Effective Automatic Parallelization and Locality Optimization Using The Polyhedral Model

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the

Graduate School of The Ohio State University

By

Uday Kumar Reddy Bondhugula, *B-Tech, M.S.*

* * * * *

The Ohio State University

2008

ABSTRACT

Multicore processors have now become mainstream. The difficulty of programming these architectures to effectively tap the potential of multiple processing units is well-known. Among several ways of addressing this issue, one of the very promising and simultaneously hard approaches is automatic parallelization. This approach does not require any effort on part of the programmer in the process of parallelizing a program.

The Polyhedral model for compiler optimization is a powerful mathematical framework based on parametric linear algebra and integer linear programming. It provides an abstraction to represent nested loop computation and its data dependences using integer points in polyhedra. Complex execution-reordering, that can improve performance by parallelization as well as locality enhancement, is captured by affine transformations in the polyhedral model. With several recent advances, the polyhedral model has reached a level of maturity in various aspects – in particular, as a powerful intermediate representation for performing transformations, and code generation after transformations. However, an approach to automatically find good transformations for communication-optimized coarse-grained parallelization together with locality optimization has been a key missing link. This dissertation presents a new automatic transformation framework that solves the above problem. Our approach works by finding good affine transformations through a powerful and practical linear cost function that enables efficient tiling and fusion of sequences of arbitrarily nested loops.

This in turn allows simultaneous optimization for coarse-grained parallelism and locality. Synchronization-free parallelism and pipelined parallelism at various levels can be extracted. The framework can be targeted to different parallel architectures, like general-purpose multicores, the Cell processor, GPUs, or embedded multiprocessor SoCs.

The proposed framework has been implemented into a new end-to-end transformation tool, PLUTO, that can automatically generate parallel code from regular C program sections. Experimental results from the implemented system show significant performance improvement for single core and multicore execution over state-of-the-art research compiler frameworks as well as the best native production compilers. For several dense linear algebra kernels, code generated from Pluto beats, by a significant margin, the same kernels implemented with sequences of calls to highly-tuned libraries supplied by vendors. The system also allows empirical optimization to be performed in a much wider context than has been attempted previously. In addition, Pluto can serve as the parallel code generation backend for several high-level domain-specific languages.

ACKNOWLEDGMENTS

I am thankful to my adviser, P. Sadayappan, for always being very keen and enthusiastic to discuss new ideas and provide valuable feedback. He also provided me a great amount of freedom in the direction I wanted to take and this proved to be very productive for me. Nasko's (Atanas Rountev) determination in getting into the depths of a problem had a very positive influence and was helpful in making some key progresses in the theory surrounding this dissertation. In addition, I was fortunate to have Ram's (J. Ramanujam) advice from time to time that complemented everything else well. Interaction and discussions with Saday, Nasko, and Ram were very friendly and entertaining. I would also like to thank Gagan (Prof. Gagan Agrawal) for being a very helpful and accessible Graduate Studies Committee chair and agreeing to serve on my dissertation committee at short notice.

Thanks to labmates Muthu Baskaran, Jim Dinan, Tom Henretty, Brian Larkins, Kamrul, Gaurav Khanna, Sriram Krishnamoorthy, Qingda Lu, and Rajkiran Panuganti for all the interesting conversations during my stay at 474 Dreese Labs: they will be remembered. I am also thankful to the researchers at the Ohio Supercomputer Center, Ananth Devulapalli, Joseph Fernando, and Pete Wyckoff, with whom I worked during the initial stages of my Ph.D., and they gave me a good start.

Many thanks to Louis-Noël Pouchet and Albert Cohen for hosting a very pleasant visit at INRIA Saclay. Good friendships have evolved with them that will last for a long time. I am also thankful to the group of Kathryn O'Brien at the IBM TJ Watson Research Center at Yorktown Heights – its group members Alex Eichenberger and Lakshminarayanan Renganarayana, as well as Kevin O'Brien and Dan Prener for a memorable experience during my internship there, and a mutually beneficial association. Thanks also to Nicolas Vasilache for some discussion from time to time.

I have to acknowledge the contributions of the developers of CLooG, LooPo, and PipLib, without which the theory this dissertation proposes would not have translated into a complete end-to-end system. In particular, I would like to acknowledge Cédric Bastoul for having written Cloog, and Martin Griebl and the LooPo team for creating an excellent polyhedral infrastructure over the years. I just had all the necessary components to build and complete the missing links. I hope the results presented in the thesis vindicate the efforts put into these libraries and tools both from a research and an engineering standpoint. I would also like to acknowledge the anonymous reviewers of our papers published at the International Conference on Compiler Construction 2008, and the ACM SIGPLAN conference on Programming Languages Design and Implementation 2008, for several constructive comments.

I conclude by acknowledging my parents who deserve the maximum credit for this work. My father, CDR. B. Sudhakar Reddy, and my mother, Nirmala, took extraordinary interest in my education since my childhood. I am glad their efforts have been fruitful at every stage of my studies.

VITA

Sep 10, 1982	Born - India
Jul 2000 – Jul 2004	Bachelor of Technology Computer Science and Engineering Indian Institute of Technology, Madras
Sep 2004 – Aug 2008	Graduate Research Associate Computer Science and Engineering The Ohio State University
Jun 2007 – Sep 2007	Research Intern Advanced Compilation Technologies IBM TJ Watson Research Labs, Yorktown Heights, New York
Mar 2008 – Apr 2008	Visiting Researcher ALCHEMY team INRIA Saclay, Île-de-France

PUBLICATIONS

Uday Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *ACM SIGPLAN Programming Language Design and Implementation*, Jun 2008.

Uday Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International Conference on Compiler Construction*, Apr. 2008.

M. Baskaran, Uday Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs. In *ACM International Conference on Supercomputing (ICS)*, June 2008.

M. Baskaran, Uday Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic Data Movement and Computation Mapping for Multi-level Parallel Architectures with Explicitly Managed Memories. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb 2008.

Uday Bondhugula, J. Ramanujam, and P. Sadayappan. Automatic mapping of nested loops to FPGAs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Mar 2007.

S. Krishnamoorthy, M. Baskaran, Uday Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun 2007.

Uday Bondhugula, A. Devulapalli, J. Dinan, J. Fernando, P. Wyckoff, E. Stahlberg, and P. Sadayappan. Hardware/software integration for FPGA-based all-pairs shortest-paths. In *IEEE Field Programmable Custom Computing Machines*, Apr 2006.

Uday Bondhugula, A. Devulapalli, J. Fernando, P. Wyckoff, and P. Sadayappan. Parallel FPGA-based all-pairs shortest-paths in a directed graph. In *IEEE International Parallel and Distributed Processing Symposium*, Apr 2006.

S. Sur, Uday Bondhugula, A. Mamidala, H.-W. Jin, and D. K. Panda. High performance RDMA-based all-to-all broadcast for InfiniBand clusters. In *12th IEEE International Conference on High Performance Computing*, Dec 2005.

FIELDS OF STUDY

Major Field: Computer Science and Engineering

Emphasis: Automatic parallelization, Parallelizing compilers, Polyhedral model

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

CHAPTER 1

INTRODUCTION

Current trends in microarchitecture are towards larger number of processing elements on a single chip. Till the early 2000s, increasing the clock frequency of microprocessors gave most software a performance boost without any additional effort on part of the programmers, or much effort on part of compiler or language designers. However, due to power dissipation issues, it is now no longer possible to increase clock frequencies the same way it had been (Figure 1.1). Increasing the number of cores on the chip has currently become the way to use the ever increasing number of transistors available, while keeping power dissipation in control. Multi-core processors appear in all realms of computing – high performance supercomputers built out of commodity processors, accelerators like the Cell processor and general-purpose GPUs and multi-core desktops. Besides mainstream and high-end computing, the realm of embedding computing cannot be overlooked. Multiprocessor System-on-Chip (MP-SoCs) are ubiquitous in the embedding computing domain for multimedia and image processing.

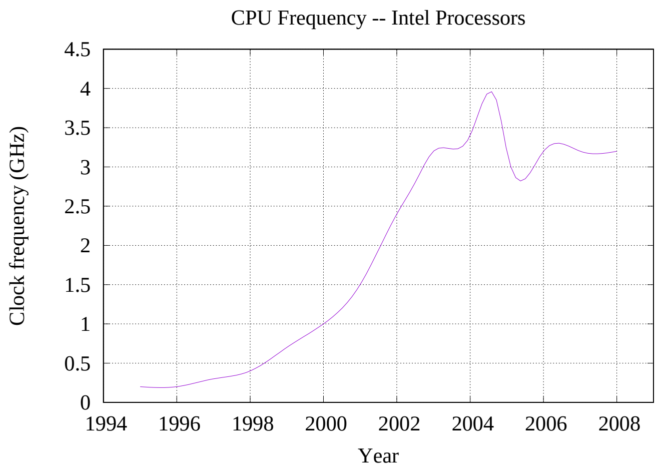


Figure 1.1: Clock frequency of Intel processors in the past decade

End of the ILP era. Dynamic out-of-order execution, superscalar processing, multiple way instruction issuing, hardware dynamic branch prediction and speculative execution, non-blocking caches, were all techniques to increase instruction-level parallelism (ILP), and thus improve performance for "free", i.e., transparent to programmers and without much assistance from compilers or programming languages. Compilers were taken for "granted" and programmers would mainly expect correct code generated for them. In general, compiler optimizations have had a narrow audience [Rob01]. Most of the credit for improved performance would go to the architecture, or to the language designer for higher productivity. However, currently, efforts to obtain more ILP have reached a dead-end with no more ILP left to be extracted. This is mainly because ILP works well within a window of instructions of bounded length. Extracting a much more coarser granularity of parallelism, thread-level parallelism, requires looking across instructions that are possibly thousands or millions of instructions away in the dynamic length of a program. The information that they can be executed in parallel is obviously something that hardware does not have, or is at least not feasible to be done by hardware alone at runtime. Analyses and transformation that can achieve the above can only be performed at a high level – by a software system like the language translator or a compiler.

Computers with multiple processors have a very long history. Special-purpose parallel computers like massively parallel processors (MPPs), systolic arrays, or vector supercomputers were the dominant form of computers or supercomputers up till the early 1990s. However, all of these architectures slowly disappeared with the arrival of single-threaded superscalar microprocessors. These microprocessors continuously delivered ever increasing performance without requiring any advanced parallel compiler technology. As mentioned before, this was due to techniques to obtain more and more ILP and increasing clock speed keeping up with Moore's law. Even a majority of the supercomputers today are built out of commodity superscalar processors [Top]. Hence, due to the absence of parallelism in the mainstream for the past decade and a half, parallel compiler and programming technology stagnated. However, with parallelism and multi-core processors becoming mainstream, there is renewed interest and urgency in supporting parallel programming at various levels – languages, compilers, libraries, debuggers, and runtime systems. It is clear that hardware architects no longer have a solution by themselves.

Parallelism and Locality. Besides multiple cores on a chip, caches or faster on-chip memories and registers have been around for a long time and are likely to stay. Often, not optimizing for locality has a direct impact on parallelization. This is because of the fact that increasing the number of threads increases the amount of memory bandwidth needed for the computation: the amount of available memory bandwidth has always lagged computational power. Hence, optimizing for parallelism and locality are the central issues in improving performance.

Difficulty of parallel programming. The difficulty of programming multi-core architectures to effectively use multiple on-chip processing units is a significant challenge. This is due to several reasons. Writing programs with a single thread of logic is quite intuitive and natural, and so nearly all programmers have been used to writing sequential programs for decades. One can think of two broad approaches of transitioning from sequential programs to parallel ones: (1) proceed incrementally, i.e., get a basic sequential code working and then parallelize it, and (2) design and code a parallel program from the start itself. With the former approach, manually parallelizing an application written to be sequential can be quite tedious. In some cases, the parallelization may be non-trivial enough to be infeasible to be done by hand, or even detect manually. The second approach goes against one of the basic philosophies of programming which says, "pre-mature optimization is the root of all evil". Thinking about parallelism from the very start of the development process may make the path to obtain a correct working version of a program itself difficult. In any case, all of the manual techniques are often not productive for the programmer due to the need to synchronize access to data shared by threads.

Automatic Parallelization. Among several approaches to address the parallel programming problem, one that is very promising but simultaneously very challenging is automatic parallelization. Automatic parallelization is the process of automatically converting a sequential program to a version that can directly run on multiple processing elements without altering the semantics of the program. This process requires no effort on part of the programmer in parallelization and is therefore very attractive. Automatic parallelization is typically performed in a compiler, at a high level where most of the information needed is available. The output of an auto-parallelizer is a race-free deterministic program that obtains the same results as the original sequential program. This dissertation deals with compile-time automatic parallelization and primarily targets shared memory parallel architectures for which auto-parallelization is significantly easier. A common feature of all of the multicore architectures we have named so far is that they all have shared memory at one level or the other.

The Polyhedral Model. Many compute-intensive applications often spend most of their execution time in nested loops. This is particularly common in scientific and engineering applications. The polyhedral model provides a powerful abstraction to reason about transformations on such loop nests by viewing a dynamic instance (iteration) of each statement as an integer point in a well-defined space called the statement's *polyhedron*. With such a representation for each statement and a precise characterization of inter and intra-statement dependences, it is possible to reason about the correctness of complex loop transformations in a completely mathematical setting relying on machinery from linear algebra and integer linear programming. The transformations finally reflect in the generated code as reordered execution with improved cache locality and/or loops that have been parallelized. The polyhedral model is readily applicable to loop nests in which the data access functions and loop bounds are affine functions (linear function with a constant) of the enclosing loop variables and parameters. While a precise characterization of data dependences is feasible for programs with static control structure and affine references and loop-bounds, codes with non-affine array access functions or code with dynamic control can also be handled, but primarily with conservative assumptions on some dependences.

The task of program optimization (often for parallelism and locality) in the polyhedral model may be viewed in terms of three phases: (1) static dependence analysis of the input program, (2) transformations in the polyhedral abstraction, and (3) generation of code for the transformed program. Significant advances were made in the past decade on dependence analysis [Fea91, Fea88, Pug92] and code generation [AI91, KPR95, GLW98] in the polyhedral model, but the approaches suffered from scalability challenges. Recent advances in dependence analysis and more importantly in code generation [QRW00, Bas04a, VBGC06, VBC06] have solved many of these problems resulting in the polyhedral techniques being applied to code representative of real applications like the spec2000fp benchmarks [CGP+05, GVB+06]. These advances have also made the polyhedral model practical in production compiler contexts [PCB$^+$06] as a flexible and powerful representation to compose and apply transformations. However, current state-of-the-art polyhedral implementations still apply transformations manually and significant time is spent by an expert to determine the best set of transformations that lead to improved performance [CGP$^+$05, GVB$^+$06]. Regarding the middle step, an important open issue is that of the choice of transformations from the huge space of valid transforms. Existing automatic transformation frameworks [LL98, LCL99, LLL01, AMP01, Gri04] have one or more drawbacks or restrictions in finding *good* transformations. All of them lack a practical and scalable cost model for effective coarse-grained parallel execution and locality as is used with manually developed parallel applications.

This dissertation describes a new approach to address this problem of automatically finding good transformations to simultaneously optimize for coarse-grained parallelism and locality. Our approach is driven by a powerful and practical linear cost function that goes beyond just maximizing the number of degrees of parallelism or minimizing the order of synchronization. The cost function allows finding good ways to tile and fuse across multiple statements coming from sequences of arbitrarily nested loops. The entire framework has been implemented into a tool, PLUTO, that can automatically generate OpenMP parallel code from regular C program sections. In this process, we also describe techniques to generate efficient tiled and parallel code, along with a number of other optimizations to achieve high performance on modern multicore architectures.

Experimental results from the implemented system show very high speedups for local and parallel execution on multicores over state-of-the-art research compiler frameworks as well as the best native production compilers. For several linear algebra kernels, code generated from Pluto beats, by a significant margin, the same kernels implemented with sequences of calls to hand-tuned BLAS libraries supplied by vendors. The system also leaves a lot of scope for further improvement in performance. Thanks to the mathematical abstraction provided by the polyhedral model, Pluto can also serve as the backend for parallel code generation with new domain-specific frontends.

The rest of this dissertation is organized as follows. Chapter 2 provides the mathematical background for the polyhedral model. Chapter 3 describes our automatic transformation framework. Chapter 4 is devoted to explaining how loop fusion is naturally handled in an integrated manner in our framework. Chapter 5 describes the implemented Pluto system along with details on parallel and tiled code generation and complementary post-processing. Chapter 6 provides an experimental evaluation of the framework. Conclusions and directions for future research are finally presented in Chapter 7. Most of the content in Chapters 3 and 5, and some results from Chapter 6 have been published in [BBK$^+$08c] and [BHRS08].

CHAPTER 2

BACKGROUND

In this chapter, we present a overview of the polyhedral model, and introduce notation used throughout the dissertation. The mathematical background on linear algebra and linear programming required to understand the theoretical aspects of this dissertation is fully covered in this chapter. A few fundamental concepts and definitions relating to cones, polyhedra, and linear inequalities have been omitted and they can be found in [Wil93] and [Sch86]. Detailed background on traditional loop transformations can be found in [Wol95, Ban93]. Overall, I expect the reader to find the background presented here self-contained. [Bas04b, Gri04] are among other sources for introduction to the polyhedral model.

All row vectors will be typeset in bold lowercase, while regular vectors are typeset with an overhead arrow. The set of all real numbers, the set of rational numbers, and the set of integers are represented by $\mathbb{R}$, $\mathbb{Q}$, and $\mathbb{Z}$, respectively.

2.1 Hyperplanes and Polyhedra

Definition 1 (**Linear**)**.** A k-dimensional function f is linear iff it can expressed in the following form:

$$linear \;\; function \quad f(\vec{v}) = M_f\vec{v} \tag{2.1}$$

where $\vec{v} = \begin{pmatrix} v_1 \\ \vdots \\ v_d \end{pmatrix}$ and $M_f \in \mathbb{R}^{k\times d}$ is a matrix with k rows and d columns. In our context, M_f is an integer matrix, i.e., $M_f \in \mathbb{Z}^{k\times d}$

Definition 2 (**Affine**)**.** A k-dimensional function f is affine iff it can expressed in the following form:

$$affine \;\; function \quad f(\vec{v}) = M_f\vec{v} + \vec{f_0} \tag{2.2}$$

where $\vec{v} = \begin{pmatrix} v_1 \\ \vdots \\ v_d \end{pmatrix}$ and $M_f \in \mathbb{R}^{k\times d}$ is a matrix with k rows and d columns, $f_0 \in \mathbb{R}^k$ is a k-dimensional vector. In all cases, we deal with affine functions with $M_f \in \mathbb{Z}^{k\times d}$ and $f_0 \in \mathbb{Z}^k$. The domain is also a set of integers: $\vec{v} \in \mathbb{Z}^d$.

Definition 3 (**Null space**)**.** The *null space* of an affine function $f(\vec{v}) = M_f\vec{v} + \vec{f_0}$ is $\left\{\vec{x} \mid f(\vec{x}) = \vec{0}\right\}$.

f is a one-to-one mapping iff M_f has full column rank, i.e., if it has as many linearly independent rows (and columns) as the number of its columns. In such a case, the null space is 0-dimensional, i.e., trivially the vector $\vec{0}$.

Definition 4 (Affine spaces). A set of vectors is an affine space iff it is closed under affine combination, i.e., if $\vec{x}, \vec{y}$ are in the space, all points lying on the line joining $\vec{x}$ and $\vec{y}$ belong to the space.

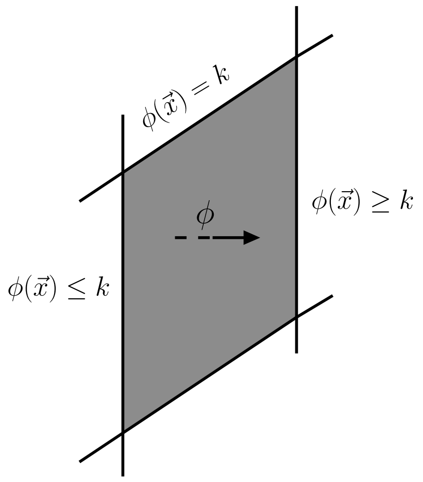
A line in a vector space of any dimensionality is a one-dimensional affine space. In 3-d space, any 2-d plane is an example of a 2-d affine sub-space. Note that 'affine function' as defined in (2.2) should not be confused with 'affine combination', though several researchers use the term affine combination in place of an affine function.

Definition 5 (Affine hyperplane). An affine hyperplane is an $n-1$ dimensional affine sub-space of an n dimensional space.

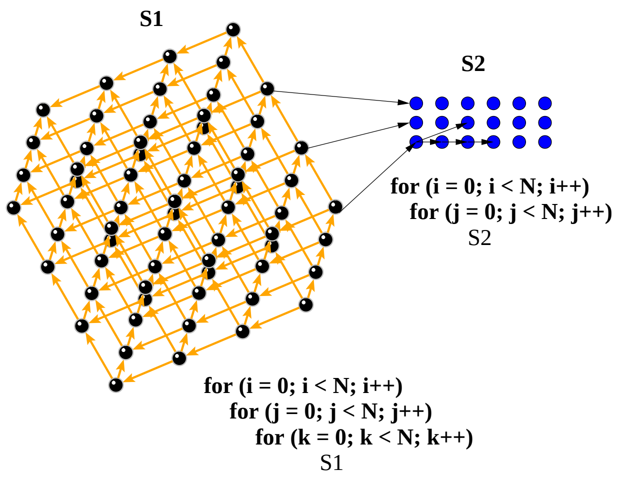
In our context, the set of all vectors $v \in \mathbb{Z}^n$ such that $\mathbf{h}.\vec{v} = k$, for $k \in \mathbb{Z}$, forms an affine hyperplane. The set of parallel *hyperplane instances* correspond to different values of k with the row vector $\mathbf{h}$ normal to the hyperplane. Two vectors $\vec{v_1}$ and $\vec{v_2}$ lie in the same hyperplane if $\mathbf{h}.\vec{v_1} = \mathbf{h}.\vec{v_2}$. An affine hyperplane can also be viewed as a one-dimensional affine function that maps an n-dimensional space onto a one-dimensional space, or partitions an n-dimensional space into $n-1$ dimensional slices. Hence, as a function, it can be written as:

$$\phi(\vec{v}) = \mathbf{h}.\vec{v} + c \tag{2.3}$$

Figure 2.1(a) shows a hyperplane geometrically. Throughout the dissertation, the hyperplane is often referred to by the row vector, $\mathbf{h}$, the vector normal to the hyperplane. A hyperplane $\mathbf{h}.\vec{v} = k$ divides the space into two *half-spaces*, the positive half-space,



(a) An affine hyperplane



(b) Polyhedra (courtesy: Loopo display tool)

Figure 2.1: Hyperplane and Polyhedron

$h.\vec{v} \geq k$, and a negative half-space, $\mathbf{h}.\vec{v} \leq k$. Each half-space can be represented by an affine inequality.

Definition 6 (**Polyhedron, Polytope**)**.** A *polyhedron* is an intersection of a finite number of half-spaces. A *polytope* is a bounded polyhedron.

Each of the half-spaces provides a face to the polyhedron. Hence, the set of affine inequalities, each representing a face, can be used to compactly represent the polyhedron. If there are m inequalities, then the polyhedron is

$$\left\{\vec{x} \in \mathbb{R}^n \mid A\vec{x} + \vec{b} \geq \vec{0}\right\}$$

where $A \in \mathbb{R}^{m\times n}$ and $\vec{b} \in \mathbb{R}^m$.

A polyhedron also has an alternate dual representation in terms of vertices, rays, and lines, and algorithms like the Chernikova algorithm [LeV92] exist to move from the face representation to the vertex one. Polylib [Pol] and PPL [BHZ] are two libraries that provide a range of functions to perform various operations on polyhedra and use the dual representation internally.

In our context, we are always interested in the integer points inside a polyhedron since loop iterators typically have integer data types and traverse an integer space. The matrix A and $\vec{b}$ for problems we will deal with also comprise only integers. So, we always have:

$$\left\{\vec{x} \in \mathbb{Z}^n \mid A\vec{x} + \vec{b} \geq \vec{0}\right\} \tag{2.4}$$

where $A \in \mathbb{Z}^{m \times n}$ and $\vec{b} \in \mathbb{Z}^m$.

Lemma 1 (**Affine form of the Farkas lemma**). *Let $\mathcal{D}$ be a non-empty polyhedron defined by p affine inequalities or faces*

$$\mathbf{a_k}.\vec{x} + b_k \geq 0, \quad k = 1, p$$

then, an affine form ψ is non-negative everywhere in $\mathcal{D}$ iff it is a non-negative linear combination of the faces:

$$\psi(\vec{x}) \equiv \lambda_0 + \sum_{k=1}^{p} \lambda_k \left(\mathbf{a_k}\vec{x} + b_k\right), \quad \lambda_0, \lambda_1, \ldots, \lambda_p \geq 0 \tag{2.5}$$

The non-negative constants λ_k are referred to as the Farkas multipliers. Proof of the *if* part is obvious. For the *only if* part, see Schrijver [Sch86]. We provide the main idea here roughly. The polyhedron $\mathcal{D}$ lies in the non-negative half-space of the hyperplane $\psi(\vec{x})$. This makes sure that λ_0 has to be non-negative if the hyperplane is pushed close enough to the polytope so that it touches a vertex of the polyhedron first without cutting the polyhedron. If a hyperplane passes through a vertex of the polyhedron and with the entire polyhedron in its non-negative half-space, the fact that it can be expressed as a non-negative linear combination of the faces of the polyhedron directly follows from the Fundamental Theorem of Linear Inequalities [Sch86].

Definition 7 (**Perfect loop nest, Imperfect loop nest**)**.** A set of nested loops is called a *perfect loop nest* iff all statements appearing in the nest appear inside the body of the innermost loop. Otherwise, the loop nest is called an *imperfect loop nest*. Figure 2.6 shows an imperfect loop nest.

Definition 8 (**Affine loop nest**)**.** *Affine loop nests* are sequences of imperfectly nested loops with loop bounds and array accesses that are affine functions of outer loop variables and program parameters.

Program parameters or structure parameters are symbolic constants that appear in loop bounds or access functions. They very often represent the problem size. N is a program parameter in Figure 2.1(b), while in Figure 2.2, N and β are the program parameters.

2.2 The Polyhedral Model

The polyhedral model is a geometrical as well as a linear algebraic framework for capturing the execution of a program in a compact form for analysis and transformation. The compact representation is primarily of the dynamic instances of statements of a program surrounded by loops in a program, the dependences between such statements, and transformations.

Definition 9 (Iteration vector). The *iteration vector* of a statement is the vector consisting of values of the indices of all loops surrounding the statement.

Let S be a statement of a program. The iteration vector is denoted by $\vec{i}_S$. An iteration vector represents a dynamic instance of a statement appearing in a loop nest that may be nested perfectly or imperfectly.

Definition 10 (Domain, Index set). The set of all iteration vectors for a given statement is the *domain* or the *index set* of the statement.

A program comprises a sequence of statements, each statement surrounded by loops in a given order. We denote the domain of a statement S by $\mathcal{D}^S$. When the loop bounds and data accesses are affine functions of outer loop indices and other program parameters, and all conditionals are statically predictable, the domain of every statement is a polyhedron as defined in (2.4). Again, conditionals that are affine functions of outer loop indices and program parameters are statically predictable. Affine loop nests with static control are also called static control programs or regular programs. These programs are readily accepted in the polyhedral model. Several of the restrictions for the polyhedral model can be overcome with tricks or conservative assumptions while still making all analysis and transformation meaningful. However, many pose a challenging problem requiring extensions to the model. Techniques developed and implemented in this thesis apply to all programs for which a polyhedral representation can be extracted. All codes used for experimental evaluation are regular programs with static control.

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    S1: A[i,j] = A[i,j]+u1[i]*v1[j] + u2[i]*v2[j];

for (k=0; k<N; k++)
  for (l=0; l<N; l++)
    S2: x[k] = x[k]+beta*A[l,k]*y[l];
```

Figure 2.2: A portion of the GEMVER kernel

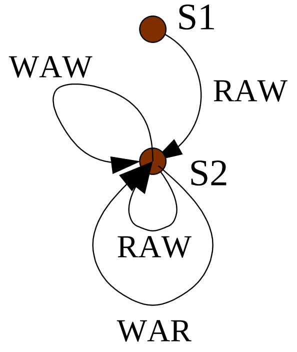


Figure 2.3: The data dependence graph

Each dynamic instance of a statement S, in a program, is identified by its iteration vector $\vec{i}_S$ which contains values for the indices of the loops surrounding S, from outermost to innermost. A statement S is associated with a polytope $\mathcal{D}^S$ of dimensionality m_S. Each point in the polytope is an m_S-dimensional iteration vector, and

$$\begin{array}{rcl} i & \geq & 0 \\ j & \geq & 0 \\ -i+N-1 & \geq & 0 \\ -j+N-1 & \geq & 0 \end{array} \qquad \mathcal{D}^{S_1}: \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 1 & -1 \\ 0 & -1 & 1 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix} \geq 0$$

Figure 2.4: Domain for statement S1 from Figure 2.2

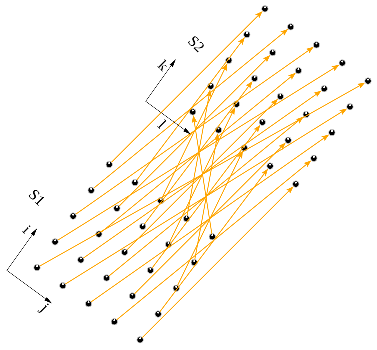
the polytope is characterized by a set of bounding hyperplanes. This is true when the loop bounds are linear combinations (with a constant) of outer loop indices and program parameters (typically, symbolic constants representing the problem size).

2.3 Polyhedral Dependences

Dependences. Two iterations are said to be dependent if they access the same memory location and one of them is a write. A true dependence exists if the source iteration's access is a write and the target's is a read. These dependences are also called read-after-write or RAW dependences, or flow dependences. Similarly, if a write precedes a read to the same location, the dependence is called a WAR dependence or an anti-dependence. WAW dependences are also called output dependences. Read-after-read or RAR dependences are not actually dependences, but they still could be important in characterizing reuse. RAR dependences are also called input dependences.

Dependences are an important concept while studying execution reordering since a reordering will only be legal if does not violate the dependences, i.e., one is allowed to change the order in which operations are performed as long as the transformed program has the same execution order with respect to the dependent iterations.

Data Dependence Graph. The Data Dependence Graph (DDG) $G = (V, E)$ is a directed multi-graph with each vertex representing a statement, i.e., $V = \mathbf{S}$. An edge, $e \in E$, from node S_i to S_j represents a dependence with the source and target



$$\left(\begin{array}{cccccc} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & -1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & -1 \\ \hline 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 \end{array}\right) \begin{pmatrix} i \\ j \\ k \\ l \\ N \\ 1 \end{pmatrix} \begin{array}{c} \\ \\ \geq \\ \\ \\ = \end{array} \left(\begin{array}{c} 0 \\ 0 \\ 0 \\ 0 \\ \hline 0 \\ 0 \end{array}\right)$$

Figure 2.5: Flow dependence from A[i][j] of S1 to A[l][k] of S2 and its dependence polyhedron (courtesy: Loopo display tool)

conflicting accesses in S_i and S_j respectively. Figure 2.3 shows the DDG for the code in Figure 2.2.

2.3.1 Dependence polyhedron.

For an edge e, the relation between the dynamic instances of S_i and S_j that are dependent is captured by the *dependence polyhedron*, $\mathcal{P}_e$. The dependence polyhedron is in the sum of the dimensionalities of the source and target statement's polyhedra along with dimensions for program parameters. If $\vec{s}$ and $\vec{t}$ are the source and target iterations that are dependent, we can express:

$$\langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_e \quad \iff \quad \vec{s} \in \mathcal{D}^{S_i}, \vec{t} \in \mathcal{D}^{S_j} \text{ are dependent through edge } e \in E \tag{2.6}$$

The ability to capture the exact conditions on when a dependence exists through linear equalities and inequalities rests on the fact that there exists a affine relation between the iterations and the accessed data for regular programs. Equalities can be replaced by two inequalities ('≥ 0' and '≤ 0') and everything can be converted to inequalities in the ≥ 0 form, i.e., the polyhedron can be expressed as the intersection of a finite number of non-negative half-spaces. Let the i^{th} inequality after conversion to such a form be denoted by $\mathcal{P}_e^i$. For the code shown in Figure 2.2, consider the dependence between the write at A[i][j] from S1 and the read A[l][k] in S2. The dependence polyhedron for this edge is shown in Figure 2.5.

In the next chapter, we see that the dependence polyhedra is the most important structure around which the problem of finding legal and good transformations centers. In particular, the Farkas lemma (Sec. 2.1) is applied for the dependence polyhedron. A minor point to note here is that the dependence polyhedra we see are often integral polyhedra, i.e., polyhedra that have integer vertices. Hence, the application of Farkas lemma for it is exact and not conservative. Even when the dependence polyhedron is not integral, i.e., when its integer hull is a proper subset of the polyhedron, the difference between applying it to the integer hull and the entire polyhedron is highly unlikely to matter in practice. If need be, one can construct the integer hull of the polyhedron and apply the Farkas lemma on it.

2.3.2 Strengths of dependence polyhedra

The dependence polyhedra are a very general and accurate representation of instance-wise dependences which subsume several traditional notions like distance vectors (also called uniform dependences), dependence levels, and direction vectors. Though a similar notion of exact dependences was presented by Feautrier [Fea91] for value-based array dataflow analysis, this notion of dependence polyhedra has only been sharpened in the past few years by researchers [CGP+05, VBGC06, PBCV07] and is not restricted to programs in single assignment form nor does it require conversion to single-assignment form. Dependence abstractions like direction vectors or distance vectors are tied to a particular syntactic nesting unlike dependence polyhedra which is more abstract and captures the relation between integer points of polyhedra. One could obtain weaker dependence representations from a dependence polyhedra.

```
for (t=0; t<tmax; t++) {
  for (j=0; j<ny; j++)
    ey [0][ j ] = t;
    for (i=1; i<nx; i++)
      for (j=0; j<ny; j++)
        ey[i ][ j ] = ey[i ][ j ] − 0.5∗(hz[i ][ j]−hz[i−1][j ]);
    for (i=0; i<nx; i++)
      for (j=1; j<ny; j++)
        ex[i ][ j ] = ex[i ][ j ] − 0.5∗(hz[i ][ j]−hz[i ][ j−1]);
    for (i=0; i<nx; i++)
      for (j=0; j<ny; j++)
        hz[i ][ j]=hz[i [ j ] −
          0.7∗(ex[i ][ j+1]−ex[i][j]+ey[i+1][j]−ey[i ][ j ]);
}
```

Figure 2.6: An imperfect loop nest

$$\begin{aligned}
0 &\leq t \leq T-1 \\
0 &\leq t' \leq T-1 \\
0 &\leq i \leq N-1 \\
0 &\leq j \leq N-1 \\
0 &\leq i' \leq N-1 \\
0 &\leq j' \leq N-1 \\
t &= t'-1 \\
i &= i'-1 \\
j &= j'
\end{aligned}$$

Figure 2.7: Dependence polyhedron: $S4(hz[i][j]) \rightarrow S2(hz[i-1][j])$

Another example. For the code shown in Figure 2.6, consider the flow dependence between S4 and S2 from the write at hz[i][j] to the read at hz[i-1][j] (later time steps). Let $\vec{s} \in \mathcal{D}^{S4}$, $\vec{t} \in \mathcal{D}^{S2}$, $\vec{s} = (t, i, j)$, $\vec{t} = (t', i', j')$; then, $\mathcal{P}_e$ for this edge is shown in Figure 2.7.

Uniform and Non-uniform dependences. Uniform dependences traditionally make sense for a statement in perfectly nested loop nest or two statements which are in the same perfectly nested loop body. In such cases a uniform dependence is a dependence where the source and target iteration in question are a constant vector distance apart. Such a dependence is also called a constant dependence and represented as a distance vector [Wol95].

For detailed information on polyhedral dependence analysis and a good survey of older techniques in the literature including non-polyhedral ones, the reader can refer to [VBGC06].

2.4 Polyhedral Transformations

A one-dimensional affine transform for statement S is an affine function defined by:

$$\begin{aligned} \phi_S(\vec{i}) &= \begin{pmatrix} c_1^S & c_2^S & \dots & c_{m_S}^S \end{pmatrix} \begin{pmatrix} \vec{i}_S \end{pmatrix} + c_0^S \\ &= \begin{pmatrix} c_1^S & c_2^S & \dots & c_{m_S}^S & c_0^S \end{pmatrix} \begin{pmatrix} \vec{i}_S \\ 1 \end{pmatrix} \end{aligned} \tag{2.7}$$

where $c_0, c_1, c_2, \dots, c_{m_S} \in \mathbb{Z}, \quad \vec{i} \in \mathbb{Z}^{m_S}$ Hence, a one-dimensional affine transform for each statement can be interpreted as a partitioning hyperplane with normal $(c_1, \dots, c_{m_S})$. A multi-dimensional affine transformation can be represented as a sequence of such ϕ's for each statement. We use a superscript to denote the hyperplane for each level. ϕ_S^k represents the hyperplane at level k for statement S. If $1 \le k \le d$, all the ϕ_S^k can be represented by a single d-dimensional affine function $\mathcal{T}_S$ given by:

$$\mathcal{T}_S \vec{i}_S = M_S \vec{i}_S + \vec{t}_S \tag{2.8}$$

where $M_S \in \mathbb{Z}^{d \times m_S}$, $\vec{t}_S \in \mathbb{Z}^d$.

$$\mathcal{T}_S(\vec{i}) = \begin{pmatrix} \phi_S^1(\vec{i}) \\ \phi_S^2(\vec{i}) \\ \vdots \\ \phi_S^d(\vec{i}) \end{pmatrix} = \begin{pmatrix} c_{11}^S & c_{12}^S & \dots & c_{1m_S}^S \\ c_{21}^S & c_{22}^S & \dots & c_{2m_S}^S \\ \vdots & \vdots & \vdots & \vdots \\ c_{d1}^S & c_{d2}^S & \dots & c_{dm_S}^S \end{pmatrix} \vec{i}_S + \begin{pmatrix} c_{10}^S \\ c_{20}^S \\ \vdots \\ c_{d0}^S \end{pmatrix} \tag{2.9}$$

Scalar dimensions. The dimensionality of $\mathcal{T}_S$, d, may be greater than m_S as some rows in $\mathcal{T}_S$ serve the purpose of representing partially fused or unfused dimensions at a level. Such a row has $(c_1, \dots, c_{m_S}) = \mathbf{0}$, and a particular constant for c_0. All statements with the same c_0 value are fused at that level and the unfused sets are placed in the increasing order of their c_0s. We call such a level a **scalar dimension**. Hence, a level is a scalar dimension if the ϕ's for all statements at that level are constant functions. Figure 2.8 shows a sequence of matrix-matrix multiplies and how a transformation captures a legal fusion: the transformation fuses ji of S1 with jk of S2; ϕ^3 is a scalar dimension.

Complete scanning order. The number of rows in M_S for each statements should be the same (d) to map all iterations to a global space of dimensionality d. To provide a complete scanning order for each statement, the number of linearly independent ϕ_S's for a statement should be the same as the dimensionality of the statement, m_S, i.e., $\mathcal{T}_S$ should have full column rank. Note that it is always possible to represent any transformed code (any nesting) with at most $2m_S^* + 1$ rows, where $m_S^* = \max_{S \in \mathbf{S}} m_S$.

Composition of simpler transformations. Multi-dimensional affine functions capture a sequence of simpler transformations that include permutation, skewing,

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
    S1: C[i,j] = C[i,j] + A[i,k] * B[k,j];
    }
  }
}
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
    S2: D[i,j] = D[i,j] + E[i,k] * C[k,j];
    }
  }
}
```

Original code

```
for (t0=0;t0<=N−1;t0++) {
  for (t1=0;t1<=N−1;t1++) {
    for (t3=0;t3<=N−1;t3++) {
      C[t1,t0]=A[t1,t3]*B[t3,t0]+C[t1,t0];
    }
    for (t3=0;t3<=N−1;t3++) {
      D[t3,t0]=E[t3,t1]*C[t1,t0]+D[t3,t0];
    }
  }
}
```

Transformed code

$$T_{S_1}(\vec{i}_{S_1}) = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$T_{S_2}(\vec{i}_{S_2}) = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

i.e.,

$$\begin{array}{ll} \phi^1_{S_1} = j & \phi^1_{S_2} = j \\ \phi^2_{S_1} = i & \phi^2_{S_2} = k \\ \phi^3_{S_1} = 0 & \phi^3_{S_2} = 1 \\ \phi^4_{S_1} = k & \phi^4_{S_2} = i \end{array}$$

Figure 2.8: Polyhedral transformation: an example

reversal, fusion, fission (distribution), relative shifting, and tiling for fixed tile sizes. Note that tiling cannot be readily expressed as an affine function on the original iterators ($\vec{i_S}$), but can be once supernodes are introduced into the domain increasing its dimensionality: this is covered in detail in a general context in Chapter 5.

Traditional transformations like unimodular transformations [Ban93, Wol95] and non-unimodular [Ram92, LP94, Ram95] ones were applied to a single perfect loop nest in isolation. They are therefore subsumed. Due to the presence of scalar dimensions, polyhedral transformations can be used to represent or transform to any kind of nesting structure. Also, they map iterations of statements to a common multidimensional space providing the ability to interleave iterations of different statements as desired.

One can notice a one-to-one correspondence between the A, B, Γ representation used for URUK/WRAP-IT [GVB$^+$06] and the one we described above, except that we have all coefficients in Γ set to zero, i.e., no parametric shifts. The motivation behind this will be clear in the next two chapters. The above representation for transformations was first proposed, though in different forms, by Feautrier [Fea92b] and Kelly et al. [Kel96], but used systematically by viewing it in terms of three components, A, B, and Γ only recently [CGP$^+$05, GVB$^+$06, Vas07].

The above notation for transformations directly fits with scattering functions that a code generation tool like CLooG [Bas04a, Clo] supports. It refers to $\mathcal{T}_S$ as a scattering function. On providing the original statement domains, $\mathcal{D}^S$, along with $\mathcal{T}_S$, Cloog can scan the domains in the global lexicographic ordering imposed by $T_S(\vec{i_S})$ across all $S \in \mathbf{S}$. The goal of automatic transformation is to find the unknown coefficients of $\mathcal{T}_S$, $\forall S \in \mathbf{S}$.

2.4.1 Why affine transformations?

Definition 11 (Convex combination). A convex combination of vectors, $\vec{x}_1$, $\vec{x}_2$, $\ldots$, $\vec{x}_n$, is of the form $\lambda_1\vec{x}_1 + \lambda_2\vec{x}_2 + \cdots + \lambda_n\vec{x_n}$, where $\lambda_1, \lambda_2, \ldots, \lambda_n \geq 0$ and $\sum_{i=1}^{n} \lambda_i = 1$.

Informally, a convex combination of two points always lies on the line segment joining the two points. In the general case, a convex combination of any number of points lies inside the convex hull of those points.

The primary reason affine transformations are of interest is that affine transformations are the most general class of transformations that preserve the collinearity and convexity of points in space, besides the ratio of distances. An affine transformation transforms a polyhedron into another polyhedron and one stays in the polyhedral abstraction for further analyses and most importantly for code generation. Code generation is relatively easier and so has been studied extensively for affine transformations. We now quickly show that if $\mathcal{D}^S$ is convex, its image under the affine function $\mathcal{T}_S$ is also convex. Let the image be:

$$T(\mathcal{D}^S) = \left\{ \vec{z} \mid \vec{z} = \mathcal{T}_S(\vec{x}), \vec{x} \in \mathcal{D}^S \right\}$$

Consider the convex combination of any two points, $\mathcal{T}_S(\vec{x})$ and $\mathcal{T}_S(\vec{y})$, of $T(\mathcal{D}^S)$:

$$\lambda_1\mathcal{T}_S(\vec{x}) + \lambda_2\mathcal{T}_S(\vec{y}), \quad \lambda_1 + \lambda_2 = 1, \lambda_1 \geq 0, \lambda_2 \geq 0$$

Now,

$$\begin{aligned}
\lambda_1 \mathcal{T}_S(\vec{x}) + \lambda_2 \mathcal{T}_S(\vec{y}) &= \lambda_1 M_S(\vec{x}) + \lambda_1 \vec{t}_S + \lambda_2 M_S(\vec{y}) + \lambda_2 \vec{t}_S \\
&= M_S(\lambda_1 \vec{x} + \lambda_2 \vec{y}) + \vec{t}_S, \quad (\because \lambda_1 + \lambda_2 = 1) \\
&= \mathcal{T}_S(\lambda_1 \vec{x} + \lambda_2 \vec{y}) \qquad (2.10)
\end{aligned}$$

Since $\mathcal{D}^S$ is convex, $\lambda_1 \vec{x} + \lambda_2 \vec{y} \in \mathcal{D}^S$. Hence, from (2.10), we have:

$$\begin{aligned}
\lambda_1 \mathcal{T}_S(\vec{x}) + \lambda_2 \mathcal{T}_S(\vec{y}) &\in T(\mathcal{D}^S) \\
\Rightarrow T(\mathcal{D}^S) \text{ is convex}
\end{aligned}$$

If M_S (the linear part of $\mathcal{T}_S$) has full column rank, i.e., the rank of M_S is m^S, $\mathcal{T}_S$ is a one-to-one mapping from $\mathcal{D}^S$ to $T(\mathcal{D}^S)$. A point to note when looking at integer spaces instead of rational or real spaces is that not every integer point in the rational domain that encloses $T(\mathcal{D}^S)$ may have an integer pre-image in $\mathcal{D}^S$, for example, transformations that are non-unimodular may create sparse integer polyhedra. This is not a problem since a code generator like Cloog can scan such sparse polyhedra by inserting modulos. Note that just like convexity, affine transformations also preserve the ratio of distances between points. Since integer points in the original domain are equally spaced, they are so in the transformed space too. Techniques for removal of modulos also exist [Vas07]. Hence, no restrictions need be imposed on the affine function $\mathcal{T}_S$. Sparse integer polyhedra also correspond to code with non-unit strides. However, these can be represented with an additional dimension as long as the stride is a constant, for eg., as $\{0 \leq i \leq n-1, \; i = 2k\}$ for i going from 0 to $n-1$ with stride two. However, if one is interested, a more direct representation for integer points in a polyhedron can be used [S.P00, GR07]. The term $\mathcal{Z}$-polyhedron is associated with such a representation which is the image of a rational polyhedron under an affine integer lattice. Closure properties with such a representation under various operations including affine image and pre-image have been proved [GR07].

2.5 Putting the Notation Together

Let us put together the notation introduced so far. Let the statements of the program be S_1, S_2, …, S_m. Let $\mathbf{S}$ be the set of all statements. Let $\vec{n}$ be the vector of program parameters, i.e., typically loop bounds or symbols appearing in the loop bounds, access functions, or the statement body.

Let $G = (V, E)$ be the data dependence graph of the original program, i.e., $V = \mathbf{S}$ and E is the set of data dependence edges. $e^{S_i \to S_j} \in E$ denotes an edge from S_i to S_j, but we will often drop the superscript on e. For every edge $e \in E$ from S_i to S_j, let the dependence polyhedron be $\mathcal{P}_e$, the fact that a source iterations $\vec{s} \in \mathcal{D}^{S_i}$ and a target iteration $\vec{t} \in \mathcal{D}^{S_j}$ are dependent are known through the equalities and inequalities in the dependence polyhedron, and we express this fact by:

$$\langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_e \quad \iff \quad \vec{s} \in \mathcal{D}^{S_i}, \vec{t} \in \mathcal{D}^{S_j} \text{ are dependent through edge } e^{S_i \to S_j} \in E \tag{2.11}$$

$\phi^k_{S_i}$ denotes the affine hyperplane or function for level k for S_i, $1 \leq k \leq d$. The set of all $\phi^k_{S_i}$, for $S_i \in \mathbf{S}$ represent the interleaving of all statement instances at level k. $\mathcal{T}_S$ is a d-dimensional affine function for each S as defined in (2.9). The subscript on ϕ^k is dropped when referring to the property of the function across all statements, since all statements instances are mapping to a target space-time with dimensions $\phi^1, \phi^2, \ldots, \phi^d$.

2.6 Legality and Parallelism

Dependence satisfaction. A dependence edge e with polyhedron $\mathcal{P}_e$ is *satisfied* at a level l iff l is the first level at which the following condition is met:

$$\forall k(1 \leq k \leq l-1) : \phi^k_{S_j}\left(\vec{t}\right) - \phi^k_{S_i}\left(\vec{s}\right) \geq 0 \quad \bigwedge \quad \phi^l_{S_j}\left(\vec{t}\right) - \phi^l_{S_i}\left(\vec{s}\right) \geq 1, \quad \langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_e$$

Legality. Statement-wise affine transformations ($\mathcal{T}_S$) as defined in (2.9) are legal iff

$$T_{S_j}(\vec{t}) - T_{S_i}(\vec{s}) \succ \vec{0}_d, \quad \langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_e, \forall e \in E \tag{2.12}$$

Definition 12 (**Permutable band**). The ϕs at levels p, $p+1$, $\ldots$, $p+s-1$ form a permutable band of loops in the transformed space iff

$$\forall k \quad (p \leq k \leq p+s-1) : \quad \phi^k_{S_j}(\vec{t}) - \phi^k_{S_i}(\vec{s}) \geq 0, \quad \langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_e, e \in E_p \tag{2.13}$$

where E_p is the set of dependences not satisfied up to level $p-1$.

The above directly follows from (2.12). Loops within a permutable band can be freely interchanged or permuted among themselves. One can see that doing so will not violate (2.12) since dependence components for all unsatisfied dependences are non-negative at each of the dimensions in the band. We will later find the above definition a little conservative. Its refinement and associated intricacies will be discussed in Section 5.4.2 of Chapter 5.

Definition 13 (**Outer parallel**). A $\{\phi_{S1}, \phi_{S2}, \ldots, \phi_{S_m}\}$ is an outer parallel hyperplane if and only if

$$\phi_{S_j}(\vec{t}) - \phi_{S_i}(\vec{s}) = 0, \quad \langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_e, \quad \forall e \in E$$

Outer parallelism is also often referred to as communication-free parallelism or synchronization-free parallelism.

```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    a[i][j] = a[i][j-1] + 1;
  }
}
```

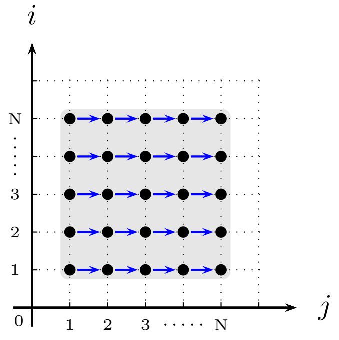


Figure 2.9: Outer parallel loop, i: hyperplane (1,0)

```
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        a[i][j] = a[i-1][N-1-j] + 1;
    }
}
```

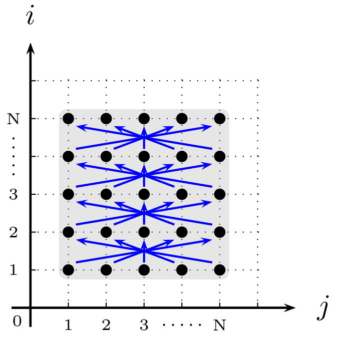


Figure 2.10: Inner parallel loop, j: hyperplane (0,1)

Definition 14 (**Inner parallel**). A $\{\phi^k_{S1}, \phi^k_{S2}, \ldots, \phi^k_{S_m}\}$ is an inner parallel hyperplane if and only if $\phi^k_{S_i}(\vec{t}) - \phi^k_{S_i}(\vec{s}) = 0$, for every $\langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_e$, $e \in E_k$, where E_k is the set of dependences not satisfied up to level $k-1$.

It is illegal to move an inner parallel loop in the outer direction since the dependences satisfied at loops it has been moved across may be violated at the new position of the moved loop. However, it is always legal to move an inner parallel loop further inside.

```
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        a[i][j] = a[i][j−1] + a[i−1][j];
    }
}
```

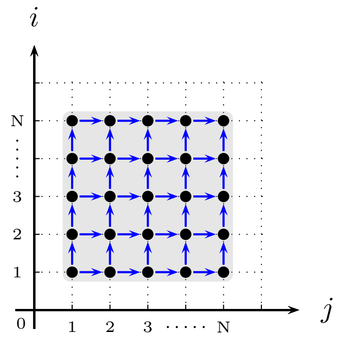


Figure 2.11: Pipelined parallel loop: i or j

Outer and inner parallelism is often referred to as doall parallelism. However, note that inner parallelism requires synchronization every iteration outer to the loop.

Pipelined parallelism. Two or more loops may have dependences that have dependence components along each of them can still be executed in parallel if one of them can be delayed with respect to the other by a fixed amount. If dependence components are non-negative along each of the dimensions in question, one just needs a delay of one. Figure 2.11 shows a code with dependences along both i and j. However, say along i, successive iterations can start with a delay of one and continue executing iterations for j's in sequence. Similarly, if there are n independent dimensions, at most $n-1$ of them can be pipelined while iterations along at least one will be executed in sequence. Pipelined parallelism is also often referred to as doacross parallelism. We will formalize conditions for this in a very general setting in Chapter 3 since it is goes together with tiling. Code generation for pipelined parallelism is discussed in Chapter 5.

Space-time mapping. Once properties of each row of $\mathcal{T}_S$ are known, some of them can be marked as space, i.e., a dimension along which iterations are executed by different processors, while others can be marked as time, i.e., a dimension that is executed sequentially by a single processor. Hence, $\mathcal{T}_S$ specifies a complete space-time mapping for S. Each of the d dimensions is either space or time. Since M_S is of full column rank, when an iteration executes and where it executes, is known. However, in reality, post-processing can be done to $\mathcal{T}_S$ before such a mapping is achieved.

CHAPTER 3

Automatic Transformations for Parallelism and Locality

The three major phases of an optimizer for parallelism and locality are:

1. Static analysis: Computing affine dependences for an input source program
2. Automatic transformation: Computing the transformations automatically
3. Code generation: Generating the new nested loop code under the computed transformations

As explained in Chapter 1, the first and last steps are currently quite stable, while no scalable and practical approach exists for the middle step that works for all polyhedral input or for input that the first and last steps are known to be quite advanced for. This chapter deals with the theory for the key middle step: automatic transformation, which is often considered synonymous with automatic parallelization.

3.1 Schedules, Allocations, and Partitions

Hyperplanes can be interpreted as schedules, allocations, or partitions, or any other term a researcher may define based on the properties it satisfies. Saying that a hyperplane is one of them implies a particular property that the new loop will satisfy in the transformed space. Based on the properties the hyperplanes will satisfy at various levels, certain further transformations like tiling, unroll-jamming, or marking it parallel, can be performed. Typically, scheduling-based works [Fea92a, Fea92b, DR96, Gri04] obtains the new set of hyperplanes as schedules and allocations, while Lim and Lam [LL98] find them as space and time partitions.

3.2 Limitations of Previous Techniques

In this section, we briefly describe the limitations of existing polyhedral transformation frameworks in the literature. Automatic parallelization efforts in the polyhedral model broadly fall into two classes: (1) scheduling/allocation-based, and (2) partitioning-based. The works of Feautrier [Fea92a, Fea92b] and Griebl [Gri04] (to some extent) fall into the former class, while Lim and Lam's approach [LL98, LCL99, LLL01] falls into the second class.

3.2.1 Scheduling + allocation approaches

Schedules specify when iterations execute. A schedule can assign a time point for every iteration, and a code generator can generate code that will scan them in the specified order. Schedules in our context are assumed to be affine functions since code generation in such cases has been studied extensively. Hence, serial order or sequentiality is implicit in a schedule. For a complete coverage of scheduling for automatic parallelization, the reader is referred to the book of Darte et al. [DRV00].

Briefly, a set of scheduling hyperplanes satisfy all dependences.

$$\theta_{S_j}(\vec{t}) - \theta_{S_i}(\vec{s}) \geq 1, \quad \langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_{e^{S_i \to S_j}}$$

Several such θ with a given order specify a multi-dimensional schedule. Allocations specify where iterations execute. An allocation is implicitly parallel. It can always be coarsened, i.e., a group of iterations mapped to a particular processor since everything that is scheduled at a time point are independent of each other. With scheduling-allocation techniques, good schedules with the minimum number of dimensions are first found [Fea92a, Fea92b], and then allocations are found [Fea94, DR96, GFG05].

If communication costs were to be zero or negligible when compared with computation and there were to be no memory hierarchy, clearly, just using optimal or near optimal affine schedules would be the solution. This is obviously not the case with any modern parallel computer architecture. Hence, reducing communication overhead and improving locality across the memory hierarchy through tiling is needed. For the iteration space depicted in Figure 3.2, an affine (fine-grained) schedule is given by:

$$\theta \begin{pmatrix} i \\ j \end{pmatrix} = 2 * i + j$$

Limitations. The approaches have the following limitations.

1. Using schedules and allocations does not naturally fit well with outer parallelism and tiling, since schedules and allocations directly imply outer sequential and inner parallel loops, i.e., they go hand-in-hand with inner parallelism (Figure 3.1). The inner loops can always be readily tiled (space tiling), but they may not give

coarse-enough parallelism. One may try to find allocations that minimize data movement, however, it may still require a high order of synchronization, or too frequent synchronization.

2. They can go against locality even if the schedule gives a coarse granularity of parallelism, since reuse will often be across the outer loops (dependences satisfied at outer levels) and no tiling can be done across those without modifying the allocations. This in turn may kill all benefits of parallelization due to limited memory bandwidth. Inability to tile along the scheduling dimensions would also affect register reuse when register tiling is done.

3. Using schedules and allocations typically leads to schedules with minimum number of schedule dimensions and maximum number of parallel loops. This misses solutions that correspond to higher dimensional schedules. These missed schedules might be sub-optimal with the typical schedule selection criteria, but still perform better due to better tiling and coarse-grained parallelization. The downsides of this cannot be undone.

One may try to “fix” the above problems by finding allocations with a certain property that will allow tiling along the scheduling dimensions too [GFG05, Gri04], but it has other undesired effects – sometimes unable to find the natural solution primarily due to the third reason listed above. All of these will be clearer to the reader through this chapter.

```
for (t1=0; i<N; i++) {
  for (t2=0; i<N; i++) {
    forall (p1=0; i<N; i++) {
      forall (p2=0; i<N; i++) {
        S1
      }
    }
    <barrier>;
  }
}
```

Figure 3.1: A typical solution with scheduling-allocation

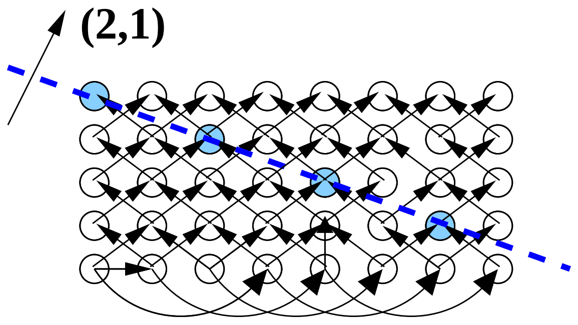


Figure 3.2: A fine-grained affine schedule

3.2.2 Existing partitioning-based approaches

A partitioning based approach would just use the code generator as a way to scan the iteration space in another order and later on mark loops as parallel or sequential, or stripmine and interchange loops at a later point. We gave an detailed background on affine hyperplanes as partitions in Section 2.4. A key limitation of existing techniques in this class is the absence of a way to find good partitions. Criteria based on finding those that maximize the number of degrees of parallelism and minimize the order of synchronization are not sufficient to enable good parallelization. One could for example obtain the partitioning shown in Figure 3.3 which is:

$$\phi^1(i,j) = i$$

$$\phi^2(i,j) = 3i + j$$

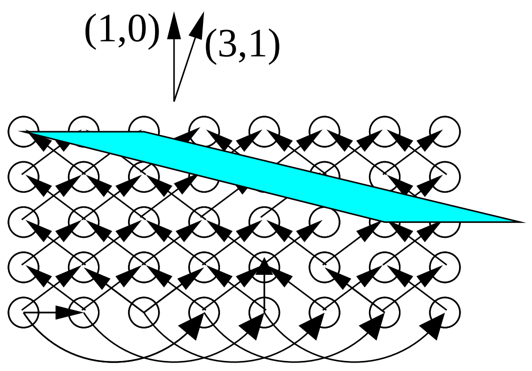


Figure 3.3: Affine partitioning with no cost function can lead to unsatisfactory solutions

However, a desirable affine partitioning here is given by the following and shown in Figure 3.4.

$$\phi^1(i,j) = i$$

$$\phi^2(i,j) = i + j$$

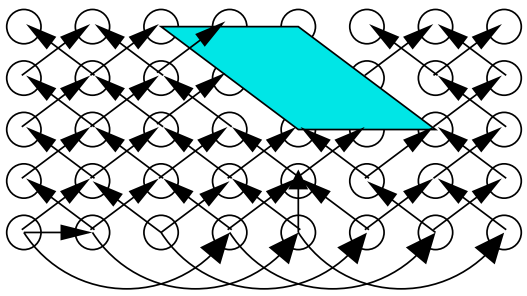


Figure 3.4: A good set of ϕs: (1,0) and (1,1)

3.3 Recalling the Notation

Let us recall the notation introduced in the previous chapter in Section 2.4 and Section 2.5 before we introduce the new transformation framework. The statements of the program are $S_1, S_2, \ldots, S_n$. The dimensionality of S_i is m_{S_i}. $\mathbf{S} = \{S_1, S_2, \ldots, S_n\}$. $\vec{n}$ is the vector of program parameters, i.e., typically loop bounds or symbols that appear in loop bounds, access functions, or the statement body.

$G = (V, E)$ is the data dependence graph for the original program, and G is a multi-graph. $V = \mathbf{S}$ and E is set of data dependence edges. $e^{S_i \rightarrow S_j} \in E$ denotes an edge from S_i to S_j, but we will often drop the superscript on e for better readability. For every edge $e \in E$ from S_i to S_j, $\mathcal{P}_e$ is the dependence polyhedron. The fact that a source iterations $\vec{s} \in S_i$ and a target iteration $\vec{t} \in S_j$ are dependent are known through the equalities and inequalities in the dependence polyhedron, and this is expressed as:

$$\langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_e \quad \Longleftrightarrow \quad \vec{s} \in \mathcal{D}^{S_i}, \vec{t} \in \mathcal{D}^{S_j} \text{ are dependent through edge } e^{S_i \to S_j} \in E \tag{3.1}$$

All inequalities and equalities in $\mathcal{P}_e$ can be written as inequalities in the ≥ 0 form. Let the LHS of each such inequality be denoted by $\mathcal{P}_e^i$, $1 \leq i \leq m_e$.

$\phi_{S_i}^k$ denotes the affine hyperplane for level k for S_i. The set of all $\phi_{S_i}^k$, for $S_i \in \mathbf{S}$ represent the interleaving of all statement instances at level k. We drop the superscript from the coefficients when a single statement is in question.

$$\phi_S\left(\vec{i}_S\right) = (c_1 \; c_2 \; \ldots \; c_{m_S})\left(\vec{i}_S\right) + c_0 \tag{3.2}$$

$$c_0, c_1, c_2, \ldots, c_{m_S} \in \mathbb{Z}, \quad \vec{i}_S \in \mathbb{Z}^{m_S}$$

The subscript on ϕ^k is dropped when referring to the property of the function across all statements, since all statements instances are mapping to a target space-time with dimensions $\phi^1, \phi^2, \ldots, \phi^d$. $\mathcal{T}_S$ is a d-dimensional affine function capturing the complete space-time transformation for statement S:

$$\mathcal{T}_S\left(\vec{i}_S\right) = M_S \vec{i}_S + \vec{t}_S$$

All row vectors are typeset in bold, while normal vectors are typeset with an overhead arrow. $\succ, \prec$ denote lexicographic comparisons. $\leq, \geq, <, >$ in the context of vectors apply component-wise.

3.4 Problem Statement

The goal of automatic transformation is to find the unknown coefficients of $\mathcal{T}_S$, $\forall S \in \mathbf{S}$. In other words, the ϕ's for all statements at each level are the unknown we wish to find. In addition, the rows that correspond to processor space and that correspond to sequential time should also be determined.

3.5 Legality of Tiling in the Polyhedral Model

Loop tiling [IT88, WL91b, Xue00] is a key transformation in optimizing for parallelism and data locality. There has been a considerable amount of research into these two transformations. Tiling has been studied from two perspectives – data locality optimization and parallelization. Tiling for locality requires grouping points in an iteration space into smaller blocks (tiles) allowing reuse in multiple directions when the block fits in a faster memory which could be registers, L1 cache, L2 cache, or L3 cache. When execution proceeds tile by tile, reuse distances are no longer a function of the problem size, but a function of the tile size. One can tile multiple times, once for each level of the memory hierarchy. Tiling for coarse-grained parallelism involves partitioning the iteration space into tiles that may be concurrently executed on different processors with a reduced frequency and volume of inter-processor communication: a tile is atomically executed on a processor with communication required only before and after execution. One of the key aspects of our transformation framework is to find good ways of performing tiling.

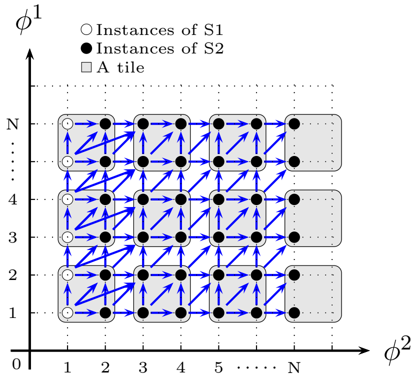


Figure 3.5: Tiling an iteration space

Theorem 1. $\{\phi_{S_1}, \phi_{S_2}, \ldots, \phi_{S_k}\}$ *is a legal (statement-wise) tiling hyperplane if and only if the following holds true for each dependence edge* $e^{S_i \to S_j} \in E$*:*

$$\phi_{S_j}\left(\vec{t}\right) - \phi_{S_i}\left(\vec{s}\right) \;\; \geq \;\; 0, \quad \left\langle \vec{s}, \vec{t} \right\rangle \in \mathcal{P}_{e^{S_i \to S_j}} \tag{3.3}$$

Proof. Tiling of a statement's iteration space defined by a set of tiling hyperplanes is said to be legal if each tile can be executed atomically and a valid total ordering of the tiles can be constructed. This implies that there exists no two tiles such that they both influence each other. Let $\{\phi^1_{S_1}, \phi^1_{S_2}, \ldots, \phi^1_{S_k}\}$, $\{\phi^2_{S_1}, \phi^2_{S_2}, \ldots, \phi^2_{S_k}\}$ be two statement-wise 1-d affine transforms that satisfy (3.3). Consider a tile formed by aggregating a group of hyperplane instances along ϕ^1 and ϕ^2. Due to (3.3), for any dynamic dependence, the target iteration is mapped to the same hyperplane or a greater hyperplane than the source, i.e., the set of all iterations that are outside of the tile and are influenced by it always lie in the forward direction along one of the independent tiling dimensions (ϕ^1 and ϕ^2 in this case). Similarly, all iterations outside of a tile influencing it are either in that tile or in the backward direction along one or more of the hyperplanes. The above argument holds true for both intra- and inter-statement dependences. For inter-statement dependences, this leads to an interleaved execution of tiles of iteration spaces of each statement when code is generated from these mappings. Hence, $\{\phi^1_{S_1}, \phi^1_{S_2}, \ldots, \phi^1_{S_k}\}$, $\{\phi^2_{S_1}, \phi^2_{S_2}, \ldots, \phi^2_{S_k}\}$ represent rectangularly tilable loops in the transformed space. If such a tile is executed on a processor, communication would be needed only before and after its execution. From the point of view of locality, if such a tile is executed with the associated data fitting in a faster memory, reuse is exploited in multiple directions.□

The above condition was well-known for the case of a single-statement perfectly nested loops from the work of Irigoin and Triolet [IT88] (as "$h^T.R \geq \mathbf{0}$"). The above is a generalization for multiple iteration spaces with possibly different dimensionalities and with affine dependences, coming from code that could be nested arbitrarily.

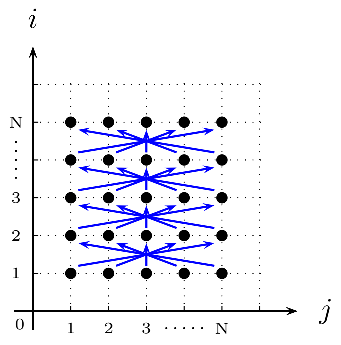


Figure 3.6: No 2-d tiling possible

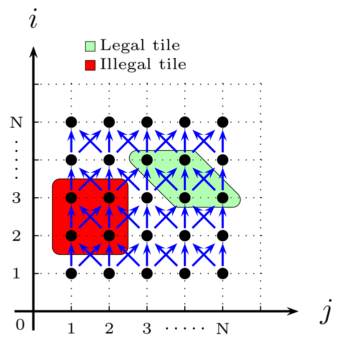


Figure 3.7: A legal and an illegal tile

Tiling hyperplanes at any level. Note that the legality condition as written in (3.3) is imposed on all dependences. However, if it is imposed only on dependences that have not been satisfied up to a certain depth, the independent ϕ's that satisfy the condition represent tiling hyperplanes at that depth, i.e., rectangular blocking (strip-mine/interchange) at that level in the transformed program is legal. More subtleties regarding tiled code generation will be discussed in Chapter 5.

Linearizing the legality condition. Condition (3.3) is non-linear in the unknowns coefficients of the ϕs and loop index variables. The condition when expanded is:

$$\left(c_1^{S_j}, c_2^{S_j}, \dots, c_{m^{S_j}}^{S_j}\right)\vec{t} - \left(c_1^{S_i}, c_2^{S_i}, \dots, c_{m^{S_i}}^{S_i}\right)\vec{s} \;\geq\; 0, \quad \langle \vec{s}, \vec{t}\rangle \in \mathcal{P}_{e^{S_i \rightarrow S_j}}$$

The affine form of the Farkas lemma (Section 2.1) can be used to linearize the above condition. Its use for scheduling purposes was first suggested by Feautrier in [Fea92a], before which the Vertex method [Qui87, Viv02] used to be employed. Using the Farkas lemma, the non-linear form in the loop variables is expressed equivalently as a non-negative linear combination of the faces of the dependence polyhedron.

$$\begin{aligned}
\left(c_1^{S_j}, c_2^{S_j}, \dots, c_{m^{S_j}}^{S_j}\right)\vec{t} - \left(c_1^{S_i}, c_2^{S_i}, \dots, c_{m^{S_i}}^{S_i}\right)\vec{s} \quad &\geq \quad 0, \quad \langle \vec{s}, \vec{t}\rangle \in \mathcal{P}_e \\
&\Longleftrightarrow \\
\left(c_1^{S_j}, c_2^{S_j}, \dots c_{m^{S_j}}^{S_j}\right)\vec{t} - \left(c_1^{S_i}, c_2^{S_i}, \dots c_{m^{S_i}}^{S_i}\right)\vec{s} \quad &\equiv \quad \lambda_{e0} + \sum_{k=1}^{m_e} \lambda_{ek}\mathcal{P}_e^k, \quad \lambda_{ek} \geq 0
\end{aligned}$$

Once this is done, the coefficients of the loop variables on the LHS and the RHS are equated to get conditions free of the original loop variables. Application of the

```
for (t=0; t<T; t++) {
  for (i=2; i<N−1; i++) {
    a[t,i] = 0.333*(a[t−1,i] + a[t−1,i−1]
                   + a[t−1,i+1]);
  }
}
```

Figure 3.8: 1-d Jacobi: perfectly nested

```
for (t = 1; t<T; t++)
  for (i=2; i<N−1; i++)
    S1: b[i] = 0.333*(a[i−1]+a[i]+a[i+1]);
  }
  for{i=2; i<N−1; i++}
    S2: a[i] = b[i];
  }
}
```

Figure 3.9: 1-d Jacobi: imperfectly nested

Farkas Lemma is done on a per-dependence basis, and the resulting constraints linear in the coefficients of the ϕs are aggregated. All Farkas multipliers can be eliminated dependence-wise, some by Gaussian elimination and the rest by Fourier-Motzkin elimination [Ban93, Sch86]. Thanks to its scalability, the Farkas lemma has been used to linearize legality constraints by nearly all polyhedral approaches in the literature [LL98, AMP01, CGP$^+$05, PBCV07].

3.6 Towards Finding Good Solutions

Consider the perfectly-nested version of 1-d Jacobi shown in Figure 3.8 as an example. This discussion also applies to the imperfectly nested version, but for convenience we first look at the single-statement perfectly nested one. The code has three uniform dependences, (1,0), (1,1) and (1,-1) when represented as distance vectors. We first describe solutions obtained by existing state of the art approaches - Lim and Lam's affine partitioning [LL98, LCL99] and Griebl's space and time tiling with Forward Communication-only placement [Gri04]. Lim and Lam define legal

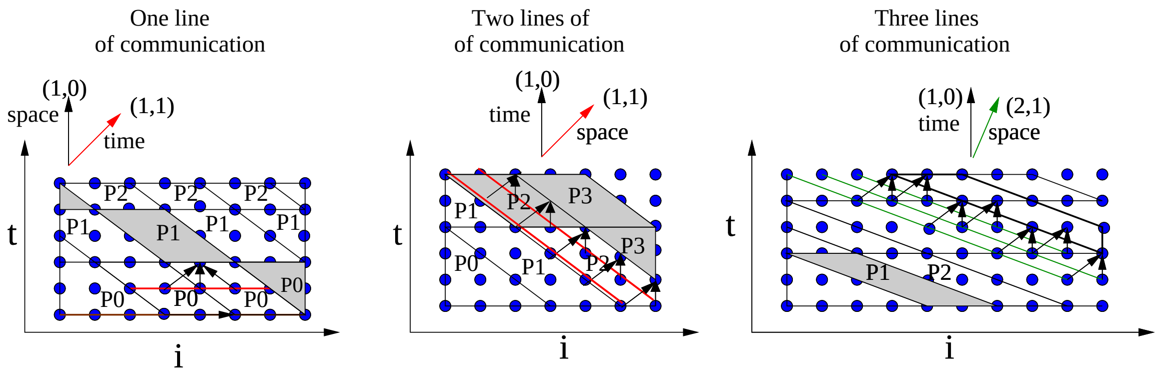


Figure 3.10: Communication volume with different valid hyperplanes for 1-d Jacobi: shaded tiles are to be executed in parallel

time partitions which have the same property of tiling hyperplanes we described in the previous section. Their algorithm obtains affine partitions that minimize the *order* of communication while maximizing the *degree* of parallelism. Using the validity constraint in Eqn 3.3, we obtain the constraints:

$$(c_t, c_i)\begin{pmatrix}1\\0\end{pmatrix} \geq 0; \quad (c_t, c_i)\begin{pmatrix}1\\1\end{pmatrix} \geq 0; \quad (c_t, c_i)\begin{pmatrix}1\\-1\end{pmatrix} \geq 0$$

i.e.,

$$\begin{aligned} c_t &\geq 0 \\ c_i + c_j &\geq 0 \\ c_i - c_j &\geq 0 \end{aligned}$$

There are infinitely many valid solutions with the same order complexity of synchronization, but with different communication volumes that may impact performance. Although it may seem that the volume may not effect performance considering the fact that communication startup time on modern interconnects dominates, for higher dimensional problems like n-d Jacobi, the ratio of communication to computation increases, proportional to tile size raised to $n-1$. Existing works on tiling [SD90, RS92, Xue97] can find good or near communication-optimal tile shapes for perfectly nested loops with constant dependences, but cannot handle arbitrarily nested loops. For 1-d Jacobi, all solutions within the cone formed by the vectors $(1,1)$ and $(1,-1)$ are valid tiling hyperplanes. For the imperfectly nested version of 1-d Jacobi, the valid cone has extremals $(2,1)$ and $(2,-1)$. Lim et al.'s algorithm [LL98] finds two valid independent solutions without optimizing for any particular criterion. In particular, the solutions found by their algorithm (Algorithm A in [LL98]) are $(2,-1)$ and $(3,-1)$ which are clearly not the best tiling hyperplanes to minimize communication volume, though they do maximize the degree of parallelism and minimize the *order* of synchronization to $O(N)$: in this case any valid hyperplane has $O(N)$ synchronization. Figure 3.10 shows that the required communication increases as the hyperplane gets more and more oblique. For a hyperplane with normal $(k,1)$, one would need $(k+1)*T$ values from the neighboring tile.

Using Griebl's technique [GFG05, Gri04], if coarser granularity of parallelism is desired with schedules, the forward communication-only (FCO) constraint finds an allocation satisfying the condition that all dependences have non-negative affine components along it, i.e., communication will be in the forward direction. Due to this, both the schedule and allocation dimensions become one permutable band of loops that can be tiled. Hence, tiling along the scheduling dimensions, called time tiling is enabled. For the code in question, we first find that only space tiling is enabled with the affine schedule being $\theta(t,i) = t$. With a forward-communication only (FCO) placement along (1,1), time tiling is enabled that can aggregate iterations into 2-d tiles decreasing the frequency of communication. However, note that communication in the processor space occurs along (1,1), i.e., two lines of the array are required. However, using (1,0) and (1,1) as tiling hyperplanes with (1,0) as space and (1,1) as inner time and a tile space schedule of (2,1) leads to only one line of communication along (1,0). The approach we will present will find such a solution. We now define a cost metric for an affine transform that captures reuse distance and communication volume.

3.7 Cost Function

Consider the affine function δ defined as follows.

$$\delta_e\left(\vec{s},\vec{t}\right) \quad = \quad \phi_{S_j}\left(\vec{t}\right) - \phi_{S_i}\left(\vec{s}\right), \quad \left\langle \vec{s},\vec{t}\right\rangle \in \mathcal{P}_{e^{S_i \rightarrow S_j}} \tag{3.4}$$

The affine function, $\delta_e(\vec{s},\vec{t})$, holds much significance. This function is also the number of hyperplanes the dependence e traverses along the hyperplane normal. It gives us a measure of the reuse distance if the hyperplane is used as time, i.e., if the hyperplanes are executed sequentially. Also, this function is a rough measure of communication volume if the hyperplane is used to generate tiles for parallelization and used as a processor space dimension. An upper bound on this function would mean that the number of hyperplanes that would be communicated as a result of the dependence at the tile boundaries would not exceed this bound. We are particularly interested if this function can be reduced to a constant amount or zero by choosing a suitable direction for ϕ: if this is possible, then that particular dependence leads to a constant or no communication for this hyperplane. Note that each δ_e is an affine function of the loop indices. The challenge is to use this function to obtain a suitable objective for optimization in a linear setting.

The constraints obtained from Eqn 3.3 above only represent validity (permutability). We discuss below problems encountered when one tries to apply a performance factor to find a good tile shape out of the several possibilities.

As explained in the previous sub-section, the Farkas lemma has been used to eliminate loop variables from constraints by getting equivalent linear inequalities. However, an attempt to minimize δ_e ends up in an objective function involving both loop variables and hyperplane coefficients. For example, $\phi(\vec{t}) - \phi(\vec{s})$ could be $c_1 i + (c_2 - c_3)j$, where $1 \le i \le N \wedge 1 \le j \le N \wedge i \le j$. One could possibly end up with such a form when one or more of the dependences are not uniform, making it infeasible to construct an objective function involving only the unknown coefficients of ϕ's.

3.8 Cost Function Bounding and Minimization

We first discuss a result that would take us closer to the solution.

Lemma 2. *If all iteration spaces are bounded, there exists an affine function in the structure parameters, $\vec{n}$, that bounds $\delta_e(\vec{s}, \vec{t})$ for every dependence edge, i.e., there exists*

$$v(\vec{n}) = \mathbf{u}.\vec{n} + w \tag{3.5}$$

such that

$$v(\vec{n}) \;-\; \big(\phi_{S_j}(\vec{t}) - \phi_{S_i}(\vec{s})\big) \quad \geq \quad 0, \quad \langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_e, \forall e \in E \tag{3.6}$$

i.e.,

$$\mathbf{u}.\vec{n} + w - \delta_e\left(\vec{s}, \vec{t}\right) \quad \geq \quad 0, \quad \langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_e, \forall e \in E \tag{3.7}$$

The idea behind the above is that even if δ_e involves loop variables, one can find large enough constants in the row vector $\mathbf{u}$ that would be sufficient to bound $\delta_e(\vec{s}, \vec{t})$. Note that the loop variables themselves are bounded by affine functions of the parameters, and hence the maximum value taken by $\delta_e(\vec{s}, \vec{t})$ will be bounded by such an affine function. Also, since $\mathbf{u}.\vec{n} + w \geq \delta_e(\vec{s}, \vec{t}) \geq 0$, $\mathbf{u}$ should either increase or stay constant with an increase in the structural parameters, i.e., the components of $\mathbf{u}$ are non-negative. The reuse distance or communication volume for each dependence is bounded in this fashion by the same affine function.

Now, we apply the affine form of the Farkas lemma to (3.7).

$$\mathbf{u}.\vec{n} + w - \delta_e(\vec{s}, \vec{t}) \;\; \equiv \;\; \lambda_{e0} + \sum_{k=1}^{m_e} \lambda_{ek} \mathcal{P}_e^k, \quad \lambda_{ek} \geq 0 \tag{3.8}$$

The above is an identity and the coefficients of each of the loop indices in $\vec{i}$ and parameters in $\vec{n}$ on the left and right hand side can be gathered and equated. We now get linear inequalities entirely in coefficients of the affine mappings for all statements, components of row vector $\mathbf{u}$, and w. The above inequalities can be at once be solved by finding a lexicographic minimal solution with $\mathbf{u}$ and w in the leading position, and the other variables following in any order, i.e., if $\mathbf{u} = (u_1, u_2, \ldots, u_k)$, then our objective function is:

$$\text{minimize}_{\prec}\ (u_1, u_2, \ldots, u_k, w, \ldots, c_i, \ldots) \tag{3.9}$$

Finding the lexicographic minimal solution is within the reach of the Simplex algorithm and can be handled by the PIP software [Fea88]. Since the structural parameters are quite large, we first want to minimize their coefficients. We do not lose the optimal solution since an optimal solution would have the smallest possible values for u's. Though the zero vector is always a trivial solution with the above objective function, it is avoided through constraints described in the next section.

The solution gives a hyperplane for each statement. Note that the application of the Farkas lemma to (3.7) is not required in all cases. When a dependence is uniform, the corresponding $\delta_e(\vec{s}, \vec{t})$ is independent of any loop variables, and application of the Farkas lemma is not required. In such cases, we just have $\delta_e \leq w$.

Avoiding the zero vector

Avoiding the zero solution statement-wise poses a challenge. In particular, to avoid the zero vector we need:

$$\left(c_1^S, c_2^S, \ldots, c_{m_S}^S\right) \neq \vec{0}, \quad \text{for each } S \in \mathbf{S} \tag{3.10}$$

The above constraint cannot be expressed as a single convex space in the c^S's even for a single statement. For multiple statements, the number of possibilities get multiplied. The above difficulties can be solved at once by only looking for non-negative transformation coefficients. Then, the zero solution can be avoided with the constraint of $\sum_{i=1}^{m^S} c_i^S \geq 1$, for each $S \in \mathbf{S}$.

Making the above practical choice leads to the exclusion of solutions that involve loop reversals or combination of reversals with other transformations. In practice, we do not find this to be a concern at all. The current implementation of Pluto [Plu] is with this choice, and scales well without any evident loss of good transformations for codes evaluated.

3.9 Iteratively Finding Independent Solutions

Solving the ILP formulation in the previous section gives us a single solution to the coefficients of the best mappings for each statement. We need at least as many independent solutions as the dimensionality of the polytope associated with each statement. Hence, once a solution is found, we augment the ILP formulation with new constraints and obtain the next solution; the new constraints ensure linear independence with solutions already found. Let the rows of H_S represent the solutions found so far for a statement S. Then, the sub-space orthogonal to H_S [Pen55, LP94, BRS07] is given by:

$$H_S^{\perp} = I - H_S^T \left(H_S H_S^T\right)^{-1} H_S \tag{3.11}$$

Note that $H_S^{\perp}.{H_S}^T = \mathbf{0}$, i.e., the rows of H_S are orthogonal to those of $H_S^{\perp}$. Assume that $H_S^{\perp}$ has been freed of any unnecessary rows so that it forms a spanning basis for the null space of H_S. Let h_S^* be the next row (linear portion of the hyperplane) to be found for statement S. Let ${H^i}_S^{\perp}$ be a row of $H_S^{\perp}$. Then, any *one* of the inequalities given by: $\forall i,\ {H^i}_S^{\perp}.\vec{h}_S^* > 0, {H^i}_S^{\perp}.\vec{h}_S^* < 0$ gives the necessary constraint to be added for statement S to ensure that h_S^* has a non-zero component in the sub-space orthogonal to H_S and thus be linearly independent to H_S. This leads to a non-convex space, and ideally, all cases have to be tried and the best among those kept. When the number of statements is large, this leads to a combinatorial explosion. In such cases, we restrict ourselves to the sub-space of the orthogonal space where all the constraints are positive, i.e., the following constraints are added to the ILP formulation for linear independence:

$$\forall i, {H^i}_S^{\perp}.{h^*}_S \geq 0 \quad \wedge \quad \sum_i {H^i}_S^{\perp} h_S^* \geq 1 \tag{3.12}$$

Example. For example, let us say there is one statement and that the first hyperplane found for it is (1, 0, 0), i.e., $(c_i, c_j, c_k) = (1, 0, 0)$.

$$H_S = (1,0,0) \qquad H_S^{\perp} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

i.e.,

$$H_S^{1\perp} = (0,1,0) \qquad H_S^{2\perp} = (0,0,1)$$

So, for linear independence we need to have:

$$c_j \neq 0 \bigvee c_k \neq 0$$

i.e.,

$$c_j \geq 1 \bigvee c_j \leq -1 \bigvee c_k \geq 1 \bigvee c_k \leq -1$$

Just forcing a positive component in the positive orthant will give a single convex space for linear independence:

$$c_j \geq 0 \wedge \; c_k \geq 0 \wedge \; c_j + c_k \geq 1$$

Note that when only the positive orthant is considered, the constraints for linear independence for all statements form a single convex space.

The mappings found are independent on a per-statement basis. When there are statements with different dimensionalities, the number of such independent mappings found for each statement is equal to its domain dimensionality. Hence, no more linear independence constraints need be added for statements for which enough independent solutions have been found – the rest of the rows get automatically filled with zeros or linearly dependent rows. The number of rows in the transformation matrix (M_S) is the same for each statement, consistent with our definition of $\mathcal{T}_S$. The depth of the deepest loop nest in the transformed code is the same as that of the source loop nest. Overall, a hierarchy of fully permutable bands (Def 12) is found, and a lower level in the hierarchy will not be obtained unless ILP constraints corresponding to dependences that have been satisfied by the parent permutable band have been removed.

3.10 Communication and Locality Optimization Unified

From the algorithm described above, both synchronization-free and pipelined parallelism is found. Note that the best possible solution to Eqn. (3.9) is with $(\mathbf{u} = \mathbf{0}, w = 0)$ and this happens when we find a hyperplane that has no dependence components along its normal, which is a fully parallel loop requiring no synchronization if it is at the outer level, i.e., it is outer parallel. It could be an inner parallel loop if some dependences were removed previously and so a synchronization is required after the loop is executed in parallel. Thus, in each of the steps that we find a new independent hyperplane, we end up first finding all synchronization-free hyperplanes; these are followed by a set of fully permutable hyperplanes that are tilable and pipelined parallel requiring constant boundary communication ($\mathbf{u} = \mathbf{0}, w > 0$) w.r.t the tile sizes. In the worst case, we have a hyperplane with $\mathbf{u} > \mathbf{0}, w \geq 0$ resulting in long communication from non-constant dependences. It is important to note that the latter are pushed to the innermost level. By bringing in the notion of communication volume and its minimization, all degrees of parallelism are found in the order of their preference.

From the point of view of data locality, note that the hyperplanes that are used to scan the tile space are same as the ones that scan points in a tile. Hence, data locality is optimized from two angles: (1) cache misses at tile boundaries are minimized for local execution (as cache misses at local tile boundaries are equivalent to communication along processor tile boundaries); (2) by reducing reuse distances, we are increasing the size of local tiles that would fit in cache. The former is due to selection of good tile shapes and the latter by the right permutation of hyperplanes that is implicit in the order in which we find hyperplanes.

3.10.1 Space and time in transformed iteration space.

By minimizing $\delta_e(\vec{s}, \vec{t})$ as we find hyperplanes from outermost to innermost, we push dependence satisfaction to inner loops and also ensure that no loops have negative dependences components so that all target loops can be blocked. Once this is done, if the outer loops are used as space (how many ever desired, say k), and the rest are used as time (note that at least one time loop is required unless all loops are synchronization-free parallel), communication in the processor space is optimized as the outer space loops are the k best ones. All loops can be tiled resulting in coarse-grained parallelism as well as better reuse within a tile. Hence, the same set of hyperplanes are used to scan points in a tile. When space loops have dependences, a transformation may be necessary in the outer tile space loops to get a schedule of tiles to generate parallel code: this will be described in Chapter 5.

3.11 Examples

3.11.1 A step by step example

Figure 3.11 shows an example from the literature [DV97] with affine non-constant dependences. We exclude the constant c_0 from the transformation function as we have a single statement. Dependence analysis produces the following dependence polyhedra:

$$\begin{aligned}
\text{flow} : a[i',j'] \rightarrow a[i,j-1] \quad & \mathcal{P}_{e_1} : i'=i, j'=j-1, 2 \le j \le N, 1 \le i \le N \\
\text{flow} : a[i',j'] \rightarrow a[j,i] \quad & \mathcal{P}_{e_2} : i'=j, j'=i, 2 \le j \le N, 1 \le i \le N, i-j \ge 1 \\
\text{anti} : a[j',i'] \rightarrow a[i,j] \quad & \mathcal{P}_{e_3} : j'=i, i'=j, 2 \le j \le N,\ 1 \le i \le N,\ i-j \ge 1
\end{aligned}$$

Note that the equalities in the dependence polyhedra can be readily used to eliminate variables from the constraints. In this case, $\mathbf{u} = (u_1)$ since we have only one parameter N.

```
for (i=0; i<N; i++) {
  for (j=1; j<N; j++) {
      a[i,j] = a[j,i]+a[i,j−1]
  }
}
```

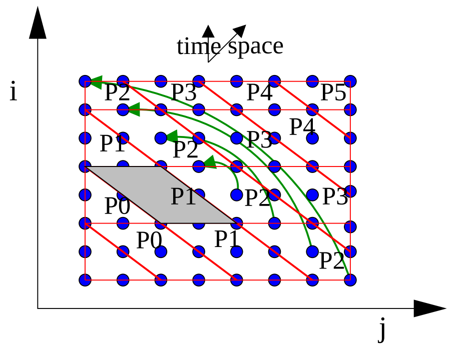


Figure 3.11: Example: Non-uniform dependences

Dependence 1: Tiling legality constraint (3.3) gives:

$$\begin{aligned}(c_i, c_j)\begin{pmatrix} i \\ j \end{pmatrix} - (c_i, c_j)\begin{pmatrix} i' \\ j' \end{pmatrix} &\geq 0, \quad \langle i, j, i', j' \rangle \in \mathcal{P}_{e_1} \\ \Rightarrow c_i i + c_j j - c_i i - c_j(j-1) &\geq 0 \\ \Rightarrow c_j &\geq 0\end{aligned}$$

Since this is a uniform dependence, the volume bounding constraint gives:

$$\begin{aligned}(c_i, c_j)\begin{pmatrix} i \\ j \end{pmatrix} - (c_i, c_j)\begin{pmatrix} i' \\ j' \end{pmatrix} &\leq w, \quad \langle i, j, i', j' \rangle \in \mathcal{P}_{e_1} \\ \Rightarrow w - c_j &\geq 0\end{aligned}$$

Dependence 2: This is not a uniform dependence and hence the application of Farkas Lemma for legality and bounding constraints cannot be avoided. The tiling legality condition is given by:

$$(c_i i + c_j j) - (c_i j + c_j i) \geq 0, \quad 2 \leq j \leq N, 1 \leq i \leq N, i - j \geq 1$$

Applying Farkas lemma, we have:

$$\begin{aligned}(c_i - c_j)i &+ (c_j - c_i)j \\ &\equiv \lambda_0 + \lambda_1(N-i) + \lambda_2(N-j) \\ &\quad +\lambda_3(i-j-1) + \lambda_4(i-1) + \lambda_5(j-2) \qquad (3.13) \\ \lambda_1, \lambda_2, \lambda_3, \lambda_4, \lambda_5 &\geq 0\end{aligned}$$

LHS and RHS coefficients for i, j, N and the constants are equated in (3.13) and the Farkas multipliers are eliminated through Fourier-Motzkin (FM) variable elimination, i.e., by equating coefficients on each side, we first obtain:

$$\begin{aligned}c_i - c_j &= \lambda_3 + \lambda_4 - \lambda_1 \\ c_j - c_i &= \lambda_5 - \lambda_3 - \lambda_2 \\ \lambda_0 - \lambda_3 - \lambda_4 - 2\lambda_5 &= 0 \\ \lambda_1 + \lambda_2 &= 0 \\ \lambda_1 &\geq 0 \\ \lambda_2 &\geq 0 \\ \lambda_3 &\geq 0 \\ \lambda_4 &\geq 0 \\ \lambda_5 &\geq 0\end{aligned}$$

The reader may verify that eliminating the λ's through Gaussian elimination and FM yields:

$$c_i - c_j \geq 0$$

Volume bounding constraint:

$$u_1 N + w - (c_i j + c_j i - c_i i - c_j j) \quad \geq \quad 0, \quad (i,j) \in \mathcal{P}_{e_2}$$

Application of Farkas lemma in a similar way as above and elimination of the multipliers yields:

$$\begin{aligned} u_1 &\geq 0 \\ u_1 - c_i + c_j &\geq 0 \\ 3u_1 + w - c_i + c_j &\geq 0 \end{aligned} \tag{3.14}$$

Dependence 3: Due to symmetry with respect to i and j, the third dependence does not give anything more than the second one.

Avoiding the zero solution: As discussed in Section 3.8, the zero solution is avoided with

$$c_i + c_j \quad \geq \quad 1$$

Finding the transformation. Aggregating legality and volume bounding constraints for all dependences, we obtain:

$$\begin{aligned} c_j &\geq 0 \\ w - c_j &\geq 0 \\ c_i - c_j &\geq 0 \\ u_1 &\geq 0 \\ u_1 - c_i + c_j &\geq 0 \\ 3u_1 + w - c_i + c_j &\geq 0 \\ c_i + c_j &\geq 1 \\ \text{minimize}_{\prec}\ (u_1, w, c_i, c_j) & \end{aligned} \tag{3.15}$$

The lexicographic minimal solution for the vector $(u_1, w, c_i, c_j) = (0, 1, 1, 1)$. The zero vector is a trivial solution and is avoided with $c_i + c_j \geq 1$. Hence, we get $c_i = c_j = 1$. Note that $c_i = 1$ and $c_j = 0$ is not obtained even though it is a valid tiling hyperplane as it involves more communication: it requires u_1 to be positive.

The next solution is forced to have a positive component in the subspace orthogonal to $(1, 1)$ given by (3.11) as:

$$H_S = (1, 1)$$

$$H_S^{\perp} = I - H_S(H_S H_S^T)^{-1} H_S^T = \begin{pmatrix} \frac{1}{2} & \frac{-1}{2} \\ \frac{-1}{2} & \frac{1}{2} \end{pmatrix}$$

We first normalize the rows of $H_S^{\perp}$, then rows that are all zero and negations of previous rows are discarded (the second row in this case) since they do not give anything new in terms of linear independence, and ultimately we are just left with

(1,-1). We thus have $c_i - c_j \geq 1$. With the trade-off discussed in Sec. 3.9, we just add $c_i - c_j \geq 1$ for linear independence.

Adding $c_i - c_j \geq 1$ to (3.15), the lexicographic minimal solution is (1, 0, 1, 0), i.e., $u_1 = 1, w = 0, c_i = 1, c_j = 0$ ($u_1 = 0$ is no longer possible). Hence, $(1,1)$ and $(1,0)$ are the best tiling hyperplanes. The transformation is given by:

$$\mathcal{T}_S\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} i \\ j \end{pmatrix}$$

(1,1) is used as space with one line of communication between processors, and the hyperplane (1,0) is used as time in a tile. The outer tile schedule is (2,1), obtained by the addition of (1,1) and (1,0): this is described in more detailed in Chapter 5 in Section 5.2.3.

This transformation is in contrast to other approaches based on schedules which obtain a schedule and then the rest of the transformation matrix. Feautrier's greedy heuristic gives the schedule $\theta(i,j) = 2i + j - 3$ which satisfies all dependences. However, using this as either space or time does not lead to communication or locality optimization. The (2,1) hyperplane has long dependence components along it. In fact, the only hyperplane that has short dependences along it is (1,1). This is the best hyperplane to be used as a space loop if the nest is to be parallelized, and is the first solution that our algorithm finds. The (1,0) hyperplane is used as time leading to a solution with one degree of pipelined parallelism with one line per tile of near-neighbor communication along (1,1) as shown in Figure 3.11.1. Hence, a good schedule that tries to satisfy all dependences (or as many as possible) is not necessarily a good loop for the transformed iteration space.

3.11.2 Example 2: Imperfectly nested 1-d Jacobi

For the code in Figure 3.9, our algorithm obtains:

$$\mathcal{T}_{S_1}\begin{pmatrix} t \\ i \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 2 & 1 \\ 0 & 0 \end{pmatrix}\begin{pmatrix} t \\ i \end{pmatrix} \qquad \mathcal{T}_{S_2}\begin{pmatrix} t' \\ j \end{pmatrix} == \begin{pmatrix} 1 & 0 \\ 2 & 1 \\ 0 & 0 \end{pmatrix}\begin{pmatrix} t' \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$$

The resulting transformation is equivalent to a constant shift of one for S2 relative to S1, fusion and skewing the i loop with respect to the t loop by a factor of two. The (1,0) hyperplane has the least communication: no dependence crosses more than one hyperplane instance along it.

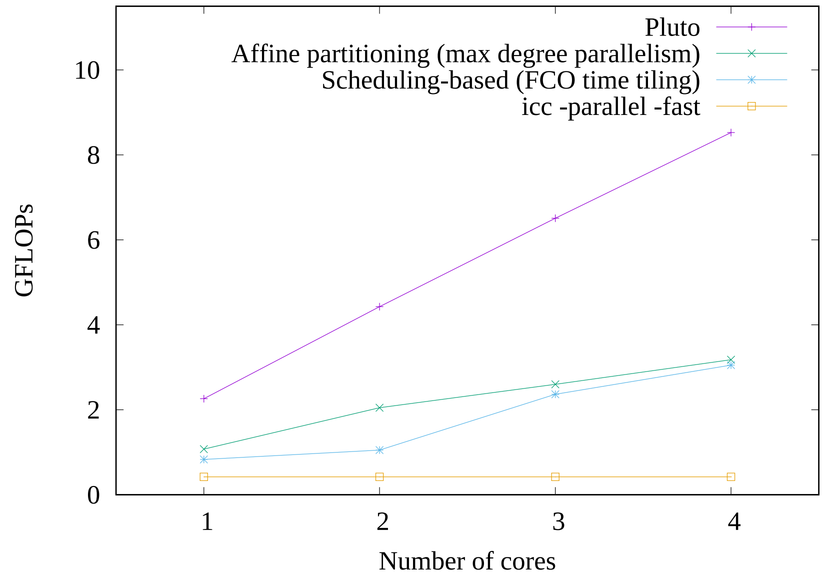


Figure 3.12: Imperfectly nested Jacobi stencil parallelization: $N = 10^6$, $T = 10^5$

3.12 Handling Input Dependences

Input dependences need to be considered for optimization in many cases as reuse can be exploited by minimizing them. Clearly, legality (ordering between dependent RAR iterations) need not be preserved. We thus do not add legality constraints (3.3) for such dependences, but consider them for the bounding objective function (3.7). Since input dependences can be allowed to have negative components in the transformed space, they need to be bounded from both above and below. If $\mathcal{P}_e$ is a dependence polyhedron corresponding to an input (RAR) dependence, we have the constraints:

$$\left|\phi_{S_j}\left(\vec{t}\right) - \phi_{S_i}\left(\vec{s}\right)\right| \quad \leq \quad \mathbf{u}.\vec{n} + w, \quad \langle\vec{s},\vec{t}\rangle \in \mathcal{P}_e$$

$$\text{i.e.,}$$

$$\phi_{S_j}\left(\vec{t}\right) - \phi_{S_i}\left(\vec{s}\right) \leq \mathbf{u}.\vec{n} + w \quad \bigwedge \quad \phi_{S_i}\left(\vec{s}\right) - \phi_{S_j}\left(\vec{t}\right) \leq \mathbf{u}.\vec{n} + w, \quad \langle\vec{s},\vec{t}\rangle \in \mathcal{P}_e$$

3.13 Refinements for Cost Function

The metric we presented here can be refined while keeping the problem within ILP. The motivation behind taking a *max* is to avoid multiple counting of the same set of points that need to be communicated for different dependences. This happens when all dependences originate from the same data space and the same order volume of communication is required for each of them. Using the sum of max'es on a per-array basis is a more accurate metric. Also, even for a single array, sets of points with very less overlap or no overlap may have to be communicated for different dependences. Also, different dependences may have source dependence polytopes of different dimensionalities. Note that the image of the source dependence polytope under the data access function associated with the dependence gives the actual set of points to be communicated. Hence, just using the communication rate (number of hyperplanes on the tile boundary) as the metric may not be accurate enough. This can be taken care of by having different bounding functions for dependences with different orders of communication, and using the bound coefficients for dependences with higher orders of communication as the leading coefficients while finding the lexicographic minimal solution. Hence, the metric can be tuned while keeping the problem linear.

3.14 Loop Fusion

We just give a brief overview here as Chapter 4 is devoted to discussion on how fusion is naturally handled with our scheme. Solving for hyperplanes for multiple statements leads to a schedule for each statement such that all statements in question are *finely* interleaved: this is indeed fusion. In some cases, it might be enabled in combination with other transformations like permutation or shifts. It is important to leave the structure parameter $\vec{n}$ out of our hyperplane form in (2.8) for the above to hold true. Leaving the parameter out allows us to control the fusion choices. Hence, a common tiling hyperplane also represents a fused loop, and reuse distances between components that are weakly connected can be reduced with our cost function. The set of valid independent hyperplanes that can be iteratively found from our algorithm for multiple statements (at a given depth) is the maximum number of loops that can be fused at that depth. This generalization of fusion is same as the one proposed in

[CGP+05, GVB+06] however we are able to automatically enable it in conjunction with other transformations. All of this is the subject of Chapter 4.

3.15 Summary

Algorithm 1 Pluto automatic transformation algorithm

```
INPUT Data Dependence graph $G = (V, E)$ with dependence polyhedra, $\mathcal{P}_e$, $\forall e \in E$
 1: $S_{max}$: statement with maximum domain dimensionality
 2: for each dependence $e^{S_i \to S_j} \in E$ do
 3:   Build legality constraints: apply Farkas Lemma on $\phi_{S_j}(\vec{t}) - \phi_{S_i}(\vec{s}) \geq 0$, under
      $\langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_e$, and eliminate all Farkas multipliers
 4:   Build bounding function constraints: apply Farkas Lemma to $\mathbf{u}.\vec{n} + w - (\phi_{S_j}(\vec{t}) -$
      $\phi_{S_i}(\vec{s})) \geq 0$ under $\langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_e$, and eliminate all Farkas multipliers
 5:   Aggregate constraints from Step 3 and Step 4 into $C_e$
 6: end for
 7: repeat
 8:   $C = \emptyset$
 9:   for each dependence edge $e \in E$ do
10:     $C \leftarrow C \cup C_e$
11:   end for
12:   Compute lexicographic minimal solution with $\mathbf{u}'s$ coefficients in the leading
      position followed by $w$; iteratively find independent solutions to $C$ with linear
      independence constraints (3.12) added after each solution is found
13:   if no solutions were found then
14:     Remove dependences between two strongly-connected components in the
        GDG and insert a scalar dimension in the transformation functions of the
        statements (Chapter 4)
15:   end if
16:   Compute $E_c$: dependences satisfied by solutions of Step 12/14
17:   $E \leftarrow E - E_c$; reform the data dependence graph $G = (V, E)$
18:   Compute $H^{\perp}_{S_{max}}$: the null space of $\mathcal{T}_{S_{max}}$
19: until $H^{\perp}_{S_{max}} = \mathbf{0}$ and $E = \emptyset$
OUTPUT The transformation function $\mathcal{T}_S$ for each statement
```

The algorithm is summarized below. Our approach can be viewed as transforming to a tree of permutable loop nests sets or bands – each node of the tree is a good permutable loop nest set. Step 12 of the repeat-until block in Algorithm 1 finds such a band of permutable loops. If all loops are tilable, there is just one node containing all the loops that are permutable. On the other extreme, if no loops are tilable, each node of the tree has just one loop and so no tiling is possible. At least two hyperplanes should be found at any level without dependence removal to enable tiling. Dependences from previously found solutions are thus not removed unless they have to be (Step 17): to allow the next permutable band to be found, and so on. Hence, partially tilable or untilable input is all handled. Loops in each node of the target tree can be stripmined/interchanged when there are at least two of them in it; however, it is illegal to move a stripmined loop across different levels in the tree. There are some intricacies involved here when moving across scalar dimensions: they will be fully described in Chapter 5. When compared to earlier approaches that found maximal sets of permutable loop nests [WL91a, DSV97, LCL99], ours is with an optimization criterion (3.9) that goes beyond maximum degrees of parallelism. Also, [WL91a, DSV97] were applicable to only perfect loop nests.

The theoretical complexity of the algorithm will be discussed in the next chapter after the notion of fusion is sharpened.

3.16 Comparison with Existing Approaches

Tiling and cost functions. Iteration space tiling [IT88, RS92, Xue00, WL91b] is a standard approach for aggregating a set of loop iterations into tiles, with each tile being executed atomically. It is well known that it can improve register reuse, improve locality and minimize communication. Researchers have considered the problem of selecting tile shape and size to minimize communication, improve locality or minimize finish time [ABRY03, BDRR94, HS02, HCF97, HCF99, RS92, SD90, Xue97]. However, these studies were restricted to very simple codes – like single perfectly nested loop nests with uniform dependences and/or sometimes loop nests of a particular depth. To the best of our knowledge, these works have not been extended to more general cases and the cost functions proposed therein not been implemented to allow a direct comparison for those restricted cases. Our work is in the direction of a practical cost function that works for the general case (any polyhedral program or one that can be approximated into it) as opposed to a more sophisticated function for restricted input. With such a function, we are able to keep the problem linear, and since sparse ILP formulations that result here are solved very quickly, we are at a sweet-spot between cost-function sophistication and scalability to real-world programs. Note that our function does not capture tile size optimization, but the results to be presented in Chapter 6 show that decoupling optimization of tile shapes and sizes is a practical and very effective approach.

Ahmed et al. Ahmed et al. [AMP00, AMP01] proposed a framework to optimize imperfectly nested loops for locality. The approach determines the embedding for each statement into a product space, which is then considered for locality optimization through another transformation matrix. Their framework was among the first to address tiling of imperfectly nested loops. However, the heuristic used for minimizing reuse distances is not scalable as described. The reuse distances in the target space for some dependences are set to zero (or a constant) with the goal of obtaining solutions to the embedding function and transformation matrix coefficients. However, there is no concrete procedure to determine the choice of the dependences and the number (which is crucial), and how a new choice is made when no feasible solution is found. Moreover, setting some reuse classes to zero (or a constant) need not completely determine the embedding function or transformation matrix coefficients. Some reuse distances may not be reducible to a constant while some may be. Exploring all possibilities here leads to a combinatorial explosion even for simple imperfect loop nests. Hence, a test-and-set approach for minimizing reuse distances is unlikely to succeed for general affine dependences. With several recent polyhedral approaches, no special embedding or sinking (conversion of an imperfect loop nest to a perfect one) is needed; the original nesting of the code can be discarded once polyhedral domains and dependences have been extracted. The transformation functions finally specify the space-time mapping into a common space that Cloog efficiently scans.

Some specialized works [SL99, YKA04] also exist on tiling a restricted class of imperfectly nested loops for locality. These works are subsumed by approaches employing the polyhedral model. This is without any additional evident complexity in dependence analysis, transformation computation, or code generation with the polyhedral tools for the input those tools handle.

Iteration space slicing [PR97, PR00] and transitive closure are techniques that potentially go beyond affine transformations, i.e., they can partition iteration spaces in a way that is not readily expressible with affine transformations. However, existing slicing techniques [PR00, PR97] do not provide a solution to the key decision problems of – which array to slice and the dimension along which to slice. Besides, the practicality of slicing for coarse-grained parallelization or computation of transitive closure has not yet been demonstrated even for simple affine loop nests.

Loop parallelization has been studied extensively. The reader is referred to the survey of Boulet et al. [BDSV98] for a detailed summary of earlier parallelization algorithms – these restricted the input loop forms, like to perfect loop nests and/or were based on weaker dependence abstractions than exact polyhedral dependences. These include Lamport's seminal hyperplane method [Lam74], Allen and Kennedy's algorithm [AK87] based on dependence levels, Wolf and Lam [WL91a], and Darte et al. [DV97, DSV97]. Automatic parallelization efforts in the polyhedral model broadly fall into two classes: (1) scheduling/allocation-based [Fea92a, Fea92b, Fea94, DR96, Gri04] and (2) partitioning-based [LL98, LCL99, LLL01]. We now compare with previous approaches from both classes.

Pure scheduling-based approaches [Fea92a, Fea92b] are geared towards finding minimum latency schedules or maximum fine-grained parallelism, as opposed to tilability for coarse-grained parallelization with minimized communication and improved locality. Clearly, on most modern parallel architectures, at least one level of coarse-grained parallelism is desired as communication/synchronization costs matter, and so is improving locality.

Griebl et al. Griebl [Gri04] presents an integrated framework for optimizing locality and coarse-grained parallelism with space and time tiling, by enabling time tiling as a post-processing step after a schedule is found. When schedules are used, the inner parallel (space) loops can be readily tiled. In addition, if coarser granularity of parallelism is desired, Griebl's forward communication-only (FCO) constraint finds an allocation satisfying the constraint that all dependences have non-negative affine components along it, i.e., communication will be in the forward direction. Due to this, both the schedule and allocation dimensions become one permutable band of loops that can be tiled. Hence, tiling along the scheduling dimensions (time tiling) is enabled. As shown in this chapter from a theoretical standpoint (Section 3.2.1), fixing schedules as loops does not naturally fit well with tiling. It also adds additional complexity to code generation. Results presented in Chapter 6 also confirm these downsides which cannot be undone regardless of how allocations are found.

Lim and Lam. Lim and Lam's approach [LL98, LCL99] was the first to take a partitioning view with affine functions. They proposed a framework that identifies outer parallel loops (communication-free space partitions) and permutable loops (time partitions) with the goal of maximizing the degree of parallelism and minimizing the order of synchronization. They employ the same machinery to tile for locality and array contraction [LLL01]. There are infinitely many affine transformations that maximize the degree of parallelism and they significantly differ in performance. Whenever there exist no communication-free partitions, their algorithm finds maximally independent solutions to time partitioning constraints without a cost metric to pick good ones. As shown in this chapter, without a cost function, solutions obtained even for simple input may be unsatisfactory with respect to communication cost and locality. It is also not clear how linear independence is ensured between permutable bands belonging to different levels in the hierarchy. Lastly, their algorithm cannot enable non-trivial fusion across sequences of loop nests that fall in different strongly-connected components having a producer-consumer relationship: this is discussed and compared in the next chapter.

Our approach is closer to the latter class of partitioning-based approaches. However, to the best of our knowledge, it is the first to explicitly model good ways of tiling in the transformation framework with an objective function that is applicable to any polyhedral program. Codes which cannot be tiled or only partially tiled are all handled, and traditional transformations are captured. Loop fusion in conjunction with all other transformations is also captured and this is described in the next chapter.

Semi-automatic and iterative techniques. In addition to model-based approaches, semi-automatic and search-based transformation frameworks in the polyhedral model also exist [KP93, KP95, CGP$^+$05, GVB$^+$06, PBCV07, PBCC08]. Cohen et al. [CGP$^+$05] and Girbal et al. [GVB$^+$06] proposed and developed a powerful framework (URUK WRAP-IT) to compose and apply sequences of transformations semi-automatically. Transformations are applied automatically, but specified manually by an expert. A limitation of the recent iterative polyhedral compilation approaches [PBCV07, PBCC08] is that the constructed search space does not include tiling and its integration poses a non-trivial challenge. Though our system now is fully model-driven, empirical iterative optimization would be beneficial on complementary aspects, such as determination of optimal tile sizes and unroll factors, and in other cases when interactions with the underlying hardware and native compiler cannot be well-captured.

3.17 Scope for Extensions

Besides the refinements suggested in Section 3.13, the process for constructing the linearly independent sub-space can be made more comprehensive. This might be possible by adding additional decision variables. In addition, the framework can be made to find negative coefficients for the $\phi's$ – we avoided this to avoid the zero vector solution easily and since the additional benefit of considering negative values is not worth the complexity they bring in. For all the codes tried so far with the implemented system, these practical choices have not caused a problem. The ILP formulations on hyperplane coefficients appear to be very sparse and can benefit from special techniques to solve them much faster [Fea06].

Index set splitting. The transformation our algorithm computes for a statement applies to the entire domain of the statement as was extracted from the input program. Griebl et al. [GFL00] proposed a method to split the original iteration domains based on dependences before space-time mappings are computed. After such a pre-processing phase, a parallelizer may be able to find better mappings with the split domains. Such a dependence-driven index set splitting technique is complementary to our algorithm and could enhance the transformations computed.

CHAPTER 4

Integrating Loop Fusion: Natural Task

In this chapter, we describe how loop fusion is naturally captured in our transformation framework and how finding legal and reasonably good fusion structures is automatically enabled.

Loop fusion involves merging a sequence of two or more loops into a fused loop structure with multiple statements in the loop body. Sequences of producer/consumer loops are commonly encountered in applications, where a nested loop statement produces an array that is consumed in a subsequent loop nest. In this context, fusion can greatly reduce the number of cache misses when the arrays are large - instead of first writing all elements of the array in the producer loop (forcing capacity misses in the cache) and then reading them in the consumer loop (incurring cache misses), fusion allows the production and consumption of elements of the array to be interleaved, thereby reducing the number of cache misses. Figure 4.4 shows an example. For parallel execution, fusion can also reduce the number of synchronizations. Hence, an automatic transformation framework should be able to find good ways of fusion in conjunction with other transformations.

Loop fusion has mainly been studied in an isolated manner by the compiler optimization community [KM93, MS97, SM97, DH99]. Following are the limitations of previous works.

1. Only fusion legality in isolation with other transformations
2. Usually studied fusion with shifts only
3. Simpler dependence models were employed that miss complex fusions that can provide better performance or reduce storage requirement

Often, more complex sequence of transformations are needed that include fusion as one of the transformations. For example, one could permute and shift to enable fusion, and then tile (partially) fused structures. The ability of enumerate fusion structures is also useful.

Fusion strongly interacts with tiling and parallelization. Maximal fusion often kills parallelism, no fusion gives maximal parallelism but can lead to lesser storage optimization or lower cache reuse. Due to complex interactions with other transformations, fusion needs to be studied in an integrated way in a compiler framework for automatic parallelization. For example, it is common for fusion to be legal with some loop shifts. In many scientific codes, fusion is legal after specific permutations. In several of those cases, multiple choices exist and not all of them may be amenable to good parallelization.

It is possible to reason about fusion and drive it completely in the polyhedral model. Traditional literature on loop fusion states that one can distribute two loop nests that were originally fused if there are no "back edges", i.e., no dependences from the second statement to the first. Similarly, one can fuse if it does not add back edges. These legality conditions are already captured with the standard legality constraints in our ILP formulation on the transformation coefficients. Note that we consider all dependences including dependences between SCCs.

Researchers have successfully represented fusion structures through affine transformations: Feautrier [Fea92b], Kelly et al. [KP95, Kel96], Cohen et al and Girbal et al [CGP+05, GVB+06]. However, there is no framework to automatically find such transformations representing good fusions. Our framework can automatically enable good fusion in conjunction with all other transformations. We present three different fusion algorithms that capture the interesting fusion choices in most cases with medium-sized loop nests.

4.1 Automatic Fusion

Let us recall again the structure of our transformation. ϕ_S for a statement was defined as:

$$\phi_S(\vec{i}) \quad = \quad \left(c_1^S \; c_2^S \; \ldots \; c_{m_S}^S\right).\vec{i}_S + \mathbf{0}.\vec{n} + c_0^S$$

where $\vec{i}_S$ is the iterators surrounding the statement in the original program and $\vec{n}$ is the vector of program parameters. ϕ does not have any coefficients for the program parameters $\vec{n}$. Also, note that trivial solutions are avoided (Sec. 3.8) through: $\sum_{i=1}^{m_S} c_i^S \geq 1$ for each statement. Recall also the definition of a scalar dimension, where $\phi_{S_i}^k$ is a constant function for each S_i, i.e., $\left(c_1^S \; c_2^S \; \ldots \; c_{m_S}^S\right) = \mathbf{0}$.

Solving for hyperplanes for multiple statements leads to a schedule for each statement such that all statements in question are *finely* interleaved: this is indeed fusion. The fine interleaving is due to the fact that ϕ does not have coefficients for the program parameters (typically loop bounds). Hence, a common tiling hyperplane also represents a fused loop, and reuse distances between components that are weakly connected can be reduced with our cost function. The set of valid independent hyperplanes that can be iteratively found from our algorithm for multiple statements (at a given depth) is the maximum number of loops that can be fused at that depth.

4.1.1 Implications of minimizing δ for fusion

Recall from (3.9) that $\phi_{S_j}(\vec{t}) - \phi_{S_i}(\vec{s})$ is bounded by $\mathbf{u}.\vec{n} + w$ and minimized at each level iteratively after addition of linear independence constraints.

One can see that the dependence distance is minimized from (1,0,0) to (0,0,1) after transformation and this is lexicographically the minimum. It is obtained since solutions are found iteratively that minimize the dependence distances each level and thus lexicographically minimize the vector of maximum dependence components.

Now, we discuss how the case of non-existence of a fused loop is handled concretely. The code in Figure 4.3 is one example.

A program's data dependence graph (DDG) can always be viewed as a DAG of strongly connected components (SCCs). Let $SCC(i)$ be the i^{th} one in the topological sort of the DAG of SCCs.

```
for (i=1; i<N−2; i++) {
    A[i] = 0.33*(In[i−1]+In[i]+In[i+1]);
}
for (i=2; i<N−3; i++) {
    Out[i] = 0.33*(A[i−1]+A[i]+A[i+1]);
}
```

(a) original code

```
for (t0=1;t0<=min(2,N−3);t0++) {
  A[t0] = 0.33*(In[1+t0]+In[t0]+In[t0−1]);
}
for (t0=3;t0<=N−3;t0++) {
   A[t0] = 0.33*(In[1+t0]+In[t0]+In[t0−1]);
   Out[t0−1] = 0.33*(A[1+t0−1]+A[t0−1]+
                    A[t0−1 −1]);
}
```

(b) Fused code

Figure 4.1: Fusion: simple example

Original	Transformed
$\phi^1_{S1}=0$, $\phi^1_{S2}=1$	$\phi^1_{S1}=i$, $\phi^1_{S2}=i+1$
$\phi^2_{S1}=i$, $\phi^2_{S2}=i$	$\phi^2_{S1}=0$, $\phi^2_{S2}=1$

Level	$\min_{\prec}(\mathbf{u},w)$	
	Original	Transformed
Level 0	1	0
Level 1	0	1

Figure 4.2: Lexicographic $\mathbf{u}.\vec{n}+w$ reduction for fusion

Definition 15 (**Cutting dependences**)**.** Two adjacent nodes in the DAG, SCC(i) and SCC($i+1$), are considered cut at a level m iff $\phi^m_{S_i}=\alpha$, $\forall S_i \in SCC(k), k \le i$, and $\phi^m_{S_j}=\alpha+1$, $\forall S_j \in SCC(k), k > i$, where α is an integer constant.

Consider the sequence of two matrix-vector multiplies in Figure 4.1.1. Applying our algorithm on it first gives us only one solution: $\phi^1_{S1}=i$, $\phi^1_{S2}=l$. This implies fusion of the i loop of S_1 and the j loop of S_2. Putting the linear independence constraint now, we do not obtain any more solutions. As per our algorithm, we now remove the dependences satisfied by the ϕ^1s, and this still does not yield a solution as the loops cannot be fused further. Now, if a scalar dimension is inserted, with $\phi^2_{S_1}=0$, and $\phi^2_{S_2}=1$, it will satisfy any remaining (unsatisfied) dependences between S1 and

```
for (i=0; i<N; i++) {
    s = s + b[i]*c[i]
}
for (i=0; i<N; i++) {
    c[i] = s + a[i];
}
```

Figure 4.3: No fusion possible

S2. This is a cut between S1 and S2 which are SCCs in the DDG by themselves. After the dependences are cut, $\phi^3_{S_1} = j$, and $\phi^3_{S_2} = l$ can be found. The remaining unfused loops are thus placed one after the other as shown in Figure 4.1.1. This generalization of fusion is same as the one proposed in [CGP+05, GVB+06].

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        S1: x[i] = x[i]+a[i,j]*y[j];

for (k=0; k<N; k++)
    for (l=0; l<N; l++)
        S2: y[k] = y[k] + a[k,l]*x[l];
```

```
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        S1: x[i] = x[i]+a[i,j]*y[j];
    }
    for (k=0; k<N; k++) {
        S2: y[k] = y[k]+a[k,i]*x[i];
    }
}
```

Figure 4.4: Two matrix vector multiplies

Now, consider a sequence of three matrix vector multiplies such as

$$y = Ax; \quad z = By; \quad w = Cz$$

Together, these three loop nests do not have a common surrounding loop. However, it is possible to fuse the first two or the last two. When our algorithm is run on all three components, no solutions are found. However, a solution is found if dependences between either the first and second MVs or the second and third MVs is cut.

4.1.2 Dependence cutting schemes

The notion of cutting dependences between SCCs exposes the choice of fusion structures. The algorithm as described is geared towards maximal fusion, i.e., dependences are cut at the deepest level (as a last resort). However, it does not restrict how the following decision is made – which dependences between SCCs to cut when fused loops are not found (Step 14)?. Aggressive fusion may be detrimental to parallelism and increases the running time of the transformation framework in the presence of a large number of statements, while separating all SCCs may give up reuse. To capture most of the interesting cases, we use three different fusion schemes:

1. **nofuse:** Whenever no more common loops are found, all SCCs in the DDG are completely separated out through a scalar dimension, i.e., dependences between all SCCs are cut

2. **smartfuse:** Dependences between SCCs are cut based on the order of reuse of the maximum dimensionality array in an SCC, i.e., SCCs with the same order of reuse are grouped together. If solutions are still not found, SCCs with the same order of reuse are separated based on the maximum dimensionality across

all statement domains in an SCC. This scheme ultimately falls back to *nofuse* if the above does not succeed in finding any fusable loops.

3. **maxfuse:** Dependences are cut very conservatively, between exactly two SCCs at a time. The scheme falls back to *smartfuse* and ultimately to *nofuse* in the presence of no fusion at all.

Our notion of a cut is equivalent to introducing a parametric shift to separate loops. Loop shifting was used for parallelization and fusion with a simplified representation of dependences and transformations by Darte et al. [DH99, DH00]. Shifting was used for correcting illegal loop transformations by Vasilache et al. [VCP07]. Example 4.4.4 explains how more sophisticated transformations can be enabled than with techniques based purely on loop shifting.

The *smartfuse* and *maxfuse* heuristics have a tendency to introduce a large number of scalar dimensions with many of them being redundant, i.e., a continuous set of them can be collapsed into just one scalar dimension and polyhedral operations on such dimensions anyway are trivial. Thanks to code generation optimizations in Cloog that remove such scalar dimensions [VBC06].

4.2 Correctness and Completeness

We now prove that Algorithm 2 terminates successfully with a legal transformation.

Theorem 2. *A transformation is always found by Algorithm 2*

Algorithm 2 Pluto algorithm (with fusion heuristics)

```
INPUT Data Dependence graph $G = (V, E)$ with dependence polyhedra, $\mathcal{P}_e$, $\forall e \in E$
 1: $S_{max}$: statement with maximum domain dimensionality
 2: for each dependence $e \in E$ do
 3:     Build legality constraints as in Algorithm 1
 4:     Build bounding function constraints as in Algorithm 1
 5:     Aggregate constraints from both into $C_e$
 6: end for
 7: if fusion heuristic is nofuse or smartfuse then
 8:     Cut dependences between SCCs appropriately (Section 4.1.2)
 9: end if
10: repeat
11:     $C = \emptyset$
12:     for each dependence edge $e \in E$ do
13:         $C \leftarrow C \cup C_e$
14:     end for
15:     Compute lexicographic minimal solution with $u's$ coefficients in the leading
        position followed by $w$ to iteratively find independent solutions to $C$
16:     if no solutions were found then
17:         Cut dependences between strongly-connected components in the GDG using
            one of the cutting schemes: {nofuse, smartfuse, maxfuse}
18:     end if
19:     Compute $E_c$: dependences satisfied by solutions of Step 15 and 17
20:     $E \leftarrow E - E_c$; reform $G = (V, E)$
21:     Compute $H^{\perp}_{S_{max}}$: the null space of $\mathcal{T}_{S_{max}}$
22: until $H^{\perp}_{S_{max}} = \mathbf{0}$ and $E = \emptyset$
OUTPUT The transformation function $\mathcal{T}_S$ for each statement
```

Proof. We show that the termination condition (Step 22) of Algorithm 2 is always reached. Firstly, every strongly-connected component of the dependence graph has at least one common surrounding loop, if the input comes from a valid computation. Hence, Step 15 is guaranteed to find at least one solution when all dependences between strongly-connected components have eventually been cut (iteratively in Step 17 whenever solutions are not found). Hence, enough linearly independent solutions are found for each statement such that $H_S^{\perp}$ eventually becomes $\mathbf{0}_{m_S \times m_S}$ for every $S \in V$, i.e., H_S becomes full-ranked for each statement. Now, we show that the condition $E = \emptyset$ is also eventually satisfied. Let us consider the two groups of dependences: (1) self-edges (or intra-statement dependences), and (2) inter-statement dependences. Since H_S becomes full-ranked and does not have a null space, all dependent iterations comprising a self-edge are satisfied at one level or the other (since $\phi(\vec{t}) - \phi(\vec{s}) \geq 0$ stays in the formulation till satisfaction). Now, consider an inter-statement dependence from S_i to S_j. If at Step 17, the dependences between S_i and S_j were cut, all unsatisfied dependences between S_i and S_j will immediately be satisfied at the scalar dimension introduced in the transformation matrices (since ϕ_{S_j} is set to one and ϕ_{S_i} to zero). However, if S_i and S_j belong to the same strongly-connected component, then a solution will be found at Step 15, and eventually they will belong to separate strongly-connected components and dependences between them will be cut (if not satisfied). Hence, both intra and inter-statement dependences are eventually satisfied, and the condition $E = \emptyset$ is met.□

Theorem 3. *The transformation found by Algorithm 2 is always legal.*

Proof. Given the proof for Theorem 2, the proof for legality is straightforward. Since we keep $\phi_{S_j}(\vec{t}) - \phi_{S_i}(\vec{s}) \geq 0$, $\langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_e$ in the formulation till $e^{S_i \Rightarrow S_j}$ is satisfied, no dependence is violated. The termination condition for the repeat-until block (Step 22) thus ensures that all dependences are satisfied. Hence, the transformations found are always legal.□

4.2.1 Bound on iterative search in Algorithm 2

Each iteration of the repeat-until block in Algorithm 2 incurs a call to PIP. The number of times this block executes depends on how dependences across strongly-connected components are handled in Step 17. Consider one extreme case when all dependences between any two strongly-connected components are cut whenever no solutions are found (*nofuse*): then, the number of PIP calls required is $2d+1$ at worst, where d is the depth of the statement with maximum dimensionality. This is because, in the worst case, exactly one solution is found at Step 15, and the rest of the d times dependences between all SCCs are cut (both happen in an alternating fashion); the last iteration of the block adds a scalar dimension that specifies the ordering of the statements in the innermost loop(s). Now, consider the other extreme case, when dependences are cut very conservatively between SCCs; in the worst case, this would increase the number of iterations of the block by the number of dependences in the program.

4.3 Theoretical Complexity of Computing Transformations

Integer linear programming is known to be NP-hard. The PIP solver employed here uses the dual-simplex algorithm with Gomory's cutting plane method [Sch86] to obtain the best integer solution. The Simplex algorithm is known to have an average-case complexity of O(n^3) and is very efficient in practice, even though its worst case complexity is exponential. In particular, it is important to note that the ILP formulations that result out of affine scheduling or partitioning in the polyhedral model are very simple and sparse, and are quickly solved. The following are the key input sizes.

n:	Sum of the dimensionalities of the domains of all statements
$\|E\|$:	Number of dependences (edges in the DDG)
m:	Depth of the statement with maximum domain dimensionality

Our algorithms needs $\Theta(m)$ to $\Theta(|E|)$ number of calls to PIP depending on the fusion heuristic. Assuming *smartfuse*, the theoretical complexity is typically $O(n^3 * m)$. This does not yet include the complexity of building the legality and cost function constraints which is a one-time process proportional to the number of dependences. Elimination of the Farkas multipliers is done dependence-wise, and therefore on systems with dimensionality proportional to nesting depth. Usually, Fourier-Motzkin elimination on these systems leads to a reduction in the number of inequalities, hence, its super-exponential theoretical complexity is of little importance for our problems. The complexity involved in building the constraints can be written as $O(|E| * m^3)$, since $\Theta(m)$ Farkas multipliers are to be eliminated from systems with (m) constraints. This is assuming that dependence polyhedra have (m) inequalities. Overall, the average-case complexity appears to be $O(n^3 * m + |E| * m^3)$.

Code	Number of statements	Number of loops	Number of dependences	Computation time
2-d Jacobi	2	6	20	0.033s
Haar 1-d	3	5	12	0.018s
LU	2	5	10	0.011s
TCE 4-index	4	20	15	0.095s
2-d FDTD	4	11	37	0.061s
Swim	57	109	633	2.8s

Table 4.1: Time to compute transformations as of Pluto version 0.3.1 on an Intel Core2 Quad Q6600

Table 4.3 shows the time taken compute transformations for several kernels.

4.4 Examples

We now give several examples to show non-trivial fusion structures are achieved from our framework. The transformed code for all of these examples have automatically been generated from Pluto with either the *maxfuse* or *smartfuse* heuristic.

4.4.1 Sequence of matrix-matrix multiplies

For the sequence of matrix-matrix multiplies in Figure 4.5, each of the original loop nests can be parallelized, but a synchronization is needed after the first loop nest is executed. The transformed loop nest has one outer parallel loop ($t0$), but reuse is improved as each element of matrix C is consumed immediately after it is produced.

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
      S1: C[i,j] = C[i,j] + A[i,k] * B[k,j]
    }
  }
}
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
      S2: D[i,j] = D[i,j] + E[i,k] * C[k,j]
    }
  }
}
```

```
for (t0=0;t0<=N−1;t0++) {
  for (t1=0;t1<=N−1;t1++) {
    for (t3=0;t3<=N−1;t3++) {
      C[t1][t0]=A[t1][t3]*B[t3][t0]+C[t1][t0];
    }
    for (t3=0;t3<=N−1;t3++) {
      D[t3][t0]=E[t3][t1]*C[t1][t0]+D[t3][t0];
    }
  }
}
```

Transformed code

	$S1$	$S2$
ϕ^1	j	j
ϕ^2	i	k
ϕ^3	0	1
ϕ^4	k	i

Figure 4.5: Sequence of MMs

C can be contracted to a single scalar. This transformation is cannot be obtained from existing frameworks. As per the algorithm, though ϕ^1, ϕ^2 fall in one permutable band, and ϕ^4 in another, it is still possible to create 3-d tiles for both MMs with an optimization discussed in the next chapter.

4.4.2 Multiple statement stencils

This code (Figure 4.6) is representative of multimedia applications. The transformed code enables immediate reuse of data produced by each statement at the next statement.

```
for (i=1; i<n−1; i++) {
  a1[i] = a0[i−1] + a0[i] + a0[i+1];
}
for (i=1; i<n−1; i++) {
  a2[i] = a1[i−1] + a1[i] + a1[i+1];
}
for (i=1; i<n−1; i++) {
  a3[i] = a2[i−1] + a2[i] + a2[i+1];
}
for (i=1; i<n−1; i++) {
  a4[i] = a3[i−1] + a3[i] + a3[i+1];
}
for (i=1; i<n−1; i++) {
  a5[i] = a4[i−1] + a4[i] + a4[i+1];
}
```

(a) original code

```
for (c1=1;c1<=min(2,n−2);c1++) {
  a1[c1]=a0[1+c1]+a0[c1]+a0[c1−1];
}
for (c1=3;c1<=min(4,n−2);c1++) {
  a1[c1]=a0[1+c1]+a0[c1]+a0[c1−1];
  a2[c1−1]=a1[1+c1−1]+a1[c1−1]+a1[c1−1 −1];
}
for (c1=5;c1<=min(6,n−2);c1++) {
  a1[c1]=a0[1+c1]+a0[c1]+a0[c1−1];
  a2[c1−1]=a1[1+c1−1]+a1[c1−1]+a1[c1−1 −1];
  a3[c1−2]=a2[1+c1−2]+a2[c1−2]+a2[c1−2 −1];
}
for (c1=7;c1<=min(8,n−2);c1++) {
  a1[c1]=a0[1+c1]+a0[c1]+a0[c1−1];} ;
  a2[c1−1]=a1[1+c1−1]+a1[c1−1]+a1[c1−1 −1];
  a3[c1−2]=a2[1+c1−2]+a2[c1−2]+a2[c1−2 −1];
  a4[c1−3]=a3[1+c1−3]+a3[c1−3]+a3[c1−3 −1];
}
for (c1=9;c1<=n−2;c1++) {
  a1[c1]=a0[1+c1]+a0[c1]+a0[c1−1];
  a2[c1−1]=a1[1+c1−1]+a1[c1−1]+a1[c1−1 −1];
  a3[c1−2]=a2[1+c1−2]+a2[c1−2]+a2[c1−2 −1];
  a4[c1−3]=a3[1+c1−3]+a3[c1−3]+a3[c1−3 −1];
  a5[c1−4]=a4[1+c1−4]+a4[c1−4]+a4[c1−4 −1];
}
```

(b) Fused code

	$S1$	$S2$	$S3$	$S4$	$S5$
$\phi^1(i)$	i	$i+1$	$i+2$	$i+3$	$i+4$
$\phi^2(i)$	0	1	2	3	4

Figure 4.6: Stencils involving multiple statements

4.4.3 CCSD (T)

The code is shown in Figure 4.7. Fusion can greatly reduce the size of the arrays here. In transformed code, both X and Y which were originally 6-d arrays, have been contracted to scalars.

```
e1 = 0;
e2 = 0;

for (a=0;a<V;a++)
  for (b=0;b<V;b++)
    for (c=0;c<V;c++)
      for (e=0;e<V;e++)
        for (i=0;i<O;i++)
          for (j=0;j<O;j++)
            for (k=0;k<O;k++)
              for (m=0;m<O;m++)
                X[a][b][c][i][j][k] = X[a][b][c][i][j][k] +
                   T2[a][b][k][m]*O1[c][m][i][j] + T2[c][e][i][j]*O2[a][b][e][k];

for (a=0;a<V;a++)
  for (b=0;b<V;b++)
    for (c=0;c<V;c++)
      for (i=0;i<O;i++)
        for (j=0;j<O;j++)
          for (k=0;k<O;k++)
            Y[a][b][c][i][j][k] = T1[c][k]*O3[a][b][i][j];

for a, b, c, i, j, k
            e1 = e1 + X[a][b][c][i][j][k]*X[a][b][c][i][j][k];

for a, b, c, i, j, k
            e2 = e2+ X[a][b][c][i][j][k]*Y[a][b][c][i][j][k];
```

Figure 4.7: CCSD (T) code

```
for (c1=0; c1<=V−1; c1++) {
  for (c2=0; c2<=V−1; c2++) {
    for (c3=0; c3<=V−1; c3++) {
      for (c4=0; c4<=O−1; c4++) {
        for (c5=0; c5<=O−1; c5++) {
          for (c6=0; c6<=O−1; c6++) {
            Y = T1[c3][c6]∗O3[c1][c2][c4][c5];
            for (c8=0; c8<=V−1; c8++) {
              for (c9=0; c9<=O−1; c9++) {
                X = X + T2[c3][c8][c4][c5]∗O2[c1][c2][c8][c6] +
                          T2[c1][c2][c6][c9]∗O1[c3][c9][c4][c5];
              }
            }
            e1 = e1 + X∗X;
            e2 = e2 + X∗Y;
          }
        }
      }
    }
  }
}
```

Figure 4.8: CCSD (T): Pluto transformed code: maximally fused. X and Y have been reduced to scalars from 6-dimensional arrays

4.4.4 TCE four-index transform

This is a sequence of four loop nests, each of depth five (Figure 4.9), occurring in Tensor Contraction Expressions that appear in computational quantum chemistry problems [CS90]. Our tool transforms the code as shown in Figure 4.10, where the producing/consuming distances between the loops have been reduced. One of the dimensions of arrays T1, T3 can now be contracted. There are other maximal fusion structures that can be enumerated, but we do not show them due to space constraints.

It is extremely tedious to reason about the legality of such a transformation manually. With a semi-automatic framework accompanied with loop shifting to automatically correct transformations [VCP07], such a transformation cannot be found unless the expert has applied the right permutation on each loop nest before fusing them. In this case, correction purely by shifting after straightforward fusion will introduce shifts at the outer levels itself, giving up reuse opportunity.

4.4.5 GEMVER

The GEMVER kernel is a combination of outer products and matrix vector products. It is used for householder bidiagonalization. The BLAS version of the GEMVER kernel from Siek et al. [SKJ08] along with the linear algebraic specification is shown in Figure 4.11. A, B are matrices, while x, y, w, $u1$, $v1$, $u2$, $v2$ are vectors and α, β, γ are scalars. The nested loop code is in Figure 4.12. Permuting and fusing the first two loop nests is the key. Figure 4.13 shows the performance comparison. More detailed results will be presented in Chapter 6.

4.5 Past Work on Fusion

Traditional works on loop fusion [KM93, MS97, SM97, QK06] are restricted in their ability to find complex fusion structures. This is mainly due to the lack of a powerful representation for dependences and transformations. Hence, the non-polyhedral approaches typically study fusion in an manner isolated with other transformations. This is the case for several kernels discussed in Section 4.4 and for which results will be presented in Chapter 6.

```
for (a=0; a<N; a++)
  for (q=0; q<N; q++)
    for (r=0; r<N; r++)
      for (s=0; s<N; s++)
          for (p=0; p<N; p++)
            T1[a][q][r][s] = T1[a][q][r][s] + A[p][q][r][s]*C4[p][a];

for (a=0; a<N; a++)
  for (b=0; b<N; b++)
    for (r=0; r<N; r++)
      for (s=0; s<N; s++)
        for (q=0; q<N; q++)
          T2[a][b][r][s] = T2[a][b][r][s] + T1[a][q][r][s]*C3[q][b];

for (a=0; a<N; a++)
  for (b=0; b<N; b++)
    for (c=0; c<N; c++)
      for (s=0; s<N; s++)
        for (r=0; r<N; r++)
          T3[a][b][c][s] = T3[a][b][c][s] + T2[a][b][r][s]*C2[r][c];

for (a=0; a<N; a++)
  for (b=0; b<N; b++)
    for (c=0; c<N; c++)
      for (d=0; d<N; d++)
        for (s=0; s<N; s++)
          B[a][b][c][d] = B[a][b][c][d] + T3[a][b][c][s]*C1[s][d];
```

Figure 4.9: TCE 4-index transform (original specification)

```
for (c1=0;c1<=N−1;c1++) {
  for (c2=0;c2<=N−1;c2++) {
    for (c4=0;c4<=N−1;c4++) {
      for (c5=0;c5<=N−1;c5++) {
        for (c8=0;c8<=N−1;c8++) {
          {T1[c1][c5][c4][c2]=A[c8][c5][c4][c2]∗C4[c8][c1]+T1[c1][c5][c4][c2];} ;
        }
        for (c8=0;c8<=N−1;c8++) {
          {T2[c1][c8][c4][c2]=T1[c1][c5][c4][c2]∗C3[c5][c8]+T2[c1][c8][c4][c2];} ;
        }
      }
    }
    for (c4=0;c4<=N−1;c4++) {
      for (c5=0;c5<=N−1;c5++) {
        for (c8=0;c8<=N−1;c8++) {
          {T3[c1][c4][c5][c2]=T2[c1][c4][c8][c2]∗C2[c8][c5]+T3[c1][c4][c5][c2];} ;
        }
        for (c8=0;c8<=N−1;c8++) {
          {B[c1][c4][c5][c8]=T3[c1][c4][c5][c2]∗C1[c2][c8]+B[c1][c4][c5][c8];} ;
        }
      }
    }
  }
}
```

Figure 4.10: Transformed TCE code (tiling and parallelization is not shown): T1, T3 can be contracted to scalars, while T2 and B to a 2-d array and a 3-d array respectively

$$\begin{aligned} B &= A + u_1 v_1^T + u_2 v_2^T \\ x &= \beta B^T y + z \\ w &= \alpha B x \end{aligned}$$

```
dcopy(m ∗ n, A, B, 1);
dger(m, n, 1.0, u1, 1, v1 , 1, B, m);
dger(m, n, 1.0, u2, 1, v2 , 1, B, m);
dcopy(n, z, x, 1);
dgemv('T', m, n, beta, B, m, y, 1, 1.0, x, 1);
dgemv('N', m, n, alpha, B, m, x, 1, 0.0, w, 1);
```

(b) BLAS version

Figure 4.11: The GEMVER kernel

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        B[i][j] = A[i][j] + u1[i]*v1[j] + u2[i]*v2[j];

for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        x[i] = x[i] + beta* B[j][i]*y[j];

for (i=0; i<N; i++)
    x[i] = x[i] + z[i];

for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        w[i] = w[i] + alpha* B[i][j]*x[j];
```

Figure 4.12: GEMVER nested loop code

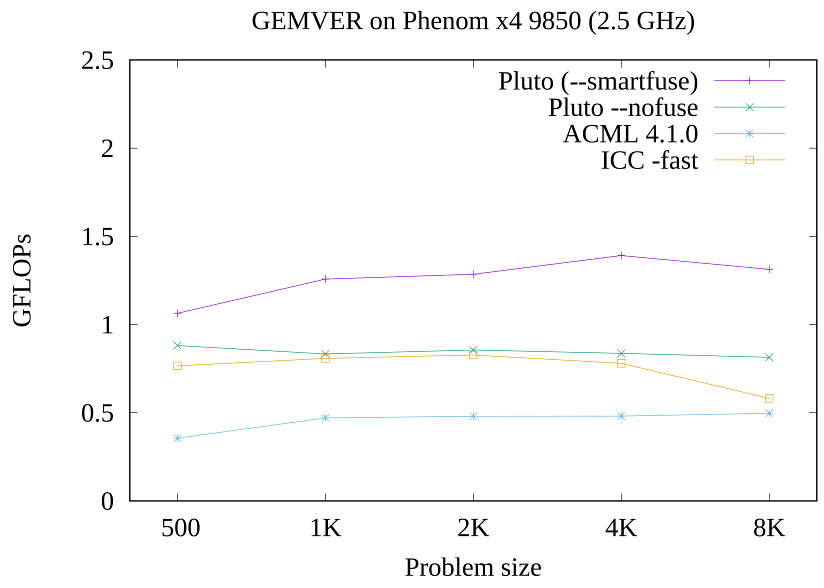


Figure 4.13: GEMVER performance: preview (detailed results in Chapter 6)

Darte et al. [DSV97, DH00] study fusion with parallelization, but only in combination with shifting. Our work on the other hand enables fusion in the presence of all polyhedral transformations like permutation and skewing. Megiddo and Sarkar [MS97] proposed a way to perform fusion for an existing parallel program by grouping components in a way that parallelism is not disturbed. Decoupling parallelization and fusion clearly misses several interesting solutions that would have been captured if the legal fusion choices were itself cast into their framework.

Siek et al. [SKJ08] built a domain specific compiler for dense linear algebra kernels; parallelization has not yet been reported by the authors. In comparison, our framework applies to all polyhedral input that encompass linear algebra kernels, and parallelization issues are all naturally handled in an integrated fashion. Darte [Dar00] studies the complexity of several variations of the loop fusion problem. The cutting choice that arises in our algorithm is of exponential complexity in the number of SCCs if one wishes to try all possibilities and choose the best based on our cost function or any other cost model. None of our heuristics explore all possible choices, but just the ones that seem to be interesting.

Affine scheduling-based approaches are geared towards finding minimum latency schedules or maximum fine-grained parallelism. However, a schedule specifying maximum parallelism need not finely interleave operations of different statements. Hence, works based on such schedules [BF03, Gri04] do not readily enable fusion.

Lim et al.'s [LCL99] affine partitioning algorithm, like our algorithm, would allow maximal fusion within an SCC. However, their algorithm does not enumerate or present a choice of fusion structures across SCCs as they treat each SCC independently. Some treatment of optimizing across two neighboring SCCs exists [LCL99] using constraints for near-neighbor communication, but the approach as presented is a test-and-set approach and is not automatable in general. Also, whether it is within an SCC or across SCCs, the choice of the partitions would have the same downsides as those described in Section 3.16 whenever communication-free partitions do not exist. For the same reasons, their framework for array contraction [LLL01] though may be successful in contracting arrays whenever it independent partitions can be found, it cannot be employed when storage along an array dimension can be shrunk to a small constant. Due to minimization of δ_es, our algorithm can enable that.

Ahmed et al. [AMP01] proposed a framework to tile imperfectly nested loops for locality. Their embedding functions can also capture fusion. However, limitations of their reuse distance minimization heuristic discussed in detail in Section 3.16 will have a direct impact on the ability to find good fusions. Also, there is no procedure to expose fusion structure choices, as is possible with our cutting schemes.

The URUK/WRAP-IT framework [CGP$^+$05, GVB$^+$06] can be used to manually specify fusion and its legality can be checked. With the automatic correction scheme proposed by Vasilache et al. [VCP07], any illegal fusion can be corrected by introducing shifts, at multiple levels if needed. However, this cannot enable, for eg., a permute and fuse, unless the expert specifying the transformation provides the right permutation to start with.

In this chapter, we proposed three different fusion heuristics that cover the diversity in choice of available fusion structures for loop nest sequences of medium size. Detailed experimental results are presented in Chapter 6. Overall, to the best of our knowledge, ours is the first work to treat fusion in an integrated manner in a transformation framework for automatic parallelization.

CHAPTER 5

The PLUTO Parallelization System

In this chapter, we first provide an overview of the new end-to-end parallelization system that we developed. Then, issues involved in generation of actual tiled and parallel code are discussed. Lastly, some complementary post-processing transformations are discussed.

5.1 Overview of PLUTO

The transformation framework described in Chapter 3 and Chapter 4 has been implemented into a tool, PLUTO [Plu]. Figure 5.1 shows the entire tool-chain. We used the scanner, parser and dependence tester from the LooPo project [Loo]. We used PipLib 1.3.6 [PIP, Fea88] as the ILP solver to find lexicographic minimal solution for 3.9. Note that the parametric functionality of PIP is not utilized, so any ILP solver can be used here. Cloog 0.14.1 [Clo] was used for code generation. The transformation framework takes as input, polyhedral domains and dependence polyhedra from LooPo's dependence tester, computes transformations and provides it to Cloog.

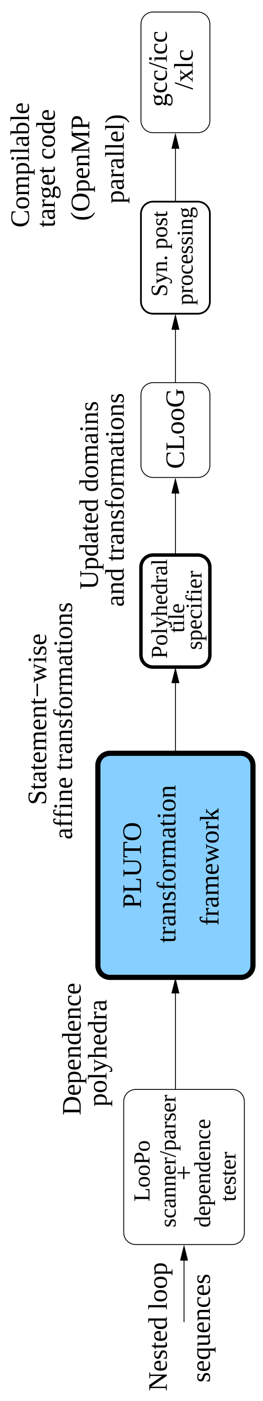

Nested loop sequences
LooPo scanner/parser + dependence tester
Dependence polyhedra
PLUTO transformation framework
Statement−wise affine transformations
Polyhedral tile specifier
Updated domains and transformations
CLooG
Syn. post processing
Compilable target code (OpenMP parallel)
gcc/icc /xlc

Figure 5.1: The PLUTO source-to-source transformation system

Compilable OpenMP parallel code is finally output after some post-processing on the Cloog code.

LooPo's dependence tester can provide flow, anti, output, and input dependences. However, if techniques to remove false dependences are applied, our framework can work with those reduced set of dependences too. The removal of transitively covered dependences is desired but not necessary, and techniques for the same exist [VBGC06, Vas07]. LooPo's tester employs the PIP library [Fea88] to test for the existence of an integer solution to a system of linear equalities and inequalities. In addition, it can also compute the last conflicting access for a given dependence that is expressed through an affine function called the *h-transformation*, which gives the last source iteration as a function of the target iteration, i.e., $\vec{s} = h_e(\vec{t})$. The equalities representing the h-transformation are included in the dependence polyhedron. The h-transformation is meaningful for all but the anti-dependences (Chapter 5.2.2 of [Gri04]). For RAW and WAW dependences, providing the h-transformation can be viewed as freeing transitive dependences from a given dependence polyhedron.

A brief description of CLooG. The affine functions, ϕ or $\mathcal{T}_S$, are called *scattering functions* in the specification of Cloog. Cloog [Clo, Bas04a] can scan a union of polyhedra, and optionally, under a new global lexicographic ordering specified as through scattering functions. Scattering functions are specified statement-wise, and the legality of scanning the polyhedron with these dimensions in the particular order should be guaranteed by the specifier – an automatic transformation system in our case. The code generator does not have any information on the dependences and hence, in the absence of any scattering functions would scan the union of the statement polyhedra in the global lexicographic order of the original iterators (statement instances are interleaved). Cloog uses Quilleré separation [QRW00] with several improvements made by Bastoul [Bas04a]. Its operations are based on PolyLib [Wil93, Pol], which in turn uses the Chernikova algorithm [LeV92]) to move between the vertex form of a polyhedron and its face form [Wil93]. The Chernikova algorithm is guaranteed to give an irredundant set of linear inequalities from the vertex form. The code generated is more efficient than that by older code generators based on Fourier-Motzkin variable elimination like Omega Codegen [KPR95] or LooPo's internal code generator [GLW98, Gri04]). Also, code generation time and memory utilization are much lower [Bas04a] besides making it feasible for cases it was earlier thought not to be. Such a powerful and efficient code generator is essential in conjunction with the transformation framework we developed. When tiling is expressed with the transformations we compute, generating code is a challenging task. In particular, Cloog's ability to optimize control for user-desired levels (-f/-l options) – trading code size with conditional depth control make code generation feasible in the presence of a large number of statements or under complex transformations for coarse-grained parallelization.

5.2 Tiled Code Generation under Transformations

In this section, we describe how tiled code is generated from transformations found by the algorithm in the previous chapter. This is an important step in generation of high performance code.

5.2.1 Syntactic tiling versus tiling scattering functions

Before proceeding further, we differentiate between using the term 'tiling' for, (1) modeling and enabling tiling through a transformation framework (as was described in the previous chapter), (2) final generation of tiled code from the hyperplanes found. Both are generally referred to as tiling. Our approach models tiling in the transformation framework by finding affine transformations that make rectangular tiling in the transformed space legal. The hyperplanes found are the new *basis* for the loops in the transformed space and have special properties that have been detected when the transformation is found – e.g. being parallel, sequential or belonging to a band of loops that can now be rectangularly tiled. Hence, the transformation framework guarantees legality of rectangular tiling in the new space. The final generation of tiled loops can be done in two ways broadly, (1) directly through the polyhedral code generator itself in one pass itself, or (2) as a post-pass on the abstract syntax tree generated after applying the transformation. Each has its merits and both can be combined too.

For transformations that possibly lead to imperfectly nested code, tiling the scattering functions is a natural way to get tiled code in one pass through the code generator. Consider the code in Figure 5.2.2(a) for example. If code is generated by just applying the transformation first, we get code shown in Figure 5.2.2(b). Even though the transformation framework obtained two tiling hyperplanes, the transformed code in Figure 5.2.2(b) has no 2-d perfectly nested kernel, and tiling it syntactically is illegal (fully distributing the two statements is also legal). The legality of syntactic tiling or unroll/jam (for register tiling) of such loops cannot be reasoned about in the target AST easily since we are typically out of the polyhedral model once we obtain the loop nests. We describe in the next section how tiled code can be obtained in one pass through the code generator by just updating the domains and scattering functions without knowing anything about the AST. For example, for the code in Figure 5.2.2(a), 2-d tiled code generated would be as shown in Figure 5.3.

5.2.2 Tiles under a transformation

Our approach to tiling is to specify a modified higher dimensional domain and specify transformations for what would be the tile space loops in the transformed space. Consider a very simple example: a two-dimensional loop nest with original iterators: i and j. Let the transformation found be $c_1 = i$, and $c_2 = i + j$, with c_1, c_2 constituting a permutable band; hence, they can be blocked leading to 2-d tiles. We would like to obtain target code that is tiled rectangularly along c_1 and c_2. The domain supplied to the code generator is a higher dimensional domain with the tile shape constraints like that proposed by Ancourt and Irigoin [AI91]; but the scatterings are duplicated for the tile space too. 'T' subscript is used to denote the corresponding tile space iterator. The tile space and intra tile loop scattering functions are specified as follows. The new domain is given by:

$$0 \le i \le N-1$$
$$0 \le j \le N-1$$
$$0 \le i - 32i_T \le 31$$
$$0 \le (i+j) - 32(i_T + j_T) \le 31$$

The scattering function is given by: $(c_{1T}, c_{2T}, c_{3T}, c_1, c_2, c_3) = (i_T, i_T + j_T, i, i+j)$

Algorithm 3 Tiling for multiple statements under transformations

```
INPUT Hyperplanes comprising a tilable band of width $k$: $\phi_S^i, \phi_S^{i+1}, \ldots, \phi_S^{i+k-1}$; original domains ($\mathcal{D}^S$), transformations ($\mathcal{T}_S$), tile sizes: $\tau_i, \tau_{i+1}, \ldots, \tau_{i+k-1}$
 1: /* Update the domains */
 2: for each statement S do
 3:     for each $\phi_S^j = \mathbf{f}^j(\vec{i}_S) + f_0$ do
 4:         Increase the domain ($\mathcal{D}^S$) dimensionality by creating supernodes for all original iterators that appear in $\phi_S^j$
 5:         Let the supernode iterators be $\vec{i}_T$
 6:         Add the following two constraints to $\mathcal{D}^S$:
            $\tau_j * \mathbf{f}^j(\vec{i}_{T_S}) \le \mathbf{f}^j(\vec{i}_S) + f_0^j$
            $\mathbf{f}^j(\vec{i}_S) + f_0^j \le \tau_j * \mathbf{f}^j(\vec{i}_{T_S}) + \tau_j - 1$
 7:     end for
 8: end for
 9: /* Update the transformation functions */
10: for each statement $S$ do
11:     Add $k$ new rows to the transformation of $S$ at level $i$
12:     Add as many columns as the number of supernodes added to $\mathcal{D}^S$ in Step 4
13:     for each $\phi_S^j = \mathbf{f}^j(\vec{i}_S) + f_0^j$, $j = i, \ldots, i+k-1$ do
14:         Create a supernode: $\phi_{T_S}^j = \mathbf{f}^j(\vec{i}_{T_S})$
15:     end for
16: end for
OUTPUT Updated domains ($\mathcal{D}^S$) and transformations ($\mathcal{T}_S$)
```

The higher-dimensional tile space loops are referred to as supernodes in the description. i_T, j_T are supernodes in the original domain, while c_{1T}, c_{2T} are supernodes in the transformed space. We will refer to the latter as *scattering supernodes*. With this, we formally state the algorithm to update the domains and transformations (Algorithm 3).

Theorem 4. *The set of scattering supernodes, $\phi^i_{T_S}$, $\phi^{i+1}_{T_S}$, ..., $\phi^{i+k-1}_{T_S}$ obtained from Algorithm 3 satisfy the tiling legality condition (3.3)*

Since, ϕ^j_S, $i \leq j \leq i+k-1$ satisfy (3.3) and since the supernodes step through an aggregation of parallel hyperplane instances, dependences have non-negative components along the scattering supernode dimensions too. This holds true for both intra and inter-statement dependences. $\{\phi^j_{T_{S_1}}, \phi^j_{T_{S_2}}, \ldots, \phi^j_{T_{S_n}}\}$ thus represent a common supernode dimension in the transformed space with the same property as that of $\{\phi^j_{S_1}, \phi^j_{S_2}, \ldots, \phi^j_{S_n}\}$. □

Figure 5.2.2 shows tiles for imperfectly nested 1-d Jacobi. Note that tiling it requires shifting S2 by one and skewing the space loops by a factor of two with respect to time, as opposed to skewing by a factor of one that is required for the single-assignment perfectly nested variant in Figure 5.2.2(a). The update statement causes the increase in skewing factor. Figure 5.3 shows the tiled code. One can notice a 2-d perfectly nested hotspot in the code which corresponds to the so-called full tiles shown in Figure 5.2.2. Such a perfectly nested hotspot stands out in the imperfectly nested code. Further optimizations like register tiling can be performed on this portion of the code syntactically. Alternatively, separation of such a full tile can be accomplished with much more compact code by modifying the code generator itself [KRSR07].

Example: 3-d tiles for LU The transformation obtained for the LU decomposition code shown in Figure 5.7 is:

$$\mathcal{T}_{S1} : \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} k \\ j \end{pmatrix} \qquad \mathcal{T}_{S2} : \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} k \\ i \\ j \end{pmatrix}$$

Hyperplanes c_1, c_2 and c_3 are identified as belonging to one tilable band. The domains are specified as:

$$\begin{array}{cc} \text{S1} & \text{S2} \\ 0 \le k \le N-1 & 0 \le k \le N-1 \\ k+1 \le j \le N-1 & k+1 \le i \le N-1 \\ & k+1 \le j \le N-1 \\ 0 \le k - 32k_T \le 31 & 0 \le k - 32k_T \le 31 \\ 0 \le j - 32j_T \le 31 & 0 \le i - 32i_T \le 31 \\ & 0 \le j - 32j_T \le 31 \end{array}$$

The scattering functions for S1 and S2 are given by:

$$\mathcal{T}_{S1} : (c_{1T}, c_{2T}, c_{3T}, c_1, c_2, c_3) = (k_T, j_T, k_T, k, j, k)$$

$$\mathcal{T}_{S2} : (c_{1T}, c_{2T}, c_{3T}, c_1, c_2, c_3) = (k_T, j_T, i_T, k, j, i)$$

Note that tile sizes for c_1 and c_3 need to be the same since they correspond to the same dimension in the original domain of $S1$. The tiled code after applying the parallelization technique described in the next sub-section is shown in the next chapter (Figure 6.8).

Tiling multiple times. The same tiling hyperplanes can be used to tile multiple times (due to Theorem 4), for registers, L1, L2 caches, and for parallelism, and the

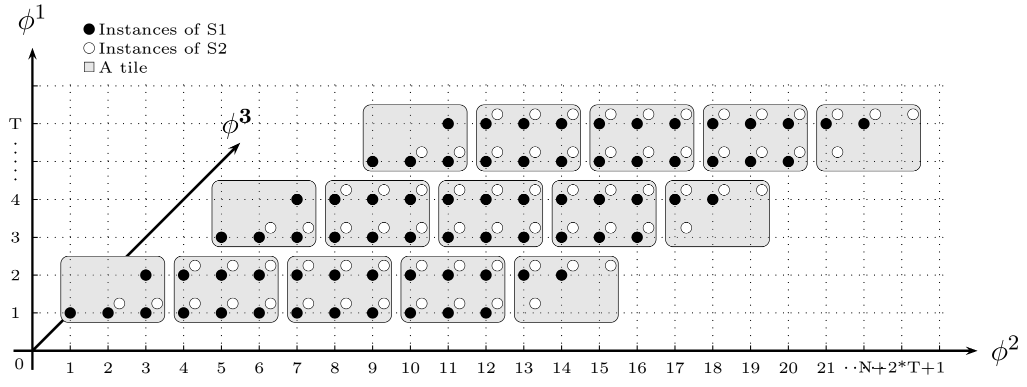
```
for (t=0; t<T; t++) {
  for (i=2; i<N−1; i++) {
    b[i] = 0.333*(a[i−1] + a[i] + a[i+1]);
  }
  for (j=2; j<N−1; j++){
    a[j] = b[j];
  }
}
```

(a) Original code

```
for (c1=0;c1<=T−1;c1++) {
    b[2]=0.333*(a[1+2]+a[2]+a[2 −1]);
    for (c2=2*c1+3;c2<=2*c1+N−2;c2++) {
        b[−2*c1+c2] = 0.333*(a[1+−2*c1+c2]+
                    a[−2*c1+c2]+a[−2*c1+c2−1]);
        a[−2*c1+c2−1]=b[−2*c1+c2−1];
    }
    a[N−2]=b[N−2];
}
```

(b) Transformed (before tiled code generation) context N >= 5

$$\mathcal{T}_{s_1}\begin{pmatrix} t \\ i \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 2 & 1 \\ 0 & 0 \end{pmatrix}\begin{pmatrix} t \\ i \end{pmatrix} \qquad \mathcal{T}_{s_2}\begin{pmatrix} t \\ j \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 2 & 1 \\ 0 & 0 \end{pmatrix}\begin{pmatrix} t \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$$



Indexing base changed to (1,1) for convenient display

(c) Tiles in the transformed space

Figure 5.2: Imperfectly nested Jacobi stencil: tiling

```
for (t0=0;t0<=floord(T,1024);t0++) {
  for (t1=max(0,ceild(2048*t0-1021,1024));
                t1<=min(floord(n+2*T,1024),floord(2048*t0+n+2046,1024));t1++) {
    if ((t0 <= floord(1024*t1-n,2048)) && (t1 >= ceild(n+2,1024))) {
      if (-n%2 == 0) {
          {a[n-1]=b[1+n-1]+b[n-1 -1]+b[n-1];} ;
      }
    }
    for (t2=max(max(ceild(1024*t1-n+1,2),1024*t0),1);
      t2<=min(min(min(512*t1-2,1024*t0+1023),floord(1024*t1-n+1023,2)),T);t2++) {
      for (t3=1024*t1;t3<=2*t2+n-1;t3++) {
        {b[-2*t2+t3]=a[-2*t2+t3]*a[1+-2*t2+t3]+a[-2*t2+t3-1];} ;
        {a[-2*t2+t3-1]=b[1+-2*t2+t3-1]+b[-2*t2+t3-1 -1]+b[-2*t2+t3-1];} ;
      }
      {a[n-1]=b[1+n-1]+b[n-1 -1]+b[n-1];} ;
    }
    for (t2=max(max(512*t1-1,1024*t0),1);
            t2<=min(min(1024*t0+1023,floord(1024*t1-n+1023,2)),T);t2++) {
      {b[2]=a[2]*a[1+2]+a[2 -1];} ;
      for (t3=2*t2+3;t3<=2*t2+n-1;t3++) {
        {b[-2*t2+t3]=a[-2*t2+t3]*a[1+-2*t2+t3]+a[-2*t2+t3-1];} ;
        {a[-2*t2+t3-1]=b[1+-2*t2+t3-1]+b[-2*t2+t3-1 -1]+b[-2*t2+t3-1];} ;
      }
      {a[n-1]=b[1+n-1]+b[n-1 -1]+b[n-1];} ;
    }
    for (t2=max(max(1,1024*t0),ceild(1024*t1-n+1024,2));
                          t2<=min(min(1024*t0+1023,T),512*t1-2);t2++) {
      for (t3=1024*t1;t3<=1024*t1+1023;t3++) {
        {b[-2*t2+t3]=a[-2*t2+t3]*a[1+-2*t2+t3]+a[-2*t2+t3-1];} ;
        {a[-2*t2+t3-1]=b[1+-2*t2+t3-1]+b[-2*t2+t3-1 -1]+b[-2*t2+t3-1];} ;
      }
    }
    for (t2=max(max(max(512*t1-1,1),1024*t0),ceild(1024*t1-n+1024,2));
                         t2<=min(min(512*t1+510,1024*t0+1023),T);t2++) {
      {b[2]=a[2]*a[1+2]+a[2 -1];} ;
      for (t3=2*t2+3;t3<=1024*t1+1023;t3++) {
        {b[-2*t2+t3]=a[-2*t2+t3]*a[1+-2*t2+t3]+a[-2*t2+t3-1];} ;
        {a[-2*t2+t3-1]=b[1+-2*t2+t3-1]+b[-2*t2+t3-1 -1]+b[-2*t2+t3-1];} ;
      }
    }
  }
}
```

Figure 5.3: Imperfectly nested Jacobi: tiled code (context: $N \geq 4$)

legality of the same is guaranteed by the transformation framework. The scattering functions are duplicated for each such level as it was done for one level. Such a guarantee is available even when syntactic tiling is to be done as a post-pass on a perfectly nest band in the target AST.

5.2.3 Pipelined parallel code generation

Once the algorithm in Sec. 5.2.2 is applied, outer parallel or inner parallel loops can be readily marked parallel (for example with OpenMP pragmas). However, unlike scheduling-based approaches, since we find tiling hyperplanes and the outer ones are used as space, one or more of the space loops may have dependences along them. Tiles created from an outermost band of tiling hyperplanes are to be scheduled appropriately.

Algorithm 4 Code generation for tiled pipelined parallelism

```
INPUT Given that Algorithm 3 has been applied, a set of $k$ (statement-wise) supernodes in the transformed space belonging to a tilable band: $\phi^1_{T_S}, \phi^2_{T_S}, \ldots, \phi^k_{T_S}$
1: To extract $m$ $(< k)$ degrees of pipelined parallelism:
2: /* Update transformation functions */
3: for each statement $S$ do
4:    Perform the following unimodular transformation on only the scattering supernodes: $\phi^1_T \rightarrow \phi^1_T + \phi^2_T + \cdots + \phi^{m+1}_T$
5:    Mark $\phi^2_T, \phi^3_T, \ldots, \phi^{m+1}_T$ as parallel
6:    Leave $\phi^1_T, \phi^{m+2}_T, \ldots, \phi^k_T$ as sequential
7: end for
OUTPUT Updated transformation functions and scatterings
```

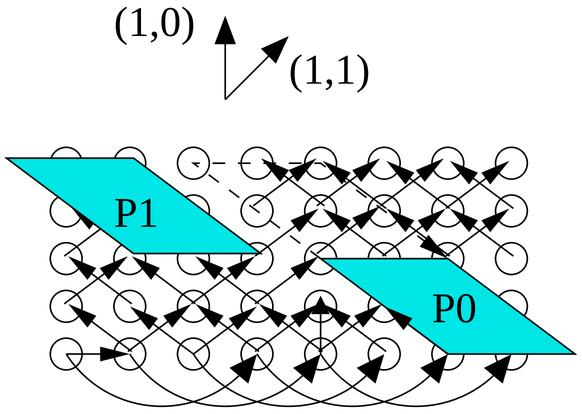


Figure 5.4: Tile wavefront along (2,1)

Treating pipelined parallelism as inner parallelism: For pipelined parallel codes, our approach to generate coarse-grained (tiled) shared memory parallel code is as described in Figure 4. Once the technique described in the previous section is applied to generate the tile space scatterings and intra-tiled loops, dependence components are non-negative direction along each of the tiling hyperplanes. Hence, the sum $\phi_T^1 + \phi_T^2 + \cdots + \phi_T^{p+1}$ satisfies all affine dependences satisfied by ϕ_T^1, ϕ_T^2, ..., ϕ_T^{p+1}, and gives a legal schedule of tiles. Since the transformation is only on the tile space, it preserves the shape of the tiles. Communication still happens along boundaries of ϕ^1, ϕ^2, ..., ϕ^p, thus preserving benefits of the optimization performed by the bounding function approach. Moreover, performing such a unimodular transformation on the tile space introduces very less additional code complexity.

In contrast, obtaining an affine (fine-grained) schedule and then enabling time tiling would lead to shapes different from above our approach. The above technique of

```
for(i=1; i<N; i++) {
  for(j=1; j<N; j++) {
    a[i,j] = a[i−1,j] + a[i,j−1];
  }
}
```

Figure 5.5: Original (sequential) code

```
for (c1=−1;c1<=floord(N−1,16);c1++) {
#pragma omp parallel for shared(c1,a) private(c2,c3,c4)
  for (c2=max(ceild(32∗c1−N+1,32),0);
              c2<=min(floord(16∗c1+15,16),floord(N−1,32)); c2++){
      for (c3=max(1,32∗c2);c3<=min(32∗c2+31,N−1); c3++) {
         for (c4=max(1,32∗c1−32∗c2); c4<=min(N−1,32∗c1−32∗c2+31); c4++) {
            S1(c2,c1−c2,c3,c4) ;
          }
      }
      < implied barrier >
  }
}
```

Figure 5.6: Shared memory parallel code generation example: a coarse-grained tile schedule

adding up 1-d transforms resembles that of [LL98] where (permutable) time partitions are summed up for maximal dependence dismissal; however, we do this in the tile space as opposed to for finding a schedule that dismisses all dependences.

Figure 5.6 shows a simple example with tiling hyperplanes (1,0) and (0,1). Our scheme allows clean generation of parallel code without any syntactic treatment. Alternate ways of generating pipelined parallel code exist that insert special post/notify or wait/signal directives to handle dependences in the space loops [LCL99, Gri04], but, these require syntactic treatment. Note that not all degrees of pipelined parallelism need be exploited. In practice, a few degrees are sufficient; using several could introduce code complexity with diminishing return.

```
for (k=0; k<N; k++) {
    for (j=k+1; j<N; j++) {
        a[k][j] = a[k][j]/a[k][k];
    }
    for (i=k+1; i<N; i++) {
        for (j=k+1; j<N; j++) {
            a[i][j] = a[i][j]−a[i][k]∗a[k][j];
        }
    }
}
```

Figure 5.7: LU decomposition (non-pivoting) form

5.2.4 Example 1: Imperfectly nested stencil.

The transformation computed by Algorithm 1 is as follows.

$$\begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \mathcal{T}_{S1}\begin{pmatrix} t \\ i \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 2 & 1 \\ 0 & 0 \end{pmatrix}\begin{pmatrix} t \\ i \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

$$\begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \mathcal{T}_{S2}\begin{pmatrix} t \\ j \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 2 & 1 \\ 0 & 0 \end{pmatrix}\begin{pmatrix} t \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$$

The transformation is updated to the following for generation of locally tiled code.

$$\mathcal{T}_{S1}\begin{pmatrix} t_T \\ i_T \\ t \\ i \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}\begin{pmatrix} t_T \\ i_T \\ t \\ i \\ 1 \end{pmatrix}$$

$$\mathcal{T}_{S2}\begin{pmatrix} t_T \\ j_T \\ t \\ j \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} t_T \\ j_T \\ t \\ j \\ 1 \end{pmatrix}$$

After applying Algorithm 4, the transformation for generation of parallelized and locally tiled code will be the following with the second hyperplane marked parallel.

$$\mathcal{T}_{S1}\begin{pmatrix} t_T \\ i_T \\ t \\ i \\ 1 \end{pmatrix} = \begin{pmatrix} 3 & 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}\begin{pmatrix} t_T \\ i_T \\ t \\ i \\ 1 \end{pmatrix}$$

$$\mathcal{T}_{S2}\begin{pmatrix} t_T \\ j_T \\ t \\ j \\ 1 \end{pmatrix} = \begin{pmatrix} 3 & 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} t_T \\ j_T \\ t \\ j \\ 1 \end{pmatrix} \tag{5.1}$$

5.2.5 Example 2: LU

For the LU decomposition code in Fig. 5.7, we show how the transformations are updated to obtain one degree or two degrees of pipelined parallelism. The transformations for S1 and S2, after the tiling for 1-d pipelined parallelism is expressed, are the following with c_2 being marked OpenMP parallel.

$$\mathcal{T}_{S1} : \begin{pmatrix} c_1 \\ \mathbf{c_2} \\ c_3 \\ c_4 \\ c_5 \\ c_6 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} k_T \\ j_T \\ k \\ j \end{pmatrix}$$

$$\mathcal{T}_{S2} : \begin{pmatrix} c_1 \\ \mathbf{c_2} \\ c_3 \\ c_4 \\ c_5 \\ c_6 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} k_T \\ i_T \\ j_T \\ k \\ i \\ j \end{pmatrix}$$

If one wishes to extract two degrees of pipelined parallelism, the transformation would be the following, with loops $c2$ and $c3$ are marked OpenMP parallel.

$$\mathcal{T}_{S1} : \begin{pmatrix} c_1 \\ \mathbf{c_2} \\ \mathbf{c_3} \\ c_4 \\ c_5 \\ c_6 \end{pmatrix} = \begin{pmatrix} 2 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} k_T \\ j_T \\ k \\ j \end{pmatrix}$$

$$\mathcal{T}_{S2} : \begin{pmatrix} c_1 \\ \mathbf{c_2} \\ \mathbf{c_3} \\ c_4 \\ c_5 \\ c_6 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} k_T \\ i_T \\ j_T \\ k \\ i \\ j \end{pmatrix}$$

5.3 Preventing Code Expansion in Tile Space

The overhead of floor and ceil operations as well as conditionals in the tile-space loops (at the outer levels) is insignificant. Hence, we would like to have compact code at the outer level while allowing code expansion in the intra-tile loops to decrease control complexity. This improves performance while keeping the code size under control.

Using the default options with tiling specified as described leads to significant code expansion since the transformed space we are tiling is a shifted and skewed space. Preventing any code expansion at all leads to an if condition in the innermost loop, resulting in very low performance. However, optimizing only the intra-tile loops for control is very effective. It also avoids large numbers in the intermediate operations performed by code generators, that could possibly lead to PolyLib exceptions for large tile sizes or deep loop nest tiling. Table 5.1 shows the sensitivity in performance for the code in Figure 5.2.2.

Cloog options	Code size (lines)	Codegen time	Execution time	Improvement over ‘icc -fast’
Full code expansion	2226	1.84s	2.57s	2.7x
Only intra-tile expansion	40	0.04s	1.6s	4.3x
No code expansion	15	0.01s	17.6s	0.39x

Table 5.1: Performance sensitivity of L1-tiled imperfectly nested stencil code with Cloog options: $N = 10^6$, $T = 1000$, tile dimensions: 2048×2048

5.4 Tilability Detection and Parallelism Detection

Our algorithm naturally transforms to hierarchy of tilable loop nest sets. The outermost band can be taken and used for coarse-grained parallelization using the scheme proposed in Sec. 5.2.3.

5.4.1 AST-based versus scattering function-based

Currently, all parallelism and tilability detection in Pluto is done on a (global) scattering function basis as opposed to being done on the transformed AST. The former is easier to implement from within the transformation framework without making any changes to the code generator and was the only reason to do so. When the AST is split using a scalar dimension, a particular inner ϕ^k for a statement may be parallel while the one for another need not be. However, if the analysis is done for all the dependences (intra-statement and inter-statement), it would lead to conservative detection. The way inner and outer parallel hyperplanes were defined in Def 14 and Def 13 was at a "global" level. The same holds for tiling too, but is not as important as the concern described in the next sub-section.

Once ϕs are known, directions of the affine dependences components can be computed on a scattering function basis, i.e., one can compute for each edge e and a given dimension k of the scattering function:

$$\begin{array}{ll}
0: & \phi^k_{S_j}(\vec{t}) - \phi^k_{S_i}(\vec{s}) = 0, \ \ \langle \vec{s}, \vec{t}\rangle \in \mathcal{P}_e \\
+: & \left(\phi^k_{S_j}(\vec{t}) - \phi^k_{S_i}(\vec{s}) \geq 0, \ \langle \vec{s}, \vec{t}\rangle \in \mathcal{P}_e\right) \wedge \neg \left(\phi^k_{S_j}(\vec{t}) - \phi^k_{S_i}(\vec{s}) = 0, \ \langle \vec{s}, \vec{t}\rangle \in \mathcal{P}_e\right) \\
-: & \left(\phi^k_{S_j}(\vec{t}) - \phi^k_{S_i}(\vec{s}) \leq 0, \ \langle \vec{s}, \vec{t}\rangle \in \mathcal{P}_e\right) \wedge \neg \left(\phi^k_{S_j}(\vec{t}) - \phi^k_{S_i}(\vec{s}) = 0, \ \langle \vec{s}, \vec{t}\rangle \in \mathcal{P}_e\right) \\
\star: & \text{Neither of } +, -, \text{or } 0
\end{array}$$

The above can be computed very easily with calls to PIP [Fea88] since ϕs are already known.

5.4.2 Intricacies in detecting tilable bands

Just tiling the bands is conservative in many cases. All previous frameworks that transform to a hierarchy of permutable loop nests [WL91a, DSV97, LCL99] suggest such tiling. We discuss some intricacies using a kernel from a real application. Consider the imperfectly nested *doitgen* kernel shown in Figure 5.8. It can be viewed as an in-place matrix-matrix multiplication.

The transformation found with Pluto's *maxfuse* is as follows.

$$\begin{aligned}
\text{S1: } (\phi^1, \phi^2, \phi^3, \phi^4, \phi^5) &= (r, q, 0, p, 0) \\
\text{S2: } (\phi^1, \phi^2, \phi^3, \phi^4, \phi^5) &= (r, q, 1, p, s) \\
\text{S3: } (\phi^1, \phi^2, \phi^3, \phi^4, \phi^5) &= (r, q, 2, p, 0)
\end{aligned}$$

The direction vectors based on the affine dependence components can now be written as:

```
for (r=0; r<N; r++) {
    for (q=0; q<N; q++) {
        for (p=0; p<N; p++) {
            sum[r][q][p] = 0;
            for (s = 0; s< N; s++) {
                sum[r][q][p] = sum[r][q][p] + A[r][q][s]*C4[s][p];
            }
        }
        for (p=0; p<N; p++) {
            A[r][q][p] = sum[r][q][p];
        }
    }
}
```

Figure 5.8: Doitgen (original code)

Dependence	Affine component direction
S1 $\rightarrow$ S2	$(0, 0, +, 0, +)$
S2 $\rightarrow$ S2	$(0, 0, 0, 0, +)$
S1 $\rightarrow$ S3	$(0, 0, +, 0, 0)$
S2 $\rightarrow$ S3	$(0, 0, +, 0, -)$
S2 $\rightarrow$ S2	$(0, 0, 0, 0, +)$
S2 $\rightarrow$ S3	$(0, 0, +, -, -)$
S1 $\rightarrow$ S2	$(0, 0, +, 0, +)$
S2 $\rightarrow$ S2	$(0, 0, 0, 0, +)$

Standard tiling based on identifying hierarchies of tilable bands would treat (ϕ^1,ϕ^2) as one band and (ϕ^4,ϕ^5) as another. Note that ϕ^3 is a scalar dimension. So the tiled space for S2 would be given by:

$$(r_T, q_T, r, q, 1, pT, sT, p, s)$$

The above does not create true 4-d tiles for S2. However, the surprising good tiling here is given by:

$$\begin{array}{ll} \text{S1:} & (r_T, q_T, 0, p_T, 0, r, q, p, 0) \\ \text{S2:} & (r_T, q_T, 1, p_T, s_T, r, q, p, s) \\ \text{S3:} & (r_T, q_T, 2, p_T, 0, r, q, p, 0) \end{array}$$

Here, it is legal to move the two outer loops all the way inside since the dependences that have negative components on ϕ^4 and ϕ^5 are satisfied at the scalar dimension ϕ^3. Hence, we have an additional consideration of ignoring negative directions for dependences that have been satisfied at a scalar dimension, since the scalar dimension can be preserved at the third level even in the tiled code. This "trick" is very useful in tiling several codes that would otherwise come out as untilable or not tilable with any good benefit. Pluto's detection can take care of the above. The tiled code is shown in Figure 5.9. Figure 5.10 compares the performance of the transformed code with the version written with BLAS calls. More detailed performance results are presented in Chapter 6.

It would also be useful to note that the following tiling would be illegal as some values of A will been overwritten before they are finished using.

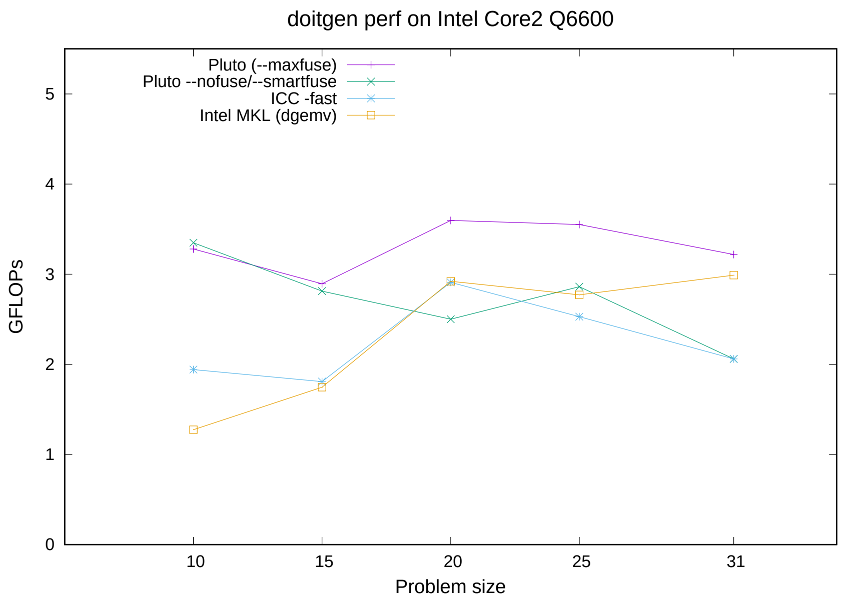
$$\begin{array}{ll} \text{S1:} & (rT, qT, 0, pT, 0, r, q, 0, p, 0) \\ \text{S2:} & (rT, qT, 0, pT, sT, r, q, 1, p, s) \\ \text{S3:} & (rT, qT, 0, pT, 0, r, q, 2, p, 0) \end{array}$$

In this case, the scalar dimension has no longer been preserved and has been considered as part of the tilable band violating the inter-statement dependences between $S2$ and $S3$.

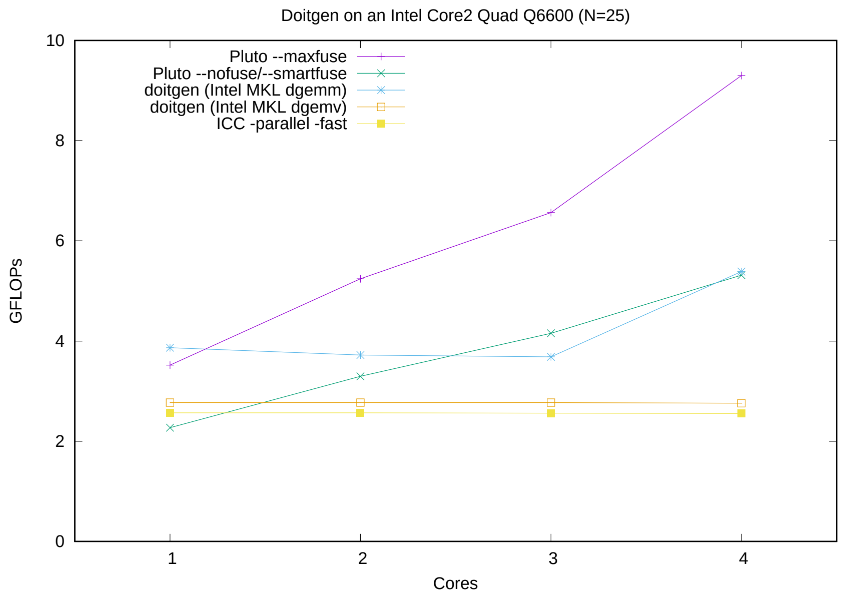
Besides doitgen, the sequence of matrix-matrix multiplies and the TCE 4-index transform examples presented in Section 4.4 are other codes for which tilability detection is improved with the above enhancement.

```
lb1=0;
ub1=floord(N−1,4);
#pragma omp parallel for shared(lb1,ub1) private(t0,t1,t2,t3,t4,t5,t6,t7,t8,t9)
for (t0=lb1; t0<=ub1; t0++) {
  for (t1=0;t1<=floord(N−1,8);t1++) {
    for (t3=0;t3<=floord(N−1,31);t3++) {
      for (t5=max(0,4∗t0);t5<=min(N−1,4∗t0+3);t5++) {
        for (t6=max(0,8∗t1);t6<=min(N−1,8∗t1+7);t6++) {
          for (t9=max(0,31∗t3);t9<=min(N−1,31∗t3+30);t9++) {
            {sum[t5][t6 ][ t9]=0;} ;
          }
        }
      }
    }
    for (t3=0;t3<=floord(N−1,31);t3++) {
      for (t4=0;t4<=floord(N−1,8);t4++) {
        for (t5=max(0,4∗t0);t5<=min(N−1,4∗t0+3);t5++) {
          for (t6=max(0,8∗t1);t6<=min(N−1,8∗t1+7);t6++) {
            for (t8=max(0,8∗t4);t8<=min(N−1,8∗t4+7);t8++) {
              for (t9=max(0,31∗t3);t9<=min(N−1,31∗t3+30);t9++) {
                {sum[t5][t6 ][ t9]=A[t5][t6 ][ t8]∗C4[t8][t9]+sum[t5][t6 ][ t9 ];}  ;
              }
            }
          }
        }
      }
    }
    for (t3=0;t3<=floord(N−1,31);t3++) {
      for (t5=max(0,4∗t0);t5<=min(N−1,4∗t0+3);t5++) {
        for (t6=max(0,8∗t1);t6<=min(N−1,8∗t1+7);t6++) {
          for (t9=max(0,31∗t3);t9<=min(N−1,31∗t3+30);t9++) {
            {A[t5][t6 ][ t9]=sum[t5][t6 ][ t9 ];}  ;
          }
        }
      }
    }
  }
}
```

Figure 5.9: Doitgen tiled code (register tiling and vector pragmas not shown)



(a) Single core



(b) Parallel: $N = 25$

Figure 5.10: Doitgen performance on an Intel Core 2 Quad: preview

5.5 Setting Tile Sizes

With our approach, tile sizes can be set in a decoupled manner as the shapes have been captured with $\mathcal{T}_S$. Pluto currently uses very rough thumb rules based on array dimensionalities to set tile sizes. In addition, custom tile sizes can be forced by the user easily through a separate file. By decoupling tile shape and size optimization, not only is the transformation framework's objective function kept linear, but also is the possibility of employing stronger cost models to determine tile sizes for locality opened. Techniques to efficiently count integer points in parametrized polytopes exist: [VSB+07] is the state-of-the-art approach that employs Barvinok's rational functions for this purpose and has been implemented into a library. It is straightforward to obtain the polyhedra corresponding to the points in the data space that are touched by a tile [BBK+08b]. Once this is done and the cache sizes are known, [VSB+07] may be used to deduce tile sizes. We intend to explore the effectiveness of such an approach.

Tile sizes for parallelism. The OpenMP "parallel for" directive(s) typically achieves block distribution of iterations of tile space loops among processor cores. Hence, whenever there is outer parallelism or pure inner parallelism, there is no special need to compute another set of tile sizes for parallelism, unless one wants to schedule tiles in a block cyclic manner. However, just performing block distribution is better for inter-tile reuse. Execution on each core is thus a sequence of L2 tiles or L1 tiles if no L2 tiling is done. However, whenever no synchronization-free parallelism exists, tile sizes for parallelism affect pipelined startup cost, load balance, and the number of synchronizations. There is a trade-off between the former two and the latter. This problem has been studied extensively, but for very restrictive input [ABRY03, BDRR94, HS02, HCF97, HCF99, SD90]. Clearly, the right tile sizes here are a function of the problem size and the number of processors, and leaving tile sizes as parameters in the final generated code is necessary for any scheme to succeed. The approach we described in this chapter is for fixed tile sizes, and extension to parametric ones [RKRS07] is needed. Meanwhile, just setting conservative (smaller) L2 tile sizes or not performing L2 tiling for smaller problem sizes seems to be a reasonable approach in practice. This avoids load imbalance and may incur higher number of synchronizations than the optimal solution, but is not likely to be a big issue since coarse-enough parallelism has already been achieved making it a case of diminishing return. Given the large size of L2 caches on current processors, for small problem sizes, L2 tiling should be disabled with Pluto to prevent load imbalance. This can also be easily automated in future by generating multiple versions of code conditional on structure parameter sizes.

5.6 Post-processing Transformations

We have integrated an annotation-driven system, Orio [Ori, NHG07], to perform syntactic transformations on the code generated from Cloog as a post-processing. These include register tiling, unrolling, and unroll-jamming. The choice of loops to perform these transformations on is specified by Pluto's core transformation module, and hence legality is guaranteed. The tool Orio itself has no information on dependences.

5.6.1 Unroll-jamming or register tiling

Trivially, the innermost loop can always be unrolled since doing so does not change the execution order. A loop can be unrolled and jammed iff it can be made the innermost loop, i.e., if it can be stripmined and the point loop can be made the innermost one. Now, a hyperplane can be made the innermost loop if all the dependences it satisfies have non-negative components on the inner hyperplanes. One can see that unroll-jamming a particular loop can affect the unroll-jamming at inner levels. To avoid all of these pathologies and to come up with a clean condition, we simply focus on consecutive sets of loops. Unroll-jamming a consecutive set of loops is the same as *register tiling*, since it can be viewed as tiling and unrolling the intra-tile loops. Hence, the legality of tiling a band of loops is same as the legality of unroll-jamming them. The following conditions seem to be adequate and powerful:

1. A parallel loop at any level (inner parallel or outer parallel) can always be unroll-jammed

2. The innermost band of permutable loops can always be unroll-jammed

Recall that our cost function pushes hyperplanes carrying higher reuse to inner levels. This goes well with register tiling the innermost band of permutable loops when there are multiple bands in the hierarchy. Orio [Ori, NHG07] can perform syntactic unrolling or unroll-jamming of specified loops, rectangular or non-rectangular, and

Pluto uses it for that purpose on the output of Cloog. The syntactic register tiling accomplished by Orio is same in effect when compared to [KRSR07, JLF02], although code from the scheme of Kim et al.[KRSR07] will be more compact. We chose to use Orio as it is annotation-driven and can be readily integrated with Pluto without the need to modify Cloog.

5.6.2 Intra-tile reordering and aiding auto-vectorization

Due to the nature of our algorithm, even within a local tile (L1) that is executed sequentially, the intra-tile loops that are actually parallel do not end up being outer in the tile (Sec. 3.10): this goes against vectorization of the transformed source for which we rely on the native compiler. However, we know that it is always legal to move a parallel loop inside (Def 14). As a consequence, a parallel loop at any level can be stripmined, the corresponding point loop moved innermost and left for auto-vectorization by the native compiler. Pluto generated code is complex enough to hinder a native compiler's dependence analysis leading to assumed vector dependences. So, with ICC for example, we use the ignore vector dependence pragma (ivdep) [Int] to mark the loop for vectorization.

In the current implementation of Pluto, a loop that is moved in and marked for vectorization as described above is not unrolled for two reasons: (1) we find that it interferes with its vectorization by the compiler (for eg. with Intel's C compiler), and (2) the loop is parallel and so the only possible reuse along it is due to input dependences. Hence, for a code like matrix-matrix multiplication in the standard ijk form, where i and j are parallel, and ijk is one tilable band, we obtain:

After tiling: (i_T, j_T, k_T, i, j, k)

After inner movement for vectorization: $(i_T, j_T, k_T, \underbrace{i, k}_{8\times 8}, j)$.

Then, register tiling is done for i and k while j will be auto-vectorized by the compiler.

Similar reordering is possible to improve spatial locality that is not considered by our cost function due to the latter being fully dependence-driven. Characterizing spatial locality for affine accesses is easy and well known [WL91b]. Bastoul et al. [BF03]'s approach of computing chunking functions is particularly well-suited to find a better execution order for a local tile to improve spatial reuse. Note that the tile shapes or the schedule in the tile space is not altered by such post-processing.

5.7 Related Work on Tiled Code Generation

Code generation under multiple affine mappings was first addressed by Kelly et al. [KPR95]. Significant advances relying on new algorithms and mathematical machinery were made by Quilleré et al. [QRW00] and by Bastoul [Bas04a], resulting in a powerful open-source code generator, Cloog [Clo]. Our tiled code generation scheme uses Ancourt and Irigoin's [AI91] classic approach to specify domains with fixed tile sizes and shape information, but combines it with Cloog's support for scattering functions to allow generation of tiled code for multiple domains under the computed transformations.

Goumas et al. [GAK02] reported an alternate tiled code generation scheme to Ancourt and Irigoin's [AI91]) to address the inefficiency involved in using Fourier-

Motzkin elimination – however, this is no longer an issue as the state-of-the-art uses efficient algorithms [QRW00, Bas04a] based on PolyLib [Wil93, Pol].

Techniques for parametric tiled code generation [RKRS07, KRSR07] were recently proposed for single statement domains for which rectangular tiling is valid. These techniques complement our system very well and we intend to explore their integration. There are two clear benefits in the context of our system. Firstly, empirical tuning can be done much faster since code need not be generated each time for a particular set of tile sizes. Secondly, big numbers in the intermediate computations of Cloog can be avoided. When large tile sizes are used with multiple levels of tiling, the numbers can sometimes grow big enough to cross 64-bit boundaries.

Regarding generation of pipelined parallel code, existing techniques [LCL99, Gri04] insert or propose the use of special post/notify or wait/signal directives to handle dependences in the space loops. Our scheme does not require any additional synchronization primitives or any syntactic treatment. This is achieved by transforming the tile space to translate pipelined parallelism into doall parallelism. Hence, only OpenMP pragmas are sufficient with our scheme.

CHAPTER 6

EXPERIMENTAL EVALUATION

This chapter presents the experimental evaluation of the Pluto automatic parallelization system, the design to implementation of which were described in the previous three chapters. The performance for several kernels is compared to the existing state-of-the-art from the research literature as well as to that of best native compilers and highly tuned vendor-supplied libraries where applicable.

6.1 Comparing with previous approaches

Several previous papers on automatic parallelization have presented experimental results. However, significant jumps were made in the process of going from the compiler framework to evaluation. A direct comparison is difficult since the implementations of those approaches (with the exception of Griebl's) is not available; further most previously presented studies did not use an end-to-end automatic implementation, but performed manual code generation based on solutions generated by a transformation framework, or by picking solutions from a large space of solutions characterized. In addition, a common missing link in the chain was the lack of a powerful and efficient code generator like Cloog, which has only recently become available.

In assessing the effectiveness of our system, we compare performance of the generated code with that generated by production compilers, as well as undertaking a best-effort fair comparison with previously presented approaches from the research community. The comparison with other approaches from the literature is in some cases infeasible because there is insufficient information for us to reconstruct a complete transformation (e.g. [AMP01]). For others [LL98, LCL99, LLL01], a complete description of the algorithm allows us to manually construct the transformation; but since we do not have access to an implementation that can be run to determine the transformation matrices, we have not attempted an exhaustive comparison for all the cases.

The current state-of-the-art with respect to optimizing code has been semi-automatic approaches that require an expert to manually guide transformations. As for scheduling-based approaches, the LooPo system [Loo] includes implementations of various polyhedral scheduling techniques including Feautrier's multi-dimensional time scheduler which can be coupled with Griebl's space and FCO time tiling techniques. We thus provide comparison for some number of cases with the state of the art – (1) Griebl's approach that uses Feautrier's schedules along with Forward-Communication-Only allocations to enable time tiling [Gri04], and (2) Lim-Lam's affine partitioning [LL98, LCL99, LLL01]. For both of these previous approaches, the input code was run through our system and the transformations were forced to be what those approaches would have generated. Hence, these techniques get all benefits of Cloog and our fixed tile size code generation scheme.

6.2 Experimental Setup

Three different processor architectures are used for the results in this section.

1. **Intel Core 2 Quad Q6600** quad-core CPU clocked at 2.4 GHz (1066 MHz FSB) with a 32 KB L1 D cache, 8MB of L2 cache (4 MB shared per core pair), and 2 GB of DDR2-667 RAM, and running Linux kernel version 2.6.22 (x86-64)

2. **AMD Opteron 2218** dual-core CPUs (2-way SMP) clocked at 2.6 GHz with a 64 KB L1 cache and 1 MB L2 cache, running Linux kernel version 2.6.18 (x86-64)

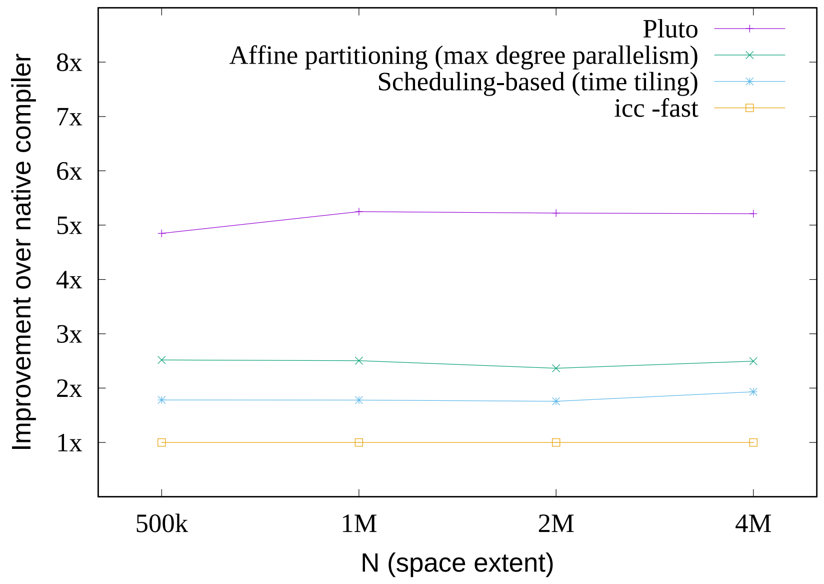
3. **AMD Phenom x4 9850** quad-core clocked at 2.5 GHz with a 64 KB L1 data cache, 512 KB private L2 cache per core, and a 2 MB shared L3 cache, running Linux kernel version 2.6.25 (x86-64)

ICC 10.0, Intel's C compiler, is used to compile the base codes as well as the source-to-source transformed codes; it was run with '-fast' (-openmp for parallelized code); the '-fast' option turns on -O3, -ipo, -static, -no-prec-div on x86-64 processors – these options also enable auto-vectorization in icc. Whenever gcc is used, it is GCC 4.1.1 with options "-O3" (-fopenmp for parallelized code). The OpenMP implementation of icc supports nested parallelism – this is needed for exploiting multiple degrees of pipelined parallelism when they exist. For easier presentation and analysis, local tiling for most codes is done for the L1 cache, with equal tile sizes used along all dimensions; they were set empirically (without any comprehensive search) and agreed with the cache size quite well. In all cases, the optimized code for our framework was obtained automatically in a turn-key fashion from the input source code. When comparing with approaches of Lim/Lam and Griebl, the same tile sizes were used for each approach and they appeared to be good ones. The OpenMP "parallel for" directive(s) achieves the distribution of the blocks of tile space loop(s) among processor cores. Hence, execution on each core is a sequence of L1 or L2 tiles. Analysis is more detailed for the first example which is simpler.

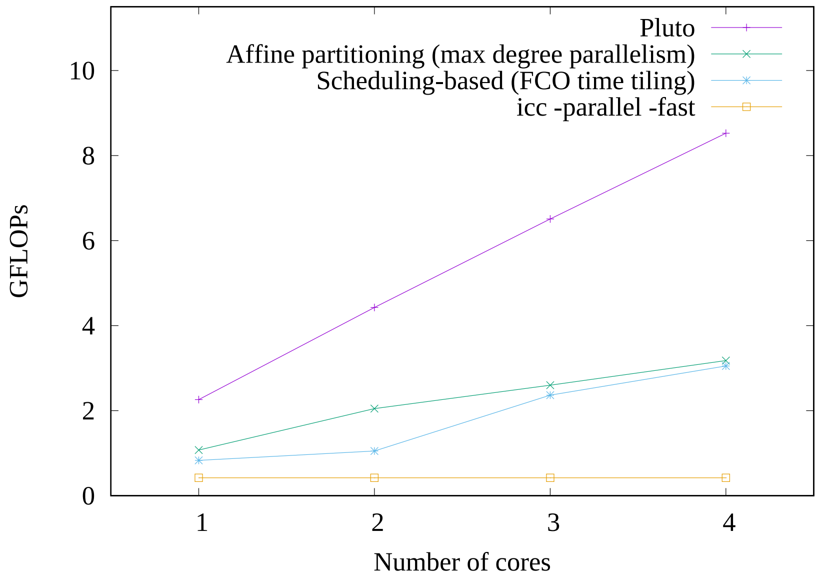
6.3 Imperfectly Nested Stencils

The original code, code optimized by our system without tiling, and optimized tiled code were shown in Figure 5.2.2. The performance of the optimized codes are shown in Figure 6.3. Speedup's ranging from 4x to 7x are obtained for single core execution due to locality enhancement. The parallel speedups are compared with Lim/Lam's technique (Algorithm A in [LL98]) which finds (2,-1), (3,-1) as the maximally independent time partitions. These do minimize the order of synchronization and maximize the degree of parallelism (O(N)), but any legal independent time partitions would have one degree of pipeline parallelism.

With scheduling-based techniques, the schedules found by LooPo's Feautrier scheduler are $2t$ and $2t+1$ for S1 and S2, respectively (note that this does not imply fusion). An FCO allocation here is given by $2t + i$, and this enables time tiling. Just space

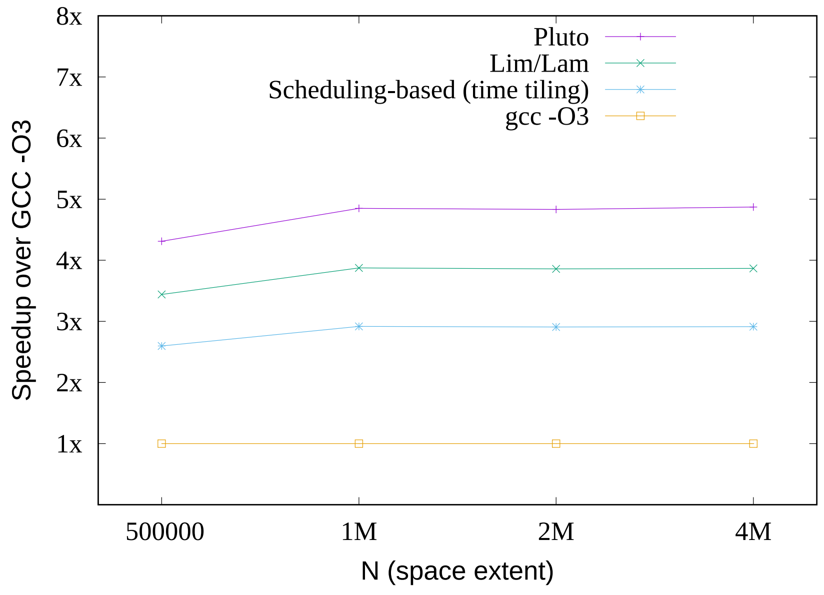


(a) Single core: $T = 10^4$

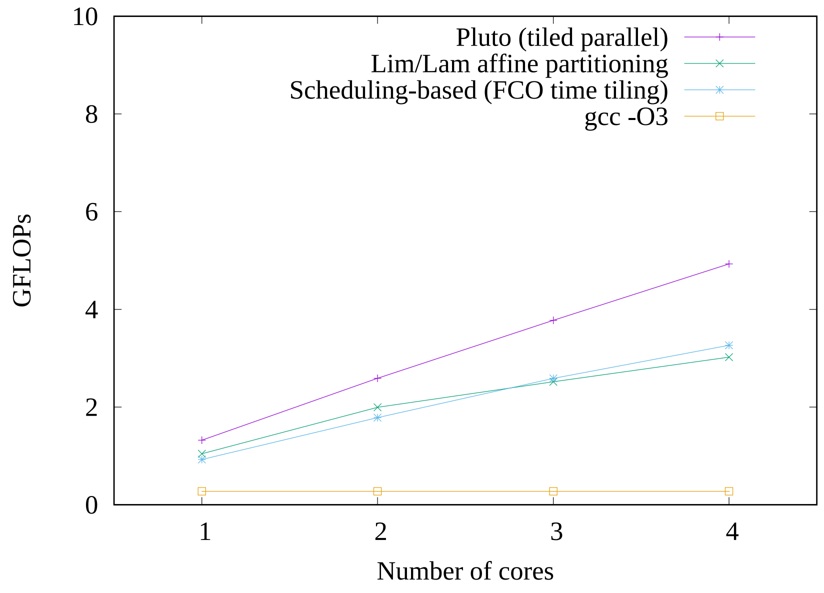


(b) Multi-core parallel: $N = 10^6, T = 10^5$

Figure 6.1: Imperfectly nested 1-d Jacobi stencil: all codes compiled with ICC



(a) Single core (with gcc): $T = 10^4$



(b) Multi-core parallel (with gcc): $N = 10^6, T = 10^5$

Figure 6.2: Imperfectly nested 1-d Jacobi stencil: all codes compiled with GCC

tiling in this case does not expose sufficient parallelism granularity and an inner space parallelized code has very poor performance. This is the case with icc's auto parallelizer; hence, we just show the sequential run time for icc in this case. Figure 6.3 shows L1 cache misses with each approach for a problem size that completely fits in the L2 cache. Though Griebl's technique incurs considerably lesser cache misses than Lim-Lam's, the schedule introduces non-unimodularity leading to modulo comparison in inner loops; it is possible to remove the modulo through an advanced technique using non-unit strides [Vas07] that Cloog does not implement yet. Our code incurs two times lesser number of cache misses than Griebl's and nearly 50 times lesser cache misses than Lim/Lam's scheme. Note that both Lim-Lam's and our transformation in this case are unimodular and hence have the same number of points in a tile for a given tile size. Comparison with gcc is provided in Figure 6.2(b) (gcc used to compile all codes) to demonstrate that the relative benefits of our source-to-source system will be available when used in conjunction with any sequential compiler.

6.4 Finite Difference Time Domain electromagnetic kernel

FDTD code is as shown in Figure 6.4. ex, ey represent electric fields in x and y directions, while hz is the magnetic field. The code has four statements - three of them 3-d and one 2-d and are nested imperfectly. Our transformation framework finds three tiling hyperplanes (all in one band - fully permutable). The transformation represent a combination of shifting, fusion and time skewing. Parallel performance results shown are for $nx = ny = 2000$ and $tmax = 500$. L1 and L2 tile sizes were

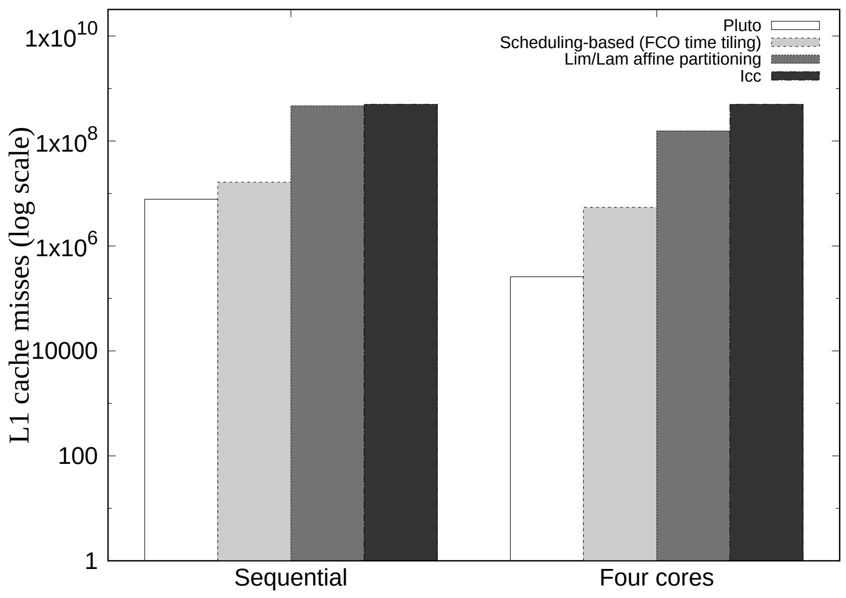


Figure 6.3: 1-d Jacobi: L1 tiling: $N = 10^5, T = 10^4$ (note the log scale on y-axis)

set automatically using rough thumb rules based on dimensionality of arrays involved and vectorization considerations. Results are shown in Figure 6.5. With polyhedral scheduling-based techniques, the outer loop is identified as the sequential schedule loop and the inner loops are all parallel – this is also the transformation applied by icc's auto parallelizer. This does not fuse the inner loops, and synchronization has to be done every time step.

With our approach, all three dimensions are tiled (due to a relative shift followed by a skew), the loops are fused, and each processor executes a 3-d tile (which itself is a sequence of 3-d L2 tiles) before synchronization. Multi-core results exhibit highly super-linear speedups. We have two degrees of pipelined parallelism here – to exploit

```
for (t=0; t<tmax; t++) {
  for (j=0; j<ny; j++) {
    ey [0][ j ] = exp(−t1);
  }

  for (i=1; i<nx; i++) {
    for (j=0; j<ny; j++) {
      ey[ i ][ j ] = ey[i ][ j ] −
          coeff1∗(hz[ i ][ j]−hz[i−1][j ]);
    }
  }

  for (i=0; i<nx; i++) {
    for (j=1; j<ny; j++){
      ex[ i ][ j ] = ex[i ][ j ]
      − coeff1∗(hz[ i ][ j]−hz[i ][ j−1]);
    }
  }

  for (i=0; i<nx; i++) {
    for (j=0; j<ny; j++) {
      hz[ i ][ j ] = hz[ i ][ j ] −
         coeff2∗(ex[ i ][ j+1]−ex[i][j ]
               +ey[i+1][j]−ey[i ][ j ]);
    }
  }
}
```

$M_S \qquad \vec{t}_S$

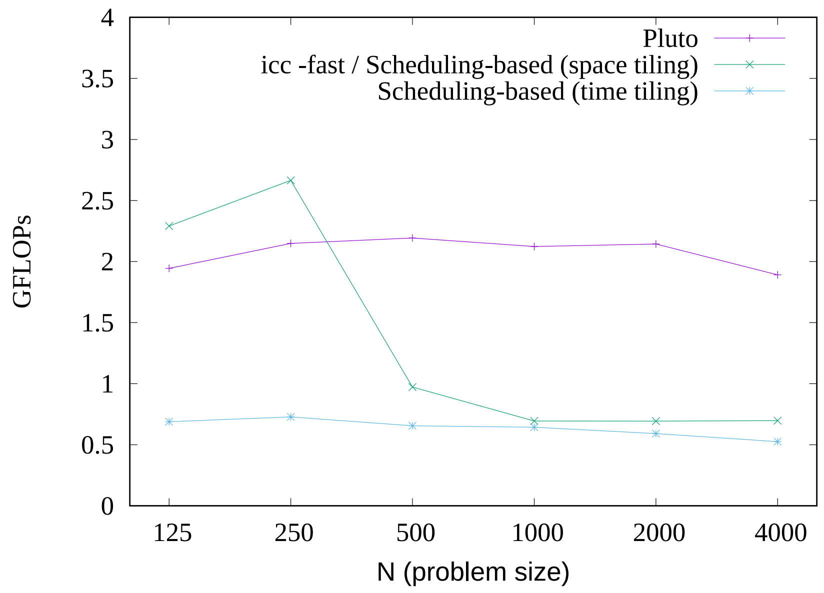
$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

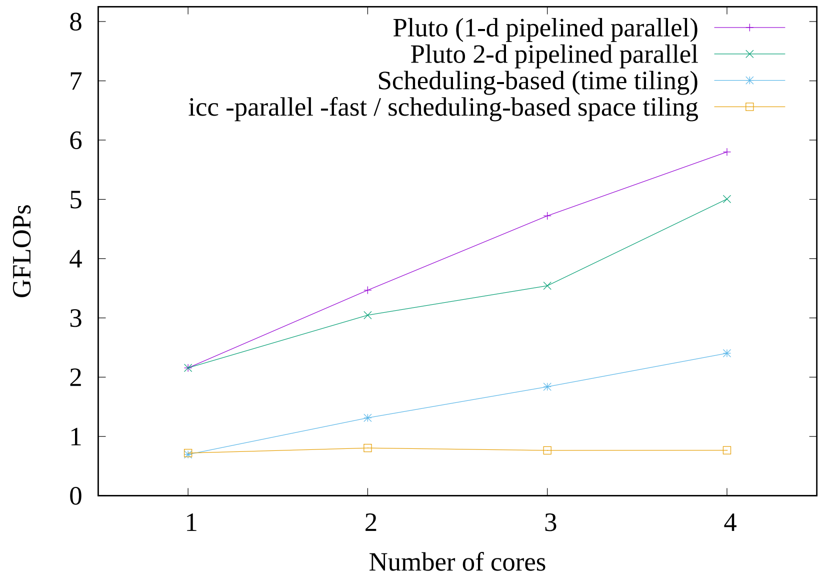
$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$$

Figure 6.4: 2-d Finite Difference Time Domain: original code and the statement-wise transformation



(a) Single core: T=500



(b) Parallel: $nx = ny = 2000$, $tmax = 500$

Figure 6.5: 2-d FDTD

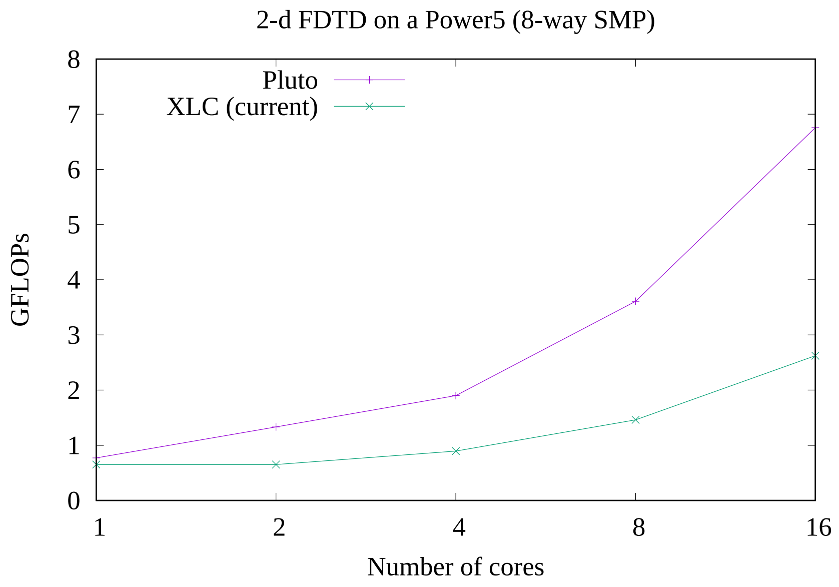


Figure 6.6: 2-D FDTD on an 8-way Power5 SMP

both, a tile space wavefront of (1,1,1) is needed; however, to exploit one, we just need a wavefront of (1,1,0) (Section 5.2.3) leading to simpler code. Note that two degrees of parallelism are only meaningful when the number of cores is not a prime number. The slight drop in performance for $N = 4000$ for the sequential case is due to sub-optimal L2 cache tile sizes.

6.5 LU Decomposition

Three tiling hyperplanes are found – all belonging to a single band of permutable loops. The first statement though lower-dimensional is naturally sunk into a a 3-dimensional fully permutable space. Thus, there are two degrees of pipelined parallelism. Exploiting both degrees of pipelined parallelism requires a tile wavefront of

```
for (k=0; k<N; k++) {
    for (j=k+1; j<N; j++) {
        a[k][j] = a[k][j]/a[k][k];
    }
    for(i=k+1; i<N; i++) {
        for (j=k+1; j<N; j++) {
            a[i][j] = a[i][j]−a[i][k]∗a[k][j];
        }
    }
}
```

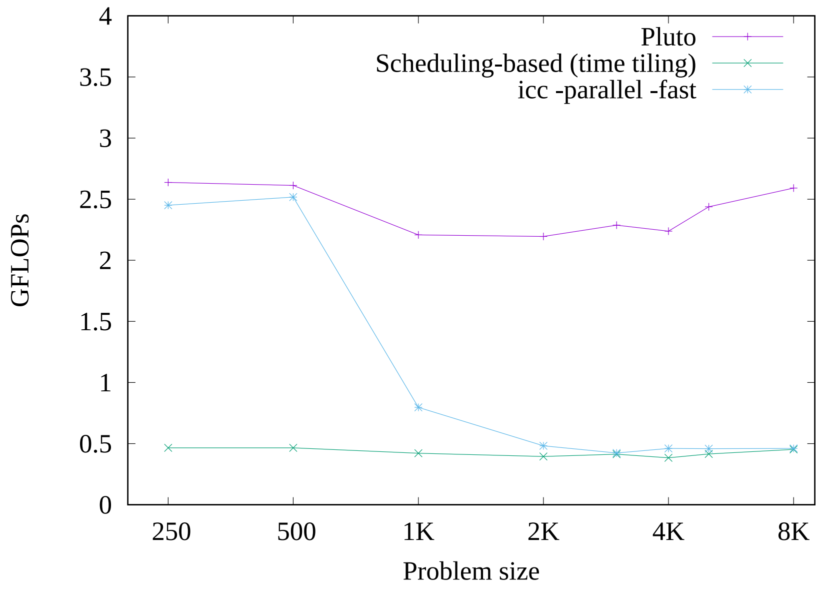
Figure 6.7: LU decomposition (non-pivoting) form

(1,1,1) while exploiting only one requires (1,1,0). The code for the latter is likely to be less complex, however, has a lesser computation to communication ratio. Performance results on the quad core machine are shown in Figure 6.9. The GFLOPs is computed using an operation count of $\frac{2N^3}{3}$. Tiling was done for both L1 and L2 caches. The transformation was shown in detail in Sections 5.2.2 and 5.2.3. The second hyperplane (0,1,0) which is the original j loop is also inner parallel. It is thus moved innermost only in the L1 tile to assist compiler auto-vectorization (Section 5.6.2).

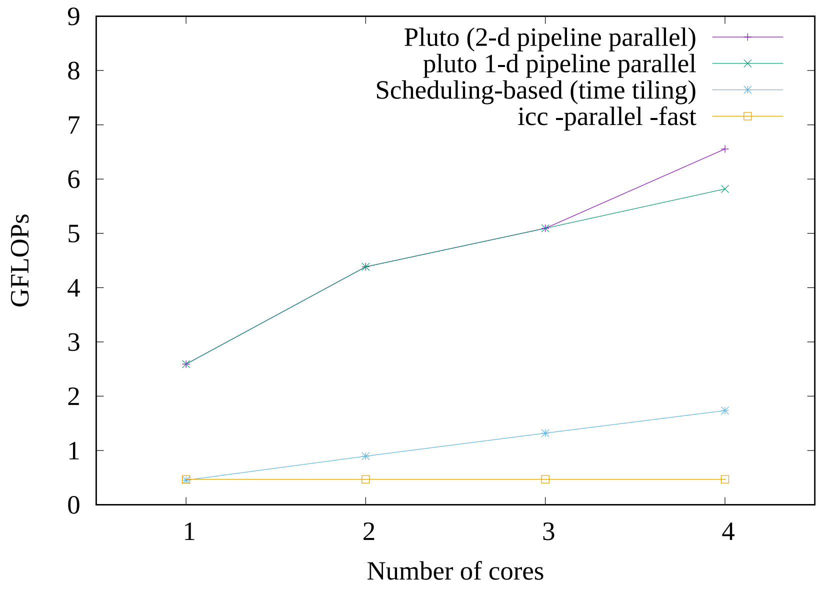
With scheduling-based approaches, the schedules will be: $\theta_{S1}(k, i, j) = 2k$ and $\theta_{S_2}(k, i, j) = 2k + 1$. In this case, time tiling is readily enabled by choosing a simple allocation along i and j since such an allocation has non-negative dependence components though non-uniform. Again, like the Jacobi code, the code complexity appears to make the schedule-based transformation perform poorly for a single thread. As for ICC, we find that it does not parallelize this code.

```
for (t0=−1;t0<=floord(9∗N−17,128);t0++) {
  lb1=max(max(ceild(8∗t0−63,72),0),ceild(16∗t0−N+2,16));
  ub1=min(floord(N−1,128),floord(16∗t0+15,16));
#pragma omp parallel for shared(t0,lb1,ub1) private(t1,t2,t3,t4,t5)
  for (t1=lb1; t1<=ub1; t1++) {
        for (t2=max(ceild(8∗t0−8∗t1−105,120),ceild(8∗t0−8∗t1−7,8));
                                                  t2<=floord(N−1,16);t2++) {
          if (t0 == t1+t2) {
            for (t3=max(0,16∗t2);t3<=min(min(16∗t2+14,N−2),128∗t1+126);t3++) {
              for (t5=max(t3+1,128∗t1);t5<=min(N−1,128∗t1+127);t5++) {
                a[t3][t5]=a[t3][t5]/a[t3][t3];
              }
              for (t4=t3+1;t4<=min(16∗t2+15,N−1);t4++) {
                for (t5=max(t3+1,128∗t1);t5<=min(N−1,128∗t1+127);t5++) {
                  a[t4][t5]=a[t4][t5]−a[t4][t3]∗a[t3][t5];
                }
              }
            }
          }
          for (t3=max(0,16∗t0−16∗t1);
                      t3<=min(min(16∗t2−1,16∗t0−16∗t1+15),128∗t1+126);t3++) {
            for (t4=16∗t2;t4<=min(N−1,16∗t2+15);t4++) {
              for (t5=max(128∗t1,t3+1);t5<=min(N−1,128∗t1+127);t5++) {
                a[t4][t5]=a[t4][t5]−a[t4][t3]∗a[t3][t5];
              }
            }
          }
          if ((−t0 == −t1−t2) &&
                  (t0 <= min(floord(16∗t1+N−17,16),floord(144∗t1+111,16)))) {
            for (t5=max(16∗t0−16∗t1+16,128∗t1);t5<=min(N−1,128∗t1+127);t5++) {
              a[16∗t0−16∗t1+15][t5]=
                        a[16∗t0−16∗t1+15][t5]/a[16∗t0−16∗t1+15][16∗t0−16∗t1+15];
            }
          }
        }
  }
}
```

Figure 6.8: LU decomposition parallelized code (register tiling, L2 tiling, and vector pragmas not shown); context: $N \geq 2$



(a) Single core (L1 and L2 tiled)



(b) On a quad core: N=8000

Figure 6.9: LU performance

6.6 Matrix Vector Transpose

```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    x1[i] = x1[i] + a[i][j]*y1[j];
  }
}

for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    x2[i] = x2[i] + a[j][i]*y2[j];
  }
}
```

(a) Original code

```
for (t0=0;t0<=N−1;t0++) {
  for (t1=0;t1<=N−1;t1++) {
      x1[t0] = x1[t0] + a[t0][t1]*y1[t1];
      x2[t1] = x2[t1] + a[t0][t1]*y2[t0];
  }
}
```

(b) Transformed

$$\mathcal{T}_{S_1} : \begin{pmatrix} t_0 \\ t_1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \qquad \mathcal{T}_{S2} : \begin{pmatrix} t_0 \\ t_1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix}$$

Figure 6.10: Matrix vector transpose

The MVT kernel is a sequence of two matrix vector transposes as shown in Figure 6.10 (a). It is encountered inside a time loop in Biconjugate gradient. The only inter-statement dependence is a non-uniform read/input on matrix A. The cost function bounding (3.7) leads to minimization of this dependence distance by fusion of the first MV with the permuted version of the second MV. As a result, the δ_e for this dependence becomes 0 for both $t0$ and $t1$). This however leads to loss of synchronization-free parallelism, since, in the fused form, each loop satisfies a de-

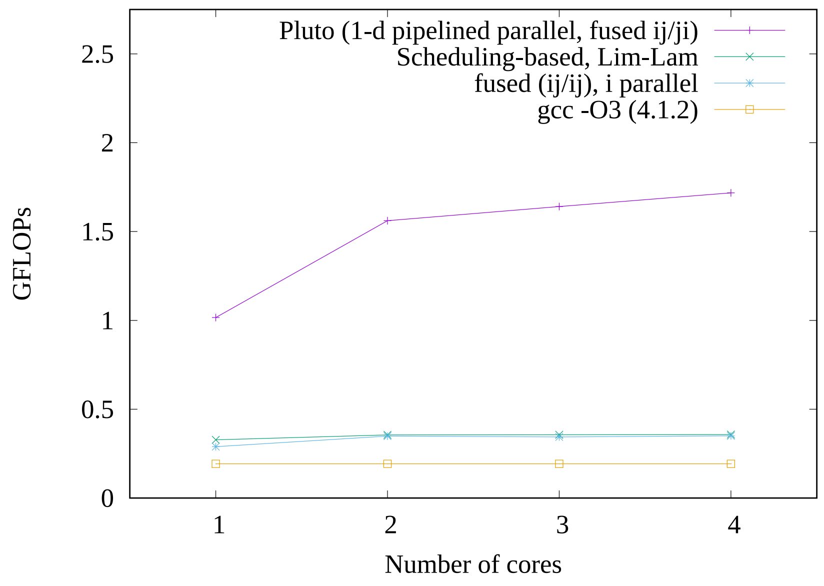


Figure 6.11: MVT performance on a quad core: N=8000

pendence. However, since these dependences have non-negative components, parallel code can be generated corresponding to one degree of pipelined parallelism.

Existing techniques, even if they consider input dependences, cannot automatically fuse the first MV with the permuted version of the second MV. Note that each of the matrix vector multiplies is one strongly connected component. Hence, previous approaches are only able to extract synchronization-free parallelism from each of the MVs separately with a barrier between the two, giving up reuse on array A. Figure 6.11 shows the results for a problem size $N = 8000$. Note that both the optimized versions were tiled for the L1 cache. Fusion of ij with ij does not exploit reuse on matrix A, whereas the code that our tool comes up with performs best – it fuses ij with ji, tiles it and exploits a degree of pipelined parallelism.

6.7 3-D Gauss-Seidel successive over-relaxation

```
for (t=0; t<=T−1; t++) {
    for (i=1; i<=N−2; i++) {
        for (j=1; j<=N−2; j++) {
            a[i][j] = (a[i−1][j−1] + a[i−1][j] + a[i−1][j+1]
                        + a[i][j−1] + a[i][j] + a[i][j+1]
                        + a[i+1][j−1] + a[i+1][j] + a[i+1][j+1])/9.0;
        }
    }
}
```

$$M_S = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 2 & 1 & 1 \end{pmatrix}$$

Figure 6.12: 3-d Seidel code (original) and transformation

The Gauss-Seidel computation (Figure 6.12) allows tiling of all three dimensions after skewing. The transformation our tool obtains skews the first space dimension by a factor of one with respect to time, while the second one by a factor of two with time and one with respect to the previous space dimension. The transformation is also shown in Figure 6.12. Two degrees of pipelined parallelism can be extracted subsequently after creating 3-d tiles. Figure 6.13 shows the performance improvement achieved with 2-d pipelined parallel space as well as 1-d: the latter is better in practice mainly due to simpler code. Again, icc is unable to parallelize this code. The GFLOPs performance is on the lower side since unroll-jamming of non-rectangular loops is not

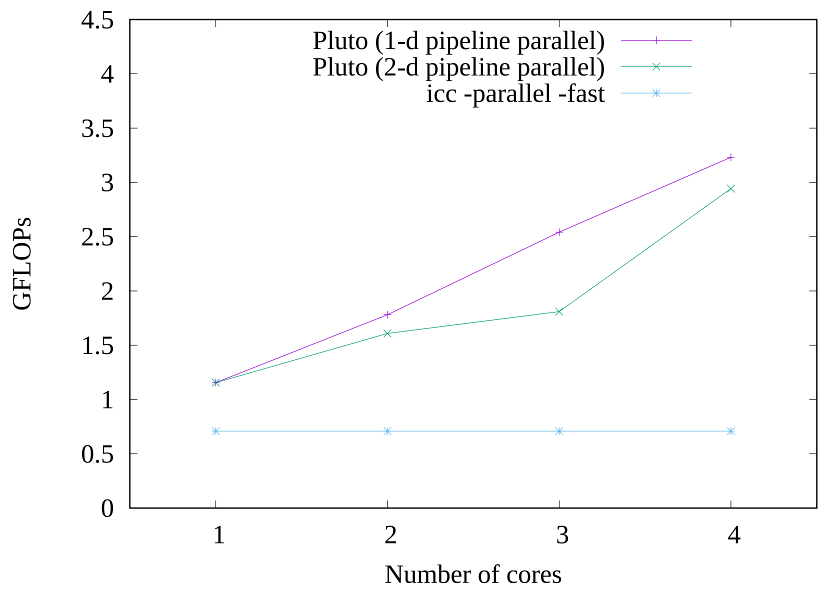


Figure 6.13: 3-D Gauss Seidel on Intel Q6600: $N_x = N_y = 2000$; $T = 1000$

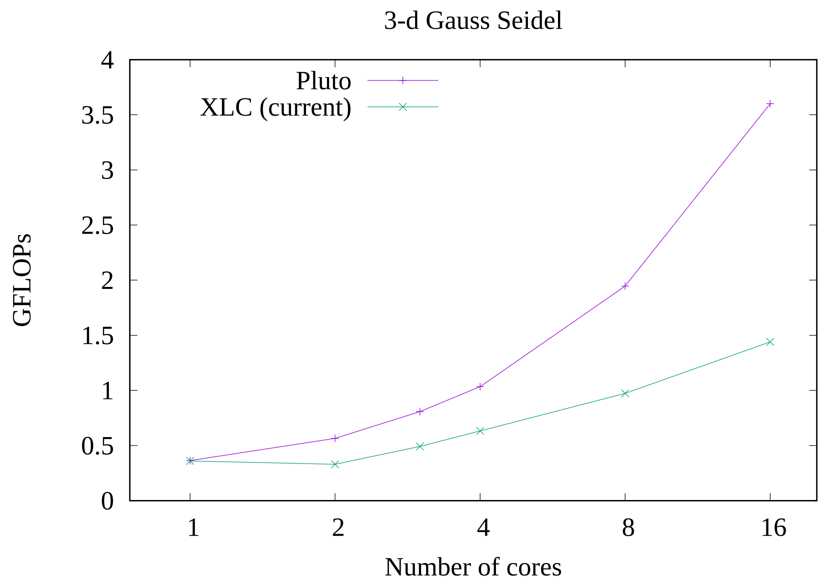


Figure 6.14: 3-D Gauss Seidel on an 8-way Power5 SMP: $N_x = N_y = 2000$; $T = 1000$

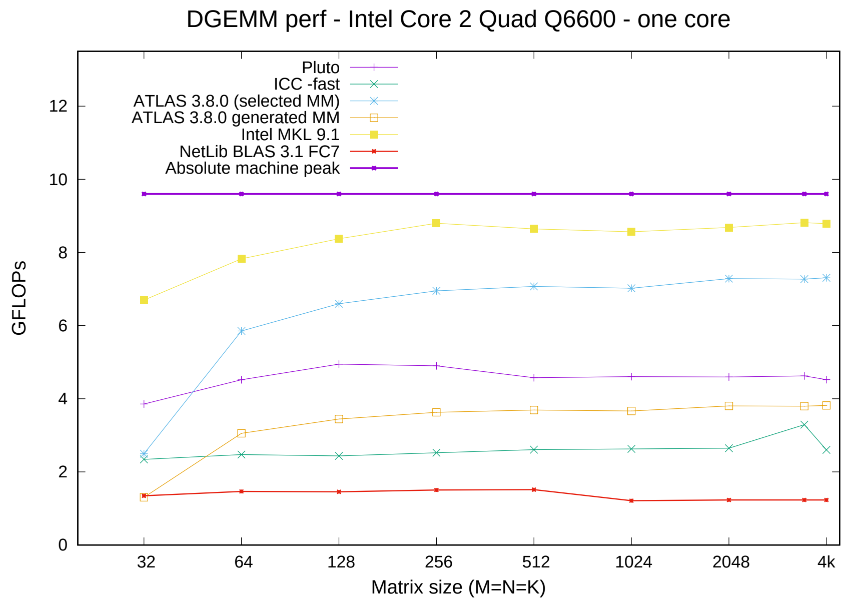
yet supported. Also, this code is not readily vectorized since none of intra-tile loops are fully parallel.

6.8 DGEMM

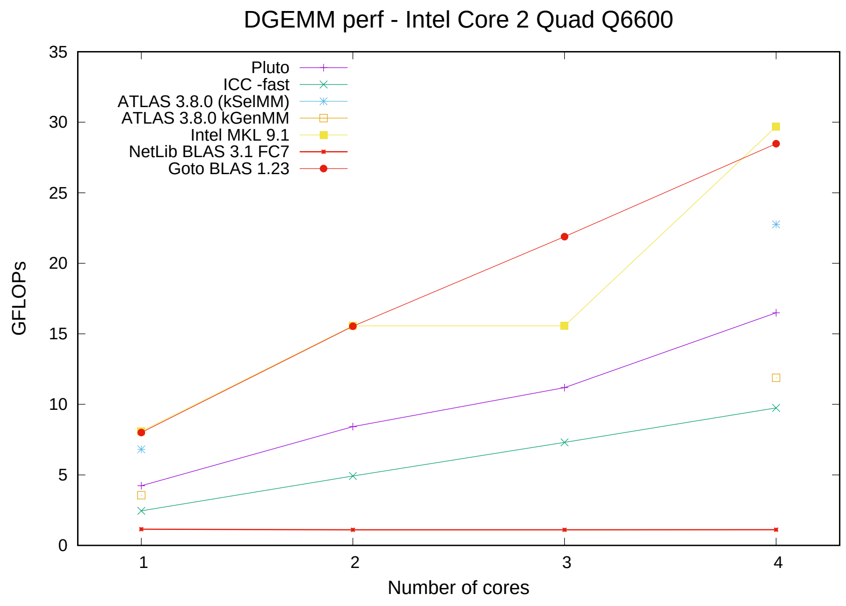
We now compare the Pluto generated code for DGEMM with several highly-tuned libraries. The purpose of this comparison is to see how far Pluto code is from the machine peak and the factors responsible for this gap. Figure 6.15 shows that the Pluto code is within 1.5 to 2x of vendor-supplied BLAS. The difference appears to be mainly due to the performance left on the table by the auto-vectorization performed on transformed Pluto code by ICC. Merging this gap is the subject of one of the threads of future work discussed briefly in the next chapter.

6.9 Experiments with Fusion-critical Codes

We now show the benefits of the techniques proposed in Chapter 4 by evaluating codes for which performing loop fusion in conjunction with other transformations is important. Some comparisons include a version of the benchmark written with vendor supplied libraries. On the Intel Core 2 machine, we used the Intel Math Kernel Library (MKL) 10.0 that includes a multithreaded implementation of the Basic Linear Algebra Software (BLAS) routines highly optimized for Intel processors. Similarly, for the AMD machines, the AMD Core Math Library (ACML) version 4.1 was used.



(a) Single core



(b) Parallel: $M = N = K = 2048$

Figure 6.15: DGEMM perf: Pluto vs. Hand-tuned

6.9.1 GEMVER

The GEMVER kernel is a combination of outer products and matrix vector products. It is used for householder bidiagonalization. The BLAS version of the GEMVER kernel from Siek et al. [SKJ08] along with the linear algebraic specification is shown in Figure 6.16. The nested loop code is in Figure 6.17. Permuting and fusing the first two loop nests is the key optimization to reduce cache misses. Pluto's fusion algorithm is able to cut between the second and third statement, and the third and fourth statement. The fused code is automatically tiled for caches and registers too. The parallel loop inside the L1 tile is made the innermost for vectorization as described in Section 5.6.2. Figures 6.9.1, 6.18, and 6.20 show performance results on three different architectures. The performance is relatively lower on the Opteron as ICC does not vectorize Pluto generated code or the original code for this architecture, while the version written with ACML is naturally a hand-vectorized one.

$$\begin{aligned} B &= A + u_1 v_1^T + u_2 v_2^T \\ x &= \beta B^T y + z \\ w &= \alpha B x \end{aligned}$$

```
dcopy(m * n, A, B, 1);
dger(m, n, 1.0, u1, 1, v1 , 1, B, m);
dger(m, n, 1.0, u2, 1, v2 , 1, B, m);
dcopy(n, z, x, 1);
dgemv('T', m, n, beta, B, m, y, 1, 1.0, x, 1);
dgemv('N', m, n, alpha, B, m, x, 1, 0.0, w, 1);
```

Figure 6.16: GEMVER (high-level specification)

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        B[i][j] = A[i][j] + u1[i]*v1[j] + u2[i]*v2[j];

for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        x[i] = x[i] + beta* B[j][i]*y[j];

for (i=0; i<N; i++)
    x[i] = x[i] + z[i];

for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        w[i] = w[i] + alpha* B[i][j]*x[j];
```

Figure 6.17: Nested loops for GEMVER

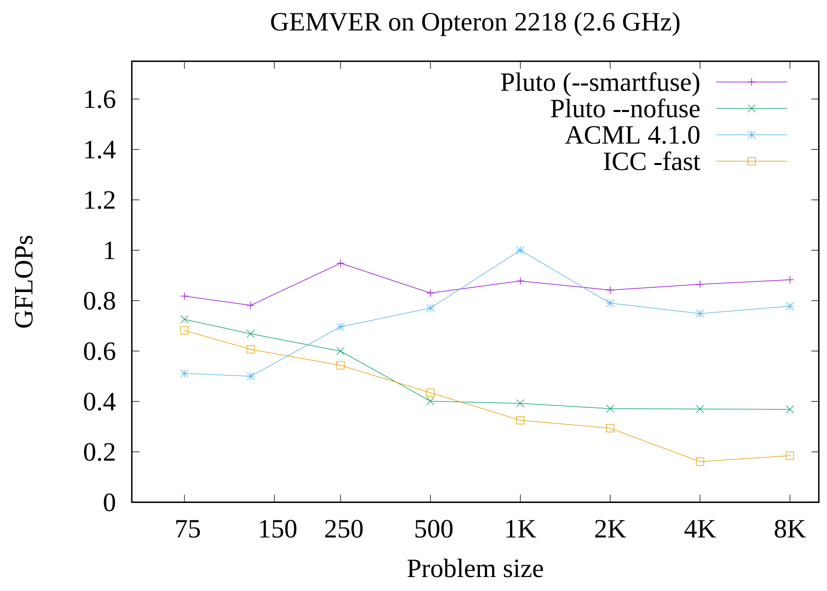


Figure 6.18: GEMVER on Opteron

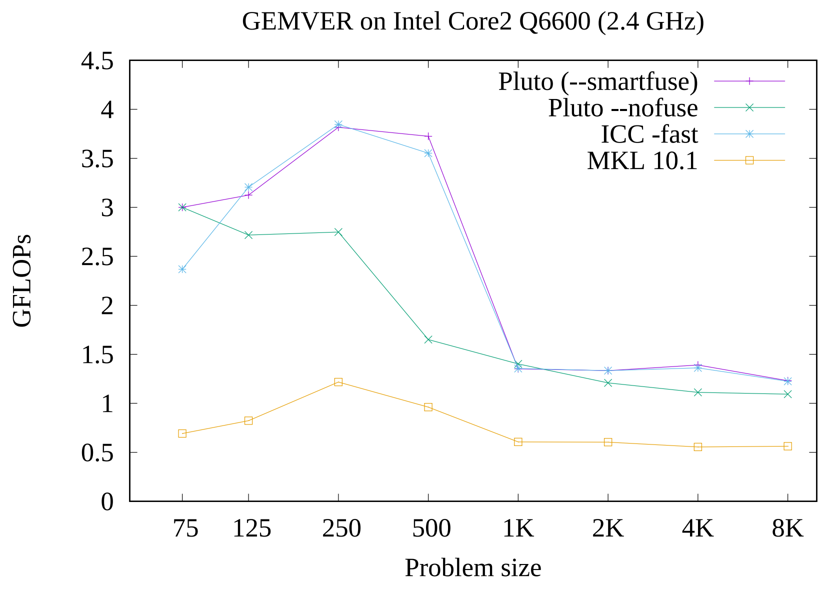


Figure 6.19: GEMVER on Intel Core 2 Quad

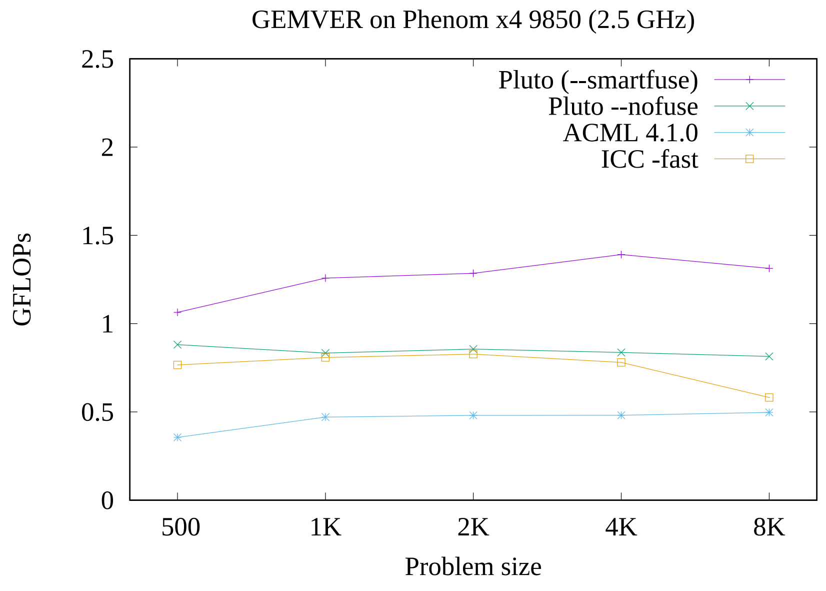


Figure 6.20: GEMVER on AMD Phenom x4 9850

6.9.2 Doitgen

The Doitgen kernel is part of MADNESS [MAD], a scientific simulation framework in many dimensions using adaptive multiresolution methods in multiwavelet bases [MAD]. The computation is quite similar to matrix-matrix multiplication and is shown in Figure 6.21. The kernel is special in that it involves small problem sizes and is called repeatedly from higher level routines. Figure 6.23 and Figure 6.24 show significantly higher performance with Pluto over the same kernel written with BLAS library calls. Register tiling of the imperfect loop nest is responsible for most of the improvement here for the smaller problem sizes. The tiled code is shown in Figure 6.22. Creating 4-d tiles for this code involves an intricacy described in Section 5.4.2. Previous approaches [WL91a, LCL99, AMP01] that just tile every band of permutable loops in a hierarchy will be unable to create 4-d tiles for the second statement.

6.10 Analysis

All the above experiments show very high speedups with our approach, both for single thread and multicore parallel execution. The performance improvement is very significant over production compilers as well as state-of-the-art from the research community. Speedup ranging from 2x to 5x are obtained over previous automatic transformation approaches in most cases, while an order of 10x improvement is obtained over native production compilers. Linear to super-linear speedups are seen for almost all compute-intensive kernels considered here due to optimization for locality

```
for (r=0; r<N; r++) {
    for (q=0; q<N; q++) {
        for (p=0; p<N; p++) {
            sum[r][q][p] = 0;
            for (s = 0; s< N; s++) {
                sum[r][q][p] = sum[r][q][p] + A[r][q][s]*C4[s][p];
            }
        }
        for (p=0; p<N; p++) {
            A[r][q][p] = sum[r][q][p];
        }
    }
}
```

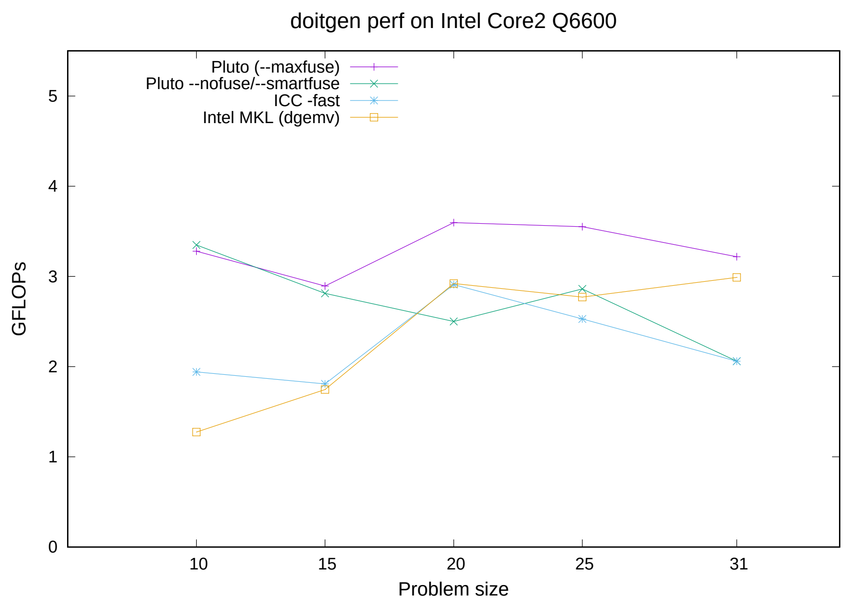
Figure 6.21: Doitgen

as well as parallelism. To the best of our knowledge, such speedup's have not been reported by any automatic compiler framework as general as ours.

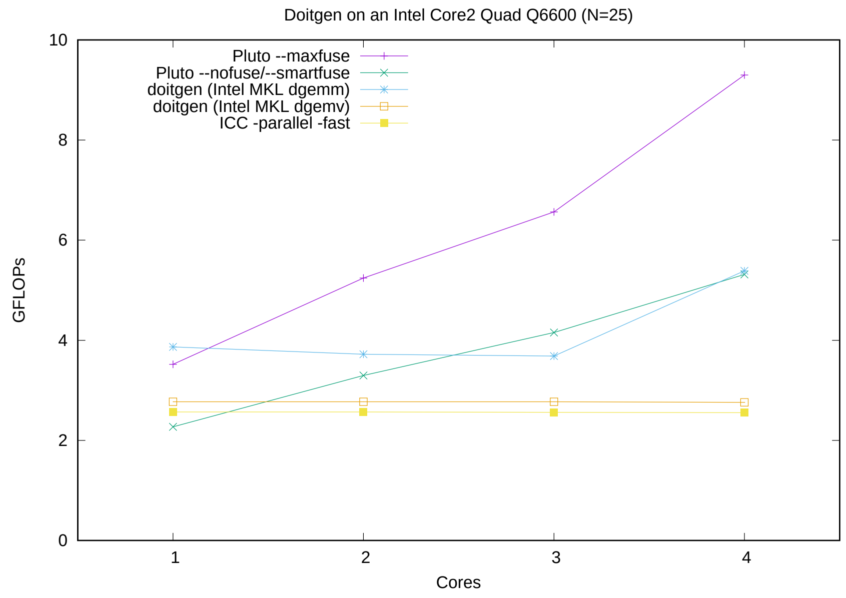
Hand-parallelization of many of the examples we considered here is extremely tedious and not feasible in some cases, especially when time skewed code has to be pipelined parallelized; this coupled by the fact that the code has to be tiled for at least for one level of local cache, and a 2-d pipelined parallel schedule of 3-d tiles is to be obtained makes manual optimization very complex. The performance of the optimized stencil codes through our system is already in the range of that of hand optimized versions reported in [KDW+06]. Also, for many of the codes, a simple parallelization strategy of exploiting inner parallelism and leaving the outer loop sequential (i.e., no time tiling) hardly yields any parallel speedup (Figure 6.5(b),

```
lb1=0;
ub1=floord(N−1,4);
#pragma omp parallel for shared(lb1,ub1) private(t0,t1,t2,t3,t4,t5,t6,t7,t8,t9)
for (t0=lb1; t0<=ub1; t0++) {
  for (t1=0;t1<=floord(N−1,8);t1++) {
    for (t3=0;t3<=floord(N−1,31);t3++) {
      for (t5=max(0,4∗t0);t5<=min(N−1,4∗t0+3);t5++) {
        for (t6=max(0,8∗t1);t6<=min(N−1,8∗t1+7);t6++) {
          for (t9=max(0,31∗t3);t9<=min(N−1,31∗t3+30);t9++) {
            {sum[t5][t6 ][ t9]=0;} ;
          }
        }
      }
    }
    for (t3=0;t3<=floord(N−1,31);t3++) {
      for (t4=0;t4<=floord(N−1,8);t4++) {
        for (t5=max(0,4∗t0);t5<=min(N−1,4∗t0+3);t5++) {
          for (t6=max(0,8∗t1);t6<=min(N−1,8∗t1+7);t6++) {
            for (t8=max(0,8∗t4);t8<=min(N−1,8∗t4+7);t8++) {
              for (t9=max(0,31∗t3);t9<=min(N−1,31∗t3+30);t9++) {
                {sum[t5][t6 ][ t9]=A[t5][t6 ][ t8]∗C4[t8][t9]+sum[t5][t6 ][ t9 ];}  ;
              }
            }
          }
        }
      }
    }
    for (t3=0;t3<=floord(N−1,31);t3++) {
      for (t5=max(0,4∗t0);t5<=min(N−1,4∗t0+3);t5++) {
        for (t6=max(0,8∗t1);t6<=min(N−1,8∗t1+7);t6++) {
          for (t9=max(0,31∗t3);t9<=min(N−1,31∗t3+30);t9++) {
            {A[t5][t6 ][ t9]=sum[t5][t6 ][ t9 ];}  ;
          }
        }
      }
    }
  }
}
```

Figure 6.22: Doitgen tiled code (register tiling and vector pragmas not shown)

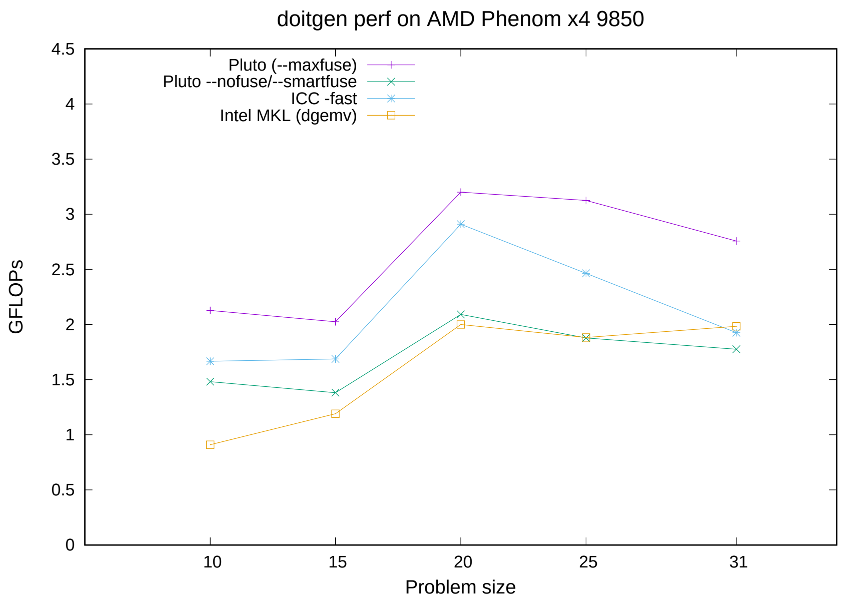


(a) Single core

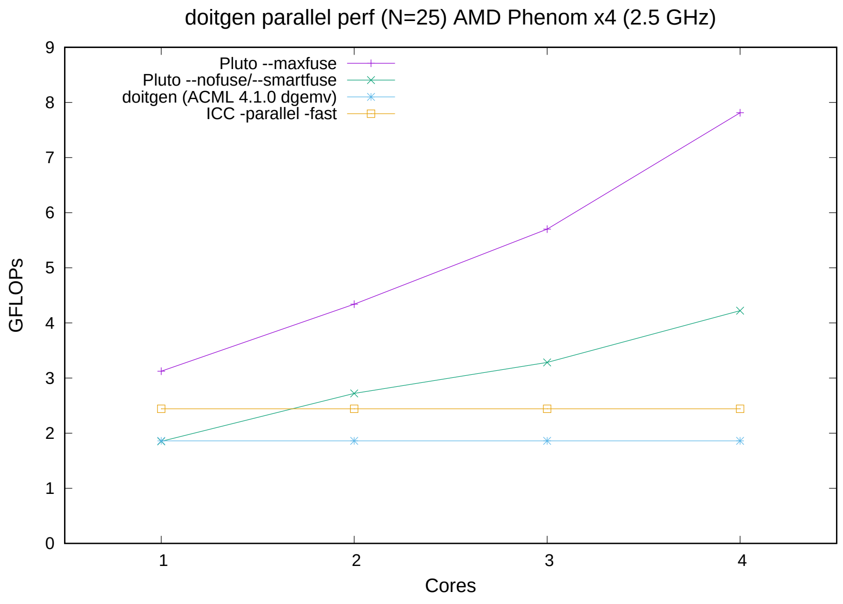


(b) Parallel: $N = 25$

Figure 6.23: Doitgen on an Intel Core 2 Quad



(a) Single core



(b) Parallel: $N = 25$

Figure 6.24: Doitgen on AMD Phenom x4

Figure 6.1(b)). Scheduling-based approaches that do not perform time tiling, or production compilers' auto-parallelizers perform such parallelization.

As mentioned before, tile sizes were not optimized through any extensive search or a model. Also, Orio does not yet support unroll-jamming for non-rectangular iteration spaces: this impacted the performance for LU and Gauss-Seidel. Using models for tile size selection, with some amount of empirical search, in a manner decoupled with the pure model-driven scheme presented is a reasonable approach to take this performance higher. For simpler codes like matrix-matrix multiplication, this latter phase of optimization, though very simple and straightforward when compared to the rest of our system, brings most of the benefits.

As for fusion for long sequences, it is almost clear that finding the highest performing parallel code taking into account fusion, tiling, and storage optimization needs some kind of iterative techniques. However, finding reasonably good fusion structures for loop nests with up to a handful of statements can be achieved purely through models and heuristics. Interactions with the hardware (mainly prefetching) create more complications and the best transformation is often architecture-specific. Presence or the lack of explicit copying also has an impact. Hence, a study of fusion has to be revisited once automatic explicit copying is implemented into Pluto.

CHAPTER 7

CONCLUSIONS AND FUTURE DIRECTIONS

In this dissertation, we presented the theory, design, and implementation of an automatic polyhedral program optimization system that can optimize sequences of arbitrarily-nested loops simultaneously for coarse-grained parallelism and locality. The framework has been fully implemented into a tool, Pluto, to generate OpenMP parallel code from regular C program sections automatically. Experimental results show significantly higher performance for single core and parallel execution on multi-cores, when compared with production compilers as well as state-of-the-art research approaches. In many cases, the parallelization performed through Pluto is infeasible to obtain by hand. A beta release of the Pluto system including all codes used for experimental evaluation in this dissertation are available at [Plu].

Past studies on loop fusion were done in an isolated fashion not integrated with other transformations for parallelization and locality optimization. We showed how our approach can come up with non-trivial fusions and presented three different fusion algorithms that cover the interesting cases. For several linear algebra kernels, code generated from Pluto beats, by a significant margin, the same kernel implemented with a sequence of calls to BLAS routines linked with hand-tuned vendor supplied libraries.

Domain-specific frontends. The transformation system built here is not just applicable to C or Fortran code, but to any input language from which polyhedral computation domains can be extracted and analyzed. Since our entire transformation framework works in the polyhedral abstraction, only the frontend and the dependence tester needs to be adapted to accept a new language. It could be applied for example to very high-level languages or domain-specific languages to generate high-performance parallel code. Designing new domain-specific frontends for Pluto can be done easily. Corequations [Cor] already is one such high-level language which accepts equations as input – a natural way to represent many scientific calculations. Pluto can directly benefit such languages by serving as their backend optimizer.

Iterative compilation and auto-tuning. The Pluto system leaves a lot of flexibility for future optimization, through iterative and empirical approaches. For very long sequences of loop nests, it appears clear that iterative compilation is needed to find the best fusion structure, mainly because the best choice for the fusion structure along with all other transformations and optimizations appears to be complex to be captured in a model. Integration of polyhedral iterative [PBCV07, PBCC08] and our pure model-driven approach is thus promising. Orthogonally, tuning parameters such as tile sizes and unroll factors, for which we currently do not have a sophisticated model is a necessity. The Orio auto-tuner [Ori, NHG07] recently developed at the

Argonne National Labs already provides a solution by providing auto-tuning capabilities on top of Pluto as one of its features. Model-driven empirical optimization and automatic tuning approaches (e.g., ATLAS [WD98, WPD00]) have been shown to be very effective in optimizing single-processor execution for some regular kernels like matrix-matrix multiplication [WPD00, YLR$^+$03]. There is interest in developing effective empirical tuning approaches for arbitrary input kernels. Pluto can enable such model-driven or guided empirical search to be applied to arbitrary affine programs, in the context of both sequential and parallel execution.

Memory optimization. The integration of false dependence removal techniques [Fea91, PW92, MAL93, TP94, BCC98, KS98] and subsequent memory optimization after transformation is a useful component of an optimizer for imperative languages. Programmers typically tend to use the same storage for each iteration of a loop introducing a number of false dependences that hinder parallelization. Removing all false dependences can cause a storage bloat, and hence the need to optimize storage after parallelization. These techniques have been studied well [RW97, SCFS98, CDRV97, Coh99, TVSA01, DSV05, QR00, ABD07]. Storage optimization techniques that are schedule dependent like that of Alias et al. [ABD07] are the natural solution to optimize storage after Pluto determines the transformation. No existing system integrates such storage optimization techniques into an automatic parallelizer yet.

More real-world applications. The application of polyhedral techniques to true real-world programs is still a significant challenge mainly due to the lack of techniques to handle irregular and non-affine programs, procedures, and dynamic control. Making interprocedural dependence analysis work in conjunction with polyhedral static analysis is now interesting to explore given that all the links have been put together for optimization of regular codes. Availability of a robust full-fledged frontend to extract polyhedra is a necessary step. There are currently efforts to bring the polyhedral model to production compilers like GCC [PCB+06]. The availability of such infrastructure is crucial for further progress. In addition, hybrid static-cum-dynamic approaches are promising to explore. [RPR07], for example, uses dynamic techniques to aid static analysis to enable parallelization that cannot be performed statically alone. However, such techniques have not been studied in conjunction with the polyhedral model.

Vector intrinsics. For most kernels evaluated in Chapter 6, there is a significant amount of performance that is left unexploited, even though Pluto generated code provides high speedup. Figure 6.15 gives some indication. The quality of auto-vectorization performed by the compiler on the polyhedral transformed code appears to be one primary reason for this gap. Usage of vector intrinsics and automatically mapping polyhedral computation to vector intrinsics is a very challenging problem. This is even more crucial with several architectures employing wider vector architectures on the horizon [AVX]. Spiral [SPI, PMJ+05] is one domain-specific project that employs such vector intrinsics to generate highly optimized implementations of digital signal processing algorithms.

More target architectures. The framework presented in this dissertation is applicable to any parallel architecture as long as at least one degree of coarse-grained parallelism is desired: this is the case with almost all modern parallel computers. Some works have used Pluto to map programs to GPGPUs [BBK$^+$08a] with additional optimizations like explicit copying for scratchpad memories [BBK$^+$08b] as well as considerations to avoid bank conflicts for SIMD threads. Similarly, the Cell processor, and embedded multiprocessor chips are also good targets. In addition to shared memory architectures, techniques presented are in a form that would also distributed memory automatic parallelization to be addressed incrementally: the incremental step being the efficient generation of message passing code.

Concluding remarks. The Polyhedral model is about two decades old now. With a practical and scalable approach for automatic transformation, all the missing links have been put together for end-to-end automatic parallelization of input that the polyhedral model can readily handle – affine loop nests with static control. There are several codes that fall into this category, parallelization and optimization for which will benefit a large number of users: [CS90, MAD] are just two examples. One can now conclude that polyhedral techniques are very effective for these domains and are the best way to obtain the highest performance parallel code with no effort on part of the programmer. The possibility of extension to more general code is now feasible and interesting to be tried. Thanks to the mathematical abstraction, the techniques are applicable to new languages or domain-specific languages. The problem of auto-parallelization has always been considered a mountain, but mountains can be approximated by polyhedra.

BIBLIOGRAPHY

[ABD07] Christophe Alias, Fabrice Baray, and Alain Darte. Bee+Cl@k: an implementation of lattice-based array contraction in the source-to-source translator Rose. In *Languages Compilers and Tools for Embedded Systems*, pages 73–82, 2007.

[ABRY03] Rumen Andonov, Stefan Balev, Sanjay Rajopadhye, and Nicola Yanev. Optimal semi-oblique tiling. *IEEE Transactions on Parallel and Distributed Systems*, 14(9):944–960, 2003.

[AI91] Corinne Ancourt and Francois Irigoin. Scanning polyhedra with do loops. In *ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 39–50, 1991.

[AK87] Randy Allen and Ken Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, 1987.

[AMP00] N. Ahmed, N. Mateev, and K. Pingali. Tiling imperfectly-nested loop nests. In *Proc. Supercomputing*, 2000.

[AMP01] N. Ahmed, Nikolay Mateev, and Keshav Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loops. *International Journal of Parallel Programming*, 29(5), October 2001.

[AVX] Intel Advanced Vector Extensions (AVX). http://softwareprojects.intel.com/avx/.

[Ban93] Utpal K. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[Bas04a] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *IEEE International Conference on Parallel Architectures and Compilation Techniques*, pages 7–16, September 2004.

[Bas04b] Cédric Bastoul. *Improving Data Locality in Static Control Programs*. PhD thesis, University Paris 6, Pierre et Marie Curie, France, December 2004.

[BBK$^+$08a] M. Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs. In *ACM International conference on Supercomputing (ICS)*, June 2008.

[BBK$^+$08b] M. Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic Data Movement and Computation Mapping for Multi-level Parallel Architectures with Explicitly Managed Memories. In *ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, February 2008.

[BBK$^+$08c] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International conference on Compiler Construction (ETAPS CC)*, April 2008.

[BCC98] Denis Barthou, Albert Cohen, and Jean-Francois Collard. Maximal static expansion. In *ACM SIGPLAN symposium on Principles of Programming Languages*, pages 98–106, 1998.

[BDRR94] Pierre Boulet, Alain Darte, Tanguy Risset, and Yves Robert. (Pen)-ultimate tiling? *Integration, the VLSI Journal*, 17(1):33–51, 1994.

[BDSV98] Pierre Boulet, Alain Darte, Georges-André Silber, and Frédéric Vivien. Loop parallelization algorithms: From parallelism extraction to code generation. *Parallel Computing*, 24(3–4):421–444, 1998.

[BF03] Cédric Bastoul and P. Feautrier. Improving data locality by chunking. In *International conference on Compiler Construction (ETAPS CC)*, pages 320–335, Warsaw, Apr. 2003.

[BHRS08] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2008.

[BHZ] Roberto Bagnara, Patricia Hill, and Enea Zaffanella. PPL: The Parma Polyhedra Library. http://www.cs.unipr.it/ppl/.

[BRS07] Uday Bondhugula, J. Ramanujam, and P. Sadayappan. Automatic mapping of nested loops to FPGAs. In *ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, March 2007.

[CDRV97] Pierre-Yves Calland, Alain Darte, Yves Robert, and Frédéric Vivien. Plugging anti and output dependence removal techniques into loop parallelization algorithm. *Parallel Computing*, 23(1-2):251–266, 1997.

[CGP$^+$05] Albert Cohen, Sylvain Girbal, David Parello, M. Sigler, Olivier Temam, and Nicolas Vasilache. Facilitating the search for compositions of program transformations. In *ACM International conference on Supercomputing*, pages 151–160, June 2005.

[Clo] CLooG: The Chunky Loop Generator. http://www.cloog.org.

[Coh99] Albert Cohen. Parallelization via constrained storage mapping optimization. *Lecture Notes in Computer Science*, 16(15):83–94, 1999.

[Cor] COREquations. http://www.corequations.com.

[CS90] Lawrence A. Covick and Kenneth M. Sando. Four-index transformation on distributed-memory parallel computers. *Journal of Computational Chemistry*, pages 1151 – 1159, November 1990.

[Dar00] Alain Darte. On the complexity of loop fusion. *Parallel Computing*, 26(9):1175–1193, 2000.

[DH99] Alain Darte and Guillaume Huard. Loop shifting for loop compaction. In *International workshop on Languages and Compilers for Parallel Computing*, pages 415–431, 1999.

[DH00] Alain Darte and Guillaume Huard. Loop shifting for loop parallelization. Technical Report RR2000-22, ENS Lyon, May 2000.

[DR96] Michèle Dion and Yves Robert. Mapping affine loop nests. *Parallel Computing*, 22(10):1373–1397, 1996.

[DRV00] Alain Darte, Yves Robert, and Frederic Vivien. *Scheduling and Automatic Parallelization.* Birkhauser Boston, 2000.

[DSV97] Alain Darte, Georges-Andre Silber, and Frederic Vivien. Combining retiming and scheduling techniques for loop parallelization and loop tiling. *Parallel Processing Letters*, 7(4):379–392, 1997.

[DSV05] Alain Darte, Robert Schreiber, and Gilles Villard. Lattice-based memory allocation. *IEEE Transactions on Computers*, 54(10):1242–1257, 2005.

[DV97] Alain Darte and Frederic Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. *International Journal of Parallel Programming*, 25(6):447–496, December 1997.

[Fea88] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

[Fea91] P. Feautrier. Dataflow analysis of scalar and array references. *International Journal of Parallel Programming*, 20(1):23–53, Feb. 1991.

[Fea92a] P. Feautrier. Some efficient solutions to the affine scheduling problem: Part I, one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, 1992.

[Fea92b] P. Feautrier. Some efficient solutions to the affine scheduling problem: Part II, multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, 1992.

[Fea94] P. Feautrier. Toward automatic distribution. *Parallel Processing Letters*, 4:233–244, 1994.

[Fea06] P. Feautrier. Scalable and structured scheduling. *International Journal of Parallel Programming*, 34(5):459–487, 2006.

[GAK02] G. Goumas, M. Athanasaki, and N. Koziris. Code Generation Methods for Tiling Transformations. *Journal of Information Science and Engineering*, 18(5):667–691, Sep. 2002.

[GFG05] Martin Griebl, Paul Feautrier, and Armin Größlinger. Forward communication only placements and their use for parallel program construction. In *Languages and Compilers for Parallel Computing*, pages 16–30. Springer-Verlag, 2005.

[GFL00] Martin Griebl, Paul Feautrier, and Christian Lengauer. Index set splitting. *International Journal of Parallel Programming*, 28(6):607–631, 2000.

[GLW98] Martin Griebl, Christian Lengauer, and S. Wetzel. Code generation in the polytope model. In *IEEE International Conference on Parallel Architectures and Compilation Techniques*, pages 106–111, 1998.

[GR07] Gautam Gupta and Sanjay Rajopadhye. The z-polyhedral model. In *ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 237–248, 2007.

[Gri04] Martin Griebl. *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. University of Passau, 2004. Habilitation thesis.

[GVB$^+$06] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations. *International Journal of Parallel Programming*, 34(3):261–317, June 2006.

[HCF97] K. Högstedt, Larry Carter, and Jeanne Ferrante. Determining the idle time of a tiling. In *ACM SIGPLAN symposium on Principles of Programming Languages*, pages 160–173, 1997.

[HCF99] K. Högstedt, Larry Carter, and Jeanne Ferrante. Selecting tile shape for minimal execution time. In *symposium on Parallel Algorithms and Architectures*, pages 201–211, 1999.

[HS02] Edin Hodzic and Weijia Shang. On time optimal supernode shape. *IEEE Transactions on Parallel and Distributed Systems*, 13(12):1220–1233, 2002.

[Int] Intel C Compiler Manuals. Intel C/C++ Optimizing Applications. Document Number: 315890-002US.

[IT88] F. Irigoin and R. Triolet. Supernode partitioning. In *ACM SIGPLAN Principles of Programming Languages*, pages 319–329, 1988.

[JLF02] Marta Jiménez, José M. Llabería, and Agustín Fernández. Register tiling in nonrectangular iteration spaces. *ACM Transactions on Programming Languages and Systems*, 24(4):409–453, 2002.

[KDW$^+$06] S. Kamil, Kaushik Datta, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yellick. Implicit and explicit optimization for stencil computations. In *ACM SIGPLAN workshop on Memory Systems Perofmance and Correctness*, 2006.

[Kel96] W. Kelly. Optimization within a unified transformation framework. Technical Report CS-TR-3725, 1996.

[KM93] K. Kennedy and K. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Languages and Compilers for Parallel Computing*, pages 301–320, 1993.

[KP93] W. Kelly and W. Pugh. Determining schedules based on performance estimation. Technical Report UMIACS-TR-9367, University of Maryland, College Park, December 1993.

[KP95] W. Kelly and W. Pugh. A unifying framework for iteration reordering transformations. Technical Report CS-TR-3430, Dept. of Computer Science, University of Maryland, College Park, 1995.

[KPR95] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *International Symp. on the frontiers of massively parallel computation*, pages 332–341, Feb. 1995.

[KRSR07] Daegon Kim, Lakshminarayanan Renganarayanan, Michelle Strout, and Sanjay Rajopadhye. Multi-level tiling: 'm' for the price of one. In *Supercomputing*, 2007.

[KS98] Kathleen Knobe and Vivek Sarkar. Array ssa form and its use in parallelization. In *ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 107–120, 1998.

[Lam74] Leslie Lamport. The parallel execution of do loops. *Communications of the ACM*, 17(2):83–93, 1974.

[LCL99] A. Lim, Gerald I. Cheong, and Monica S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *ACM International conference on Supercomputing*, pages 228–237, 1999.

[LeV92] H. LeVerge. A note on Chernikova's algorithm. Technical Report Research report 635, IRISA, February 1992.

[LL98] A. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing*, 24(3-4):445–475, 1998.

[LLL01] A. Lim, S. Liao, and M. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 103–112, 2001.

[Loo] The LooPo Project - Loop parallelization in the polytope model. http://www.fmi.uni-passau.de/loopo.

[LP94] Wei Li and Keshav Pingali. A singular loop transformation framework based on non-singular matrices. *International Journal of Parallel Programming*, 22(2):183–205, 1994.

[MAD] Madness: Multiresolution adaptive numerical scientific simulation. http://www.csm.ornl.gov/ccsg/html/projects/madness.html.

[MAL93] D. E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array-data flow analysis and its use in array privatization. In *ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 2–15, 1993.

[MS97] Nimrod Megiddo and Vivek Sarkar. Optimal weighted loop fusion for parallel programs. In *symposium on Parallel Algorithms and Architectures*, pages 282–291, 1997.

[NHG07] Boyana Norris, Albert Hartono, and William Gropp. *Annotations for performance and productivity.* 2007. Preprint ANL/MCS-P1392-0107.

[Ori] Orio. https://trac.mcs.anl.gov/projects/performance/wiki/Orio.

[PBCC08] L-N Pouchet, Cédric Bastoul, John Cavazos, and Albert Cohen. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *ACM SIGPLAN PLDI*, Tucson, Arizona, June 2008.

[PBCV07] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *International symposium on Code Generation and Optimization*, March 2007.

[PCB$^+$06] Sébastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, P. Jouvelot, G.-A. Silber, and N. Vasilache. GRAPHITE: Loop optimizations based on the polyhedral model for GCC. In *Proc. of the 4th GCC Developper's Summit*, Ottawa, Canada, June 2006.

[Pen55] Roger Penrose. A generalized inverse for matrices. *Proceedings of the Cambridge Philosophical Society*, 51:406–413, 1955.

[PIP] PIP: The Parametric Integer Programming Library. http://www.piplib.org.

[Plu] PLUTO: A polyhedral automatic parallelizer and locality optimizer for multicores. http://pluto-compiler.sourceforge.net.

[PMJ$^+$05] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proc. of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[Pol] PolyLib - A library of polyhedral functions. http://icps.u-strasbg.fr/polylib/.

[PR97] W. Pugh and Evan Rosser. Iteration space slicing and its application to communication optimization. In *International Conference on Supercomputing*, pages 221–228, 1997.

[PR00] W. Pugh and Evan Rosser. Iteration space slicing for locality. In *Languages and Compilers for Parallel Computing*, pages 164–184, London, UK, 2000. Springer-Verlag.

[Pug92] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, August 1992.

[PW92] W. Pugh and David Wonnacott. Eliminating false data dependences using the omega test. In *ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 140–151, 1992.

[QK06] Apan Qasem and Ken Kennedy. Profitable loop fusion and tiling using model-driven empirical search. In *Proc. ACM International conference on Supercomputing*, pages 249–258, 2006.

[QR00] Fabien Quilleré and Sanjay Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Transactions on Programming Languages and Systems*, 22(5):773–815, 2000.

[QRW00] Fabien Quilleré, Sanjay V. Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, 2000.

[Qui87] Patrice Quinton. The systematic design of systolic arrays. In *Centre National de Recherche Scientifique on Automata networks in computer science: theory and applications*, pages 229–260, Princeton, NJ, USA, 1987. Princeton University Press.

[Ram92] J. Ramanujam. Non-unimodular transformations of nested loops. In *Supercomputing*, pages 214–223, 1992.

[Ram95] J. Ramanujam. Beyond unimodular transformations. *Journal of Supercomputing*, 9(4):365–389, 1995.

[RKRS07] L. Renganarayana, DaeGon Kim, Sanjay Rajopadhye, and Michelle Mills Strout. Parameterized tiled loops for free. In *ACM SIGPLAN symposium on Programming Languages Design and Implementation*, pages 405–414, 2007.

[Rob01] Arch D. Robison. Impact of economics on compiler optimization. In *Proc. of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 1–10, 2001.

[RPR07] Silvius Rus, Maikel Pennings, and Lawrence Rauchwerger. Sensitivity analysis for automatic parallelization on multi-cores. In *Proc. of the 21st International Conference on Supercomputing*, pages 263–273, 2007.

[RS92] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–230, 1992.

[RW97] S. Rajopadhye and D. Wilde. Memory reuse analysis in the polyhedral model. *Parallel Processing Letters*, 7(2):207–215, June 1997.

[SCFS98] M. Strout, Larry Carter, Jeanne Ferrante, and Beth Simon. Schedule-independent storage mapping for loops. In *Proc. of the International conference on Architectural Support for Programming Languages and Operating Systems*, pages 24–33, October 3–7, 1998.

[Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, 1986.

[SD90] R. Schreiber and J. Dongarra. Automatic blocking of nested loops. Technical report, University of Tennessee, Knoxville, TN, August 1990.

[SKJ08] Jeremy G. Siek, Ian Karlin, and E. R. Jessup. Build to order linear algebra kernels. In *IPDPS: HIPS-POHLL workshop*, April 2008.

[SL99] Yonghong Song and Zhiyuan Li. New tiling techniques to improve cache temporal locality. In *ACM SIGPLAN symposium on Programming Languages Design and Implementation*, pages 215–228, 1999.

[SM97] S. Singhai and K. McKinley. A Parameterized Loop Fusion Algorithm for Improving Parallelism and Cache Locality. *The Computer Journal*, 40(6):340–355, 1997.

[S.P00] S.P.K. Nookala and Tanguy Risset. A library for $\mathcal{Z}$-polyhedral operations. Technical Report PI 1330, IRISA, Rennes, France, 2000.

[SPI] SPIRAL: Software/Hardware Generation for DSP Algorithms. http://www.spiral.net.

[Top] The Top 500 Supercomputers. http://www.top500.org.

[TP94] Peng Tu and David A. Padua. Automatic array privatization. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 500–521, 1994.

[TVSA01] William Thies, Frédéric Vivien, Jeffrey Sheldon, and Saman P. Amarasinghe. A unified framework for schedule and storage optimization. In *ACM SIGPLAN symposium on Programming Languages Design and Implementation*, pages 232–242, 2001.

[Vas07] Nicolas Vasilache. *Scalable Program Optimization Techniques in the Polyhedral Model*. PhD thesis, Université de Paris-Sud, INRIA Futurs, September 2007.

[VBC06] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. Polyhedral code generation in the real world. In *International conference on Compiler Construction (ETAPS CC)*, pages 185–201, March 2006.

[VBGC06] Nicolas Vasilache, Cédric Bastoul, Sylvain Girbal, and Albert Cohen. Violated dependence analysis. In *ACM International conference on Supercomputing*, June 2006.

[VCP07] Nicolas Vasilache, Albert Cohen, and Louis-Noel Pouchet. Automatic correction of loop transformations. In *IEEE International Conference on Parallel Architectures and Compilation Techniques*, 2007.

[Viv02] Frédéric Vivien. On the optimality of Feautrier's scheduling algorithm. In *Proc. of the 8th International Euro-Par Conference on Parallel Processing*, pages 299–308, London, UK, 2002. Springer-Verlag.

[VSB$^+$07] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica*, 48(1):37–66, June 2007.

[WD98] R. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software (ATLAS). In *Proc. Supercomputing '98*, 1998.

[Wil93] D. K. Wilde. A library for doing polyhedral operations. Technical Report RR-2157, IRISA, 1993.

[WL91a] M. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, 1991.

[WL91b] M. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN symposium on Programming Languages Design and Implementation*, pages 30–44, 1991.

[Wol95] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[WPD00] R. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing*, 2000.

[Xue97] Jingling Xue. Communication-minimal tiling of uniform dependence loops. *Journal of Parallel and Distributed Computing*, 42(1):42–59, 1997.

[Xue00] Jingling Xue. *Loop tiling for parallelism.* Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[YKA04] Qing Yi, Ken Kennedy, and Vikram Adve. Transforming complex loop nests for locality. *Journal of Supercomputing*, 27(3):219–264, 2004.

[YLR$^+$03] Kamen Yotov, Xiaoming Li, Gang Ren, Michael Cibulskis, Gerald DeJong, Maria Garzaran, David A. Padua, Keshav Pingali, Paul Stodghill, and Peng Wu. A comparison of empirical and model-driven optimization. In *ACM SIGPLAN symposium on Programming Languages Design and Implementation*, pages 63–76, 2003.

Index