t,i) & \to & \text{f2}(t,i) \\ \text{anti: f1}(t,i) & \to & \text{f2}(t,i+1) \end{cases}$$

$$\mathcal{D}_{\text{f2}\to\text{f1}} = \begin{cases} \text{flow: f2}(t,i) & \to & \text{f1}(t+1,i-1) \\ \text{flow: f2}(t,i) & \to & \text{f1}(t+1,i) \\ \text{flow: f2}(t,i) & \to & \text{f1}(t+1,i+1) \\ \text{anti: f2}(t,i) & \to & \text{f1}(t+1,i) \end{cases}$$

The spatial components of the dependence vectors between dependent statements are computed by subtracting the target iteration from the source iteration, yielding the following vectors

$$d_{\text{f1}\to\text{f2}} = \begin{cases} \delta_T = 0, \delta_i =< 0 > \\ \delta_T = 0, \delta_i =< -1 > \\ \delta_T = 0, \delta_i =< 0 > \\ \delta_T = 0, \delta_i =< 1 > \end{cases} \qquad d_{\text{f2}\to\text{f1}} = \begin{cases} \delta_T = 1, \delta_i =< -1 > \\ \delta_T = 1, \delta_i =< 0 > \\ \delta_T = 1, \delta_i =< 1 > \\ \delta_T = 1, \delta_i =< 0 > \end{cases}$$

The spatial components of the distance vectors are then coalesced into a tuple for each dependence such that the coalesced tuple $C^{fs \to ft} =< \delta_L, \delta_U >$ where δ_L is the maximum spatial distance and δ_U is the minimum spatial distance between two dependent statements fs and ft. For the Jacobi 1D example the tuples for each dependence are identical, $C^{\text{f1}\to\text{f2}} = C^{\text{f2}\to\text{f1}} =< 1, -1 >$. These tuples are used to label edges in the DSG, along with a separate label for the time distance δ_T. Assembling the coalesced tuples, time distances, dependences, and statements leads to the DSG shown in Figure 6.6 for the Jacobi 1D example.



$<\delta_L,\delta_U> = <1,-1>$
$\delta_T = 0$
Compute (f1)
Copy (f2)
$<\delta_L,\delta_U> = <1,-1>$
$\delta_T = 1$

Figure 6.6: Dependence Summary Graph (DSG) for Jacobi 1D

This DSG is subsequently used in Section 6.3.1 to build validity constraints for split tiles and in Section 6.3.1 to compute slopes and statement offsets.

Building Validity Constraints

We seek to constrain the legal values of tile slope and statement offsets by assembling a system of linear inequalities based upon the DSG and the loop bounds of the split tiled code we will generate. Pseudocode for the loop nests of Jacobi 1D upright tile is shown in Listing 6.5. Informally, the validity constraints state that, for any pair of dependent statements, given a region over which the target statement is executed, the source statement will be executed over a region that is, *at minimum*, large enough to satisfy the dependence.

```
for (tt =...) {
  for (ii =...) {
    for (t = 0; t < $T_T$; t++) {
      for (i = ii + $o_L^{\text{f1}}$ + $\alpha$*t; i <= ii + $T_U$ + $o_U^{\text{f1}}$ + $\beta$*t; i++) {
        f1: a1[i] = 0.33*(a0[i-1] + a0[i] + a0[i+1]);
      }
      for (i = ii + $o_L^{\text{f2}}$ + $\alpha$*t; i <= ii + $T_U$ + $o_U^{\text{f2}}$ + $\beta$*t; i++) {
        f2: a0[i] = a1[i];
      }
    }
  }
}
```

Listing 6.5: Loop nests for Jacobi 1D upright tile. Time tile size is T_T; upright space tile size is T_U. Offsets from lower bound are o_L^{f1} and o_L^{f2}; offsets from upper bound are o_U^{f1} and o_U^{f2}. Slope of lower bound is α; slope of upper bound is β.

From the DSG, we note that to compute statement `f2` over some range $[A, B]$ at timestep T we require that statement `f1` be executed over the range $[A-1, B-(-1)]$ at timestep T. Similarly, to compute statement `f1` over some range $[C, D]$ at timestep

T we require that statement `f1` be executed over the range $[C-1, D-(-1)]$ at timestep $T-1$.

Combining the dependence information from the DSG with the loop bounds of Listing 6.5 gives us validity constraints on values the loop bounds may take, and results in the following system of inequalities for lower bounds:

$$\begin{aligned} ii + o_L^{\mathrm{f1}} + \alpha * t &\leq ii + o_L^{\mathrm{f2}} + \alpha * t - 1 \\ ii + o_L^{\mathrm{f2}} + \alpha * (t-1) &\leq ii + o_L^{\mathrm{f1}} + \alpha * t - 1 \end{aligned}$$

The following system constrains the upper bounds:

$$\begin{aligned} ii + T_U + o_U^{\mathrm{f1}} + \beta * t &\geq ii + T_U + o_U^{\mathrm{f2}} + \beta * t + 1 \\ ii + T_U + o_U^{\mathrm{f2}} + \beta * (t-1) &\geq ii + T_U + o_U^{\mathrm{f1}} + \beta * t + 1 \end{aligned}$$

Simplifying and rearranging these systems of inequalities yields the following system of difference constraints for lower bounds:

$$\begin{aligned} o_L^{\mathrm{f1}} - o_L^{\mathrm{f2}} &\leq -1 \\ o_L^{\mathrm{f2}} - o_L^{\mathrm{f1}} &\leq \alpha - 1; \end{aligned}$$

the corresponding constraints for upper bounds are shown below:

$$\begin{aligned} o_U^{\mathrm{f2}} - o_U^{\mathrm{f1}} &\leq -1 \\ o_U^{\mathrm{f1}} - o_U^{\mathrm{f2}} &\leq -\beta - 1. \end{aligned}$$

These systems are used in Section 6.3.1 to compute valid offsets for all statements.

Computing Slopes and Offsets

To determine legal values for slopes α and β we compute, respectively, maximum and minimum cycle ratios [1, 9] on the DSG. A cycle ratio $\rho_L(C)$ on the DSG is computed by finding a cycle C, summing δ_L values over the cycle, and dividing by the sum of δ_T values. A cycle ratio $\rho_U(C)$ is calculated in a similar fashion with δ_U values. We set $\alpha = max(\rho_L(C))$ and $\beta = min(\rho_U(C))$ for all cycles C in the DSG.

We examine the only cycle in our example DSG, C_0, between `f1` and `f2`. The DSG tells us that computing `f1` on some interval $[A, B]$ at time T allows us to compute `f2` on the interval $[A + \delta_L^{f1 \to f2}, B + \delta_U^{f1 \to f2}]$ at time $T + \delta_T^{f1 \to f2}$ without violating any dependences. Continuing along the cycle, computing `f2` on the interval $[A + \delta_L^{f1 \to f2}, B + \delta_U^{f1 \to f2}]$ at time $T + \delta_T^{f1 \to f2}$ allows us to compute `f1` on the interval $[A + \delta_L^{f1 \to f2} + \delta_L^{f2 \to f1}, B + \delta_U^{f1 \to f2} + \delta_U^{f2 \to f1}]$ at time $T + \delta_T^{f1 \to f2} + \delta_T^{f2 \to f1}$.

Substituting in known values for the various δ variables shows us that computing `f1` over the interval $[A, B]$ at time T allows us to compute `f1` over the interval $[A + 2, B - 2]$ at time $T + 1$. Thus, we see a slope of 2 on the lower bound and a slope of -2 on the upper bound.

Equivalently, summing δ_L values and dividing by the sum of δ_T values gives us $\rho_L(C_0) = 2$; a similar calculation gives us $\rho_U(C_0) = -2$. Since there is only one cycle in the DSG, these values are $max(\rho_L(C))$ and $\min(\rho_U(C))$, and we set $\alpha = max(\rho_L(C)) = 2$ and $\beta = \min(\rho_U(C)) = -2$.

These values for α and β are substituted into the systems of difference constraints, and a solution to each of these systems is obtained using the Bellman-Ford algorithm [1, 8]. For the Jacobi 1D example we obtain $o_L^{\text{f1}} = -1$, $o_U^{\text{f1}} = 1$, and $o_L^{\text{f2}} = o_U^{\text{f2}} = 0$.

6.3.2 General Method

Our general algorithm closely follows the principles we have explained for Jacobi 1D. Since stencils may be multidimensional slopes and offsets are separately computed for each dimension. Further, different stencil functions may apply to different subgrids, which may be disjoint, overlapping, or identical. Because of this we conservatively assume all stencil functions are executed at all points in the problem domain when calculating dependences. This is an over-approximation of data dependences and has no impact on the final correctness of the generated code.

Algorithm 6.2 is a general algorithm for computing slopes and offsets for a given input program. This algorithm produces lower/upper slope vectors with one slope per dimension, and lower/upper offset vectors with one offset per stencil statement per dimension.

Algorithm 6.2: Backslice()

```
Input: $P$: SDSL program
Output: $(\vec{\alpha}, \vec{\beta}, \vec{o_L}, \vec{o_U})$: Vectors of slopes alpha, beta. Vectors of lower and upper offsets in
        each spatial dimension for each stencil statement
1  $\mathcal{D} \leftarrow$CalculateDependences($P$)
2  $\mathcal{S} \leftarrow$CalculateDependenceDistances($\mathcal{D}$)
3  $DSG \leftarrow$BuildDSG($\mathcal{S}$,$P$)
4  foreach dimension $d$ of $P$ do
5      $\alpha_d \leftarrow$ComputeMaxCycleRatio($DSG$,$d$)
6      $\beta_d \leftarrow$ComputeMinCycleRatio($DSG$,$d$)
7      $\vec{\alpha}[d] \leftarrow \alpha_d$
8      $\vec{\beta}[d] \leftarrow \beta_d$
9      $(\vec{v_L}, \vec{v_U}) \leftarrow$BuildValidityConstraints($DSG$,$\alpha_d$,$\beta_d$)
10     $\vec{o_L}[d] \leftarrow$SolveForOffsets($\vec{v_L}$)
11     $\vec{o_U}[d] \leftarrow$SolveForOffsets($\vec{v_U}$)
12 end
13 return $(\vec{\alpha}, \vec{\beta}, \vec{o_L}, \vec{o_U})$
```

Algorithm 6.2 takes an arbitrary SDSL program as input. Dependence analysis is performed on this program to determine all flow and anti dependences, and dependence distance vectors are calculated. The dependence distance vectors are used to build the DSG, where each vertex is an SDSL statement and each edge is labeled with a time distance. Edges are also labeled with coalesced distances, computed for each spatial dimension as in Section 6.3.1 and placed in vectors $(\vec{\delta_L}, \vec{\delta_U})$.

After the DSG is built, slopes and offsets are computed along each dimension d. For each dimension of the DSG we compute α and β using maximum and minimum cycle ratios. The α_d and β_d values are then added to $\vec{\alpha}$ and $\vec{\beta}$ as the slopes for d. Once the slopes have been computed for a dimension, validity constraints are constructed as shown in Section 6.3.1.

After the slopes and validity constraints are known for a given dimension, `Solve-ForOffsets()` rearranges the validity constraints into a system of differences and solves for offset values using the Bellman-Ford algorithm. Offsets for each statement in the current dimension are appended to the offset vectors. This process is repeated until slopes and offsets have been computed for all dimensions.

6.3.3 Proof of Correctness

Upright Tiles: The validity constraints guarantee that no dependences in an upright tile will be violated. In this section we prove that lower bound slopes calculated with the maximum cycle ratio always lead to a system of constraints that has a solution; the proof for the upper bound and minimum cycle ratio is identical.

A system of difference constraints not solvable by the Bellman-Ford algorithm would contain a negative weight cycle. We show that this is not possible.

We begin by constructing a graph isomorphic to the DSG where each vertex is a statement and each directed edge is labeled with its corresponding validity constraint for the lower bound v_L^e on some dimension d.

All validity constraints take one of the following forms:

$$o^s \leq o^t - \delta_e \quad \Rightarrow \quad o^s - o^t \leq -\delta_e \tag{6.1}$$

$$o^s \leq o^t - \delta_e + \alpha \quad \Rightarrow \quad o^s - o^t \leq -\delta_e + \alpha \tag{6.2}$$

In (6.1) the dependence associated with the constraint has source and target statements at the same timestep; in (6.2) the statements are 1 timestep apart. The unknown source statement offset is o^s, the unknown target statement offset is o^t, the dependence distance/edge weight is δ_e, and α is the unknown slope.

We next show that any cycle of length k on this graph restricts the possible values of α to be of the following form:

$$\alpha \geq \frac{1}{t} \sum_{i=1}^{k} \delta_i . \tag{6.3}$$

Note that the maximum cycle ratio simply maximizes α subject to the constraint in (6.3). In (6.3), δ_i is the weight of an edge in the cycle and t is the number of edges where source and target are separated by 1 timestep. Every vertex n in the cycle is entered through an edge f and exited through an edge g. On edge e, vertex n is the target of the dependence; on edge g, vertex n is the source of the dependence. Summing validity constraints $v_L^f + v_L^g$ eliminates the offset o^n. The accumulated sum $\sum_{i=1}^{k} v_L^i$ eliminates all offsets and leaves $0 \geq \sum_{i=1}^{k} \delta_i - t * \alpha$, which is equivalent to (6.3).

We now show that a negative weight cycle in the graph used by the Bellman-Ford algorithm to solve the system of difference constraints requires a value of α that violates (6.3).

All constraints in the system of differences we are solving take one of two forms, Equation 6.1 or Equation 6.2. The graph used in Bellman-Ford, after removing the start vertex and zero-weight edges emanating from it, is isomorphic to the DSG and the validity constraint graph described above. Each vertex is a statement offset, and each edge is weighted by the difference between the source and the target offset. The weight of any length k cycle is shown in (6.4).

$$\sum_{i=1}^{k} -\delta_i + t * \alpha \tag{6.4}$$

Assume there is a negative weight cycle of length k. This requires (6.5) to be true:

$$\sum_{i=1}^{k} -\delta_i + t * \alpha < 0 \tag{6.5}$$

Rearranging (6.5), we have α constrained by (6.6):

$$\alpha < \frac{1}{t} \sum_{i=1}^{k} \delta_i \tag{6.6}$$

The same cycle, on the validity graph, shows that α is constrained by (6.3). This is a contradiction as we know α was constructed by maximized subject to (6.3) thus our assumption of a negative weight cycle in the graph used to compute offsets was wrong and the system of differences constraints used to solve for offsets will always have a solution.

Inverted Tiles: The loop bounds for each statement in an inverted tile are set to fully span the gap between adjacent upright tiles (or a domain boundary). The proof of satisfaction of all dependences during execution of an inverted tile is a direct consequence of this "completeness" property of the span of spatial loops of an inverted tile. Consider all instances at time t, of an arbitrary statement S_q in an inverted tile. Any dependence on instances of any statement S_p mean that S_p textually precedes S_q, or that the dependence is from instances of S_q at some earlier time step. There are three possibilities for any such statement instance from which there exists a dependence: i) it belongs in some upright tile, ii) it belongs in the same inverted tile, iii) it belongs in some other inverted tile. If we have case (i), the dependence is satisfied since all upright tiles are executed before any inverted tiles. If we have case (ii), again the dependence is satisfied since statements are executed in textual order within the inverted tile for a time step, and in increasing time order across time steps. Finally, case (iii) is impossible since other inverted tiles are separated from the current inverted tile by at least one upright tile, whose slopes are guaranteed to prevent any dependence edge crossing their boundaries.

Multiple Spatial Dimensions: In the proof of correctness, so far we have focused on the case of a single spatial dimension. For multi-dimensional stencils, as explained in Section 6.2.2 and Section 6.2.3, tiles may have different characteristics along the different spatial dimensions. For example, with two spatial dimensions and nested split tiling, there are four possibilities: Upright-Upright, Upright-Inverted, Inverted-Upright, Inverted-Inverted. With hybrid split tiling, one or more spatial dimensions may be tiled using standard parallel tiles (using the slope β computed for the right boundary in that dimension by the back-slicing analysis). The proof of validity of execution in the multi-dimensional case is a consequence of the fact that the loop bounds along each spatial dimension are independent of iterators of other spatial dimensions. Consider, for example, a stencil with three spatial dimensions i, j, and k, where standard tiling is used along k, and split tiling along i, and j. Let a particular tile be upright along i, and inverted along j. At some time step t, the set of all instances of a statement S_q will necessarily correspond to a cross product of ranges along the three spatial extents: [kl:ku,jl:ju,il:iu]. The instances of any statement S_p that S_q depends upon, can be expressed as $[kl + \delta_L^k : ku+\delta_U^k, jl+\delta_L^j : ju+\delta_U^j, il+\delta_L^i : iu+\delta_U^i]$. We can prove that all such dependences are satisfied by proving that the extents along each dimension are satisfied. For upright or inverted tiles along any dimension, we have proved that the required extents will be covered by the generated slopes/offsets. For a standard tiled dimension too, the dependences will be satisfied: on the lower side, the instances will either lie in the same tile or in an earlier numbered tile that would have been executed before this tile; on the upper side, the instance will necessarily lie within the same tile because the slope and offset from the back-slicing computation is used.

6.4 Code Generation

Before detailing the code generation algorithm, the structure of generated code is shown in Listing 6.6. The example shows code for a single-statement 1D stencil. Upright and inverted inter-tile loops can be found, respectively, at lines 11 and 26. Note that the upright tile loop covers an extent from the lowest bound of any stencil function to the highest bound of any stencil function, and the inverted tile tile loop extends beyond this on both sides. This is needed to ensure that boundary cells are computed as upright tiles slope away from them. It is enabled by the `max()` and min() expressions that prevent intra-tile loop bounds from taking values outside of the original loop's domain (lines 17 and 32). Constraints on tile size values such as UPR_TILE_SIZE and INV_TILE_SIZE are imposed as needed to ensure correctness of generated code; this is explained below.

```
 1  // Time tile loop
 2  for (tt = 0; tt < T; tt += TT_SIZE) {
 3    // Upright tile loop
 4    // F1 is a stencil function, lb_F1 (ub_F1) is the lower (upper) bound of
 5    // the grid coordinate where F1 is applied
 6    upr_lb = lb_F1; // for multiple stencil functions F1, F2, ..., it is
 7                    // actually  min(lb_F1,lb_F2,...)
 8    upr_ub = ub_F1; // Similarly max(lb_F1, lb_F2...)
 9    // UPR_TILE_SIZE (INV_TILE_SIZE) is the size of the upright (inverted)
10    // tile base
11    for (ii = upr_lb; ii < upr_ub; ii += UPR_TILE_SIZE + INV_TILE_SIZE) {
12      // Time loop
13      for (t = tt; t < min(tt+TT_SIZE,T); t++) {
14        tile_lb_F1 = ii + offset_F1_lb + upr_alpha*(t-tt);
15        tile_ub_F1 = ii + UPR_TILE_SIZE +
16        offset_F1_ub + upr_beta*(t-tt) - 1;
17        for (i = max(tile_lb_F1, lb_F1); i <= min(tile_ub_F1, ub_F1); i++) {
18            // 1D stencil function code.
19        }
20      }
21    }
22    // Inverted tile loop
23    inv_lb = upr_lb - INV_TILE_SIZE;
24    inv_ub = ub_F1 + INV_TILE_SIZE; // for multiple stencil functions
25                                    // F1, F2, ... it is max(ub_F1,ub_F2,...)
26    for (ii = inv_lb; jj <  inv_ub; ii += UPR_TILE_SIZE + INV_TILE_SIZE) {
27      // Time loop
28      for (t = tt; t < min(tt+TT_SIZE,T); t++) {
29        tile_lb_F1 = ii + inv_offset_F1_lb + inv_alpha*(t-tt);
30        tile_ub_F1 = ii + INV_TILE_SIZE +
31        inv_offset_F1_ub + inv_beta*(t-tt) - 1;
32        for (i = max(tile_lb_F1, lb_F1); i <= min(tile_ub_F1, ub_F1); i++) {
33           // 1D stencil function code
34        }
35      }
36    }
37  }
```

Listing 6.6: Structure of generated code for 1D case

6.4.1 Basic Code Generation

Algorithm 6.3 is the overall algorithm to perform DLT and split tiling on a program that can be expressed in SDSL. It is a more detailed version of the overview algorithm shown in Section 6.2.4.

Algorithm 6.3: LayoutTransformAndSplitTile()

```
Input: $P$: SDSL program, $L_{vec}$: Integer length of a vector (in elements)
Output: $P_{sdlt}$: C code for Split tiled, DLT version of the input SDSL program
$P_{sdlt}$ ←Copy($P$)
// Convert arrays to DLT.
$A$ ←GetArrays($P_{sdlt}$)
$l_{time}$ ←GetTimeLoop($P_{sdlt}$)
foreach array $a$ in $A$ do
    $dims$ ←GetDimensions($a$,$P_{sdlt}$)
    $l_{copyin}$ ←GenCopyInLoop($a$,$dims$,$L_{vec}$)
    $l_{copyout}$ ←GenCopyOutLoop($a$,$dims$,$L_{vec}$)
    $l_{time}$.Prepend($l_{copyin}$)
    $l_{time}$.Append($l_{copyout}$)
end
// Vectorize statements.
$S$ ←GetStatements($P_{sdlt}$)
foreach statement $s$ in $S$ do
    $s_{vec}$ ←Vectorize($s$,$L_{vec}$)
    $l_{enc}$ ←GetEnclosingLoop($s_{vec}$,$P_{sdlt}$)
    ChangeBoundsToDLT($l_{enc}$)
    $s_{vec}^{start}$ ←GenMaskedWriteStartStmt($s_{vec}$,$L_{vec}$)
    $l_{start}$ ←GenMaskedStartLoop($s_{vec}^{start}$,$l_{enc}$)
    $l_{enc}$.Prepend($l_{start}$)
    $s_{vec}^{end}$ ←GenMaskedWriteEndStmt($s_{vec}$,$L_{vec}$)
    $l_{end}$ ←GenMaskedEndLoop($s_{vec}^{end}$,$l_{enc}$)
    $l_{enc}$.Append($l_{end}$)
end
// Backslice original code.
($\vec{x}$) ←ComputeOffsetSlopes($P$)
AddBacksliceResults($P_{sdlt}$,$\vec{x}$)
// Create nested split tile loops
$l_{tt}$ ←CreateTimeTileLoop()
$d$ ←GetSpaceDimensionCount($P_{sdlt}$)
GenNestedSplitTileLoops($l_{tt}$,$d$)
GenTileBodies($l_{tt}$,$P_{sdlt}$,$d$)
ReplaceTimeLoop($l_{tt}$,$P_{sdlt}$)
return $P_{sdlt}$
```

Functions `GetXXX` retrieve properties of the SDSL program required for code generation (e.g., name of arrays, number of dimensions, etc.). Functions `GenCopyXXXLoop` copies the data from the original layout to the DLT layout. Function `vectorize` generates a sequence of vector intrinsics call (SSE or AVX in our experiments) to compute the same arithmetic operation as the corresponding statement in the stencil function, using SIMD vectors. Function `ChangeBoundsToDLT` adapts/creates the loop structure scanning the dimension-lifted data arrays. Functions `GenMaskedXXX` use SIMD masked write operations to cope with vectors in the generated code where one of the vector slot (necessarily the first or last slot only) contains ghost data. The reader may refer to [26] for more details. Functions `GenNestedSplitTileLoops` and `GenTileBodies` are described below.

Algorithm 6.4 provides detail for the code generation of split tiled code, in the context of a program on which DLT has been performed first. We remark that in order for the code generation of the `masked writes` required by DLT to be correct, we impose a constraint relating the size of the innermost dimension to the tile sizes:

$$|d_{innermost}| \quad \text{mod } (upr\_tile\_size + inv\_tile\_size) = 0$$

Adding this constraint greatly simplifies the code generation, avoiding the need to handle complex corner cases at the tile boundaries. This limitation is not a problem in practice, since padding can be used to comply with this constraint.

Function `GenInvertedFullBndryTileCondition` generates a conditional to determine whether to execute a full inverted tile or an boundary inverted tile that includes two loop nests for the start and end boundaries of DLT codes. Function `AdjustTimeAndSpaceLoopBounds` alters the loop boundaries of the loops copied from the original

Algorithm 6.4: GenTileBodies()

```
Input: $P_{sdlt}$: Stencil program with DLT performed and backslicing results inserted, $l$: A
       time tile loop with empty nested split tile loops inside, $d$: Integer representing
       current loop depth (1 innermost)
Output: $l$: with split tile loop bodies generated
if $d == 1$ then
    $l_{orig}$ ←GetOriginalLoopBody($P_{sdlt}$)
    // Generate upright tile body
    $l_{upr}$ ←GetUprightTileLoop($l$)
    CopyTimeAndSpaceLoops($P_{sdlt}$,$l_{upr}$,1)
    RemoveMaskedWriteLoops($l_{upr}$)
    AdjustTimeAndSpaceLoopBounds($l_{upr}$,$P_{sdlt}$)
    $l_{inv}$ ←GetInvertedTileLoop($l$)
    ($l_{inv}^{full}$,$l_{inv}^{bndry}$) ←$l$.GenInvertedFullBndryTileCondition()
    // Generate inverted full tile body
    CopyTimeAndSpaceLoops($P_{sdlt}$,$l_{inv}^{full}$,1)
    RemoveMaskedWriteLoops($l_{inv}^{full}$)
    AdjustTimeAndSpaceLoopBounds($l_{inv}^{full}$,$P_{sdlt}$)
    // Generate inverted boundary tile body
    CopyTimeAndSpaceLoops($P_{sdlt}$,$l_{inv}^{bndry}$,2)
    AdjustTimeAndSpaceLoopBounds($l_{inv}^{bndry}$,$P_{sdlt}$)
    TrimMaskedWriteLoops($l_{inv}^{bndry}$)
else
    $l_{upr}$ ←GetUprightTileLoop($l$)
    GenTileBodies($l_{upr}$,$P_{sdlt}$,$d-1$)
    $l_{inv}$ ←GetInvertedTileLoop($l$)
    GenTileBodies($l_{inv}$,$P_{sdlt}$,$d-1$)
end
return $l$
```

code to tile loop boundaries using the results of backslicing. Function `TrimMasked-WriteLoops` removes the masked write loops at the start and end of the inverted boundary tile loops, leaving the masked writes between them.

We conclude by presenting the algorithm for generating split tiled loop nests shown in Algorithm 6.5. Hybrid split tiling code generation requires minimal changes to Algorithms 6.4 and 6.5. Algorithm 6.5 requires a conditional to check if it is generating code for the outermost loop. If so, a standard tile loop is generated instead

Algorithm 6.5: GenNestedSplitTileLoops()

```
Input: $l$: Loop to add nested upright and inverted tile loops to, $d$: Integer representing
        current loop depth (1 innermost)
Output: $Arrays$: the set of arrays to be dimension-lifted
1  if $d == 1$ then
2      $l_{upr}$ ←$l$.AddUprightTileLoop()
3      $l_{inv}$ ←$l$.AddInvertedTileLoop()
4  else
5      $l_{upr}$ ←$l$.AddUprightTileLoop()
6      GenNestedSplitTileLoops($l_{upr}$, $d - 1$)
7      $l_{inv}$ ←$l$.AddInvertedTileLoop()
8      GenNestedSplitTileLoops($l_{inv}$, $d - 1$)
9  end
10 return $l$
```

of a split tiled loop. Algorithm 6.4 requires a change to the `AdjustTimeAndSpaceLoop-Bounds()` procedure to correctly traverse the standard tile dimension.

6.4.2 Further Optimizations

Macro Statements: To minimize the number of inner loops, a point function calls is treated as a single macro statement with its associated read and write sets used in dependence calculation.

Standard tile loop sinking: In hybrid-split tiling, the outermost standard tile loop is sunk to the innermost tile loop level. This avoids excessive inter-thread synchronization required each time a set of upright or inverted tiles is completed.

Loop boundary code hoisting: Tile loop boundaries are complex integer expressions with multiple min, max, addition, and subtraction operations. The calculation of these boundaries is hoisted to the time loop of a tile, allowing all spatial boundaries to be computed once per timestep.

Full vector tile isolation: In order to achieve maximum performance, expensive masked write instructions must be avoided. Since grid and subgrid sizes are parametric, code generated according to the algorithms in Section r̃efsec:codegen assumes the worst case and uses masked writes for all statements. A run time check determines if a tile can be executed without masked writes. If so, code branches to tile code with the masked writes removed.

6.5 Experimental Results

The effectiveness of the both nested split tiling and hybrid split tiling applied in conjunction with the dimension-lifting transformation was experimentally evaluated on several hardware platforms using a variety of stencil kernels. We compare performance to the diamond-tiling system used by Pluto [5], the cache-oblivious tiling system used by Pochoir [52], and the Intel C Compiler v13.0.

6.5.1 Experimental Setup

Hardware: Experiments were performed on three hardware platforms with DVFS features disabled on all of them.

AMD Phenom II X6 1100T (K10 micro-architecture) is a 6-core x86-64 chip, clocked at 3.3GHz; single-precision peak performance of 26.4 GFlop/s/core (158.4 GFlop/s aggregate); double-precision peak performance of 13.2 GFlop/s/core (79.2 GFlop/s aggregate).

Intel Core i7-920 (Nehalem micro-architecture) is a quad-core x86-64 chip, running at 2.66 GHz; single-precision peak performance of 21.28 GFlop/s/core (85.12 GFlop/s aggregate); double-precision peak performance of 10.64 GFlop/s/core (42.56 GFlop/s aggregate).

Intel Core i7-2600K (Sandy Bridge micro-architecture) is a quad-core core x86-64 chip, running at 3.4 GHz; single-precision peak performance of 54.4 GFlop/s/core (217.6 GFlop/s aggregate); double-precision peak performance of 27.2 GFlop/s/-core (108.8 GFlop/s aggregate).

Programs were compiled using the Intel C Compiler v13.0 with the '`-O3 -ipo -xHost`' optimization flags was used for split tiled, Pluto, and Pochoir codes on all machines. Auto-parallelization and auto-vectorization was enabled for ICC results with the '`-parallel -O3 -ipo -xHost`' optimization flags and appropriate vectorization pragmas.

Benchmarks: The following stencil codes were used (with the names used to refer to them in parentheses): Jacobi 1D (jac-1d-3), Jacobi 2D (jac-2d-5), Jacobi 3D (jac-3d-7); Laplacian, Gradient and Poisson 2D; FDTD 2D [42]; Heat 1D/2D/3D distributed with Pochoir [52].

All array dimensions were set to be significantly larger than last level cache on all micro-architectures. For all stencils, the footprint of each array was set to 244.14MB for single precision and 488.28MB for double precision; this was achieved using 1D arrays with $64 * 10^7$ scalar elements, 2D arrays with 8000^2 elements, and 3D arrays with 400^3 elements. The number of time steps was set to 100 for all benchmarks.

Tile sizes were autotuned for Pluto with diamond tiling, as well as for our split tiling work. The autotuning was done over a sampling of the set of tile size combinations that respect the various constraints (i.e., multiple of vector length) of each framework. Autotuning runs were performed for a maximum of 4 hours per SIMD unit / benchmark combination (how to speed up tile size autotuning, for instance using acceleration/search heuristics is beyond the scope of this dissertation, we simply tested all tile sizes in our subset). For split tiled codes all threads (one thread per core) were assigned to the outermost parallel loop using OpenMP parallel for pragmas.

6.5.2 Experimental Results

Absolute performance for single and double precision experiments across all platforms and codes is given in Figure 6.7.

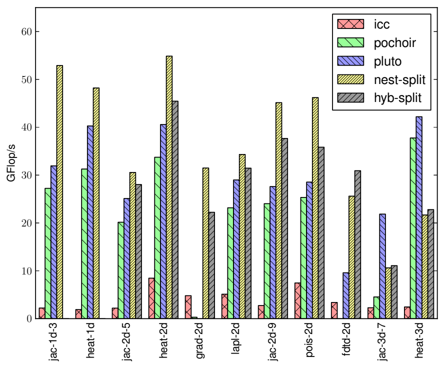
In all cases Pochoir, nested split tiled, and hybrid split tiled codes are significantly faster than sequential C code auto-parallelized and vectorized by Intel's ICC compiler. Not all benchmarks could be optimized by Pochoir, which can only generate optimized tiled parallel code for single statement stencil codes. Multi-statement, multi-loop stencil codes like FDTD, with three inter-related stencils updating Ex[t] using Ex[t-1] and Hz[t-1], Ey[t] using Ey[t-1] and Hz[t-1], and Hz[t] using Hz[t-1], Ex[t], and Ey[t]), cannot be expressed in Pochoir.

1D Benchmarks

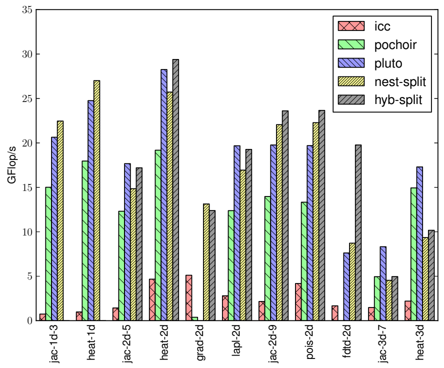
Having only one split tiled dimension allowed tiles to fit in cache and provided ample parallelism. High performance was expected of these codes and was observed. Nested split tiling + DLT outperformed ICC, Pochoir, and Pluto on both 1D benchmarks across all platforms. Improvement over Pochoir ranged from a low of $1.27\times$ for double precision Jacobi-1D on Nehalem to a high of $2.2\times$ for single precision Heat-1D on Sandy Bridge. Hybrid split tiling does not apply to 1D benchmarks as the inner loop is split tiled for both fine grain vector parallelism and coarse grain thread-level parallelism.

2D Benchmarks

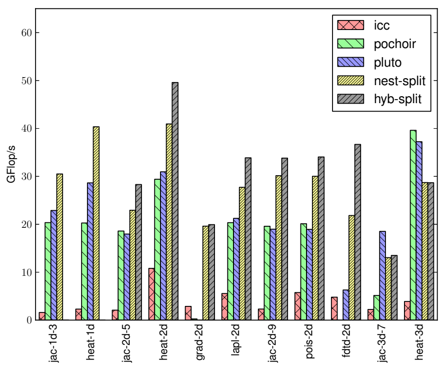
Across all 2D benchmarks hybrid split tiling outperformed ICC, Pochoir, Pluto, and nested split tiling on both quad core Intel platforms. For the hexacore AMD,



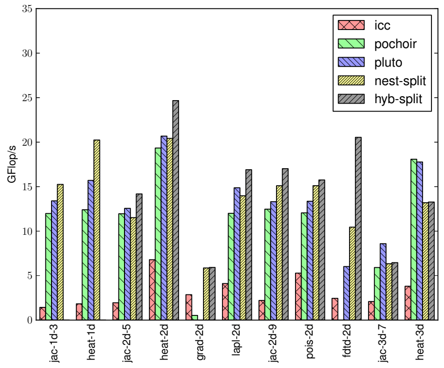
(a) AMD Phenom II X6 SSE2 single precision



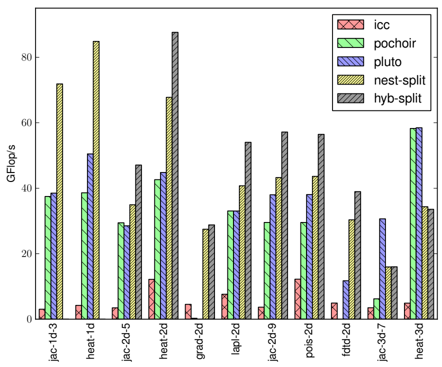
(b) AMD Phenom II X6 SSE2 double precision



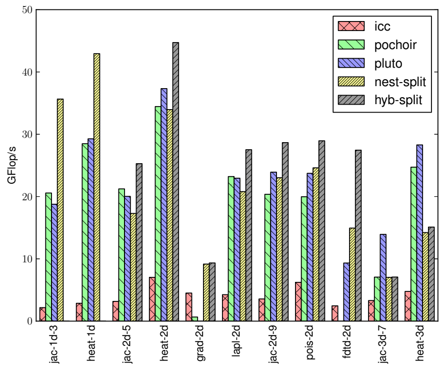
(c) Intel Core i7-920 SSE4 single precision



(d) Intel Core i7-920 SSE4 double precision



(e) Intel Core i7-2600K single precision



(f) Intel Core i7-2600K double precision

Figure 6.7: Split-tiling Performance

performance of hybrid split tiling + DLT fell behind both nested split tiling and Pluto on several benchmarks, both single and double precision.

This performance anomaly is due to load balancing issues. DLT effectively turned the dimension of the arrays involved from 8000 $\times$ 8000 float/double scalars to 8000 $\times$ 2000 float vectors and 8000 $\times$ 4000 double vectors. With hybrid split tiling, tiles from the smaller dimension (subject to the constraints described in Section 6.2.2) had to be distributed across threads. Smaller tiles with better load balancing characteristics limited reuse, while larger tiles with significant reuse were not plentiful enough to adequately distribute load across cores.

Nested split tiling exhibited better load balancing because only tiles along the larger dimension were distributed across cores. The outer dimension was large enough to allow both large tiles for significant reuse and a large quantity of tiles for load balancing. Hybrid split tiling on quad core Intel platforms did not have any load balancing issues because the smallest dimension was divisible by 4, allowing for the same number of tiles to be distributed across all cores.

3D Benchmarks

Both hybrid and nested split tiling fell behind Pochoir and Pluto on both 3D benchmarks across all platforms. While hybrid split tiling was able to significantly reduce the pressure on the memory system for 2D benchmarks, its benefit was not seen when adding a third dimension. In 3D, both inner dimensions are split tiled, thus spatial tile sizes in both inner dimensions must increase when time tile size is increased.

For both the Heat-3D and Jacobi-3D benchmarks, an increase of one in the time tile size leads to an upright tile size increase of four in each dimension. Tile sizes grow fast enough that by the time significant gains can be achieved from data reuse, the code is bound by memory latency and bandwidth consumption from spatial tiles that have grown significantly larger than L1 cache.

The diamond tiles used by Pluto overcome this limitation by halving the amount of data required for a given time tile size. For a time tile size of 16 and slopes of 2 on either side, an upright tile used in nested and hybrid split tiling must be at least 64 elements at its base. A diamond tile with a time tile size of 16 begins at a point, expands to 32 elements at its widest and narrows to a point again. Pluto is able to reuse data significantly more than both Pochoir and nested and hybrid split tiling.

Further exacerbating the problem is the fact that DLT causes tile sizes in the innermost dimension to be multiplied by vector size. Coupled with the constraints on tile size described in Section 6.2.2 this leads to split tiling + DLT tile sizes (in KB) that are much larger than similar tile sizes (size of each dimension). In Heat-3D both Pluto and Pochoir are able to achieve high performance across all platforms. It is likely that the smaller tile sizes (in KB) enable Pochoir to achieve very high performance on this code.

CHAPTER 7

SDSL—Stencil Domain Specific Language

SDSL (**S**tencil **D**omain **S**pecific **L**anguage) is a domain specific language for expressing stencil computations. SDSL was loosely based on the RNPL [38] and SNPL [37] languages used for the rapid prototyping of partial differential equation solvers, although the resemblance is mostly cosmetic and no code is shared between the projects.

This chapter begins with the motivation for the existence of SDSL and uses a simple 5-point, 2-dimensional Jacobi stencil as a running example to both illustrate the usage of and provide explanations for all major language features.

7.1 Motivation

Regardless of architecture, high performance stencil codes require the implementation of complex tiling schemes [19, 24, 27, 31, 34] to effectively deal with memory subsystem pressure created by streaming large arrays of data through the CPU. These tiling schemes require altering rectangular loop iteration domains to more complex shapes.

Further, modern shared-memory multicore, GPU, and distributed memory systems require the systematic exploitation of parallelism to achieve the highest performance. Parallel program is notoriously difficult even for expert programmers, and leads to long development and debug cycles. Modern CPU architectures and accelerators [29] also require fine-grain SIMD vector code in order to operate at or near machine peak performance.

The purpose of SDSL is to provide a programming language that allows for a simpler specification of non-trivial stencil computations in a form that enables the generation of high-performance tiled, parallel, and/or vector implementations that can be generated in a performance portable manner across multiple platforms.

7.2 Example—2D 5-point Jacobi Stencil

Figure 7.1 is a complete SDSL program to compute a 2-dimensional, 5 point Jacobi stencil. This section contains an overview of the program; the remainder of this chapter uses this example to illustrate important language features.

```
 1  int dim0;
 2  int dim1;
 3
 4  float TOL = .00001f;
 5
 6  grid g [dim1][dim0];
 7
 8  float griddata a on g at 0,1;
 9
10  pointfunction five_point_avg(p) {
11    float ONE_FIFTH;
12    ONE_FIFTH = 0.2f;
13    [1]p[0][0] =
14      ONE_FIFTH*([0]p[-1][0] + [0]p[0][-1] + [0]p[0][0] + [0]p[0][1] + [0]p[1][0]);
15  }
16
17  iterate 1000 {
18    stencil jacobi_2d {
19      [0][0:dim0-1]       : [1]a[0][0] = [0]a[0][0];
20      [dim1-1][0:dim0-1] : [1]a[0][0] = [0]a[0][0];
21      [1:dim1-2][0]       : [1]a[0][0] = [0]a[0][0];
22      [1:dim1-2][dim0-1] : [1]a[0][0] = [0]a[0][0];
23
24      [1:dim1-2][1:dim0-2] : five_point_avg(a);
25    }
26
27    reduction max_diff max {
28      [0:dim1-1][0:dim0-1] : fabs([1]a[0][0] - [0]a[0][0]);
29    }
30  } check (max_diff < TOL) every 4 iterations
```

Listing 7.1: Jacobi 2D 5-point stencil in SDSL

The program begins with a declaration of two int parameters, `dim0` and `dim1` on lines 1 and 2, and a constant `TOL` with the value of `0.00001f` on line 4. The parameters are used to define a 2-dimensional `grid` of size `dim1`×`dim0` on line 6. Line 8 declares floating point `griddata` in the shape of the `grid` defined on line 6. This data is also declared to exist at two different offsets from the current timestep, `0` and `1`.

Lines 10–15 define a 5 point stencil as a `pointfunction` named `five_pt_avg`. This function takes a parameter `p`, the grid data used to calculate the stencil, and uses a local variable `ONE_FIFTH`. `five_pt_avg` averages 5 neighboring points in one timestep of `p` and writes the result to a point in the next timestep of `p`. Lines 17–30 define an iterative loop with a convergence check. The `iterate` construct on line 16 specifies that the iterative loop should run, at most, 1000 times. Lines 18–25 define a stencil computation over subgrids of the original grid. Lines 19–22 specify the value of the edges of grid data `a` to be the same at both timesteps at which `a` is defined. Line 24 specifies that the point function `five_pt_avg` be applied at every point over the interior of `a`.

Lines 27–29 define a `reduction` to be performed at every point on `a`. The reduction variable is `max_diff`, and it contains the largest difference between the value of an element of `a` at two successive timesteps.

Finally, line 30 defines a convergence condition for the `iterate` and specifies how frequently this check is to be performed. In this program the condition is that the largest difference between successive time values of an element of `a` is less than the constant `TOL` and that this condition should be checked every 4 iterations.

7.3 Language Features

The subsections that follow provide detailed information on the various constructs of the SDSL language. A full grammar (ANTLR format) can be found in Appendix B.

7.3.1 Comments

SDSL programs can be commented with C/C++ style comments. C-style comments begin with the characters '`/*`' and continue until terminated by the '`*/`' characters. C++-style comments begin with the characters '`//`' and continue until terminated by a new line.

7.3.2 Literal Values

Floating point and integer literal values can be explicitly represented in SDSL. These values are used in SDSL expressions, described in Section 7.3.4. Both floating point and integer literals are always interpreted as base 10 numbers.

In the current definition of the language, the syntax and types of these literals correspond to a subset of the `int`, `long`, `float`, and `double` literals of the C/C++ language. Future refinements of SDSL may introduce a richer set of literal values (e.g., hex/octal literals).

7.3.3 Identifiers

C-style identifiers are used to name parameters, constants, grids, grid data, point functions, point function parameters, variables in point functions, stencils, and reductions. The following production defines legal identifiers in SDSL:

```
ID  :  (‘a’..‘z’|‘A’..‘Z’|‘_’) (‘a’..‘z’|‘A’..‘Z’|‘0’..‘9’|‘_’)*
```

The first character of an identifier must be a letter or an underscore. Subsequent characters must be letters, numbers, or underscores.

7.3.4 Arithmetic Expressions

Arithmetic expressions contain literals, constants, parameters, grid data references, and function calls. Addition, subtraction, multiplication, division, and modulo operations are permitted. In the current implementation, calls to side-effect-free external functions from C’s libmath (defined in `math.h`) are permitted by default;

a more general mechanism for accessing external code from within SDSL code will be designed in the future.

Expression terms may be grouped together using parentheses. Operator associativity and precedence are similar to that in C/C++.

7.3.5 Parameters and Constants

Parameters and constants are the first items declared and defined in a SDSL program. Both parameters and constants can be of type `int`, `long`, `float`, or `double`. A parameter declaration includes the type and identifier of the parameter, while a constant is declared with the type, name, and value. Parameters are bound to values at the time of program execution. Examples of parameter and constant declarations are shown below.

```
// Parameters
int dim0;
long n;
float alpha;
double gamma;

// Constants
int samples = 112;
long distance = 50000000000L;

float G = 9.81f;
double PI = 3.14159265358979323846264338327950288841971;
```

7.3.6 Grids

A `grid` defines the geometry of a domain over which a stencil is to be computed. Grids are restricted to being rectangular, and can be 1-, 2-, or 3-dimensional. Dimension cardinality may be specified using integer parameters, integer constants, integer literals or expressions combining these. Parameters used to define grid dimension sizes must be declared as the first statements of an SDSL program and must appear in order from inner to outer. Currently, an SDSL program may contain exactly 1 grid.

```
int dim0;
int dim1;

grid g [dim1][dim0];
```

The above SDSL statements defines a grid with the identifier `g`. This 2D grid is defined by an outer dimension having a size of `dim1` points and the inner dimension having a size of `dim0` points.

7.3.7 Grid Data

`griddata` is a concrete instance of a `grid` and has a type of `int`, `long`, `float`, or `double`. Grid data is defined over one or more offsets from the current timestep. Timestep offsets must be consecutive integers.[4]

```
float griddata a on g at 0,1;
```

[4]A convenient way to view timestep offsets is as another dimension of a grid. A 2-dimensional, `dim1`$\times$`dim0` grid defined on timesteps 0 and 1 can be thought of as a 3-dimensional grid with the third dimension having a size of 2.

The above `griddata` declaration states that we are creating a concrete instance of `grid g` of type `float` with the identifier `a` defined over timesteps `0` and `1`.

7.3.8 Grid Data References

Grid data is referenced in point functions, iterates, and reductions. A grid data reference consists of a temporal offset, the identifier of the grid data being referenced, and a spatial offset for each dimension of the grid data.

The temporal offset of a grid data reference is the offset with respect to the current timestep. Suppose a point function is currently being executed at time t. A temporal offset of 1 corresponds to grid data at time $t+1$. At the next time iteration $t' = t + 1$, a temporal offset of 0 would be used to access the same data accessed with a temporal offset of 1 at time t.

When grid data is referenced spatially, it is always in relation to a point in an n-dimensional grid. All offsets are calculated from this point. Suppose a point function is currently being applied at point i in a one dimensional grid. An offset of 1 corresponds to the point $i + 1$.

Below is a simple 2-dimensional grid data references in SDSL representing the current point being processed, 1 timestep in the future.

```
[1]data[0][0]
```

The temporal offset, enclosed by brackets, is first. After the temporal offset is the identifier of the grid data being referenced. Finally, the offsets of the referenced point in each spatial dimension, with each offset enclosed in brackets, are given. Another 2-dimensional reference is shown below. This reference is offset by -1 in the outer spatial dimension.

```
[1]data[-1][0]
```

7.3.9 Subgrids

Subgrids can be specified in a `stencil` and a `reduction` such that different computations are performed according to the location of a point on a grid. A subgrid is defined in each spatial dimension and is two integer expressions, separated by a colon, enclosed in brackets. The expressions define the start and end coordinates of the subset in the given dimension. An example subgrid containing all but the edges of a 2-dimensional `dim1`$\times$`dim0` grid is shown below.

```
[1:dim1-2][1:dim0-2]
```

Subgrids may also consist of just one expression, in which case the subgrid exists only at the coordinate obtained by evaluating the expression. The example below illustrates this by defining a subset representing the last column of an `dim1`$\times$`dim0` grid.

```
[0:dim1-1][dim0-1]
```

More examples of subgrids can be seen in Section 7.3.11 and Section 7.3.11.

7.3.10 Point Functions

Point functions contain stencil implementations. A `pointfunction` takes `griddata` as parameters and executes a series of computations using grid data references. When a point function is called, it is at an arbitrary point in a `grid`. All spatial components of grid data references within the point function are offsets from this point.

A `pointfunction` may compute an arbitrary number of points on an arbitrary number of grid data references. Variables of type `int`, `long`, `float`, and `double` may be declared and used. Constant values and parameters of the SDSL program can be used and should not be included in the parameter list of the point function. Conditional statements are currently not allowed.

```
pointfunction five_point_avg(p) {
  float ONE_FIFTH;
  ONE_FIFTH = 0.2f;
  [1]p[0][0] = ONE_FIFTH*([0]p[-1][0] +
            [0]p[0][-1] + [0]p[ 0][0] + [0]p[0][1] +
                          [0]p[ 1][0]);
}
```

Above is a simple 2D 5 point Jacobi stencil that averages the point being processed and four of its neighboring points. It begins with the `pointfuction` keyword followed by the identifier `five_point_avg`. Only one grid data parameter, `p`, is passed. The body of the point function contains the declaration of a floating point variable `ONE_FIFTH` and the subsequent assignment of the value `0.2f` to that variable. The last line updates the current point in `p` at timestep 1 with the value of itself and its top, bottom, left, and right neighbors at timestep `0`.

7.3.11 Iterate and Check Every

The outer iterative loop of a stencil computation is defined using the `iterate` construct. An `iterate` contains a sequence of `stencil` and `reduction` definitions. The defined stencils are executed once every outer loop iteration, and are executed in the order in which the `stencil` definitions appear in the `iterate`. Reductions are performed every n-th iteration as specified by the `check ... every` clause. The defined reductions are also executed in the order in which the `reduction` definitions appear in the `iterate`. The general form of an `iterate` is shown below.

```
iterate /* maximum iterations */ {
  /* 1 or more stencils and / or reductions */
} /* optional check every clause */
```

An iterate contains a maximum number of iterations, which must be a positive integer literal. The only way for an iterate to terminate before the given number of iterations is through the definition of a convergence check via the optional `check ... every` clause following the iterate. The general form of the clause is shown below.

```
  check (/* conditional expression */)
  every /* frequency */ iterations
```

The conditional expression uses the `>`, `<`, `<=`, `>=`, `==`, and `!=` operators to form expressions using reduction variables, constants, and literal values. This expression evaluates to a boolean 'TRUE' or 'FALSE' value. If the condition evaluates to 'TRUE' the outer iterative loop stops executing. This condition is evaluated at the end of every $frequency$ iterations, as specified in the `every` part of the clause. $frequency$ must be a positive integer literal.

Inside of the iterate, stencils and reductions can be specified. These form the bulk of the computational work and are described in the following sections.

Stencils

A stencil is always located inside an iterate and defines an iteration over programmer-defined elements of an abstract n-dimensional grid. This iteration is not guaranteed to occur in any order and can occur in parallel. The dimensionality of the grid is determined by the dimensionality of the grid subsets defined within the stencil. A stencil has the general form shown below.

```
stencil /* identifier */ {
  /* sequence of (subgrid,expression) pairs */
}
```

A stencil begins with the `stencil` keyword followed by a unique identifier. Inside the body of the stencil are one or more statements where each statement consists of a subgrid followed by the ':' character followed by either an assignment expression or a point function call.

```
stencil jacobi_2d {
  [0][0:Nx-1]     : [1]a[0][0] = [0]a[0][0];
  [Ny-1][0:Nx-1]  : [1]a[0][0] = [0]a[0][0];
  [0:Ny-1][0]     : [1]a[0][0] = [0]a[0][0];
  [0:Ny-1][Nx-1]  : [1]a[0][0] = [0]a[0][0];

  [1:Ny-2][1:Nx-2] : five_point_avg(a);
}
```

An example stencil is shown above. The stencil begins with the `stencil` keyword followed by the identifier `jacobi_2d`. The body of the stencil sets the top, bottom, left, and right edges of grid data `a` at timestep `1` to their values at timestep `0`. Finally the body of the grid is computed with the point function `five_point_avg()`.

Reductions

A reduction consists of the `reduction` keyword, an identifier that names the reduction, and a reduction operator. Finally, a reduction contains a sequence of grid subsets and an operation that is to be performed on that subset. Reductions cannot contain point function calls.

The results of the computation at each grid point are combined with the reduction operator into a reduction variable. The name of the reduction acts as this reduction variable and can be used in the conditional statement of a `check ... every` clause. Legal reduction operators are '`+`', '`*`', '`max`', and '`min`'.

```
reduction max_diff max {
  [0:dim1-1][0:dim0-1] : fabs([1]a[0][0] - [0]a[0][0]);
}
```

A simple reduction is shown above. This reduction uses a reduction variable named `max_diff` to store the largest absolute difference between a value of an element of `a` at timesteps `0` and `1`.

7.4 Detailed Example

This section contains a detailed example of porting a non-trivial stencil, Rician denoising [21], from C to embedded SDSL. Rician denoising is used to remove a particular type of noise commonly found in MRI images and is one step in the CDSC medical imaging pipeline.

7.4.1 C Reference Code

A C function implementing Rician denoising is shown in Listing 7.2. In this section we describe the important features of the code; in subsequent sections we port this C code to an equivalent SDSL version.

Function Signature

The Rician denoise function signature (line 1) is shown below:

```
int rician3d(double *U, double *F, double sigma, double lambda,
             int dim0, int dim1, int dim2)
```

The `rician3d` function returns an integer status and takes the following parameters:

- `double *U`: Pointer to the denoised image (output)
- `double *F` ($0 \leq$ `F[i][j][k]` ≤ 1): Pointer to the original noisy image (input)
- `double sigma` (> 0): Parameter indicating type of Rician noise expected (input)
- `double lambda` (≥ 0): Controls denoising strength, smaller `lambda` implies stronger denoising (input)

```c
int rician3d(double *U, double *F, double sigma, double lambda, int dim0, int dim1, int dim2) {
  // U0, G are local arrays
  double *U0 = (double*)malloc(dim0*dim1*dim2*sizeof(double));
  double *G = (double*)malloc(dim0*dim1*dim2*sizeof(double));
  memset(G, 0, sizeof(double)*dim0*dim1*dim2);

  // Copy F to U
  memcpy(U, F, dim0*dim1*dim2*sizeof(double));

  // Constants
  const double DT = 5.0;
  const double EPSILON = 1.0E-20;
  const double sigma2 = sigma*sigma;
  const double gamma = lambda/sigma2;

  // Convergence
  const int max_iter = 50;
  const double TOL = 0.00001;
  bool converged;

  // Macro for 3D array indexing
  #define idx(z,y,x) ((x) + (y)*dim0 + (z)*dim0*dim1)
  // Gradient descent loop
  for (int t = 0; t < max_iter; ++t) {
    /* Copy U to U0 */
    for (int i = 0; i < dim2; ++i) {
      for (int j = 0; j < dim1; ++j) {
        for (int k = 0; k < dim0; ++k) {
          U0[idx(i,j,k)] = U[idx(i,j,k)];
        }}}
    // Approximate G = 1/|grad U|
    for (int i = 1; i < dim2 - 1; ++i) {
      for (int j = 1; j < dim1 - 1; ++j) {
        for (int k = 1; k < dim0 - 1; ++k) {
          G[idx(i,j,k)] = 1.0/sqrt(EPSILON +
          (U0[idx(i,j,k)] - U0[idx(i,j+1,k)])*(U0[idx(i,j,k)] - U0[idx(i,j+1,k)]) +
          (U0[idx(i,j,k)] - U0[idx(i,j-1,k)])*(U0[idx(i,j,k)] - U0[idx(i,j-1,k)]) +
          (U0[idx(i,j,k)] - U0[idx(i,j,k+1)])*(U0[idx(i,j,k)] - U0[idx(i,j,k+1)]) +
          (U0[idx(i,j,k)] - U0[idx(i,j,k-1)])*(U0[idx(i,j,k)] - U0[idx(i,j,k-1)]) +
          (U0[idx(i,j,k)] - U0[idx(i+1,j,k)])*(U0[idx(i,j,k)] - U0[idx(i+1,j,k)]) +
          (U0[idx(i,j,k)] - U0[idx(i-1,j,k)])*(U0[idx(i,j,k)] - U0[idx(i-1,j,k)]));
        }}}
    // Update U by a semi-implict step
    converged = true;
    for (int i = 1; i < dim2 - 1; ++i) {
      for (int j = 1; j < dim1 - 1; ++j) {
        for (int k = 1; k < dim0 - 1; ++k) {
          // Evaluate r = I1(U*F/sigma^2) / I0(U*F/sigma^2)
          // with a cubic rational approximation.
          double r = U0[idx(i,j,k)]*F[idx(i,j,k)]/sigma2;
          r = ( r*(2.38944 + r*(0.950037 + r)) )
          / ( 4.65314 + r*(2.57541 + r*(1.48937 + r)) );
          // Update U
          U[idx(i,j,k)] = (U0[idx(i,j,k)] +
                             DT*(U0[idx(i,j+1,k)]*G[idx(i,j+1,k)] +
                                 U0[idx(i,j-1,k)]*G[idx(i,j-1,k)] +
                                 U0[idx(i,j,k+1)]*G[idx(i,j,k+1)] +
                                 U0[idx(i,j,k-1)]*G[idx(i,j,k-1)] +
                                 U0[idx(i+1,j,k)]*G[idx(i+1,j,k)] +
                                 U0[idx(i-1,j,k)]*G[idx(i-1,j,k)] +
                                 gamma*F[idx(i,j,k)]*r))
                           /
                            (1.0 + DT*(G[idx(i,j+1,k)] + G[idx(i,j-1,k)] +
                                       G[idx(i,j,k+1)] + G[idx(i,j,k-1)] +
                                       G[idx(i+1,j,k)] + G[idx(i-1,j,k)] +
                                       gamma));
          if (fabs(U0[idx(i,j,k)] - U[idx(i,j,k)]) > TOL) {
            converged = false;
          }
        }}}
    if (converged) {
      break;
    }
  }
  #undef idx
  free(U0);
  free(G);
  return 0;
}
```

Listing 7.2: Rician Denoising, original C implementation.

- `int dim0`, `int dim1`, `int dim2`: Dimension sizes of `U` and `F` (e.g. `U[dim2][dim1][dim0]`) (input)

We will create a drop-in replacement for this function in C and SDSL.

Arrays

In addition to the arrays passed as parameters, temporary arrays `U0` and `G` are allocated on lines 3 and 4. Array `U0` is used to hold intermediate values of the denoised image `U` in a Jacobi iteration. Array `G` is used to hold the result of a curvature approximation as described in [21]. The output image `U` is initialized to the value of the input image `F` on line 11. All arrays are 1D data structures interpreted as 3D data structures and are accessed with the indexing macro defined on line 22.

Constants and Variables

Lines 11–14 in Listing 7.2 contain the declarations of a number of constants. `DT`, `EPSILON`, `sigma2`, and `gamma` are used throughout the denoise calculation. The purpose of these variables is beyond the scope of this document; please see [21] for more details.

Lines 17–19 contain declarations and definitions that control how many times the denoise operation will run. `max_iter` is the maximum number of iterations the denoise operation can run. `TOL`, defined on line 18, is used in the convergence check on line 67. Semantically, `TOL` is the maximum absolute difference allowed between `U[i][j][k]` and `U0[i][j][k]` for all $0 <$ `i` $<$ `dim2`, $0 <$ `j` $<$ `dim1`, $0 <$ `k` $<$ `dim0` for the denoise operation to converge. Whether or not the denoising has converged is tracked by the boolean variable `converged`, defined on line 19.

Main Iterative Loop

The main loop nest for Rician denoising runs from lines 24–74 in Listing 7.2. The outer loop runs, at most, for `max_iter=50` iterations, and has four phases:

- `U` copy (lines 26–30) — Copies `U` to `U0` temp array
- `G` stencil (lines 32–42) — Computes value for `G` using a stencil on `U0`
- `U` stencil (lines 45–70) — Computes value for `U` using a stencil on `U0`, `G`, and `F`
- Convergence check (lines 71–73) — Breaks out of the main loop if `U` converged

The `U` copy phase simply copies the contents of array `U` to `U0` as part of the Jacobi iteration. The `G` stencil, `U` stencil, and convergence check are described in Section 7.4.1, Section 7.4.1, and Section 7.4.1, respectively.

`G` Stencil

The `G` stencil computes values for all elements of the `G` array except the first and last elements in each dimension. Values for array element `G[i][j][k]` are computed with a 7-point 3D stencil on array `U0` using a center element (`U0[i][j][k]`) and its neighbors along each spatial dimension (`U0[i-1][j][k]`, `U0[i+1][j][k]`, `U0[i][j-1][k]`, `U0[i][j+1][k]`, `U0[i][j-1]`, and `U0[i][j][k+1]`). While the loop body statement appears to use substantially more than 7 points, close examination reveals that there are six subexpressions of the form $(center{-}neighbor)^2$ using each of the 7 stencil points multiple times.

`U` Stencil

The `U` stencil computes values for all elements of the `U` array except the first and last elements in each dimension. For each element of `U[i][j][k]` a scalar value `r` is computed using `U0[i][j[k]` and `F[i][j[k]` at lines 50–52. This is followed by an update of `U` at lines 54–66 using a 7-point 3D stencil on `U0`. This stencil is the same "center and neighbors in each dimension" pattern as described in Section 7.4.1. The `U` update also uses a 6-point 3D stencil on `G` that is the same shape as above without the center point. After each element of `U` is computed, at lines 67–69 the result is compared to the previous value of `U` (stored in `U0`) to check for convergence.

Convergence Check

At lines 71–73 the `converged` flag, set at lines 44 and 68, is checked. If the absolute difference between successive values of `U` at any given element was greater than `TOL` then the converge flag will be set to false at line 68 and the main loop will continue executing. If, however, absolute difference for *every* computed value of `U` and its predecessor in `U0` was less than `TOL` the `converged` flag is never set to false and the calculation is considered to be converged. Control flow breaks from the main loop at line 72 and proceeds to cleanup.

7.4.2 Creating a C/SDSL Skeleton Function

We wish to create a drop-in replacement for the Rician denoise C function described in Section 7.4.1. We begin by creating a thin C wrapper function around an empty `#pragma sdsl` as shown below.

```
int sdsl_rician3d(double *U, double *F, double sigma, double lambda,
                  int dim0, int dim1, int dim2) {
  #pragma sdsl begin
  #pragma sdsl end
  return 0;
}
```

The function from Listing 7.2 is renamed to `sdsl_rician3d` and its signature is preserved. Begin and end SDSL pragmas are added inside the function followed by a return statement.

7.4.3 Passing Data from C to SDSL

With the skeleton in place, we need to pass parameters and arrays from C to SDSL.

Parameters and Constants

We begin by declaring SDSL parameters for each of the function parameters as shown in Listing 7.3. Lines 5–9 contain SDSL parameter declarations binding the declared parameters to identically named C variables.

```
int sdsl_rician3d(double *U, double *F, double sigma, double lambda,
                  int dim0, int dim1, int dim2) {
  #pragma sdsl begin
  // Parameters
  int dim0;
  int dim1;
  int dim2;
  double sigma;
  double lambda;

  // Constants
  double DT = 5.0;
  double EPSILON = 1.0E-20;
  double sigma2 = sigma*sigma;
  double gamma = lambda/sigma2;
  double TOL = 0.00001;
  #pragma sdsl end
  return 0;
}
```

Listing 7.3: Parameters passed from C to SDSL, constants defined in SDSL.

Arrays

The SDSL source with all code to pass the example arrays is shown in Listing 7.4. Passing arrays from C to SDSL requires both C and SDSL code. In C, on lines 4–5 we declare the temporary array `G` and initialize its contents to 0. Line 8 initializes the contents of `U`, the denoised output image, with the contents of `F`, the noisy input image. Note that `U0` is not declared or defined in C.

In SDSL, line 26 declares a `grid g` with with the same dimension sizes as the C arrays being passed. Lines 28–30 contain SDSL `griddata` declarations for `U`, `G`, and `F`. `U` is defined on 2 timesteps, representing arrays `U0` and `U` from the original C code. `G` and `F` are defined on 1 timestep.

```
int sdsl_rician3d(double *U, double *F, double sigma, double lambda,
                  int dim0, int dim1, int dim2) {
  // G is a local array
  double *G = (double*)malloc(dim0*dim1*dim2*sizeof(double));
  memset(G, 0, sizeof(double)*dim0*dim1*dim2);

  // Copy F to U
  memcpy(U, F, dim0*dim1*dim2*sizeof(double));

  #pragma sdsl begin
  // Parameters
  int dim0;
  int dim1;
  int dim2;
  double sigma;
  double lambda;

  // Constants
  double DT = 5.0;
  double EPSILON = 1.0E-20;
  double sigma2 = sigma*sigma;
  double gamma = lambda/sigma2;
  double TOL = 0.00001;

  // Grid and griddata
  grid g[dim2][dim1][dim0];

  double griddata U on g at 0,1;
  double griddata G on g at 0;
  double griddata F on g at 0;
  #pragma sdsl end
  free(G);
  return 0;
}
```

Listing 7.4: Arrays passed from C to SDSL.

7.4.4 Implementing `pointfunctions`

For Rician denoise we use separate `pointfunctions` to perform the stencil computations that produce elements of `G` and `U`. The next two sections describe the SDSL used to implement these stencil computations.

`approx_g`

The `approx_g` `pointfunction` is shown on lines 32–40 of Listing 7.5. This `pointfunction` implements the "`G` stencil" described in Section 7.4.1. At line 32 the `pointfunction` is declared as taking two `griddata` parameters, `u` and `g`. Lines 33–39 contain a single statement that calculates a point of `g` using neighbor points from `u`.

Note that the `griddata` references in this statement address points via offsets from the current timestep and point, e.g. `[0]u[-1][0][0]` is the value of `u` at the current timestep (`U0` from the original) offset from the current point by -1, 0, and 0 in the three spatial dimensions. Further note that the statement uses the `sqrt()` function from `math.h`, thereby taking advantage of `math.h` function availability in SDSL. Finally, note that the constant `EPSILON` is used without including it as a parameter. This is required; all parameters to a `pointfunction` *must* be `griddata`. All constants and parameters are available within a `pointfunction`.

```
int sdsl_rician3d(double *U, double *F, double sigma, double lambda,
                  int dim0, int dim1, int dim2) {
  // G is a local array
  double *G = (double*)malloc(dim0*dim1*dim2*sizeof(double));
  memset(G, 0, sizeof(double)*dim0*dim1*dim2);

  // Copy F to U
  memcpy(U, F, dim0*dim1*dim2*sizeof(double));

  #pragma sdsl begin
  // Parameters
  int dim0;
  int dim1;
  int dim2;
  double sigma;
  double lambda;

  // Constants
  double DT = 5.0;
  double EPSILON = 1.0E-20;
  double sigma2 = sigma*sigma;
  double gamma = lambda/sigma2;
  double TOL = 0.00001;

  // Grid and griddata
  grid g[dim2][dim1][dim0];

  double griddata U on g at 0,1;
  double griddata G on g at 0;
  double griddata F on g at 0;

  pointfunction approx_g(u,g) {
    [0]g[0][0][0] = 1.0 / sqrt(EPSILON +
        ([0]u[0][0][0] - [0]u[ 0][ 1][ 0]) * ([0]u[0][0][0] - [0]u[ 0][ 1][ 0]) +
        ([0]u[0][0][0] - [0]u[ 0][-1][ 0]) * ([0]u[0][0][0] - [0]u[ 0][-1][ 0]) +
        ([0]u[0][0][0] - [0]u[ 0][ 0][ 1]) * ([0]u[0][0][0] - [0]u[ 0][ 0][ 1]) +
        ([0]u[0][0][0] - [0]u[ 0][ 0][-1]) * ([0]u[0][0][0] - [0]u[ 0][ 0][-1]) +
        ([0]u[0][0][0] - [0]u[ 1][ 0][ 0]) * ([0]u[0][0][0] - [0]u[ 1][ 0][ 0]) +
        ([0]u[0][0][0] - [0]u[-1][ 0][ 0]) * ([0]u[0][0][0] - [0]u[-1][ 0][ 0]) )
  }
  #pragma sdsl end
  free(G);
  return 0;
}
```

Listing 7.5: Point function to calculate values of `G`.

`update_u`

The SDSL highlighted at lines 42–62 of Listing 7.6 corresponds to the "`U` stencil" described in Section 7.4.1. This `pointfunction` is defined starting at line 42 and takes three `griddata` parameters, `u`, `g`, and `f`. Lines 44–45 compute the local variable `r`, while lines 46–61 are a single statement used to compute the value of the current point of `u` at the next timestep, `[1]u[0][0][0]`.

```
int sdsl_rician3d(double *U, double *F, double sigma, double lambda,
                  int dim0, int dim1, int dim2) {
  // G is a local array
  double *G = (double*)malloc(dim0*dim1*dim2*sizeof(double));
  memset(G, 0, sizeof(double)*dim0*dim1*dim2);

  // Copy F to U
  memcpy(U, F, dim0*dim1*dim2*sizeof(double));

  #pragma sdsl begin
  // Parameters
  int dim0;
  int dim1;
  int dim2;
  double sigma;
  double lambda;

  // Constants
  double DT = 5.0;
  double EPSILON = 1.0E-20;
  double sigma2 = sigma*sigma;
  double gamma = lambda/sigma2;
  double TOL = 0.00001;

  // Grid and griddata
  grid g[dim2][dim1][dim0];

  double griddata U on g at 0,1;
  double griddata G on g at 0;
  double griddata F on g at 0;

  pointfunction approx_g(u,g) {
    [0]g[0][0][0] = 1.0 / sqrt(EPSILON +
                           ([0]u[0][0][0] - [0]u[ 0][ 1][ 0]) * ([0]u[0][0][0] - [0]u[ 0][ 1][ 0]) +
                           ([0]u[0][0][0] - [0]u[ 0][-1][ 0]) * ([0]u[0][0][0] - [0]u[ 0][-1][ 0]) +
                           ([0]u[0][0][0] - [0]u[ 0][ 0][ 1]) * ([0]u[0][0][0] - [0]u[ 0][ 0][ 1]) +
                           ([0]u[0][0][0] - [0]u[ 0][ 0][-1]) * ([0]u[0][0][0] - [0]u[ 0][ 0][-1]) +
                           ([0]u[0][0][0] - [0]u[ 1][ 0][ 0]) * ([0]u[0][0][0] - [0]u[ 1][ 0][ 0]) +
                           ([0]u[0][0][0] - [0]u[-1][ 0][ 0]) * ([0]u[0][0][0] - [0]u[-1][ 0][ 0]) );
  }

  pointfunction update_u(u,g,f) {
    double r = [0]u[0][0][0]*[0]f[0][0][0]/sigma2;
    r =( r*(2.38944 + r*(0.950037 + r)) )
           / ( 4.65314 + r*(2.57541 + r*(1.48937 + r)) );
    [1]u[0][0][0] =   ([0]u[0][0][0] +
                       DT*([0]u[ 0][ 1][ 0]*[0]g[ 0][ 1][ 0] +
                           [0]u[ 0][-1][ 0]*[0]g[ 0][-1][ 0] +
                           [0]u[ 0][ 0][ 1]*[0]g[ 0][ 0][ 1] +
                           [0]u[ 0][ 0][-1]*[0]g[ 0][ 0][-1] +
                           [0]u[ 1][ 0][ 0]*[0]g[ 1][ 0][ 0] +
                           [0]u[-1][ 0][ 0]*[0]g[-1][ 0][ 0] +
                           gamma*[0]f[0][0][0]*r))
                     /
                      (1.0 + DT*[0]g[ 0][ 1][ 0] +
                                [0]g[ 0][-1][ 0] +
                                [0]g[ 0][ 0][ 1] +
                                [0]g[ 0][ 0][-1] +
                                [0]g[ 1][ 0][ 0] +
                                [0]g[-1][ 0][ 0] +
                                gamma);
  }
  #pragma sdsl end
  free(G);
  return 0;
}
```

Listing 7.6: Point function to calculate values of `U`.

7.4.5 Defining the `iterate`

Listing 7.7 shows the complete replacement for the reference Rician denoise code.

```
int sdsl_rician3d(double *U, double *F, double sigma, double lambda,
                  int dim0, int dim1, int dim2) {
  // G is a local array
  double *G = (double*)malloc(dim0*dim1*dim2*sizeof(double));
  memset(G, 0, sizeof(double)*dim0*dim1*dim2);

  // Copy F to U
  memcpy(U, F, dim0*dim1*dim2*sizeof(double));

  #pragma sdsl begin
  // Parameters
  int dim0;
  int dim1;
  int dim2;
  double sigma;
  double lambda;

  // Constants
  double DT = 5.0;
  double EPSILON = 1.0E-20;
  double sigma2 = sigma*sigma;
  double gamma = lambda/sigma2;
  double TOL = 0.00001;

  // Grid and griddata
  grid g[dim2][dim1][dim0];

  double griddata U on g at 0,1;
  double griddata G on g at 0;
  double griddata F on g at 0;

  pointfunction approx_g(u,g) {
    [0]g[0][0][0] = 1.0 / sqrt(EPSILON +
                           ([0]u[0][0][0] - [0]u[ 0][ 1][ 0]) * ([0]u[0][0][0] - [0]u[ 0][ 1][ 0]) +
                           ([0]u[0][0][0] - [0]u[ 0][-1][ 0]) * ([0]u[0][0][0] - [0]u[ 0][-1][ 0]) +
                           ([0]u[0][0][0] - [0]u[ 0][ 0][ 1]) * ([0]u[0][0][0] - [0]u[ 0][ 0][ 1]) +
                           ([0]u[0][0][0] - [0]u[ 0][ 0][-1]) * ([0]u[0][0][0] - [0]u[ 0][ 0][-1]) +
                           ([0]u[0][0][0] - [0]u[ 1][ 0][ 0]) * ([0]u[0][0][0] - [0]u[ 1][ 0][ 0]) +
                           ([0]u[0][0][0] - [0]u[-1][ 0][ 0]) * ([0]u[0][0][0] - [0]u[-1][ 0][ 0]) );
  }

  pointfunction update_u(u,g,f) {
    double r = [0]u[0][0][0]*[0]f[0][0][0]/sigma2;
    r =( r*(2.38944 + r*(0.950037 + r)) )
           / ( 4.65314 + r*(2.57541 + r*(1.48937 + r)) );
    [1]u[0][0][0] =   ([0]u[0][0][0] +
                       DT*([0]u[ 0][ 1][ 0]*[0]g[ 0][ 1][ 0] +
                           [0]u[ 0][-1][ 0]*[0]g[ 0][-1][ 0] +
                           [0]u[ 0][ 0][ 1]*[0]g[ 0][ 0][ 1] +
                           [0]u[ 0][ 0][-1]*[0]g[ 0][ 0][-1] +
                           [0]u[ 1][ 0][ 0]*[0]g[ 1][ 0][ 0] +
                           [0]u[-1][ 0][ 0]*[0]g[-1][ 0][ 0] +
                           gamma*[0]f[0][0][0]*r))
                     /
                      (1.0 + DT*[0]g[ 0][ 1][ 0] +
                                [0]g[ 0][-1][ 0] +
                                [0]g[ 0][ 0][ 1] +
                                [0]g[ 0][ 0][-1] +
                                [0]g[ 1][ 0][ 0] +
                                [0]g[-1][ 0][ 0] +
                                gamma);
  }

  iterate 50 {
    stencil gs {
      [1:dim2-2][1:dim1-2][1:dim0-2] : approx_g(U,G);
    }
    stencil us {
      [1:dim2-2][1:dim1-2][1:dim0-2] : update_u(U,G,F);
    }
    reduction max_diff max {
      [1:dim2-2][1:dim1-2][1:dim0-2] : fabs([1]U[0][0][0] - [0]U[0][0][0]);
    }
  } check (max_diff < TOL) every 10 iterations
  #pragma sdsl end
  free(G);
  return 0;
}
```

Listing 7.7: Complete SDSL function.

Lines 64–74 define an `iterate` that implements the main iterative loop control described in Section 7.4.1. Line 64 specifies that the iterate runs for *at most* 50 iterations. Subsequent sections describe the stencils, reduction, and convergence check used in the iterate.

Stencils

The `iterate` contains two `stencils`. `stencil gs`, defined on lines 65–67, corresponds to the code described in Section 7.4.1; `stencil us`, defined on lines 68–70, corresponds to the loop nest described in Section 7.4.1. Each stencil defines a single region over which it is applied, `[1:dim2-2][1:dim1-2][1:dim0-2]`. This region corresponds to the spatial loop bounds of the original `G` and `U` stencils located at lines 32–34 and 45–47 of the original C implementation in Listing 7.2. At each point of the defined region, both stencils call the appropriate `pointfunction`. At line 66 `approx_g` is called with `griddata` parameters `U` and `G`. At line 69 `update_u` is called with `griddata` parameters `U`, `G`, and `F`

`reduction`

The `reduction` defined at lines 71–73 of Listing 7.7 is responsible for computing the absolute difference between successive values of `U` over all points in `[1:dim2-2][1:dim1-2][1:dim0-2]`. Instead of computing these differences per point as the `U` stencil is being executed, as described in Section 7.4.1, the `reduction` performs a separate sweep of the region and stores the maximum absolute difference in the reduction variable `max_diff` using the reduction operation `max`. The value of `max_diff` is subsequently used in the convergence check at line 74.

Convergence Check

The convergence check for the SDSL version of Rician denoise is defined at line 74 of Listing 7.7. It specifies that if the reduction variable `max_diff` is less than the constant `TOL` the `iterate` should terminate. It is specified that this check occurs every 10 iterations. By increasing the number of iterations between convergence checks we enable the SDSL compiler to perform aggressive time tiling optimizations. If the exact behavior of the original Denoise implementation was desired, we would modify the convergence check to occur every iteration.

7.4.6 Generating Code

The Rician denoise functions shown in Listing 7.2 and Listing 7.7 are combined with the driver shown in Listing 7.8 and accessory functions for timing and correctness checks (not shown) to produce a complete program suitable for compilation and execution.

```
int main() {
  // Problem size
  const int dim0 = 128;
  const int dim1 = 128;
  const int dim2 = 128;

  // Timing
  double start, finish;

  // Data arrays
  double *U = (double*)malloc(dim0*dim1*dim2*sizeof(double));
  double *F = (double*)malloc(dim0*dim1*dim2*sizeof(double));
  #ifdef CHECK_REF
  double *Uref = (double*)malloc(dim0*dim1*dim2*sizeof(double));
  double *Fref = (double*)malloc(dim0*dim1*dim2*sizeof(double));
  #endif

  // Populate arrays with random values
  srand(45);
  for (int i = 0; i < dim0*dim1*dim2; ++i) {
    F[i] = (double)rand() / (double)(RAND_MAX);
    #ifdef CHECK_REF
    Fref[i] = F[i];
    #endif
  }

  #ifdef CHECK_REF
  // Reference denoise call
  start = timestamp();
  rician3d(Uref, Fref, 0.05, 0.065, dim0, dim1, dim2);
  finish = timestamp();
  printf(" REF: %.6f sec\n",finish-start);
  #endif

  // SDSL denoise call
  start = timestamp();
  sdsl_rician3d(U, F, 0.05, 0.065, dim0, dim1, dim2);
  finish = timestamp();
  printf("SDSL: %.6f sec\n",finish-start);

  int retval = 0;
  #ifdef CHECK_REF
  // Check results
  retval = correctness_check(Uref,U,dim0,dim1,dim2);
  #endif

  // Cleanup
  free(U);
  free(F);
  #ifdef CHECK_REF
  free(Uref);
  free(Fref);
  #endif
  return retval;
}
```

Listing 7.8: SDSL driver code.

CHAPTER 8

Related Work

8.1 Data Layout Transformations

Jaeger and Barthou [30] propose a method to optimize stencil computations on CPU and GPU by strategically padding array dimensions in order to enforce vector alignment constraints on CPU and reduce bank conflicts on GPU. An ILP formulation is developed to minimize the amount of padding necessary for a given stencil. In their framework, stream alignment conflicts in all array dimensions except the fastest varying are resolved by adding padding at the end of each dimension. The method proposed in this dissertation pads only on the fastest varying dimension and resolves all stream alignment conflicts except at problem boundaries.

Sung, Stratton, and Hwu [51] describe a compile time data layout transformation for structured grid codes on GPU. Their approach uses array index permutation and strip mining as primitive operations that define a legal space of data layouts. A series of rules are applied to determine optimal layout for a given GPU configuration and access pattern. Finally, a dataflow analysis is performed to generate access functions for the new data layout. Their layout transformation does not address the constraints imposed by a stream alignment conflict on CPU as the method focuses on maximizing utilization of available memory bandwidth on GPU.

8.2 Data Locality Optimizations

Wolf and Lam[58] introduce loop tiling as a unimodular transformation for increasing data locality in perfectly nested loops, including stencils. A heuristic algorithm is used to determine and search legal loop skewing, reversal, and permutation transformations for the compound transformation that yields the minimal number of memory accesses per loop iteration. Permutable loop nests are then tiled with a manually determined tile size. Their technique shows considerable performance gains on a two dimensional successive overrelaxation kernel when combined with pipelined parallelism.

McCalpin and Wonnacott [39] focus on stencils and develop a method using skewing and tiling. Their "time skewing" utilizes skewing and loop fusion to enable the tiling of imperfectly nested stencils. Wonnacott [59] further extends this method to include a more general class of affine codes that include stencils.

Song and Li [48] also develop a compiler technique based on skewing and tiling for use with imperfectly nested stencils. Song and Li introduce the concepts of slope and offset when discussing tile geometry. As in this dissertation, they use slope to refer to the change in tile start and end points at different timesteps. Their method computes offsets per-loop nest; offsets in this work are computed per-statement-per-dimension. Song and Li construct a data structure similar to the DSG used in this work, although a different algorithm is used to compute both slope and offsets.

The tiles produced by their framework are one dimensional and have parallelogram geometries, so a given tile only has one slope, as opposed to the one or two slopes computed, respectively, for the parallelogram and trapezoidal tiles along each dimension in this work.

Li and Song [36] later extend their technique to include multidimensional tiling. A data structure similar to the DSG is used and the algorithm used to compute slopes is that of Ahuja et al. [1] as in this work. As in their previous work, tiles produced have parallelogram geometries along any individual dimension, leading to a parallelotope shape in multiple dimensions; the split tiling methods described in this dissertation will generally produce nested trapezoidal tiles and only produce parallelogram tiles in the outermost spatial dimension when performing hybrid split tiling. Li and Song again compute one slope and set of offsets but expand the number of dimensions in the more recent work.

Taken together, the methods of Wolf, Lam, McCalpin, Wonnacot, Song, and Li produce tiles with parallelotope shape and are referred to as "standard" tiling in Chapter 6. As descibed in Section 6.1 the data layout transformation described in this work is incompatible with these methods of tiling due to inter-tile dependences and are amenable to pipeline parallelism as opposed to the concurrent start parallelism of the split tiling techniques described in this work.

"Cache-oblivious" techniques for tiling stencils do not consider tile or cache sizes and have been used by a number of researchers. Prokop [45] introduces a cache-oblivious technique for recursively tiling a one-dimensional three point Jacobi stencil. Frigo and Strumpen [19, 20] develop a cache-oblivious tiling technique for an n-dimensional stencil. In their technique, the stencil iteration space is recursively subdivided along time and space dimensions into smaller trapezoids and parallelograms. The recursive subdivision ends when tiles are one timestep in height; spatial tile sizes are fixed at this point. Tile slopes are manually determined and used to determine exact tile shapes. In split tiling, time and space tile sizes remain parametric and are set by the user at run time; tile slopes are determined automatically via backslicing. The domain specific language Pochoir [52] also uses cache-oblivious tiling techniques and is described further in Section 8.3.

Overlapped tiling is proposed by Krishnamoorthy et al. [34]. In overlapped tiling adjacent tiles perform identical computations in order to eliminate inter-tile dependences. Their technique uses trapezoidal tiles with overlapping components where computations are replicated between two tiles. This technique uses polyhedral compiler analysis to define hyperplanes that tile the iteration space as in standard tiling. Further polyhedral analysis is performed to determine companion hyperplanes that eliminate inter-tile dependences. Overlapped tiling is able to achieve concurrent start of parallel tiles as in the split tiling techniques described in this work. All tiles in overlapped tiling are shaped as upright trapezoids, while in this work a mix of non-overlapping upright and inverted trapezoidal tiles are used. Holewinski, Pouchet, and Sadayappan [27] further develop overlapped tiling for optimizing stencils on GPU, while Guo et al. [24] explore the use of overlapped tiling in the context of the Hierarchically Tiled Array data type.

Split tiling was first proposed by Krishnamoorthy et al. [34]. As in their overlapped tiling method, polyhedral analysis determines hyperplanes defining a standard tiling and further analysis determines companion hyperplanes to eliminate inter-tile dependences. Split tiling uses the companion hyperplanes to define new tiles that can be computed independently of the original tiles. Their method creates the characteristic upright and inverted tiles of split tiling, although upright tiles are triangular instead of trapezoidal as in this work. Additionally, their approach uses a polyhedral, linear algebraic technique to determine tile geometries while this work uses a graph-theoretic approach. Finally, their technique assumes a scalar or auto-vectorized innermost loop while the technique in this work is adapted for a DLT transformed innermost loop.

Grosser et al. [23] propose a variant of split tiling for GPU with substantially different tile shaping and code generation techniques than this work. Their method first constructs an upright pyramid tile from the dependences of a single arbitrary point. This tile is transformed with a number of geometric operations to produce an abstract set of multidimensional parallel tiles. These abstract tiles are mapped to the iteration domain of the stencil and code is generated for GPU with polyhedral tools. The end result is split tile shapes that are trapezoidal and very similar to those created using the nested split tiling technique defined in this work. Since GPU are not affected by the stream alignment conflict described in this work no data layout transformation is used in their technique.

Bandishti, Pananilath, and Bondhugula [5] propose a "diamond" tiling technique. In their technique polyhedral analysis is used to determine hyperplanes that divide the iteration space into a set of non-overlapping tiles shaped like diamonds. Like the split tiling variants in this work their technique enables the concurrent start of tiles across parallel processing nodes. The diamond tile shape enables more reuse than the trapezoidal tiles used in this work. No layout transformation is performed in their work and vectorization of scalar C code is performed by the platform compiler as opposed to the vector intrinsic code generated by the system described in this work.

Grosser et al. [22] develop a technique for tiling stencils on GPU similar to the diamond tiles of Bandishti, Pananilath, and Bondhugula and the hybrid split tiling method described in this work. In their work, non-standard tiling is only performed on the innermost dimension. Tiles resembling the diamond tiles of Bandishti, Pananilath, and Bondhugula with the points at high and low time values replaced by planes. All other dimensions are tiled using standard tiling to reduce shared memory footprint and optimize for other GPU specific issues.

While this section has focused on the data locality optimizations most directly comparable to the work in this dissertation, a number of other groups have made substantial contributions to the state of the art. Prokop, Frigo and Strumpen, Strzodka et al. [50], and Tang et al. [52] develop various methods for tiling without knowledge of or tuning for a specific cache size (“cache-oblivious” methods). Micikevicius et al. [41] hand-tune a 3D finite difference computation stencil and achieved an order of magnitude performance increase over existing CPU implementations on GT200-based Tesla GPU. Datta et al. [11] developed an optimization and auto-tuning framework for stencil computations, targeting multi-core systems, NVidia GPU, and Cell SPU. Treibig Wellein and Hager[53] use a parallel wavefront tiling technique on both Jacobi and Gauss-Seidel stencils to achieve high performance on multicore CPU architectures with shared last level caches. [7, 25, 53].

8.3 Domain Specific Languages and Compilers

Among the numerous research efforts to optimize stencil computations some develop a specialized DSL and compiler. In general, these DSL allow specification of a limited set of stencils and generate code for one or more architectures. In this section the stencil DSL most similar to SDSL are examined in and other related DSL are briefly summarized.

Christen, Schenk, and Burkhart [7] propose PATUS, a framework that uses a stencil description, optimization strategy, and target machine descriptions to generate and autotune code for efficient execution on CPU and GPU. Their DSL allows for the specification of a domain over which a multidimensional stencil can be applied, similar to a `pointfunction` in SDSL. PATUS refers to this as an *operation*. In PATUS, a user defines a strategy for optimizing a stencil that specifies low-level tiling and parallelism decisions as well as autotunable parameters such as tile sizes. In SDSL decisions about parallelism and tiling are made on a per-backend basis. The approach used by PATUS allows the user to exercise a certain degree of control over the execution of their program however is limited in its ability to describe advanced optimizations like DLT, standard tiling, split tiling, and overlapped tiling.

Tang et al. [52] propose the Pochoir DSL and stencil compiler. Their language uses a DSL that is legal C++ code. This DSL is compiled to optimized Cilk [28] code for compilation by the Intel C Compiler. Their DSL allows specification of a multidimensional stencil similar to a `pointfunction` in SDSL in addition to boundary conditions for this stencil. Pochoir generated code uses a cache-oblivious recursive decomposition of the iteration domain into multidimensional trapezoids for parallel execution on shared memory systems. SDSL is directly compared to Pochoir in this dissertation and shown to achieve comparable or better performance. Neither PATUS nor Pochoir can generate optimized time-tiled code for multi-statement, multi-stencil computations such as the finite difference time domain (FDTD) stencil.

A DSL compiler that uses overlapped tiling for parallel execution of stencils on GPU is described by Holewinski, Pouchet, and Sadayappan [27]. This DSL uses the concept of a rectangular grid bounding the entire computation along with subgrids defining a partition of the grid. Their language features recursively defined data types similar to `structs` in the C programming language. Similar to SDSL, their system can generate code for multistatement stencils operating over subsets of a grid.

The extension of Chapel for the description of dense and sparse stencils is investigated by Barrett et al. [6]. The Chapel work enables automated distributed memory parallelization of stencils but does not address time-tiling or vectorization. Ragan-Kelly et al. [46] propose the Halide DSL for increasing the performance of image processing pipelines. Their approach focuses on executing a DAG of single timestep stencils thus is not directly comparable to PATUS, Pochoir, or SDSL.

8.4 Other Stencil Optimizations

Two works decompose a stencil statement into component operations and optimize their order of execution. Deitz, Chamberlain, and Snyder [14] propose a method called array subexpression elimination to avoid redundant computation of partial sums in stencils. Stock et al. [49] implement a retiming optimization that takes advantage of the commutativity and associativity of operations in stencils to reorder component operations of single timestep stencils in order to maximize register reuse in higher order stencils. This work uses the DLT layout transformation described in this dissertation to enable rotating register reuse in convolutions.

Vasilache et al. [54] use an ILP formulation to concurrently optimize for parallelism, data locality, contiguity of data access, and data layout. Their method is a more general optimization technique applicable to a wider variety of programs than the DLT and split tiling transformations described in this dissertation. Kong et al. [33] also proposes an integrated optimization framework for a class of computations including stencils. In their work high level tiling transformation precedes the transformation and generation of tile-sized, SIMD optimized codelets.

CHAPTER 9

Future Work

This chapter examines the integration of DLT and split tiling with a number of known iterative methods for computing stencils. Nearly all of the recent literature on stencil computations focuses on the Jacobi iterative method. With Jacobi iterations a stencil computation is performed and output is written to one or more destination arrays. After each outer iteration the destination array is either copied back to the source array or a pointer swap is executed to convert output data for one iteration to input data for the next iteration.

This iterative method has a number of flaws. It is slower to converge to a solution when used in solvers. It also doubles the amount of memory required to represent a problem because of the separation of input and output data. Other methods exist without these weaknesses, however in general they have not been widely explored in recent stencil optimization literature.

In Section 9.1 we explore the potential of applying DLT to the Gauss-Seidel iterative method, and in Section 9.2 we look at opportunities for applying DLT to the Red-Black variant of the Gauss-Seidel method. We conclude in Section 9.3 with a discussion of one possible

9.1 Gauss-Seidel

The Gauss-Seidel iterative method replaces the input and output arrays of the Jacobi method with a single array where both input and output are stored simultaneously and calculations are performed in-place. In general this method converges on a solution more quickly than Jacobi when used in iterative solvers. Further, this method uses half of the memory used by the Jacobi method for intermediate results.

Gauss-Seidel has traditionally been viewed as difficult to parallelize because of the dependency pattern. A straightforward spatial decomposition of the iteration space is not possible because of transitive dependences on data computed earlier in a given timestep.

By introducing a skew to the iteration space, however, we can successfully parallelize the computation along a wavefront. A method for performing DLT on successive wavefronts has been demonstrated as being reducible to a Red / Black Gauss Seidel. While this method requires non-optimized computation of a prologue and epilogue, we believe it represents a promising avenue of research into the acceleration and parallelization of Gauss-Seidel computations, a topic that has been virtually ignored in recent literature.

9.2 Red-Black Gauss-Seidel

The Red-Black Gauss-Seidel iterative method is a compromise between the Jacobi and Gauss-Seidel iterative methods. Alternating data elements are labeled ″red″ and ″black″. Solvers alternate between computing ″red″ output using ″black″ input and ″black″ output using ″red″ input. This technique breaks the spatial loop

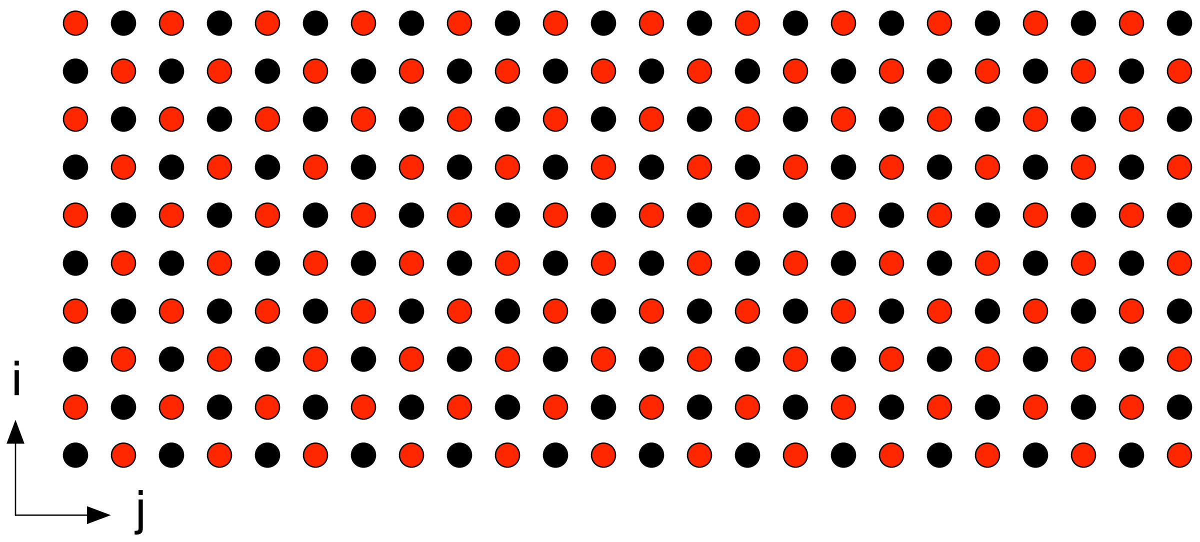


Figure 9.1: Red-Black Gauss-Seidel on 10 × 24 grid

carried dependences in the Gauss-Seidel method, thus enabling the parallelization of the code. While the code is parallel, it still suffers from a stream alignment conflict and will not be able to be vectorized efficiently without a data layout transformation or further optimization.

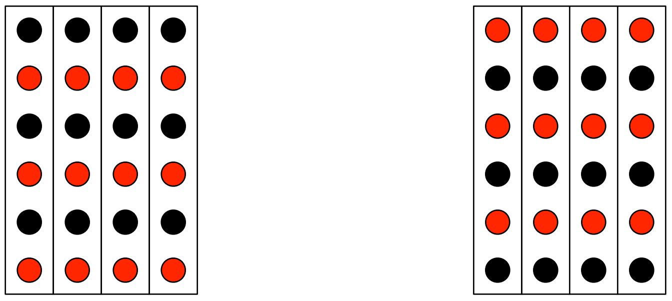

(a) Even rows **(b)** Odd rows

Figure 9.2: Red-Black Gauss-Seidel on 10 × 24 grid after DLT

DLT can be applied to Red-Black Gauss-Seidel in a very straightforward manner. In Figure 9.2 the DLT layout transformation is applied to one dimension of a red and black colored array. Note that all red elements are grouped together in vectors as well as all black elements. This grouping enables red and black results to be computed on SIMD units without incurring the overhead of redundant unaligned loads or unnecessary shuffles.

9.3 Distributed Memory DLT and Split Tiling

Split tiling and DLT can be applied in a distributed memory context. In a distributed memory context we seek coarse grain parallelism at the inter-node level, load balanced coarse grain parallelism at the intra-node level, data locality within the caches of each core, and fine grain parallelism for SIMD acceleration. We propose one possible approach to achieving these goals with multilevel nested split tiling and DLT.

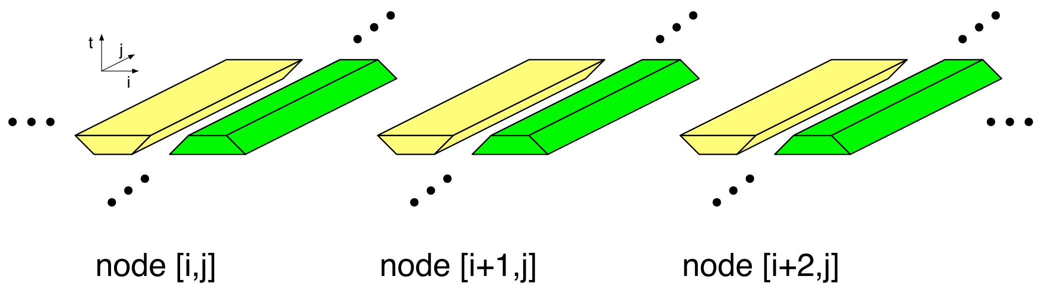


Figure 9.3: Steady state per node-tiling for distributed memory nested split tiled code in two dimensions. Each node contains an upright and inverted tile in the slowest varying dimension.

We first perform a spatial decomposition of all n-dimensional arrays onto an n-dimensional Cartesian grid of nodes in a distributed memory cluster. DLT is applied globally to all arrays involved in the stencil, thus DLT boundary conditions only appear on boundary nodes along the fastest varying dimension. Each node contains exactly one upright and one inverted split tile in the slowest varying dimension. An example of this tiling for a two spatial dimension stencil is shown in Figure 9.3. These tiles are recursively split tiled such that exactly one upright and one inverted tile is present per-node in each dimension down to and including the slowest varying. This is the first level of tiling that enables coarse grain parallelism across cluster nodes. Tile sizes are manually chosen such that a significant proportion of each node's main memory is used.

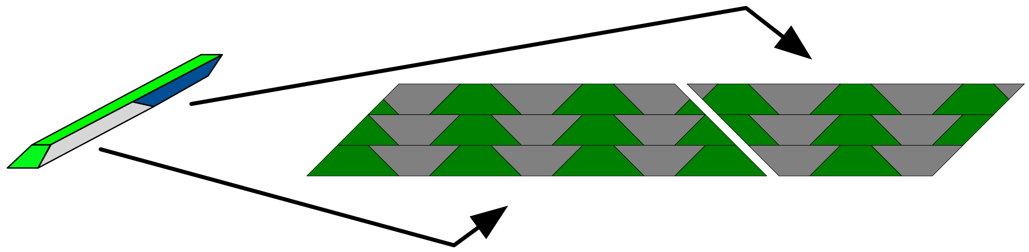
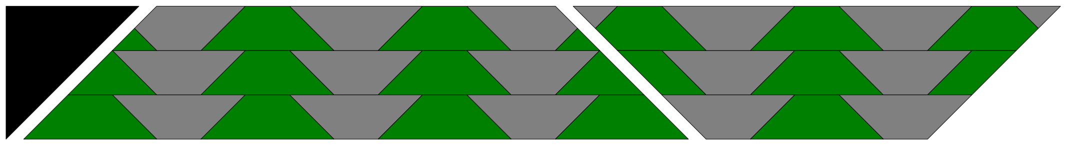
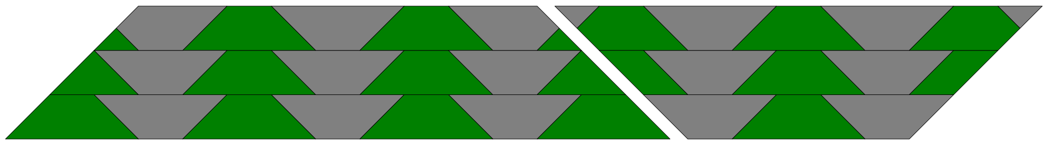


Figure 9.4: Cross section of a per-node upright tile in the slowest varying dimension of a two dimensional problem. Nested split tiling is applied on the fastest varying dimension such that only one upright and one inverted tile are present per node; nested split tiling is again performed for intra-node parallelism and locality.

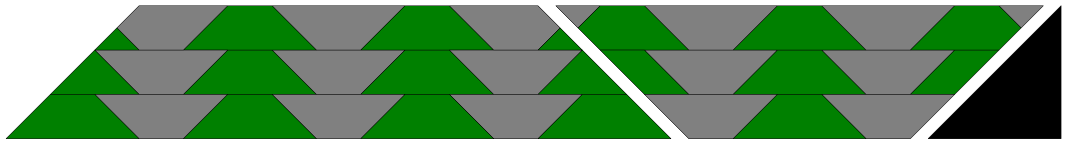
The second level of tiling enables intra-node parallelization and cache locality. This is shown in Figure 9.4 for upright and inverted tiles extending along the fastest varying dimension of a two dimensional problem. In this level, the tiles produced by first level split tiling are recursively split tiled. We want to achieve locality and load balanced parallelism across cores, so we do not restrict the number of tiles to a single upright and inverted tile as in the first level; instead we attempt to size tiles such that they are cache resident. The global DLT layout transformation enables fine gain SIMD parallelism, giving an inner loop with no stream alignment conflicts.



(a) Lower bound nodes



(b) Steady state nodes



(c) Upper bound nodes

Figure 9.5: Cross sections of multilevel boundary and steady state tiles for distributed memory nested used in split tiling. Partial tiles on boundary nodes are shown in black.

Nodes at the boundaries of the Cartesian grid require an extra partial tile to be computed; in the fastest varying dimension these tiles will contain DLT boundary conditions. These partial tiles are shown in Figure 9.5(a) and Figure 9.5(c). After completing an outer level split tile on any dimension a node must send values to its neighbor, in that dimension, adjacent to the computed tile. Because of this, global synchronization is not required and lower overhead point-to-point synchronization can be used instead.

CHAPTER 10

Conclusion

This work has developed a data layout transformation coupled and two different methods of tiling to substantially increase the performance of stencil computations with respect to the state-of-the-art on modern short-vector SIMD architectures. These optimizations are made available to domain specialists through the SDSL programming language, allowing performance gains to be achieved with substantially less effort than hand-coding the optimizations described in this work.

APPENDIX A

SDSL Implementation Limitations

- Only one `grid` may be defined in an SDSL program
- `griddata` may only be defined on timesteps 0 and 1
- `iterate` and `check ... every` trip counts must be integer literals
- Parameters, constants, grids, griddata, point function parameters, point function variables, and reduction variables all share the same namespace thus identifiers must be unique. For example, you cannot declare a constant `float pi = 3.14f` and in a point function definition declare a local variable `float pi = 3.1415f`

APPENDIX B

SDSL ANTLR Grammar

SDSL ANTLR Grammar:

```
grammar SDSL;

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
// Parser rules
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
sdsl: decl_list;


////////////////////////////////////////////////////////////////////////////////
// Lists
////////////////////////////////////////////////////////////////////////////////
decl_list: decl*;

time_list: expr time_list_elem*;
time_list_elem: COMMA expr;

grid_data_list: ID grid_data_list_elem*;
grid_data_list_elem: COMMA ID;

arg_list: expr arg_list_elem*;
arg_list_elem: COMMA expr;

iter_list: computable iter_list_elem*;
iter_list_elem: computable;
computable: stencil_def
          | reduce_def;


////////////////////////////////////////////////////////////////////////////////
// Declarations and definitions
////////////////////////////////////////////////////////////////////////////////
```

```
decl: param_decl
    | const_def
    | grid_def
    | grid_data_def
    | pf_def
    | iter_def;


////////////////////////////////////////////////////////////////////////////////
// Parameters / constants
////////////////////////////////////////////////////////////////////////////////
param_decl: type ID SEMI;
const_def: type ID ASSIGN expr SEMI;

type: I_TYPE
    | L_TYPE
    | F_TYPE
    | D_TYPE;


////////////////////////////////////////////////////////////////////////////////
// Grid definition
////////////////////////////////////////////////////////////////////////////////
grid_def: GRID ID const_region SEMI;


////////////////////////////////////////////////////////////////////////////////
// Grid Data definition
////////////////////////////////////////////////////////////////////////////////
grid_data_def: type GRID_DATA name=ID ON grid=ID (AT time_list)? SEMI;


////////////////////////////////////////////////////////////////////////////////
// Point Function definition
////////////////////////////////////////////////////////////////////////////////
pf_def: POINT_FUNC ID OPAREN grid_data_list? CPAREN OBRACE pf_stmt+ CBRACE;


////////////////////////////////////////////////////////////////////////////////
// Iteration definition
////////////////////////////////////////////////////////////////////////////////
iter_def: ITERATE INT OBRACE iter_list CBRACE
          (CHECK OPAREN expr CPAREN EVERY INT ITERATIONS)?;


////////////////////////////////////////////////////////////////////////////////
// Stencil definition
////////////////////////////////////////////////////////////////////////////////
stencil_def: STENCIL ID OBRACE stencil_stmt+ CBRACE;


```

```
/////////////////////////////////////////////////////////////////////////////////
// Reduction definition
/////////////////////////////////////////////////////////////////////////////////
reduce_def: REDUCTION ID assoc_op OBRACE reduce_stmt+ CBRACE;

assoc_op: PLUS
        | TIMES
        | MAX
        | MIN;


/////////////////////////////////////////////////////////////////////////////////
// Region
/////////////////////////////////////////////////////////////////////////////////
region: subregion+;
const_region: const_subregion+;

subregion: const_subregion
         | relative_subregion;

const_subregion   : OBRACK expr (COLON expr)? CBRACK;
relative_subregion: OBRACE expr CBRACE;


/////////////////////////////////////////////////////////////////////////////////
// Statements
/////////////////////////////////////////////////////////////////////////////////
pf_stmt: local_var_decl
       | assign_stmt;

stencil_stmt: region COLON assign_stmt
            | region COLON pf_call_stmt;

reduce_stmt: region COLON expr SEMI;

assign_stmt: ID ASSIGN expr SEMI
           | gd_ref ASSIGN expr SEMI;

pf_call_stmt: pf_call SEMI;

local_var_decl: type ID SEMI
              | type ID ASSIGN expr SEMI;


/////////////////////////////////////////////////////////////////////////////////
// Expression
/////////////////////////////////////////////////////////////////////////////////
add_op: PLUS
      | MINUS;

mul_op: TIMES
```

```
      | DIV
      | MOD;

pwr_op: PWR;

lgt_op: GT
      | LT
      | GTE
      | LTE;

enq_op: EQ
      | NE;

expr: enq_expr;

enq_expr: lgt_expr (enq_op^ lgt_expr)*;
lgt_expr: add_expr (lgt_op^ add_expr)*;
add_expr: mul_expr (add_op^ mul_expr)*;
mul_expr: una_expr (mul_op^ una_expr)*;

una_expr: PLUS una_expr
        | MINUS una_expr
        | una_expr_not_pm;

una_expr_not_pm: primary;

primary: OPAREN expr CPAREN
       | INT
       | LONG
       | FLOAT
       | DOUBLE
       | ID
       | pf_call
       | f_call
       | gd_ref;

pf_call: ID OPAREN grid_data_list CPAREN;
f_call : ID OPAREN arg_list CPAREN;


////////////////////////////////////////////////////////////////////////////////
// References
////////////////////////////////////////////////////////////////////////////////
gd_ref: (time_ref)? ID space_ref+;

time_ref: OBRACK expr CBRACK;

space_ref: const_ref
         | relative_ref;


```

```
const_ref: OBRACE expr CBRACE;
relative_ref : OBRACK expr CBRACK;


////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
// Lexer rules
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
I_TYPE : 'int';
L_TYPE : 'long';
F_TYPE : 'float';
D_TYPE : 'double';

ASSIGN : '=';
PLUS   : '+';
MINUS  : '-';
TIMES  : '*';
DIV    : '/';
MOD    : '%';
PWR    : '^';
MAX    : 'max';
MIN    : 'min';

GT  : '>';
LT  : '<';
GTE : '>=';
LTE : '<=';
EQ  : '==';
NE  : '!=';

SEMI  : '\;';
COMMA : ',';
COLON : ':';
ON    : 'on';
AT    : 'at';

OBRACK : '[';
CBRACK : ']';
OBRACE : '{';
CBRACE : '}';
OPAREN : '(';
CPAREN : ')';

GRID       : 'grid';
GRID_DATA  : 'griddata';
POINT_FUNC : 'pointfunction';
STENCIL    : 'stencil';
REDUCTION  : 'reduction';
ITERATE    : 'iterate';
CHECK      : 'check';
```

```
EVERY      : ’every’;
ITERATIONS : ’iterations’;

ID : (’a’..’z’|’A’..’Z’|’_’) (’a’..’z’|’A’..’Z’|’0’..’9’|’_’)*;

INT    : ’0’..’9’+;
LONG   : ’0’..’9’+ ’l’;
FLOAT  : (’0’..’9’)+ ’.’ (’0’..’9’)* EXPONENT? ’f’
       | ’.’ (’0’..’9’)+ EXPONENT? ’f’
       | (’0’..’9’)+ EXPONENT ’f’;
DOUBLE : (’0’..’9’)+ ’.’ (’0’..’9’)* EXPONENT?
       | ’.’ (’0’..’9’)+ EXPONENT?
       | (’0’..’9’)+ EXPONENT;

COMMENT : ’//’ ~(’\n’|’\r’)* ’\r’? ’\n’ {$channel=HIDDEN;}
        | ’/*’ ( options {greedy=false;} : . )* ’*/’ {$channel=HIDDEN;};

WS : ( ’ ’
       | ’\t’
       | ’\r’
       | ’\n’
     ) {$channel=HIDDEN;};

STRING : ’”’ ( ESC_SEQ | ~(’\\’|’”’) )* ’”’;

fragment
EXPONENT : (’e’|’E’) (’+’|’-’)? (’0’..’9’)+;

fragment
HEX_DIGIT : (’0’..’9’|’a’..’f’|’A’..’F’);

fragment
ESC_SEQ : ’\\’ (’b’|’t’|’n’|’f’|’r’|’\”’|’\”|’\\’)
        | UNICODE_ESC
        | OCTAL_ESC;

fragment
OCTAL_ESC : ’\\’ (’0’..’3’) (’0’..’7’) (’0’..’7’)
          | ’\\’ (’0’..’7’) (’0’..’7’)
          | ’\\’ (’0’..’7’);

fragment
UNICODE_ESC : ’\\’ ’u’ HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT;
```

BIBLIOGRAPHY

[1] Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[2] Allen, J. R., and Kennedy, K. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[3] Allen, R., and Kennedy, K. Automatic translation of Fortran programs to vector form. *Transactions on Programming Languages and Systems (TOPLAS) 9*, 4 (Oct. 1987), 491–542.

[4] Augustin, W., Heuveline, V., and Weiss, J.-P. Optimized stencil computation using in-place calculation on modern multicore systems. In *International Euro-Par Conference on Parallel Processing (Euro-Par)* (Delft, The Netherlands, 2009), Springer-Verlag, pp. 772–784.

[5] Bandishti, V., Pananilath, I., and Bondhugula, U. Tiling stencil computations to maximize parallelism. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)* (Salt Lake City, UT, USA, 2012), IEEE Computer Society Press, pp. 40:1–40:11.

[6] Barrett, R. F., Roth, P. C., and Poole, S. W. Finite difference stencils implemented using Chapel. Tech. Rep. TM-2007/119, Oak Ridge National Laboratory, Oak Ridge, TN, USA, 2007.

[7] Christen, M., Schenk, O., and Burkhart, H. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)* (Phoenix, AZ, USA, May 2011), IEEE Computer Society Press, pp. 676–687.

[8] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, 2001.

[9] Dantzig, G. B., Blattner, W., and Rao, M. Finding a cycle in a graph with minimum cost to time ratio with application to a ship routing problem. Tech. Rep. 66-1, Stanford University, Stanford, CA, USA, Nov. 1966.

[10] Datta, K., Kamil, S., Williams, S., Oliker, L., Shalf, J., and Yelick, K. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Review 51*, 1 (Feb. 2009), 129–159.

[11] Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., and Yelick, K. Stencil computation optimization and autotuning on state-of-the-art multicore architectures. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)* (Austin, TX, USA, 2008), IEEE Computer Society Press, pp. 4:1–4:12.

[12] Datta, K., Williams, S., Volkov, V., Carter, J., Oliker, L., Shalf, J., and Yelick, K. Auto-tuning the 27-point stencil for multicore. In *International Workshop on Automatic Performance Tuning (iWAPT)* (Tokyo, Japan, 2009), Springer-Verlag.

[13] De La Cruz, R., Araya-Polo, M., and Cela, J. M. Introducing the semi-stencil algorithm. In *International Conference on Parallel Processing and Applied Mathematics (PPAM)* (Wroclaw, Poland, 2009), Springer-Verlag, pp. 496–506.

[14] Deitz, S. J., Chamberlain, B. L., and Snyder, L. Eliminating redundancies in sum-of-product array computations. In *International Conference on Supercomputing (ICS)* (Sorrento, Naples, Italy, 2001), ACM Press, pp. 65–77.

[15] Dursun, H., Nomura, K., Wang, W., Kunaseth, M., Peng, L., Seymour, R., Kalia, R. K., Nakano, A., and Vashishta, P. In-core optimization of high-order stencil computations. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)* (Las Vegas, NV, USA, 2009), H. R. Arabnia, Ed., CSREA Press, pp. 533–538.

[16] Dursun, H., Nomura, K.-I., Peng, L., Seymour, R., Wang, W., Kalia, R. K., Nakano, A., and Vashishta, P. A multilevel parallelization framework for high-order stencil computations. In *International Euro-Par Conference on Parallel Processing (Euro-Par)* (Delft, The Netherlands, 2009), Springer-Verlag, pp. 642–653.

[17] Eichenberger, A., Wu, P., and O'Brien, K. Vectorization for SIMD architectures with alignment constraints. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Washington, DC, USA, 2004), ACM

Press, pp. 82–93.

[18] Fireman, L., Petrank, E., and Zaks, A. New algorithms for SIMD alignment. In *International Conference on Compiler Construction (CC)* (Braga, Portugal, 2007), Springer-Verlag, pp. 1–15.

[19] Frigo, M., and Strumpen, V. Cache oblivious stencil computations. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)* (Seattle, WA, USA, 2005), ACM Press, pp. 361–366.

[20] Frigo, M., and Strumpen, V. The memory behavior of cache oblivious stencil computations. *The Journal of Supercomputing 39*, 2 (2007), 93–112.

[21] Getreuer, P. riciandenoise: 2D and 3D total variation based Rician denoising. `http://cdsc-image-processing-pipeline.googlecode.com/files/riciandenoise.pdf`, Sept. 2009. Retrieved July 31, 2013.

[22] Grosser, T., Cohen, A., Holewinski, J., Sadayappan, P., and Verdoolaege, S. Hybrid hexagonal/classical tiling for gpus. In *International Symposium on Code Generation and Optimization (CGO)* (Orlando, FL, USA, 2014), ACM Press, p. 66.

[23] Grosser, T., Cohen, A., Kelly, P. H., Ramanujam, J., Sadayappan, P., and Verdoolaege, S. Split tiling for GPUs: Automatic parallelization using trapezoidal tiles. In *Workshop on General Purpose Processing Using Graphics Processing Units (GPGPU)* (Houston, TX, USA, 2013), ACM Press, pp. 24–31.

[24] Guo, J., Bikshandi, G., Fraguela, B. B., and Padua, D. Writing productive stencil codes with overlapped tiling. *Concurrency and Computation: Practice and Experience 21*, 1 (Jan. 2009), 25–39.

[25] Han, D., Xu, S., Chen, L., and Huang, L. PADS: A pattern-driven stencil compiler-based tool for reuse of optimizations on GPGPUs. In *International Conference Parallel and Distributed Systems (ICPADS)* (Tainan, Taiwan, 2011), IEEE Computer Society Press, pp. 308–315.

[26] Henretty, T., Stock, K., Pouchet, L.-N., Franchetti, F., Ramanujam, J., and Sadayappan, P. Data layout transformation for stencil computations on short-vector SIMD architectures. In *International Conference on Compiler Construction (CC)* (Saarbrücken, Germany, 2011), Springer-Verlag, pp. 225–245.

[27] Holewinski, J., Pouchet, L.-N., and Sadayappan, P. High-performance code generation for stencil computations on GPU architectures. In *International Conference on Supercomputing (ICS)* (Venice, Italy, 2012), ACM Press, pp. 311–320.

[28] Intel Corporation. Intel Cilk Plus. `https://software.intel.com/en-us/intel-cilk-plus`. Retrieved August 9, 2014.

[29] Intel Corporation. Intel Xeon Phi. `http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html`. Retrieved July 31, 2013.

[30] Jaeger, J., and Barthou, D. Automatic efficient data layout for multithreaded stencil codes on cpus and gpus. In *International Conference on High Performance Computing (HiPC)* (2012), IEEE Computer Society Press, pp. 1–10.

[31] Kamil, S., Datta, K., Williams, S., Oliker, L., Shalf, J., and Yelick, K. Implicit and explicit optimizations for stencil computations. In *Workshop on Memory*

System Performance and Correctness (MSPC) (San Jose, CA, USA, 2006), ACM Press, pp. 51–60.

[32] Kamil, S., Husbands, P., Oliker, L., Shalf, J., and Yelick, K. Impact of modern memory subsystems on cache optimizations for stencil computations. In *Workshop on Memory System Performance (MSP)* (Chicago, IL, USA, 2005), ACM Press, pp. 36–43.

[33] Kong, M., Veras, R., Stock, K., Franchetti, F., Pouchet, L.-N., and Sadayappan, P. When polyhedral transformations meet simd code generation. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Seattle, Washington, USA, 2013), ACM Press, pp. 127–138.

[34] Krishnamoorthy, S., Baskaran, M., Bondhugula, U., Ramanujam, J., Rountev, A., and Sadayappan, P. Effective automatic parallelization of stencil computations. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (San Diego, CA, USA, 2007), ACM Press, pp. 235–244.

[35] Larsen, S., and Amarasinghe, S. P. Exploiting superword level parallelism with multimedia instruction sets. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Vancouver, BC, Canada, 2000), ACM Press, pp. 145–156.

[36] Li, Z., and Song, Y. Automatic tiling of iterative stencil loops. *Transactions on Programming Laguages and Systems (TOPLAS) 26*, 6 (Nov. 2004), 975–1028.

[37] Mars, R. L., Wilson, A. G., and Choptuik, M. W. SNPL reference manual. http://laplace.physics.ubc.ca/People/agwilson/SNPL/snplref2.pdf, July 2006. Retrieved July 31, 2013.

[38] Marsa, R., and Choptuik, M. The RNPL user's guide. http://laplace.physics.ubc.ca/People/arman/files/RNPL_ref.pdf, May 1995. Retrieved July 31, 2013.

[39] Mccalpin, J., and Wonnacott, D. Time skewing: A value-based approach to optimizing for memory locality. Tech. Rep. DCS-TR-379, Rutgers University, New Bruswick, NJ, USA, 1999.

[40] Meng, J., and Skadron, K. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In *International Conference on Supercomputing (ICS)* (Yorktown Heights, NY, USA, 2009), ACM Press, pp. 256–265.

[41] Micikevicius, P. 3D finite difference computation on GPUs using CUDA. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)* (Washington, DC, USA, 2009), ACM Press, pp. 79–84.

[42] Orozco, D., and Gao, G. R. Mapping the FDTD application to many-core chip architectures. In *International Conference on Parallel Processing (ICPP)* (Vienna, Austria, 2009), IEEE Computer Society, pp. 309–316.

[43] Pouchet, L.-N. PoCC. http://pocc.sourceforge.net. Retrieved August 20, 2014.

[44] Pouchet, L.-N. PolyOpt/C. http://hpcrl.cse.ohio-state.edu/downloads/polyopt/doc/polyopt-c-0.2.1.pdf. Retrieved August 20, 2014.

[45] Prokop, H. Cache-oblivious algorithms. Master's thesis, Massachusetts Institute of Technology, 1999.

[46] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA, 2013), ACM Press, pp. 519–530.

[47] Solar-Lezama, A., Arnold, G., Tancau, L., Bodik, R., Saraswat, V., and Seshia, S. Sketching stencils. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (San Diego, CA, USA, 2007), ACM Press, pp. 167–178.

[48] Song, Y., and Li, Z. New tiling techniques to improve cache temporal locality. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Atlanta, Georgia, USA, 1999), ACM Press, pp. 215–228.

[49] Stock, K., Kong, M., Grosser, T., Pouchet, L.-N., Rastello, F., Ramanujam, J., and Sadayappan, P. A framework for enhancing data reuse via associative reordering. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Edinburgh, UK, 2014), ACM Press, ACM Press, p. 10.

[50] Strzodka, R., Shaheen, M., Pajak, D., and Seidel, H.-P. Cache oblivious parallelograms in iterative stencil computations. In *International Conference on Supercomputing (ICS)* (Tsukuba, Japan, 2010), ACM Press, pp. 49–59.

[51] Sung, I.-J., Stratton, J. A., and Hwu, W.-M. W. Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)* (Vienna, Austria, 2010), ACM Press, pp. 513–522.

[52] Tang, Y., Chowdhury, R. A., Kuszmaul, B. C., Luk, C.-K., and Leiserson, C. E. The pochoir stencil compiler. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)* (San Jose, CA, USA, 2011), pp. 117–128.

[53] Treibig, J., Wellein, G., and Hager, G. Efficient multicore-aware parallelization strategies for iterative stencil computations. *Journal of Computational Science 2*, 2 (May 2011), 130–137.

[54] Vasilache, N., Meister, B., Baskaran, M., and Lethin, R. Joint scheduling and layout optimization to enable multi-level vectorization. In *Internationol Workshop on Polyhedral Compiler Techniques (IMPACT)* (Paris, France, Jan. 2012).

[55] Venkatasubramanian, S., and Vuduc, R. W. Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems. In *International Conference on Supercomputing (ICS)* (Yorktown Heights, NY, USA, 2009), ACM Press, pp. 244–255.

[56] Wellein, G., Hager, G., Zeiser, T., Wittmann, M., and Fehske, H. Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. In *International Computer Software and Applications Conference (COMPSAC)* (Seattle, WA, USA, 2009), IEEE Computer Society, pp. 579–586.

[57] Wittmann, M., Hager, G., Treibig, J., and Wellein, G. Leveraging shared caches for parallel temporal blocking of stencil codes on multicore processors and clusters. *Parallel Processing Letters 20*, 4 (Dec. 2010), 359–376.

[58] Wolf, M. E., and Lam, M. S. A data locality optimizing algorithm. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Toronto, ON, Canada, 1991), ACM Press, pp. 30–44.

[59] Wonnacott, D. Achieving scalable locality with time skewing. *International Journal of Parallel Programming (IJPP) 30*, 3 (June 2002), 181–221.

[60] Wu, P., Eichenberger, A. E., and Wang, A. Efficient SIMD code generation for runtime alignment and length conversion. In *International Symposium on Code Generation and Optimization (CGO)* (San Jose, CA, USA, 2005), IEEE Computer Society, pp. 153–164.