COMPILE-TIME AND RUN-TIME OPTIMIZATIONS FOR ENHANCING LOCALITY AND PARALLELISM ON MULTI-CORE AND MANY-CORE SYSTEMS

DISSERTATION

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

By

Muthu Manikandan Baskaran, B.E., M.S.

Graduate Program in Computer Science and Engineering

The Ohio State University

2009

Dissertation Committee:

Dr. P. Sadayappan, Adviser

Dr. D. K. Panda

Dr. Atanas Rountev

Dr. J. Ramanujam
© Copyright by
Muthu Manikandan Baskaran
2009
ABSTRACT

Current trends in computer architecture exemplify the emergence of multiple processor cores on a chip. The modern multiple-core computer architectures that include general-purpose multi-core architectures (from Intel, AMD, IBM, and Sun), and specialized parallel architectures such as the Cell Broadband Engine and Graphics Processing Units (GPUs) have very high computation power per chip. A significant challenge to be addressed in these systems is the effective load-balanced utilization of the processor cores. Memory subsystem has always been a performance bottleneck in computer systems and it is more so, with the emergence of processor subsystem with multiple on-chip processor cores. Effectively managing the on-chip and off-chip memories and enhancing data reuse to maximize memory performance is another significant challenge in modern multiple-core architectures.

Our work addresses these challenges in multi-core and many-core systems, through various compile-time and run-time optimization techniques. We provide effective automatic compiler support for managing on-chip and off-chip memory accesses, with the compiler making effective decisions on what elements to move in and move out of on-chip memory, when and how to move them, and how to efficiently access the elements brought into on-chip memory. We develop an effective tiling approach for mapping computation in regular programs on to many-core systems like GPUs. We develop an automatic approach for compiler-assisted dynamic scheduling of computation to enhance load balancing for parallel tiled execution on multi-core systems.

There are various issues that are specific to the target architecture which need attention to maximize application performance on the architecture. First, the levels of parallelism available and the appropriate granularity of parallelism needed for the target architecture have to be considered while mapping the computation. Second, the memory access model may be inherent to the architecture and optimizations have to be developed for the specific memory access model. We develop compile-time transformation approaches to address performance factors related to parallelism and data locality that are GPU architecture-specific, and develop an end-to-end compiler framework for GPUs.
Dedicated to my parents
ACKNOWLEDGMENTS

I want to convey my heartfelt gratitude to my adviser, Dr. P. Sadayappan, for guiding me through my PhD journey. He showed faith on me that I could do competitive research. I am indebted to him for inducting me to his group and I have tried to give my full effort to keep up his confidence. He is a great mentor when it comes to steering students to formulate good research problems and think along the correct lines to solve the problems. He is a great person by heart and has always boosted my morale whenever I had a lean patch in my research. To sum up, I would have been nowhere in research without him.

I am very thankful to my co-advisers, Dr. J. (Ram) Ramanujam and Dr. Atanas (Nasko) Rountev. They have helped me to a great extent in getting my research work to a good shape. Ram has made himself available anytime I want. Though he was physically away in Louisiana, I have never felt it. I have gained a lot through the umpteen telephone conversations I have had with him. Nasko is a very systematic and methodical mentor and I am inspired to acquire those qualities to complement my research career.

I am thankful to Dr. Rajesh Bordawekar for taking me as his Summer Intern at IBM TJ Watson Research Center and giving me all flexibility and support to work on an interesting problem. He is not only a good mentor, but also a very great friend who made my stay in New York a very pleasant one.

My roommates, Shantanu Singh and Shirish Tatikonda, need a very special mention. I am very fortunate to have them as my friends. They have always taken care of me with so much love and affection that I have always felt my stay in US as home away from home. I have always cherished and benefited from the technical and personal conversations we have had.

I am thankful to my friends, and current and previous research members of Dr. Saday’s research group - Swarup, Gaurav, Naga, Aniruddh, Uday, Sriram, Raj, Albert, Qingda, Xiaoyang, Jim, Brian, Giri, Karthi, Bhargavi, Siddharth, Tom, and Justin - for making me enjoy my time in DL474. I want to express my sincere thanks to my lab mates Uday, Sriram and Raj for having helped me to concretize my research ideas at various stages of my PhD.
and working with me closely on various publications. I also want to thank my colleagues Naga and Albert for working with me on research publications. I want to convey my special thanks to my friend, undergraduate classmate and colleague in Dr. Saday’s group, Swarup, who introduced me to Dr. Saday. A very special thanks to my friends Karthi, Sai, Santhosh and Vignesh who made my final year of PhD very memorable in various ways.

To conclude my acknowledgment, I want to express my deepest thanks to the people to whom I owe everything in my life - my parents and sister. I would have never achieved anything in life without their love and support. I want to thank my brother-in-law for his support and technical advice. I want to thank my cute little nephew as his presence and thought has always energized me. Last, but not the least, I want to thank the Almighty for giving me the knowledge, strength, energy, and confidence for achieving whatever I have.
VITA

Sep 1998-May 2002 . . . . . . . . . . . . . . . . . . . . . . . . . B.E. Computer Science & Engineering
National Institute of Technology, Trichy, India

Jul 2002-Aug 2004 . . . . . . . . . . . . . . . . . . . . . . . . . . Software Engineer
Siemens Information Systems Limited, India

Sep 2004-present . . . . . . . . . . . . . . . . . . . . . . . . . . . . Graduate Research Associate
Computer Science and Engineering
The Ohio State University

Jun 2008-Sep 2008 . . . . . . . . . . . . . . . . . . . . . . . . . . Research Intern
IBM TJ Watson Research Center
Hawthorne, New York

PUBLICATIONS

Research Publications


**FIELDS OF STUDY**

Major Field: Computer Science and Engineering
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>iv</td>
</tr>
<tr>
<td>Vita</td>
<td>vi</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xi</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xii</td>
</tr>
</tbody>
</table>

**Chapters:**

1. **Introduction**                                                      | 1    |
   1.1 Challenges in Multi-core and Many-core Architectures             | 3    |
   1.2 Key Contributions                                                | 4    |
   1.3 Organization                                                     | 5    |

2. **Overview of Polyhedral Model**                                      | 6    |

3. **Compiler Support for Automatic Data Management in On-chip Memories**| 14   |
   3.1 Details of the Approach                                           | 15   |
      3.1.1 Determining Local Memory Storage                             | 18   |
      3.1.2 Determining Local Memory Array Access Functions             | 20   |
      3.1.3 Generating Data Movement Code                               | 20   |
      3.1.4 Optimizing Data Movement                                    | 22   |
   3.2 Related Work                                                      | 23   |
3.3 Conclusions ................................................................. 25

4. Compile-time and Run-time Techniques for Enhancing Parallelism .......... 27
   4.1 Tiling for Multiple Levels of Parallelism .......................... 28
      4.1.1 Details of the Approach ......................................... 28
      4.1.2 Optimal Placement of Data Movement Code .................. 30
      4.1.3 Tile Size Search Algorithm ..................................... 33
   4.2 Approach for Compiler-Assisted Dynamic Scheduling .................... 35
      4.2.1 Task Graph Generator ........................................... 37
      4.2.2 Task Scheduler .................................................. 43
      4.2.3 Run-time Execution .............................................. 46
   4.3 Experimental Results for Compiler-Assisted Dynamic Load-balanced Scheduling .................................................. 47
   4.4 Related Work .......................................................... 55
   4.5 Conclusions .............................................................. 57

5. Compile-time Transformations for GPUs ..................................... 58
   5.1 GPU Architecture and the CUDA Programming Model .................. 59
      5.1.1 GPU Computing Architecture .................................... 59
      5.1.2 CUDA Programming Model ....................................... 60
      5.1.3 GPU Execution Model ............................................ 62
   5.2 Performance Characterization of GPGPU ............................. 64
      5.2.1 Global Memory Access .......................................... 64
      5.2.2 Shared Memory Access .......................................... 67
      5.2.3 Degree of Parallelism vs Register Pressure .................... 69
   5.3 Optimizing Global Memory Access .................................... 71
      5.3.1 Global Memory Access Coalescing .............................. 71
      5.3.2 Examples .......................................................... 73
      5.3.3 Effective Use of Register and Non-register Memories .......... 75
      5.3.4 Optimized Copy from Global Memory to Shared Memory ........ 76
   5.4 Optimizing Shared Memory Access .................................... 76
   5.5 Approach for Selecting Optimal Tile sizes and Unroll factors .......... 78
      5.5.1 Model to Estimate Memory Traffic .............................. 80
      5.5.2 Model-driven Empirical Search .................................. 81
      5.5.3 Experimental Evaluation for Model-driven Empirical Search .... 82
   5.6 Experimental Results .................................................. 84
      5.6.1 Multi-level Tiling ............................................... 84
      5.6.2 Compile-time Transformations for GPUs ....................... 90
   5.7 Related Work .......................................................... 93
   5.8 Conclusions .............................................................. 95

ix
6. End-to-end C-to-CUDA System ........................................ 96
   6.1 Design of C-to-CUDA Generator ............................... 97
      6.1.1 Multi-level Parallel Tiled Code Generation .............. 99
      6.1.2 Data Movement Between Off-chip and On-chip Memories .. 102
      6.1.3 Syntactic Post-processing ................................. 105
   6.2 Experimental Results ........................................... 107
      6.2.1 Coulombic Potential (cp) ............................... 108
      6.2.2 N-Body Simulation (nbody) ............................ 110
      6.2.3 MRI kernels ........................................... 111
      6.2.4 Stencil Computation Kernels .......................... 113
      6.2.5 Gauss Seidel Successive Over Relaxation .............. 114
   6.3 Related Work .................................................. 115
   6.4 Conclusions .................................................... 116

7. Conclusions and Future Work ....................................... 118
   7.1 Optimizations for Irregular Applications .................... 119
      7.1.1 Overview of Issues for Irregular Applications on GPUs ... 119
   7.2 Heterogeneous Computing Platforms ........................... 120
Bibliography ............................................................ 122
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Architectural configurations of NVIDIA GeForce 8800 and 280 GTX</td>
<td>61</td>
</tr>
<tr>
<td>5.2</td>
<td>Architectural constraints of NVIDIA GeForce 8800 and 280 GTX</td>
<td>63</td>
</tr>
<tr>
<td>5.3</td>
<td>Global memory bandwidth for block and cyclic access patterns</td>
<td>65</td>
</tr>
<tr>
<td>5.4</td>
<td>Performance comparison (in GFLOPS) of mv kernel</td>
<td>92</td>
</tr>
<tr>
<td>5.5</td>
<td>Performance comparison (in GFLOPS) of tmv kernel</td>
<td>93</td>
</tr>
<tr>
<td>6.1</td>
<td>Performance of nbody benchmark (in GFLOPS)</td>
<td>110</td>
</tr>
<tr>
<td>6.2</td>
<td>Performance of MRI-Q (in GFLOPS)</td>
<td>112</td>
</tr>
<tr>
<td>6.3</td>
<td>Performance of MRI-FHD (in GFLOPS)</td>
<td>112</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>7</td>
</tr>
<tr>
<td>Transpose matrix vector multiply (tmv) kernel</td>
<td>7</td>
</tr>
<tr>
<td>3.1</td>
<td>26</td>
</tr>
<tr>
<td>Example of data allocation and movement</td>
<td>26</td>
</tr>
<tr>
<td>4.1</td>
<td>29</td>
</tr>
<tr>
<td>A Multi-level Parallel Architecture</td>
<td>29</td>
</tr>
<tr>
<td>4.2</td>
<td>31</td>
</tr>
<tr>
<td>Example of Multi-level Tiling - Original Code</td>
<td>31</td>
</tr>
<tr>
<td>4.3</td>
<td>32</td>
</tr>
<tr>
<td>Example of Multi-level Tiling - Multi-level Tiled Code</td>
<td>32</td>
</tr>
<tr>
<td>4.4</td>
<td>35</td>
</tr>
<tr>
<td>Enabling load-balanced execution on multi-core systems</td>
<td>35</td>
</tr>
<tr>
<td>4.5</td>
<td>38</td>
</tr>
<tr>
<td>Example with LU decomposition</td>
<td>38</td>
</tr>
<tr>
<td>4.6</td>
<td>41</td>
</tr>
<tr>
<td>Dependences in the tiled domain for LU code</td>
<td>41</td>
</tr>
<tr>
<td>4.7</td>
<td>42</td>
</tr>
<tr>
<td>Code for DAG generation</td>
<td>42</td>
</tr>
<tr>
<td>4.8</td>
<td>44</td>
</tr>
<tr>
<td>Inter-tile dependence DAG for LU decomposition</td>
<td>44</td>
</tr>
<tr>
<td>4.9</td>
<td>45</td>
</tr>
<tr>
<td>Dynamic schedule vs. affine schedule (time steps)</td>
<td>45</td>
</tr>
<tr>
<td>4.10</td>
<td>49</td>
</tr>
<tr>
<td>Performance of LU on 4 cores</td>
<td>49</td>
</tr>
<tr>
<td>4.11</td>
<td>50</td>
</tr>
<tr>
<td>Performance of LU on 8 cores</td>
<td>50</td>
</tr>
<tr>
<td>4.12</td>
<td>52</td>
</tr>
<tr>
<td>Performance of Cholesky</td>
<td>52</td>
</tr>
<tr>
<td>4.13</td>
<td>53</td>
</tr>
<tr>
<td>Performance of LU for various task weights</td>
<td>53</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>5.1</td>
<td>Organization of GPU Architecture</td>
</tr>
<tr>
<td>5.2</td>
<td>CUDA Execution Model</td>
</tr>
<tr>
<td>5.3</td>
<td>Global memory bandwidth for different access strides across threads</td>
</tr>
<tr>
<td>5.4</td>
<td>Shared memory access time for different access strides across threads</td>
</tr>
<tr>
<td>5.5</td>
<td>mv and tmv kernels</td>
</tr>
<tr>
<td>5.6</td>
<td>Shared memory bank access time for varying simultaneous bank requests</td>
</tr>
<tr>
<td>5.7</td>
<td>Performance of MM kernel for various code versions</td>
</tr>
<tr>
<td>5.8</td>
<td>Execution time of MPEG4 ME for various problem sizes</td>
</tr>
<tr>
<td>5.9</td>
<td>Execution time of 1D Jacobi for various problem sizes</td>
</tr>
<tr>
<td>5.10</td>
<td>Performance (GFLOPS) of MPEG4 ME for various problem sizes</td>
</tr>
<tr>
<td>5.11</td>
<td>Performance (GFLOPS) of 1D Jacobi for various problem sizes</td>
</tr>
<tr>
<td>5.12</td>
<td>Execution time of MPEG4 ME kernel for varying tile sizes</td>
</tr>
<tr>
<td>5.13</td>
<td>Execution time of 1D Jacobi for varying thread blocks (small problem sizes)</td>
</tr>
<tr>
<td>5.14</td>
<td>Execution time of 1D Jacobi for varying tile sizes (large problem sizes)</td>
</tr>
<tr>
<td>5.15</td>
<td>Performance of Matrix kernels</td>
</tr>
<tr>
<td>6.1</td>
<td>The C-to-CUDA Code Generation Framework</td>
</tr>
<tr>
<td>6.2</td>
<td>Original code structure for Coulombic Potential (cp) benchmark</td>
</tr>
<tr>
<td>6.3</td>
<td>Performance of cp benchmark</td>
</tr>
<tr>
<td>6.4</td>
<td>Performance variation of nbody benchmark w.r.t Optimizations</td>
</tr>
<tr>
<td>6.5</td>
<td>Performance of 2D Jacobi</td>
</tr>
<tr>
<td>6.6</td>
<td>Performance of 2d FDTD</td>
</tr>
</tbody>
</table>
6.7 Performance of Gauss Seidel .................................................................. 115

6.8 Parallel tiled code structure (with data movement) for cp benchmark .... 117
List of Algorithms

1. Reuse benefit calculation algorithm ........................................ 18
2. Automatic data allocation algorithm ....................................... 19
3. Run-time Execution ............................................................ 47
4. Finding transformations enabling coalesced global memory access 74
5. Finding Optimal Padding Factor .............................................. 79
6. Multi-level Parallel Tiled Code Generation .............................. 102
7. Generation and Placement of Data Movement Code .................. 104
8. Parallel CUDA Code Generation ............................................ 107
CHAPTER 1

INTRODUCTION

The computer architecture world has been experiencing a significant change over the years. During 1980’s and 1990’s, new generations of super fast processors were coming out every 18 to 24 months, where the processor clock rate of the new version of hardware was twice or even more faster than that of the previous one. Processor clock rates increased over 100x over the two decades. However it was realized that more cycles were being spent not in the processor core execution, but in the memory subsystem which includes a multilevel caching structure, and the so-called Memory Wall problem started to evolve quite significantly due to the fact that the increase in memory speed did not match that of processor cores. During the past decade, it became evident that it is very hard and not economical to speed the processor clock further due to tremendous increase in power consumption and power dissipation by advanced computer systems. However, since transistor density still continues to grow unabated in accordance with the Moore’s Law, the computer processor designs shifted to a new direction of providing increased overall performance by changing the structure of the processor subsystem to utilize several processor cores on a single chip, while keeping the clock rate of individual processor cores at a reasonable level. As a result of this architectural change, it is possible to provide further improvements in performance while keeping the power consumption of the processor subsystem almost constant. The
number of cores is expected to increase at a near-exponential rate over the next several years.

Some of the modern multiple-core computer architectures include general-purpose multi-core architectures (from Intel, AMD, IBM, and Sun), and specialized parallel architectures such as the Cell Broadband Engine, Graphics Processing Units (GPUs), ClearSpeed, Field-Programmable Gate Arrays (FPGAs), and Multiprocessor System-on-Chip (MPSoC). These architectures have very high computation power per chip, for e.g. the NVIDIA GeForce 8800 GTX GPU chip uses over 680 million transistors and has a peak performance of over 350 GFLOPS [66].

Memory subsystem has always been a performance bottleneck in computer systems and it is more so, with the emergence of processor subsystem with multiple on-chip processor cores. Hence architects are driven to ensure that memory bandwidth and latency are optimized to utilize the full benefits of the tremendous computational power available in the system. While general-purpose multi-core architectures have settled with hardware-managed multi-level caches, which have been the traditional solutions to alleviate memory bottleneck, modern parallel architectures such as GPUs and the Cell processor use software-managed fast on-chip memories, often referred to as scratchpad memories or local memory stores. Hence the memory subsystem in modern architectures have conventional cache-based memory hierarchy or local store memory hierarchy or both.

To summarize the major features of multi-core architectures, (1) they have very high computation power, and (2) they have cache-based and/or local store memory hierarchy to alleviate memory performance bottleneck. However these features put forward various significant challenges to solve in order to utilize the full benefits of the features.
1.1 Challenges in Multi-core and Many-core Architectures

As the number of cores per chip is expected to increase drastically, one of the pressing issues is to utilize the high computation power available in the chip and in the system. The challenging problem in addressing this issue is to extract the parallelism in the application and map it effectively to the target system. Some applications that inherently require more synchronization might lead to significant load imbalance on multi-core and many-core systems, and hence might under-utilize the computation power. Hence, a significant challenge to be addressed in the multi-core and many-core architectures is the effective load-balanced utilization of the cores.

Another significant challenge in modern multiple-core architectures is to effectively utilize the on-chip and off-chip memories and enhance data reuse to maximize memory performance. In architectures with on-chip scratchpads, the scratchpads provide benefits in performance and power reduction, but at the same time increase the burden on application developers to explicitly maintain memory buffers in scratchpads.

The current multi-core systems are so architecturally diverse that to realize high performance, the applications have to be specialized for the underlying system using architecture-specific optimization strategies. In other words, there are various issues that are specific to the target architecture which need much effort and expertise to address in order to maximize application performance on the architecture. Many of the architecture-specific optimization strategies involve optimizations targeted towards improving memory throughput of an application. The memory access model may be inherent to the architecture and optimizations have to be developed for the specific memory access model. Furthermore, the levels of parallelism available and the appropriate granularity of parallelism needed for the target
architecture have to be considered, while mapping the computation, for effectively utilizing the computational power of the system.

1.2 Key Contributions

In this dissertation, we develop compile-time and run-time optimizations that enable efficient execution of programs on multi-core and many-core architectures by effectively addressing the afore-mentioned challenges.

- We provide automatic compiler support for managing data in scratchpad memory, with the compiler making effective decisions on what elements to move in and move out of scratchpad memory, when and how to move them, and how to efficiently access the elements brought into scratchpad memory.

- We develop an effective tiling approach for efficient mapping of computation in regular programs on to many-core systems. We develop an approach for automatic dynamic determination of inter-tile dependencies, and extend their use to enhance load balancing for parallel tiled execution on multi-core systems.

- In addition to addressing issues that are common across multiple-core architectures, it is also necessary to address performance-influencing factors that are specific to an architecture. We identify and characterize key factors that affect performance of general-purpose computations on GPU architecture, develop compile-time transformation approaches to address performance factors related to parallelism and data locality that are GPU architecture-specific, and develop an end-to-end compiler framework for GPUs.
1.3 Organization

In Chapter 2, we introduce the polyhedral compiler model for representing programs, data dependences, and program transformations, which we extensively use in the development of our compiler techniques that are discussed in further Chapters. In Chapter 3, we describe our automatic compiler support for managing data in scratchpad memory. In Chapter 4, we discuss our approaches for enhancing parallelism in multi-core and many-core systems. We then present our compile-time transformations developed to enhance parallelism and memory access on GPUs in Chapter 5. We present our end-to-end compiler framework developed for GPUs in Chapter 6. Finally, we discuss some of the future directions of research in Chapter 7.
CHAPTER 2

OVERVIEW OF POLYHEDRAL MODEL

There has been significant progress over the last two decades in the development of powerful compiler frameworks for dependence analysis and transformation of loop computations with affine bounds and array access functions [4, 74, 60, 45, 35, 76, 73, 10]. For such regular programs, compile-time optimization approaches have been developed using affine scheduling functions with a polyhedral abstraction of programs and data dependencies. Although the polyhedral model of dependence abstraction and program transformation is much more powerful than the traditional model of data dependencies currently used in production optimizing compilers, early polyhedral approaches were not practically efficient. Recent advances in dependence analysis and code generation [76, 10, 92] have addressed many of these issues, resulting in polyhedral techniques being applied to code representative of real applications such as the spec2000fp benchmarks. CLooG [10, 27] is a powerful state-of-the-art code generator that captures most of these advances. Building on these developments, Bondhugula et al. have developed the Pluto compiler framework that enables end-to-end automatic parallelization and locality optimization of affine programs for general-purpose multi-core targets [15, 16, 71].
for (i=0; i < N; i++) {
    P: x[i] = 0;
    for (j=0; j < N; j++)
        Q: x[i] += a[j][i]*y[j];
}

(a) Original code

\[
D_{Q}^{\text{orig}} \cdot \begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix} \geq \vec{0} \quad D_{Q}^{\text{tiled}} \cdot \begin{pmatrix} it \\ jt \\ i \\ j \\ N \\ 1 \end{pmatrix} \geq \vec{0}
\]

(c) Original and tiled iteration space

for (it=0; it <= floord(N-1,32); it++) {
    for (jt=0; jt <= floord(N-1,32); jt++) {
        if (jt == 0) {
            for (i=max(32*it,0);
                 i <= min(32*it+31,N-1);
                 i++)
                P: x[i] = 0;
            Q: x[i] += a[0][i]*y[0];
        }
        for (i=max(32*it,0);
             i <= min(32*it+31,N-1); i++)
            for (j=max(32*jt,1);
                 j <= min(32*jt+31,N-1); j++)
                Q: x[i] += a[j][i]*y[j];
        }
    }
}

(b) Tiled code

Figure 2.1: Transpose matrix vector multiply (tmv) kernel

In this Chapter, we provide some background information on the polyhedral compiler model for representing programs, data dependencies, and program transformations, which we have used for the development of our compiler techniques.

A hyperplane in \(n\) dimensions is an \(n-1\) dimensional affine subspace of the \(n\)-dimensional space and can be represented by an affine equality. A halfspace consists of all points of an \(n\)-dimensional space that lie on one side of a hyperplane (including the hyperplane); it can be represented by an affine inequality. A polyhedron is the intersection of finitely many halfspaces. A polytope is a bounded polyhedron.

In the polyhedral model, a statement \(s\) surrounded by \(m\) loops is represented by an \(m\)-dimensional polytope, referred to as an iteration space polytope. The coordinates of
a point in the polytope (referred to as the iteration vector $\vec{i}_s$) correspond to the values of the loop indices of the surrounding loops, starting from the outermost one. In this work we focus on programs where loop bounds are affine functions of outer loop indices and global parameters (e.g., problem sizes). Similarly, array access functions are also affine functions of loop indices and global parameters. Hence the iteration space polytope $D_s$ of a statement $s$ can be defined by a system of affine inequalities derived from the bounds of the loops surrounding $s$. Each point of the polytope corresponds to an instance of statement $s$ in program execution. Using matrix representation to express systems of affine inequalities, the iteration space polytope is defined by

$$D_s \begin{pmatrix} \vec{i}_s \\ \vec{n} \\ 1 \end{pmatrix} \geq \vec{0},$$

where $D_s$ is a matrix representing loop bound constraints and $\vec{n}$ is a vector of global parameters.

Affine array access functions can also be represented using matrices. Let $a[F_{ras}(\vec{i}_s)]$ be the $i^{th}$ reference to an array $a$ in statement $s$ whose corresponding iteration vector is $\vec{i}_s$. Then

$$F_{ras}(\vec{i}_s) = F_{ras} \begin{pmatrix} \vec{i}_s \\ \vec{n} \\ 1 \end{pmatrix},$$

where $F_{ras}$ is a matrix representing an affine mapping from the iteration space of statement $s$ to the data space of array $a$. Row $i$ in the matrix $F_{ras}$ (often referred to as the access matrix) defines a mapping corresponding to the $i$th dimension of the data space. When the rank of the access matrix of an array reference is less than the iteration space dimensionality of the statement in which it is accessed, the array is said to have an order of magnitude (or higher-order) reuse due to that reference. Thus, the condition for higher-order reuse of an
array \( a \) due to a reference \( F_{\text{ras}}(i_s) \) is:

\[
\text{rank}(F_{\text{ras}}) < \text{dim}(i_s)
\]  \hspace{1cm} (2.1)

Loops whose iterators do not occur in the affine access function of a reference are said to be \textit{redundant loops} for the reference.

Given an iteration space polytope \( D \) and a set of array access functions \( F_1, F_2, \ldots, F_k \) of \( k \) references to an array in the iteration space, the set of array elements accessed in the iteration space or the \textit{accessed data space} is given by \( DS = \bigcup_{j=1}^{k} F_j D \), where \( F_j D \) is the image of the iteration space polytope \( D \) formed by the affine access function \( F_j \) and it gives the set of elements accessed by the reference \( F_j \) in \( D \).

\textbf{Example.} Consider the code in Figure 2.1(a). The iteration space polytope of statement \( Q \) is defined by \( \{ i, j \mid 0 \leq i \leq N - 1 \ \wedge \ 0 \leq j \leq N - 1 \} \). In matrix representation, this polytope is given by

\[
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
1 & 0 & 1 & -1 \\
0 & -1 & 1 & -1
\end{pmatrix}
\begin{pmatrix}
i_Q \\
N
\end{pmatrix}
\geq
\begin{pmatrix}
0 \\
1
\end{pmatrix},
\]

where \( i_Q = \binom{i}{j} \) is the iteration vector of statement \( Q \). The access function of the reference to array \( a \) in statement \( Q \) is represented as

\[
F_{1aQ}(i_Q) = \begin{pmatrix}
0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0
\end{pmatrix}
\begin{pmatrix}
i_Q \\
N
\end{pmatrix}.
\]

\textbf{Dependences.} There has been a significant body of work on dependence analysis in the polyhedral model [35, 74, 93]. An instance of statement \( s \), corresponding to iteration vector \( i_s \) within iteration domain \( D_s \), depends on an instance of statement \( t \) (with iteration
vector \( \vec{i}_t \) in domain \( D_t \), if (1) \( \vec{i}_s \) and \( \vec{i}_t \) are valid points in the corresponding iteration space polytopes, (2) they access the same memory location, and (3) \( \vec{i}_s \) is executed before \( \vec{i}_t \). Since array accesses are assumed to be affine functions of loop indices and global parameters, the constraint that defines conflicting accesses of memory locations can be represented by an affine equality (obtained by equating the array access functions in source and target statement instances). Hence all constraints to capture a data dependence can be represented as a system of affine inequalities/equalities with a corresponding polytope (referred to as a dependence polytope). The dependence polytope is defined by

\[
\begin{pmatrix}
D_s & 0 \\
0 & D_t \\
-I d & H
\end{pmatrix}
\begin{pmatrix}
\vec{i}_s \\
\vec{i}_t \\
\vec{n}
\end{pmatrix}
\begin{pmatrix}
\geq \vec{0} \\
= \vec{0}
\end{pmatrix}
\]

where \( I d \) represents an identity matrix, and \( H \) (referred to as the h-transformation of the dependence) relates the target statement instance to a source statement instance that last accessed the conflicting memory location:

\[
H \cdot \begin{pmatrix}
\vec{i}_t \\
\vec{n}
\end{pmatrix} = \begin{pmatrix}
\vec{i}_s \\
\vec{n}
\end{pmatrix}
\]

**Affine Transforms.** A one-dimensional affine transformation of a statement \( s \) is given by

\[
\phi_s(\vec{i}_s) = C_s \cdot \begin{pmatrix}
\vec{i}_s \\
\vec{n}
\end{pmatrix}
\]

where \( C_s \) is a row vector and the affine mapping \( \phi_s \) represents an affine hyperplane that maps each instance of statement \( s \) to a point in a dimension of a target iteration space.
An affine transformation is valid only if it preserves the dependences in the original program. An \(m\)-dimensional affine mapping can be represented using a matrix with \(m\) rows, where each row represents a one-dimensional mapping. A set of linearly independent one-dimensional affine functions \((\phi^1_s, \phi^2_s, \ldots, \phi^k_s)\) maps each instance of statement \(s\) into a point in the multi-dimensional target space. The transformation matrix captures a composition of transformations like fusion, skewing, reversal and shifting.

Using the polyhedral model to find (affine) program transformations has been widely used for improvement of sequential programs (source-to-source transformation) as well as automatic parallelization of programs \([36, 60, 45, 38, 16]\). Lim et al. \([60]\) use affine transformation framework for automatic parallelization. They define two different categories of constraints on the transformations: \textit{space partition constraints} and \textit{time partition constraints}. The space partition constraints ensure that any two dependent statement instances are always placed in the same space partition, i.e., they are mapped to the same virtual processor in processor space. The time partition constraints ensure that if a statement instance \(r\) depends on a statement instance \(s\), then \(r\) is executed at the same time point as \(s\) or at a later time point than \(s\). Feautrier \([36, 37]\) defines affine \textit{time schedule}, which is one-dimensional (single sequential loop in the transformed program) or multi-dimensional (nested sequential loops in the program), and the schedule associates a timestamp with each statement instance. Statement instances are executed in increasing order of timestamps to preserve data dependencies. Two statement instances that have the same timestamp can be executed in parallel. The \textit{time schedule constraint} in Feautrier’s framework, needed to preserve a data dependence, is as follows.

\[
\forall \vec{x}_s \in D_s, \forall \vec{y}_t \in D_t \text{ s.t. } \vec{y}_t \text{ depends on } \vec{x}_s, \quad \theta_t(\vec{y}_t) - \theta_s(\vec{x}_s) > 0
\]
Using such constraints, one can define a system that characterizes the time schedule coefficients, taking into account all dependencies. The system is then solved to find the legal time schedules. There has been a significant body of work (e.g., [36, 60]) on the procedure to solve a system of constraints for affine partition mappings, using the affine form of Farkas’ Lemma and Fourier-Motzkin projection algorithm.

One of the key affine transformations for optimizing parallelism and data locality is tiling. 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 (registers, L1, or L2 cache). Tiling for coarse-grained parallelism partitions 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. When tiling is performed, in the tiled iteration space, statement instances are represented by higher dimensional statement polytopes involving supernode iterators and intra-tile iterators. The code in Figure 2.1(b) represents the tiled version of the code in Figure 2.1(a). The original iteration space and the transformed iteration space are illustrated in Figure 2.1(c).

**Pluto.** Recently, Bondhugula et al. have developed Pluto [71, 16], a state-of-the-art automatic transformation system, that finds effective affine statement-wise transforms and optimizes sequences of imperfectly nested loops, simultaneously for parallelism and locality. Although the polyhedral framework provided a uniform basis for unifying several loop transformations, previous polyhedral automatic transformation frameworks [60, 45] had significant drawbacks or restrictions that limited their effectiveness. A significant problem was the lack of a realistic cost function to choose among the large space of legal transformations that are suitable for tiled, coarse-grained parallel execution. Most previous
approaches also did not consider data locality and parallelism together. Another significant shortcoming was that tiling was not directly integrated into previous polyhedral systems, but was performed as a post-processing step on fully permutable nests, after other transformations had been applied within the polyhedral framework. Pluto employs a model-driven approach (that uses a more concrete optimization criteria) to determine tiling hyperplanes that optimize both for data reuse and parallelism, and generates tiled code for statement domains of arbitrary dimensionalities using statement-wise affine transformations. Given an input sequential code, Pluto can automatically generate tiled parallel OpenMP code for multi-core processors. The effectiveness of the transformation system has been demonstrated on a number of non-trivial application kernels for multi-core processors.
CHAPTER 3

COMPILER SUPPORT FOR AUTOMATIC DATA MANAGEMENT IN ON-CHIP MEMORIES

Modern high-performance computer architectures have increasing numbers of processing elements on chip. Architects must ensure that memory bandwidth and latency are also optimized to exploit the full benefits of the available computational resources. Introducing a cache hierarchy has been the traditional way to alleviate the memory bottleneck. Caches are hardware-controlled, and it is difficult to model their exact behavior and to predict program execution times. While using caches, useful data may be evicted from the cache and replaced with other data without the programmer’s control. Due to this and other reasons concerning performance and power, various modern parallel architectures have fast explicitly managed on-chip (local) memories, often referred to as scratchpad memories, in addition to slower off-chip (global) memory in the system. The scratchpad memories are software-managed and hence software has complete control over the movement of data into and out of such memories. The execution times of programs using scratchpad memories can be more accurately predicted and controlled. Scratchpad memories help to minimize memory load/store latency and maximize on-chip bandwidth by providing more paths without any issues about coherency.
Numerous challenges arise for a compiler writer provided with an architecture with explicitly managed memories. The compiler has to make good decisions on what elements to move in and move out of local memory, when and how to move them, and how to efficiently access the elements brought into local memory, while ensuring program correctness. Programmers using architectures with scratchpad memories shoulder the burden of orchestrating the movement of data between global and local memory.

In this chapter, we develop compiler support for effective automatic data management in scratchpad memories. Our approach automatically creates buffers in local memory for storing portions of data that are accessed in a block of computation, determines array access functions for local buffer references, and automatically generates code to move data between local memory and global memory [8].

On general-purpose single-core or multi-core machines with caches, explicit copying of data (i.e., sub-arrays) into contiguous memory space is extremely important to avoid both conflict misses and TLB trashing. In particular, the benefits of data locality optimizations such as tiling and fusion are not fully seen without explicit copying. We have observed that the speedup with explicit copying for tiles is often in the range of 1.5 to 3 times over code without copying. The approach that we develop in this Chapter for creating local buffers and copying data into local buffers can be directly applied to enhance data locality in general-purpose CPUs.

3.1 Details of the Approach

In this section we discuss in detail the proposed approach (based on the polyhedral model discussed in Chapter 2) for effective automatic data management in scratchpad memories. We discuss three different aspects of the approach: (1) automatic allocation
of storage space (in the form of arrays) in scratchpad memories for holding portions of data accessed in a block of a program, (2) determination of access functions of references to arrays in scratchpad memories, and (3) automatic generation of code for moving data between scratchpad (local) memory and off-chip (global) memory. The local memory storage allocation is done independently of whether the access functions of various array references are uniformly generated or non-uniformly generated. We create local memory storage for each non-overlapping region of the data space of an array that is accessed in a program block. By default, data is moved in to local memory at the start of the block and moved out after the completion of the block. Data dependences are not violated because of the framework’s automatic storage allocation and data movement.

In parallel architectures such as the Cell processor, any data that is accessed in a computation has to be moved into scratchpad memory before access, as data cannot be accessed from global memory during computation. But in architectures such as GPUs, data can be accessed from both global memory and scratchpad memory during computation. For such architectures, the framework considers moving data that have sufficient reuse.

Our approach takes as input the iteration spaces of all statements in a program block as well as the access functions of all array references. The procedure explained below is applied to all arrays (one at a time) in the block.

For an array $A$, let $S_1, S_2, \ldots, S_q$ be the statements in the given program block containing references to $A$, and let $I_k$ represent the iteration space of statement $S_k$ $(1 \leq k \leq q)$. Let $F^1_k, F^2_k, \ldots, F^p_k$ be the matrices representing the affine read reference functions of $A$ in statement $S_k$, and let $G_k$ be the matrix representing the affine write reference function of $A$ (if there is a write reference of $A$ in $S_k$).
As discussed in Chapter 2, the data space accessed by a reference (represented by an access function matrix \( F \)) in a statement represented by an iteration space \( I \) is given by \( FI \).

Hence the set of all data spaces accessed by all read references of \( A \) in all statements in the program block is

\[
\mathcal{DS}_r^A = \{F^l_kI_k \mid 1 \leq l \leq p \land 1 \leq k \leq q\}
\]

Similarly, the set of all data spaces accessed by all write references of \( A \) in the block is

\[
\mathcal{DS}_w^A = \{G_kI_k \mid 1 \leq k \leq q\}
\]

The set of all data spaces accessed by all references of \( A \) in the block is

\[
\mathcal{DS}_{rw}^A = \mathcal{DS}_r^A \cup \mathcal{DS}_w^A
\]

We partition the set of all data spaces \( \mathcal{DS}_{rw}^A \) into maximal disjoint sets such that each partition has a subset of data spaces each of which is non-overlapping with any data space in other partitions, i.e., if \( \mathcal{DS}_{rw_{d_1}}^A, \mathcal{DS}_{rw_{d_2}}^A, \ldots, \mathcal{DS}_{rw_{d_m}}^A \) are the maximal disjoint sets,

\[
\forall d \in \mathcal{DS}_{rw_{d_i}}, \forall e \in \mathcal{DS}_{rw_{d_j}} \quad \text{s.t.} \quad 1 \leq i, j \leq m \land j \neq i: \quad d \cap e = \emptyset
\]

The problem of partitioning is solved by mapping it to an equivalent problem of finding connected components of an undirected graph. The undirected graph is created with vertices representing each data space in the set \( \mathcal{DS}_{rw}^A \); an edge exists between two vertices if the intersection of the data spaces corresponding to the vertices is not empty.

We then find the convex union or the minimum convex polytope that encloses all data spaces in a partition \( \mathcal{DS}_{rw_{d_i}}^A \). The convex union is given by

\[
\mathcal{CDS}_{rw_{d_i}}^A = \text{ConvexHull}(\mathcal{DS}_{rw_{d_i}}^A)
\]
Algorithm 1 Reuse benefit calculation algorithm

**Input:** Set of data spaces \((D)\), Number of references in \(D\) \((N_{\text{ref}})\), Iteration space dimensionality of each reference \((IS_i, 1 \leq i \leq N_{\text{ref}})\), Rank of affine function of each reference \((R_i, 1 \leq i \leq N_{\text{ref}})\)

1: \textbf{for} \(i=1\) to \(N_{\text{ref}}\) \textbf{do}
2: \quad \textbf{if} \(R_i < IS_i\) \textbf{then}
3: \quad \quad Mark yes
4: \quad \textbf{if} Not Marked yes \textbf{then}
5: \quad \quad \textbf{if} Volume(overlapped regions in \(D\)) > \(\delta \times \text{Volume}(D)\) \textbf{then}
6: \quad \quad \quad Mark yes

**Output:** Yes, if beneficial reuse, No, otherwise

3.1.1 Determining Local Memory Storage

We now describe the procedure that automatically allocates storage in local memory for portions of array \(A\) that are accessed in the given program block. We generate one local memory array for each partition \(DS_{rwd_i}^A\).

Algorithm 1 explains a procedure to determine if a partition of data spaces has sufficient reuse in a program block. If a given partition of data spaces has at least one reference whose accessed data space has higher-order reuse (i.e., it satisfies Condition (2.1)), then the partition is marked as beneficial to be copied to scratchpad memory. Otherwise we check if the data spaces in the partition have significant constant reuse. Constant reuse in the set is estimated by considering each pair of data spaces, determining the volume of their intersection, and summing up these volumes. If the sum of the volumes constitutes a significant portion, determined by a fraction \(\delta\), of the total volume of the set of all data spaces, then the partition is marked as beneficial. We have empirically fixed a value of 30% for \(\delta\) for GPUs.

The procedure explained below is applied to each partition of data spaces \(DS_{rwd_i}^A\) to determine local memory storage space for the partition. For architectures such as GPUs,
Algorithm 2 Automatic data allocation algorithm

Input: Iteration space polyhedra, Affine array access functions

1: for each array A do
2:   for each reference of the array do
3:     Find the data space accessed by the reference
4:   Partition the set of all data spaces into maximal disjoint sets such that each partition
   has a subset of data spaces each of which is non-overlapping with any data space in
   other partitions
5:   for each partition of data spaces do
6:     Find the convex union of data spaces in the partition
7:     Find the lower and upper bounds of each dimension of the convex union, as an
     affine function of program parameters. Let the number of dimensions be n. Let
     the dimension variables be $i_1, i_2, \ldots, i_n$. Let $lb_k$ and $ub_k$ be the lower and upper
     bounds of the dimension variable $i_k$.
8:   Define the local storage for a partition of accessed data spaces of array A as an
     array of dimension n and size $(ub_1 - lb_1 + 1) \times (ub_2 - lb_2 + 1) \times \cdots \times (ub_n -
     lb_n + 1)$

Output: Local memory storage for each non-overlapping accessed region of original ar-
arrays

only those partitions that are marked as beneficial by Algorithm 1 are considered. The
overall approach is summarized in Algorithm 2.

For each partition of data spaces $DS_{rw_d_i}^A$ we consider the corresponding convex hull $CDS_{rw_d_i}^A$, and find the lower and upper bounds of each dimension of this convex hull. The
bounds are determined in the form of an affine function of parameters of the program block,
using the Parametric Integer Programming (PIP) software [34]. These bounds determine
the size of the local memory array created for the partition. Let the dimensionality of the
convex hull be n. Let $i_1, i_2, \ldots, i_n$ represent the variables denoting each dimension of the
convex hull and let $lb_k$ and $ub_k$ be the lower and upper bounds of dimension variable $i_k$.

Local memory storage created for a partition of accessed data spaces of array A is an array
of dimension n with size $(ub_1 - lb_1 + 1) \times (ub_2 - lb_2 + 1) \times \cdots \times (ub_n - lb_n + 1)$. The order
of array dimensions of the local memory array follow that of the global memory array.
3.1.2 Determining Local Memory Array Access Functions

For each reference to the original array $A$ in the given program block, we find the corresponding access function for the local memory array reference. The dimensionality (say $m$) of the original global memory array $A$ might be greater than the dimensionality (say $n$) of the local memory array created for a partition of accessed data spaces of $A$. We use CLooG [27], an efficient code generator tool, to find the bound expressions of each dimension of the convex hull of the partition of accessed data spaces. The bound expressions generated from CLooG are represented as a loop structure with $n$ nested loops. The dimensions of the original data space that do not appear in the convex union polytope (convex hull) are represented as affine functions of dimensions that appear in the polytope, and program parameters, in their respective positions within the loop structure.

For any array access $f(\vec{y})$, each row of the access function matrix $F$ represents the array subscript of a dimension in the original data space. Let $F'$ be the matrix that is derived from $F$ by removing the rows (if any) corresponding to the dimensions in the original array that do not appear in the local memory array. (Note that $F' = F$ when $m = n$).

For any reference $A[f(\vec{y})]$ in the original program block, the corresponding local memory array access function is

$$f'(\vec{y}) - g, \text{ where } g = (lb_1, lb_2, \ldots, lb_n)^T$$

3.1.3 Generating Data Movement Code

This subsection describes the procedure to generate code for data movement. This procedure is applied to each partition of data spaces $\mathcal{DS}_{r_w,d_i}^A$ for which a local memory array is created.
From $\mathcal{D}_s^A$, we select the data spaces that are accessed by read references. We generate the loop structure of the code that moves data from global memory to local memory by scanning the selected data spaces using CLooG. Similarly, from $\mathcal{D}_w^A$, we select the data spaces that are accessed by write references and use CLooG to generate the loop structure of the code that moves data from local memory to global memory by scanning the selected data spaces. CLooG scans the data spaces in an efficient way such that the generated loop structure leads to single load/store of each data element that is read/written even if the accessed data spaces of references are overlapping.

Having generated the loop structures, the loop body of the data move in and move out code is generated as follows. Let the dimensionality of the original array and the local memory array be $m$ and $n$, respectively. Let $H$ be a $m \times n$ matrix derived from an $n \times n$ identity matrix $Id_n$ by adding a row (in the respective position) for each of the $m - n$ dimensions that do not appear in the convex union polytope $C\mathcal{D}_s^A$. The added row represents the corresponding dimension as affine function of dimensions that appear in the polytope, and program parameters. (Note that $H = Id_n$ when $m = n$). Let $\tilde{y}$ be the iteration vector of the loop structure of data move in (and data move out) code.

The loop body of the data move in code for each local array $L_i$ created for $A$ is

$$L_i[Id_n,\tilde{y} - g] = A[H,\tilde{y}]$$

and the loop body of the data move out code is

$$A[H,\tilde{y}] = L_i[Id_n,\tilde{y} - g]$$

where $g = (lb_1, lb_2, \ldots, lb_n)^T$.
To estimate an upper bound on the volume of data moved in to the local memory array created for $\mathcal{DS}^A_{rw_{d_i}}$, we partition the set of data spaces in $\mathcal{DS}^A_{rw_{d_i}}$, that are accessed by read references, into maximal non-overlapping subsets of data spaces (as explained earlier in the section), and find the space required in the local memory array for each such partition (using the procedure explained in Algorithm 2). The upper bound on the volume of data moved in to the local array is given by the total space needed in the local array for all such partitions. Similarly, we estimate the upper bound on the volume of data moved out of the local memory array by finding the total space needed for all non-overlapping partitions of data spaces in $\mathcal{DS}^A_{rw_{d_i}}$, that are accessed by write references.

Figure 3.1 shows an example that illustrates automatic data allocation in local memory storage and code generation for data movement.

3.1.4 Optimizing Data Movement

All data that are accessed by read references in a program block need not be moved in to scratchpad memory; similarly, all data that are accessed by write references need not be written out from scratchpad memory. Only those data elements that are read by statement instances in the block but whose values are written by earlier statement instances outside the block, need to be moved in. Of course, data elements corresponding to an input array (array that is only read in the program but not written) also need to be brought in. Similarly, data elements that are written by statement instances in the block but not read by any statement instance outside the block need not be copied out to global memory from scratchpad memory, unless the data elements belong to an output array (array that is only written in the program but not read).
The optimal strategy for determining data elements that need to be copied in and copied out requires data dependence information. We are given the iteration spaces of all statements in a program block and the set of all dependence polyhedra. For each true dependence (defined by a polyhedron), we identify each statement instance in the block that is a target of the dependence but whose corresponding source statement instance does not belong to the block. The data accessed due to the array read references involved in true dependences in the set of all identified statement instances constitute the data that needs to be copied into local memory (in addition to data belonging to input arrays accessed in the block) for the computation in the block. Similarly, for each true dependence, we identify each statement instance in the block that is a source of the dependence but whose corresponding target statement instance does not belong to the block. The data accessed due to the array write references involved in true dependences in the set of all identified statement instances constitute the data that needs to be copied out from local memory (in addition to data belonging to output arrays written in the block) after the computation in the block.

3.2 Related Work

In this section we discuss prior work that has addressed compiler issues in architectures with explicitly managed scratchpad memories.

Schreiber and Cronquist [90] have proposed an approach to do near-optimal allocation of scratchpad memories and near-optimal re-indexing of array elements in scratchpad memories. Their approach generates separate storage efficient local arrays for each equivalent class of uniformly generated references. Hence if data accessed due to two references, belonging to different classes, are overlapping, then two different local arrays would be created to hold the overlapping accessed data spaces. Anantharaman and Pande [3] perform
data partitioning on arrays into portions to be kept in local memory and global memory. They compute a bounding box for each equivalent group of uniformly generated references as in the case of [90]. Eisenbeis et al. [31] consider elements to move to local memory from a view of individual iteration of a loop nest instead of an atomic unit of computation of the program. Kandemir et al. [50] propose an approach for dynamically managing scratchpad memories, but they handle only uniformly generated affine references.

The idea of estimation of the number of references to an array in order to predict cache effectiveness has been discussed by Ferrante et al. [39] and Gallivan et al. [40]. The idea of finding image of the iteration space onto the array space to optimize global transfers has been discussed in [40]; but only a framework for estimating bounds for the number of elements accessed was given. Ferrante et al. gave exact values for uniformly generated references but did not consider multiple references. Also, for non-uniformly generated references, arbitrary correction factors were given for arriving at lower and upper bounds for the number of distinct references. Clauss[26] and Pugh [75] have presented more expensive but exact techniques to count the number of distinct accesses.

There has been a significant amount of research on memory reduction and optimization of data locality for embedded single-processor-on-chip (SOC) systems. In the case of memory optimizations, Panda et al., Balasa et al., and the IMEC group have derived several transformations for improving memory performance on embedded systems [6, 23, 68, 78, 95]. Their work is a collection of techniques that form a custom memory management methodology referred to as data transfer and storage exploration (DTSE). There is a large body of work on estimating the memory requirements of loops [6, 23, 78, 95] (and references therein). Most of these works assume the given sequential execution order and
find the memory requirements. A number of works have addressed scratchpad memory management \([47, 49, 50]\) (to name a few).

### 3.3 Conclusions

In this Chapter, we have developed an automatic compiler support for effective data management in on-chip memories. Our automatic data management support includes: (1) allocation of storage space in scratchpad memory to hold portions of data accessed in a computational block, (2) determination of access functions of local memory array references, and (3) generation of code for moving relevant portions of data resident in slow off-chip memory to fast scratchpad memory and vice versa.
Original Code:

A[200][200]; B[200][200];
for (i=10;i<=14;i++)
    for (j=10;j<=14;j++)
        for (k=11;k<=20;k++)
            B[i][j+k]=A[i][k]+B[i+j][k];
        }
    }

Modified Code:

LA0[5][10]; LA1[9][5];
LB0[5][14]; LB1[9][10];
for (i=10;i<=14;i++)
    for (j=10;j<=14;j++)
        LA0[i-10][j-11]=
            LA1[i+j-20][j+1-11]+3;
        for (k=11;k<=20;k++)
            LB0[i-10][j+k-21]=
                LA0[i-10][k-11]+LB1[i+j-20][k-11];

Local Array LA0:

lb (i) = 10; ub (i) = 14
lb (j) = 11; ub (j) = 20

Local Array LA1:

lb (i) = 20; ub (i) = 28
lb (j) = 11; ub (j) = 15

/* Data Move in code */
for (i=10;i<=14;i++)
    for (j=11;j<=20;j++)
        LA0[i-10][j-11] = A[i][j];
for (i=20;i<=28;i++)
    for (j=max(i-13,11);
            j=min(15,i-9);j++)
        LA1[i-20][j-11] = A[i][j];
/* Data Move out code */
for (i=10;i<=14;i++)
    for (j=11;j<=15;j++)
        A[i][j] = LA0[i-10][j-11];

Local Array LB0:

lb (i) = 10; ub (i) = 14
lb (j) = 21; ub (j) = 34

Local Array LB1:

lb (i) = 20; ub (i) = 28
lb (j) = 11; ub (j) = 20

/* Data Move in code */
for (i=20;i<=28;i++)
    for (j=11;j<=20;j++)
        LB1[i-20][j-11] = B[i][j];
/* Data Move out code */
for (i=10;i<=14;i++)
    for (j=21;j<=34;j++)
        B[i][j] = LB0[i-10][j-21];

Figure 3.1: Example of data allocation and movement
CHAPTER 4

COMPILE-TIME AND RUN-TIME TECHNIQUES FOR
ENHANCING PARALLELISM

A significant problem to be addressed in modern high-performance many-core architectures is the exploitation of available parallelism. Parallelization of arbitrary regular programs in these architectures is challenging as a number of constraints have to be addressed in order to effectively map the computation onto the parallel units of the architecture. In the presence of on-chip memories, the computation mapping should utilize the benefits of on-chip memories and hence the mapping is constrained by the amount of on-chip memory available. Some application kernels that inherently require more synchronization, if not effectively scheduled for execution on multi-core systems, might lead to significant load imbalance, and hence might under-utilize the computation power. For e.g. multi-statement programs with statements of different dimensionality such as Cholesky or LU decomposition are such programs whose tiled parallel execution can result in significant load imbalances. Enabling the run-time system to enhance load balance on the processor cores for such application kernels is non-trivial.

In this Chapter, we develop an effective tiling approach to extract required levels of parallelism and map the computation on to the target system. We also handle the impact of on-chip memory availability on the selection of tile sizes at different levels [8]. To address
possible load imbalance caused by certain applications executed on multi-core systems, we
develop compiler support using the polyhedral model to extract inter-tile data dependences
in the form of a Directed Acyclic Graph (DAG), which can enable a run-time system to
dynamically schedule tiles based on critical path analysis [9].

4.1 Tiling for Multiple Levels of Parallelism

In this Section, we present in detail the approach of mapping computation in an affine
program on to a multi-level parallel architecture that has explicitly managed on-chip mem-
ories. We use a multi-level tiling approach that extracts required levels of parallelism while
addressing the constraints imposed on tile sizes by the availability of on-chip memory.

4.1.1 Details of the Approach

Given any input program, the first step is to find the parallelism available in the com-
putation. Our approach uses the Pluto framework developed by Bondhugula et al. [16] for
this purpose. Given any affine input program, Pluto finds an optimal set of affine transfor-
mations (or, equivalently, tiling hyperplanes) for each statement to minimize the volume
of communication between tiles and also to improve data reuse in each tile. Pluto finds
bands of permutable loops that can be tiled and also identifies points in execution where
synchronization is required. A band can have a single sequential loop, or it can have multi-
ple loops found in the increasing order of communication volume induced due to the loop.
In our approach, we consider the outermost band that has multiple permutable loops, and
treat the communication-free loops in the band, if any, as space loops. If there are no
communication-free loops in the outermost band, we treat all but the last loop as space
loops, in order to achieve pipeline parallelism. Having identified the bands of permutable
loops and the space and time loops, we proceed to perform multi-level tiling of the space loops to distribute the available parallelism across the various levels of parallel units in the system. By default, we do as many levels of tiling as the number of levels of parallel units. But due to constraints imposed by memory availability at different levels (as explained later), we perform additional levels of tiling when needed. For these additional levels of tiling, which are done within tiles that are distributed across parallel processes, we tile all permutable loops.

The prototypical architecture we consider for further explanation has the following components (as shown in Fig. 4.1): (1) a slow global (off-chip) memory, (2) a set of parallel units at an outer level that communicate with each other through the global memory space, (3) a set of parallel units within each outer-level parallel unit, and (4) a local fast explicitly managed scratchpad memory within each outer-level parallel unit shared by the inner-level parallel units. The computation is performed by the inner-level parallel units and the data required for computation is copied into scratchpad memory (the data movement being coordinated by the inner-level parallel units) from global memory and the data produced by the computation is copied back into global memory.
The procedure to perform multi-level tiling for the two-level parallel architecture considered, is as follows. We fix the number of parallel processes at outer and inner levels to be a multiple of the number of physical parallel processors at the level. We first perform an outer level of tiling of the space loops that equally distributes tiles across outer-level parallel processes. If the tile in an outer-level process is large enough such that it requires more local memory than the available amount, it becomes necessary to introduce one more level of tiling, in order to limit the amount of needed local memory. In this case we split the tile in an outer-level parallel process into sub-tiles (that are executed sequentially within the outer-level tile) such that each sub-tile requires an amount of local memory that is no higher than the fixed upper limit. We find an optimal set of tile sizes that defines an atomic unit of computation in an outer-level tile under the constraint of limited local memory availability, using the algorithm described in Section 4.1.3. Once the outer level tiling is done, we perform an inner level of tiling of the space loops that equally distributes the iterations of an atomic unit of computation executed in an outer-level parallel process among the inner-level parallel processes.

Figure 4.2 and Figure 4.3 illustrate an example in which multiple levels of tiling are done to exploit various levels of parallelism available in the system.

4.1.2 Optimal Placement of Data Movement Code

We now discuss the approach to optimally place the data movement code (code that moves data between on-chip and off-chip memories) along with the computation code. With the tiling hyperplanes (that determine the shape of a tile executed atomically in an outer-level parallel process) known, the iteration spaces of statements in a tile, parameterized by tile sizes, are determined. The iteration spaces of statements in a tile, along with
FORALL \( i = 1, N_i \)
FORALL \( j = 1, N_j \)
FOR \( k = 1, WS \)
FOR \( l = 1, WS \)
S1
END FOR
END FOR
END FOR
END FOR
END FOR

Figure 4.2: Example of Multi-level Tiling - Original Code

the affine array access functions for each reference in the tile are given as input to the automatic on-chip memory data management support described in Chapter 3 to determine the local memory storage needed for data accessed in the tile (as a function of tile sizes) and to generate code to move data between local memory and global memory. By default, code to move data in to local memory allocated for a tile is placed in the program structure at the beginning of execution of the tile (computational block) and code to move modified data out to global memory is placed at the end of execution of the tile. The tiling loops form a loop nest over the data movement code and tile code.

When an array access function does not depend on a loop iterator in the iteration space, then the loop is a redundant loop for the reference. If all references of a local memory array have one or more common redundant loops in the loop nest of tiling loops, then the data movement code of the array is hoisted and placed at a level in the loop nest of tiling loops such that any tiling loop that is below the level in the loop nest is a redundant loop identified for the array. By doing such hoisting, the data in the local memory array is reused, if possible, across various computational blocks. Placing data movement code outside
// Tiling to distribute at the outer level
FORALL iT = 1, Ni, Ti
    FORALL jT = 1, Nj, Tj
        // Tiling to satisfy local memory limit
        FOR i’ = iT, min(iT+Ti−1,Ni), ti’
            FOR j’ = jT, min(jT+Tj−1,Nj), tj’
                FOR k’ = 1, WS, tk’
                    FOR l’ = 1, WS, tl’
                        <Data move in Code>

// Tiling to distribute at the inner level
FORALL it = i’, min(i’+ti’−1,Ni), ti
    FORALL jt = j’, min(j’+tj’−1,Nj), tj
        FOR i = it, min(it+ti−1,Ni)
            FOR j = jt, min(jt+tj−1,Nj)
                FOR k = k’, min(k’+tk’−1,WS)
                    FOR l = l’, min(l’+tl’−1,WS)
                        S1
        END FOR
    END FOR
END FOR
END FOR
Figure 4.3: Example of Multi-level Tiling - Multi-level Tiled Code

redundant loops helps in the selection of optimal tile sizes (for tiles that are sequentially executed in an outer-level parallel process).
4.1.3 Tile Size Search Algorithm

The goal of this algorithm is to find optimal sizes for tiles that determine the atomic units of computation executed in an outer-level parallel process, under the constraint that the active local memory used by the process does not exceed a given upper limit $M_{up}$. The algorithm tries to minimize data movement cost between local memory and global memory.

The data movement cost is directly affected by the volume of data that is being moved and the number of occurrences of data movement. The movement is done in parallel by the inner-level processes and there has to be a synchronization among these processes every time the data is moved. Hence the data movement cost, $C$, is modeled as

$$C = N \times (S + V \times L)$$

where $N$ is the number of occurrences of data movement, $S$ is the synchronization cost per occurrence of data movement, $V$ is the volume of data moved each time, and $L$ is the cost of transferring a data element. The values $S$ and $L$ depend on the number of inner-level processes $P$.

Consider tiling of a loop nest with maximum depth $m$. Let the index range of the $i^{th}$ loop in the nest be $N_i$ and let $t_i$ be the optimal tile size to be found for the $i^{th}$ loop in the nest. Let the number of local memory arrays created in a tile be $nl$ and $r_j$ be the optimal position in the loop nest where data movement code of $j^{th}$ local memory array is placed. Using Algorithm 2, we determine the size of each local memory array created in a tile, as a function of the tile sizes. Let $M_1, M_2, \ldots, M_{nl}$ represent the size of each local memory array. Let $V_{j^{in}}$ and $V_{j^{out}}$ be the upper bounds on the volume of data moved in to and moved out of $j^{th}$ local memory array, respectively, determined as a function of tile sizes, as explained in Section 3.1.3. Let $I$ denote the set of arrays to which data is moved in and $O$ denote the
set of arrays from which data is moved out. The tile size search algorithm is phrased as an optimization problem that minimizes the cost of data movement between local memory and global memory. The constraints ensure that (1) all tile sizes are greater than zero but lesser than or equal to the corresponding loop index range, (2) total memory required by a tile is within the given upper limit $M_{up}$, and (3) tile sizes are large enough to keep all inner-level processes busy.

The optimization problem is formulated below.

Variables:

$$t_1, t_2, \ldots, t_m$$

Constraints:

$$\forall i : 1 \leq i \leq m, t_i > 0$$

$$\forall i : 1 \leq i \leq m, t_i \leq N_i$$

$$\sum_{i=1}^{nl} M_i \leq M_{up}$$

$$t_1 \times t_2 \times \cdots \times t_m \geq P$$

Objective function:

$$\text{minimize} \quad \sum_{k \in I} \left( \prod_{i=1}^{r_k} \frac{N_i}{t_i} \times (S+V_{k1}^{in} \times L) \right) + \sum_{k \in O} \left( \prod_{i=1}^{r_k} \frac{N_i}{t_i} \times (S+V_{k1}^{out} \times L) \right)$$

The above optimization problem is a nonlinear constrained optimization problem that can be solved by a technique such as sequential quadratic programming (SQP). Note that we need to relax the integer constraints on $(t_1, t_2, \ldots, t_m)$ to lie in $\mathbb{R}^m$ instead of $\mathbb{Z}^m$, solve
the optimization problem, then round off the result to the closest integral vector. A detailed discussion on SQP is presented in [11].

We show the effectiveness of our multi-level tiling approach through experiments on GPUs which are representatives of high performance multi-level parallel architectures with explicitly managed on-chip memories, in Chapter 5.

4.2 Approach for Compiler-Assisted Dynamic Scheduling

We discussed how we generate, at compile-time, multi-level tiled code for architectures with multiple levels of parallelism. In this Section, we highlight the need for run-time enhancement for parallel programs that exhibit loss of efficiency on multi-core systems due to excessively constraining inter-task barrier synchronization. We develop a completely automatic parallelization approach that can transform input sequential codes (with affine dependences) for asynchronous, load-balanced parallel execution.
Recent work at the University of Tennessee [30, 22, 21] with LAPACK codes for several linear algebra functions highlights two key challenges for effective parallelization of such codes. First, effective use of modern multi-core hardware requires the introduction of tasks that operate on small portions of data in order to improve data locality. For affine code, fully automatic introduction of such tasks can be easily done with general polyhedral transformation tools such as Pluto, or with similar semi-automatic approaches such as [42]. The tiles generated by Pluto naturally correspond to such tasks, as they are defined through a polyhedral-based cost model with the explicit goal of reducing communication by finding profitable directions for the tiling hyperplanes. (For the rest of the discussion, we will use “task” and “tile” interchangeably.) A second critical issue highlighted in [30, 22, 21] is that of asynchronicity: the presence of synchronization points has significant negative impact on the performance of the parallel implementations. Their PLASMA project [70] addresses this problem through a run-time scheduling framework and manual rewriting of LAPACK routines to use dynamic scheduling for improved scalability.

For automatic transformation frameworks such as Pluto, the generated parallel code (e.g., OpenMP parallel loops) contains barriers that can lead to excessively constrained inter-task synchronization. This problem cannot be solved by any purely-compile-time scheduling approach. Thus, the benefits of automatic, general, and effective parallelization in the polyhedral model cannot be fully realized. This fundamental weakness of these parallelization approaches presents a significant challenge, since it is imperative to effectively schedule the parallel tiles on the processor cores to avoid load imbalance and resource under-utilization.

We propose a novel fully-automatic approach for generating efficient parallel code that can be executed on a multicore system in an asynchronous, load-balanced manner. Our
approach generates, at compile-time, additional program code whose role at run-time is to generate a directed acyclic graph (DAG) of tasks and their dependencies, and analyze the DAG to facilitate dynamic scheduling of the tasks on the processor cores for improved scalable execution. The key insight behind this idea is that the DAG-generating code can be generated at compile-time by constructing a dependence polytope that captures the inter-tile dependences. The DAG-generating code is generated in such a way that, at run-time, it would traverse the points in this polytope. Each such point is essentially a pair of inter-dependent tiles and thus represents an edge in the task dependence DAG.

The developed system is illustrated in Figure 4.4. The task graph generator identifies the tile to be executed by a processing unit at a given time, automatically determines inter-tile dependence information, and generates code that at run-time generates a DAG representing these dependences. The task scheduler adds code that at run-time analyzes the task dependence DAG and infers priorities for dynamically scheduling the tasks. Thus, the input source code is transformed into code encompassing (1) a task code segment (core computation code) to be executed by a processor core, (2) a task dependence DAG generation code segment, and (3) a task scheduling code segment. The run-time execution of the transformed code generates the DAG, analyzes it to infer priorities to be used for scheduling, and executes the tiles on the processing units based on these priorities, maintaining load balance across the processor cores.

4.2.1 Task Graph Generator

The task graph generator component is developed on top of Pluto. As mentioned in Section 2, given an input sequential code, Pluto generates locality-optimized tiled code. The resulting tiles can be effectively scheduled on the processing units by using our dynamic
for (k=0; k<N; k++)
for (j=k+1; j<N; j++)
    S1: a[k][j] = a[k][j]/a[k][k];

for (i=k+1; i<N; i++)
for (j=k+1; j<N; j++)
    S2: a[i][j] = a[i][j] - a[i][k]*a[k][j];

(a) Original LU code

for (c1=0; c1<=floord(N-2,32); c1++)
for (c2=max(ceild(16*c1-15,16),0);
c2<=floord(N-1,32); c2++)
    for (c3=max(ceild(16*c1-465,496),
ceild (16*c1-15,16));
c3<=floord(N-1,32); c3++)
        for (c4 =...)
            for (c5 =...)
                S1(c1,c2,c4,c5)

(b) Tiled LU code

Figure 4.5: Example with LU decomposition

scheduling approach, as opposed to using the compile-time affine scheduling currently employed by Pluto. This component has two sub-components: an inter-tile dependence extractor and a DAG code generator.

**Inter-tile Dependence Extractor**

A dependence polytope captures dependences involving pairs of statement instances accessing a common reference. It is represented as a system of inequalities and equalities capturing the domains of the statements involving the references, affine functions of the references, and ordering imposed by the dependence. In the tiled iteration space, statement instances are represented by higher dimensional statement polytopes involving supernode iterators and intra-tile iterators. Similarly, a dependence between two references in the tiled iteration space is captured by a higher dimensional dependence polytope – it represents a dependence between iterations belonging to the same tile or different tiles. The polytope
that characterizes dependences between iterations in the tiled domain can be generated with the following information:

1. Inequalities describing the iteration spaces of the source and target statement in the original domain.

2. Inequalities defining a tile of the source statement and that defining a tile of the target statement, given by the affine tiling transformation from Pluto.

3. Equalities relating the source statement iterators and target statement iterators with respect to the dependence (h-transformation of the dependence).

Let \(D_s\) and \(\vec{x}_s\) represent the iteration space matrix and iteration vector, respectively, of a statement \(s\) in original domain. Let \(DT_s\) represent the iteration space matrix of the statement in the tiled domain, derived from the tiling transformation generated by Pluto. \(DT_s\) embeds information that defines a tile of the statement (\(T_s\)) and also that defines the original domain of the statement (\(D_s\)). Let \(\vec{x_T}_s\) represent the iteration vector of supernode iterators. Then the domain of the statement in the tiled iteration space is given by

\[
\begin{pmatrix}
\vec{x_T}_s \\
\vec{x}_s \\
\vec{n}
\end{pmatrix} \geq \vec{0}, \quad \text{where} \quad DT_s = \begin{pmatrix}
T_s & 0 \\
0 & D_s
\end{pmatrix}
\]

If there exists a dependence between two statements \(s\) and \(t\), and if \(H\) represents the h-transformation of the dependence, then the dependence polytope in the tiled domain is given by
In our approach for dynamic scheduling of tiles on multi-core parallel systems, we are interested in dependences between tiles, i.e. dependences between iterations belonging to different tiles, to define a dependence preserving schedule of tiles across processor cores. The basic idea to derive inter-tile dependence from a dependence polytope in the tiled domain is to project out the dimensions belonging to intra-tile iterators from the dependence polytope to derive a system of inequalities/equalities involving only inter-tile or supernode iterators. The projection of intra-tile dimensions is done using Fourier-Motzkin elimination. This system is further projected to eliminate tiling dimensions that do not involve in the distribution of tiles across processor cores. The projection procedure is repeated for all dependence polytopes in the tiled domain. A projected dependence polytope has the form

\[
\begin{pmatrix}
T_s & 0 & 0 & 0 \\
0 & D_s & 0 & 0 \\
0 & 0 & T_t & 0 \\
0 & 0 & 0 & D_t \\
0 & -Id & 0 & H
\end{pmatrix}
\left(\begin{pmatrix}
\vec{x}_s \\
\vec{x}_t \\
\vec{x}_t \\
\vec{n} \\
1
\end{pmatrix}
\right)
\geq \bar{0}
\]

**Example.** Figure 4.5(a) shows sequential code for LU decomposition and Figure 4.5(b) shows the corresponding tiled code. The tiles represent the computational tasks, with each task uniquely identified by its tile number. In the LU code in Figure 4.5(b), the pair of values for the outer tile variables (c1,c2) uniquely defines a task. The dependences in the tiled domain for the LU code are illustrated in Figure 4.6(a). The dependences across tiles after the projection of intra-tile dependences are shown in Figure 4.6(b).
DAG Code Generator

At the beginning of the run-time execution, the inter-tile dependences are captured and represented in the form of a task dependence DAG. The tiles are vertices of the DAG and the inter-tile dependences are edges between the corresponding vertices. We could associate a weight with each vertex based on the expected execution time of the tile and similarly, a weight with each edge based on the time to communicate data between the incident tasks.
for (c1=0; c1 < floord(N,32); c1++)
for (c2=max(ceild(16*c1-15,16),0);
c2 < floord(N-1,32); c2++)
dag_add_vertex (c1, c2, 1.0);
(a) Creating all DAG vertices

for (s1=0; s1 <= floord(N-3,32); s1++)
for (s2=max(0,ceild(16*s1-15,16));
s2 < min(floord(N-2,32),s1+1);
s2++)
for (t1=max(max(ceild(16*s1-15,16),
ceild (32*s1-29,32)),0);
t1 < min(min(floord(32*s2+31,32),
s1+1), floord (N-2,32));t1++)
dag_add_edge(s1, s2, t1, t2, 0.0);
(b) Creating all DAG edges

Figure 4.7: Code for DAG generation

However, in our current implementation, we associate unit weights with the vertices and zero weights with the edges.

Recall that our goal is an approach to generate code at compile-time; this code, at run-time, generates the DAG of inter-tile dependences. Since the tiles represent the vertices, the following technique is used to generate code that creates the vertices in the DAG. The iteration space polytopes of all statements in the tiled domain, projected to contain only the supernode iterators, are provided to CLooG. CLooG scans the union of all polytopes and generates a loop nest that enumerates all tiles, and hence all vertices of the DAG. Figure 4.7(a) shows the compile-time-generated code that (at run-time) creates the vertices of the inter-tile dependence graph for the LU decomposition example. Note that this is a fully automatic and general approach: given any automatically (or manually) constructed
tiling, as defined by a set of valid tiling hyperplanes, this technique can directly generate code that enumerates the tiles at run-time in order to construct the DAG vertices.

The following technique is used to generate code that creates the edges in the DAG. The inter-tile dependence extractor outputs a set of projected dependence polytopes (one corresponding to each dependence) that capture the inter-tile dependences. Each dependence polytope contains (1) supernode iterators of the tile containing the source statement instance, and (2) supernode iterators of the tile containing the target statement instance. Each dependence polytope is scanned using CLooG to create a loop nest with source tile iterators as the outer loops and target tile iterators as the inner loops. In this manner, all pairs of inter-dependent tiles are enumerated, and thus all edges of the DAG can be constructed. Figure 4.7(b) shows the code that generates edges of the inter-tile dependence graph corresponding to one of the dependences in the LU decomposition code. The actual DAG generated at run-time is shown in Figure 4.8. A key advantage of this approach is that it is automatic and general. It takes full advantage of all advances in polyhedral dependence analysis and code generation, while at the same time enables parallelizing compilers to go beyond the limitations of purely-compile-time affine scheduling.

4.2.2 Task Scheduler

The task scheduler component adds code that, at run-time, analyzes the task dependence graph to assign priorities to the tasks and dynamically schedule them on cores/processors. The scheduling strategy used in our approach is as follows. Two metrics are associated with each vertex in the DAG (say $G$): *top level* and *bottom level*. The top level of a vertex $v$ in $G$, denoted by $topL(v)$, is defined as the length of the longest path from the source vertex (i.e., the vertex with no predecessors) in $G$ to $v$, excluding the vertex weight of $v$. The length of a
Figure 4.8: Inter-tile dependence DAG for LU decomposition

path in $G$ is the sum of the weights of the vertices and edges along that path. In our current implementation, since we have associated unit weights with the vertices and zero weight with the edges, the length of a path is the number of tasks that need to be executed along that path. The bottom level of a vertex $v$ in $G$, denoted by $bottomL(v)$, is defined as the length of the longest path from $v$ to the sink (vertex with no children), including the vertex weight of $v$. Any vertex $v$ with maximum value of the sum of $topL(v)$ and $bottomL(v)$ belongs to a critical path in $G$.

The tasks are prioritized based on the sum of their top and bottom levels or just the bottom level, and a priority queue of ready-to-run tasks is maintained. A task is ready to run if all its predecessors have completed. Upon completion, each task sets a flag to denote
its completion, computes amongst its children the set of tasks that are ready to run, and adds them to the priority queue. Tasks from the priority queue are executed in priority order on processors/cores as and when they become idle.

Example. In the inter-tile dependence DAG for LU decomposition shown in Figure 4.8, the vertices are marked with $\text{bottomL}(v)$. The figure illustrates the dynamic scheduling strategy based on critical path analysis that prioritizes tasks based on $\text{bottomL}(v)$. The tasks are scheduled for execution based on the (1) completion of predecessor tasks, (2) $\text{bottomL}(v)$ priority, and (3) availability of processor core. Figure 4.9(a) shows the scheduling of the tasks represented in the DAG in Figure 4.8 for a dual-core system, using our dynamic scheduling approach. As evident from the figure, the ready-to-run tasks (tasks...
whose predecessor tasks are completed) with higher $bottomL(v)$ are given priority and the
two cores are fully utilized as long as there are enough ready tasks to be scheduled. When
two tasks with equal priority are ready for scheduling, the one at the top of the priority
queue is chosen.

Figure 4.9(b) shows the affine polyhedral schedule produced by using the default ap-
proach in Pluto. The affine schedule for the case shown in Figure 4.9(b) is derived using
a time schedule $\theta(i, j) = i + j$, where $(i, j)$ is the 2-tuple denoting the task number. Tasks
are executed in a strictly increasing order of $i + j$ values. Tasks with the same $i + j$ value
are executed in parallel based on the availability of processor cores. Comparing the two
schedules in Figure 4.9, it is easy to see the benefits of effective dynamic scheduling over
compile-time affine scheduling. Furthermore, this result is not specific to Pluto or to this
particular affine schedule — in general, such benefits will be observed when compared with
any compile-time affine scheduling approach.

4.2.3 Run-time Execution

As explained earlier, at compile-time our approach generates code that has three seg-
ments: 1) the core computation or task code segment to be executed by a processor core, 2) the
dependence DAG generation code segment, and 3) the task scheduling code segment.
Algorithm 3 lists the steps that are performed at run-time while executing the code gener-
ated by our approach. The steps 5-10 are performed in parallel asynchronously by threads
executing on different cores. The DAG generation code is executed first to create the DAG.
Then $topL(v)$ and $bottomL(v)$ are calculated based on critical path analysis to prioritize the
tasks/vertices. A priority queue is maintained to insert tasks based on priority and extract
them for execution. Each parallel process waits for a task to be ready for execution and
Algorithm 3 Run-time Execution

1: Execute DAG generation code to create a DAG $G$
2: Calculate $\text{topL}(v)$ and $\text{bottomL}(v)$ for each vertex $v \in G$, to prioritize the vertices
3: Create a Priority Queue $PQ$
4: $PQ.insert$ (vertices with no parents in $G$)
5: while not all vertices in $G$ are processed do
6: $taskid = PQ.extract()$
7: Execute $taskid$ // Compute code
8: Remove all outgoing edges of $taskid$ from $G$
9: $PQ.insert$ (vertices with no parents in $G$)

executes it by calling the task code. On completion of the task, all the tasks dependent on it have their wait-count decremented to indicate the completion of one of the parent tasks. A successor task is inserted into the priority queue if its wait-count is zeroed.

4.3 Experimental Results for Compiler-Assisted Dynamic Load-balanced Scheduling

This Section assesses the effectiveness of the developed automatic dynamic scheduling approach using two linear algebra computations: LU and Cholesky decomposition. The sequential code for these computations is first transformed using the Pluto framework to generate locality-optimized tiled code (code that is tiled for data locality optimization at the levels of L1 cache and L2 cache), followed by processing through our task generator component to identify the computational tiles and inter-tile dependences, and generate code to create task dependence DAG, then followed by processing through our task scheduler component to add code that performs dynamic scheduling. The DAG is created at run-time and the computation is dynamically scheduled as described in Section 4.2.2. The reported performance for the dynamically scheduled versions of LU and Cholesky include all the overheads due to dynamic DAG generation as well as dynamic scheduling.
The experiments were conducted on two systems: a) a quad-core Intel Core 2 Quad Q6600 CPU clocked at 2.4 GHz (1066 MHz FSB) with a 32 KB L1 D cache, 8MB of L2 cache (4MB shared per core pair), and 2 GB of DDR2-667 RAM, running Linux kernel version 2.6.22 (x86-64), and b) a dual quad core Intel Xeon(R) E5345 CPU clocked at 2.33 GHz with each chip having a 8MB L2 cache (4MB shared per core pair) and 6 GB RAM, running Linux kernel version 2.6.18. The performance of the parallel code generated by our approach (which enables effective dynamic scheduling at run-time) was compared against that of the parallel code generated by Pluto (which is executed based on static affine polyhedral schedule). ICC 10.x was the primary compiler used to compile the Pluto generated code as well as the code generated by our approach; it was run with -fast -funroll-loops (-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.

An empirical study was carried out on the influence of L2 tile sizes on the statically scheduled (Pluto generated) LU code and our dynamically scheduled LU code, since L2 tiles are the ones that are scheduled for execution on different processor cores. We fixed the problem size as 8K and L1 tile size as $16 \times 300 \times 16$ (kji) and varied the L2 tile sizes. We found that dynamically scheduled parallel code always yielded better performance than statically scheduled parallel code. When the L2 tile sizes were very small ($16 \times 300 \times 16$), the performance of both statically and dynamically scheduled LU was poor, due to high synchronization overheads. The performance improved as the L2 tile sizes were increased, up till a point, after which (from tile sizes larger than or equal to $256 \times 600 \times 256$) the performance of both statically and dynamically scheduled LU saturated with increasing number of cores, due to contention for the shared L2 cache. We found $64 \times 300 \times 64$, $128 \times$
Figure 4.10: Performance of LU on 4 cores
Figure 4.11: Performance of LU on 8 cores
300 × 128 and 256 × 300 × 256 to be good L2 tile sizes for both statically and dynamically scheduled LU.

Figures 4.10 and 4.11 show the performance of LU in GFLOPS and the parallel speedup achieved on the two experimental systems for problem size N=8K. The L1 tile size was fixed as 16 × 300 × 16 and the L2 tile size was fixed as 64 × 300 × 64. As the L2 cache is shared between a core-pair and threads are typically scheduled first to cores that do not share the L2 cache, running an application on up to 2 cores in the quad core system and up to 4 cores in the dual quad core system, will not result in sharing of the L2 cache. We see that for these cases, the dynamically scheduled LU is able to achieve near perfect scaling. Thus dynamic scheduling is very effective in balancing the load on the cores. Even beyond 2 cores in the quad core system and 4 cores in the dual quad core system, dynamically scheduled LU is able to achieve significant performance improvement over statically scheduled LU.

We evaluated the usefulness of dynamic scheduling with another linear algebra computation: Cholesky decomposition. After an empirical evaluation of tile sizes, we fixed the L1 tile size as 8 × 16 × 8 (kij) and the L2 tile size as 64 × 64 × 64. We observed similar trends as for LU decomposition. Figure 4.12 shows the parallel speedup achieved for both statically and dynamically scheduled Cholesky (for a problem size of 8K) on the two experimental systems. We see that dynamic scheduling enables the parallel application to scale very well, achieving close to linear speedups.

As mentioned earlier, the performance measurements of the dynamically scheduled versions of LU and Cholesky include the inter-tile dependence DAG generation overhead and the dynamic scheduling overhead (due to task priority calculation and priority queue maintenance). The overheads introduced by our approach are quite insignificant and do
Figure 4.12: Performance of Cholesky
not affect performance. We measured separately the various overheads involved in our approach using the LU benchmark. The run-time DAG generation takes only around 0.001%, 0.003%, and 0.005% of the total execution time on 2, 4, and 8 processors, respectively. The task priority calculation and priority queue maintenance overhead account for only around 0.013%, 0.023%, and 0.036% of the total execution time on 2, 4, and 8 processors, respectively.

We also conducted experiments to assess the robustness of the dynamic scheduling strategy. Although we do not use empirically measured performance data to model task/vertex weights, assigning unit weights could still capture the priorities effectively. This is because the critical path analysis through inter-tile dependences could effectively capture the task priorities in spite of the less accurate estimate of task weight. Figure 4.13 shows
the performance in GFLOPS for LU decomposition (for a problem size of 8K) for various higher/lower weights assigned to the tasks that perform more computation. As before, the L1 tile size was fixed as $16 \times 300 \times 16$ and the L2 tile size was fixed as $64 \times 300 \times 64$. The performance remains almost the same for various weights assigned to tasks performing more computation, clearly indicating that the scheduling strategy is robust enough even with unit weights assigned to tasks.

We conclude this section with a discussion on the absolute performance achieved relative to machine peak. Although the results presented above demonstrate excellent scalability, the absolute achieved GFLOPS performance is currently lower than the machine peak by over a factor of 2. The single-node performance of the generated tiled code is only about half of the machine peak because vectorization is sub-optimal. The Pluto system currently does not incorporate much sophistication in the approach to vectorization, relying primarily on the vectorization capability of the icc compiler. Work is in progress to implement a much more effective vectorization strategy using vector intrinsics. Another approach that we plan to pursue is that of using tuned kernels such as BLAS routines. The key idea is that of automatically recognizing when the tiled code generated by Pluto can be replaced by pre-optimized kernels. The dominant operation for Cholesky and LU decomposition is the multiply-add, and the core of the tiled code generated by Pluto is essentially a DGEMM. The use of DGEMM to replace the tiled code generated by Pluto requires the automatic separation and extraction of full rectangular tiles from the general polyhedral tiles and the identification of suitable pre-optimized kernels to substitute for the full tiles.
4.4 Related Work

A number of works that use dependence abstractions weaker than those in polyhedral models have addressed loop parallelization [18, 2, 29, 28, 94]. In the context of loop parallelization in polyhedral models, several scheduling-driven works have developed techniques for finding minimum latency schedules or schedules with maximum fine-grained parallelism [36, 37, 29, 45]; these approaches are not aimed at coarse-grain parallelization or locality enhancement. Some works have used fine-grain schedules to determine loop structures which are then tiled to create coarse-grain tasks [36, 37, 45]. In contrast to these, partitioning-driven parallelization is addressed in works of Lim et al. [62, 61, 59] and in Pluto [71, 16, 15, 17, 14]. Note that due to synchronization/communication costs on most modern parallel architectures, at least one level of coarse-grained parallelism is desirable, in addition to enhanced locality. The Pluto approach is the first to explicitly model tiling in a polyhedral transformation framework, which allows us to address two key issues: (i) effective extraction of coarse-grained parallelism, and (ii) data locality optimization.

Nevertheless, depending on the structure of the loops and their parallelization, the tiled output code from any of the above approaches may still suffer from load-imbalance; therefore, dynamic scheduling of tiles is key to improving performance. The approach we pursue here has some similarities to the inspector/executor approach used in runtime compilation [86, 87, 72] in that an analyzer is created at compile-time for execution at run-time to facilitate optimized execution. However, a fundamental difference is that run-time compilation approaches typically use inspectors to obtain essential information (e.g., dependence information) that can only be known at run-time. In contrast, in our context, all dependence information is completely known at compile-time for the affine computations that we address. The problem is that the affine schedule generated by the Pluto framework (or any
other existing automatic parallelization framework) is overly constraining due to the use of a static parallel loop structure with implicit barrier synchronization. The same problem exists with the parallel implementations in LAPACK routines, as highlighted by the recent research from the University of Tennessee [30, 22, 21]. The solution approach we pursue has been inspired by that work, with the main difference being that we seek to generate the dynamically self-scheduling code completely automatically by compiler transformations from sequential code for the computation.

Several efforts have targeted dynamic run-time parallelization [24, 57, 79, 82] as well as speculative parallelization [25, 77, 81]. The basic difference between these approaches and our work is that we use dynamic scheduling to improve performance of loop computations that are amenable to compile-time characterization of dependences.

A plethora of work has been published on the topic of DAG scheduling [1, 89, 88, 41, 54, 51]. Although more sophisticated DAG scheduling algorithms could have been used in our work, we found that a straightforward bottom-level based critical-path dynamic DAG scheduling algorithm was very effective. The focus of our work has not been on exploring alternative scheduling algorithms, but on developing an approach to automatic compile-time generation of DAG generation code to be executed at run-time to facilitate dynamic load balancing of tiled parallel code.

Multi-level tiling approach has been employed in various contexts such as tiling for various levels of memory hierarchy [5, 13, 33], and tiling for parallelism and locality [80, 12]. Multi-level tiling has become a key technique for high-performance computation. There has been work on generating efficient multi-level tiled code for polyhedral iteration spaces that handle tile sizes at compile time [48] and that handle tile sizes as symbolic parameters [52].
4.5 Conclusions

In this Chapter, we have employed an effective multi-level tiling strategy that extracts required levels of parallelism and maps the computation on to the target system, while effectively handling the impact of on-chip memory availability on tile sizes at various levels. We have also developed an approach using the powerful polyhedral model to generate efficient parallel tiled code that can effectively utilize the cores on a multi-core system to minimize the execution time. The developed approach automatically generates a directed acyclic graph characterizing inter-tile dependences that is used for dynamically determining efficient load-balanced schedules to execute the tiles on various cores of a multi-core system.
CHAPTER 5

COMPILE-TIME TRANSFORMATIONS FORGPUS

Graphics Processing Units (GPUs) are now among the most powerful computational systems on a chip. For example, the NVIDIA GeForce 8800 GTX GPU chip uses over 680 million transistors and has a peak performance of over 350 GFLOPS [66]. In addition to the primary use of GPUs in accelerating graphics rendering operations, there has been considerable interest in General Purpose computation on GPUs (GPGPU) [32, 44, 43]. Until very recently, general-purpose computations on GPUs were performed by transforming matrix operations into specialized graphics processing, such as texture operations. The introduction of the CUDA (Compute Unified Device Architecture) programming model by NVIDIA provided a general-purpose multi-threaded SIMD/MIMD architectural model for implementation of general-purpose computations on GPUs. Although more convenient than previous graphics programming APIs for developing GPGPU codes, the manual development of high-performance codes with the CUDA model is still much more complicated than the use of parallel programming models such as OpenMP for general-purpose multicore systems. It is therefore of great interest, for enhanced programmer productivity and for software quality, to develop compiler support to facilitate the automatic transformation of sequential input programs into efficient parallel CUDA programs. However, building such a compiler framework for GPUs requires attention to several additional issues. In
this Chapter we identify and characterize key factors that affect GPGPU performance and develop compile-time transformation approaches for GPGPU optimization [7].

5.1 GPU Architecture and the CUDA Programming Model

In this Section, we discuss about the GPU parallel computing architecture and the CUDA programming interface.

5.1.1 GPU Computing Architecture

The GPU parallel computing architecture comprises of a set of multiprocessor units called the streaming multiprocessors (SMs), each one containing a set of scalar processor cores (called the streaming processors (SPs)). There are various memories available in GPUs for a programmer. The memories are organized in a hybrid cache and local-store hierarchy. The memories are as follows: (1) off-chip global memory, (2) off-chip local memory, (3) on-chip shared memory, (4) off-chip constant memory with on-chip cache, (5) off-chip texture memory with on-chip cache, and (6) on-chip registers. Fig. 5.1 illustrates the organization of processing units and memories in GPUs.

The global memory is a large memory and has a very high latency of 400 to 600 cycles. The shared memory is present in each SM and is organized into banks. When multiple addresses belonging to the same bank are accessed at the same time, it results in bank conflict. Each SM has a set of registers. The constant and texture memories are read-only regions in the global memory space and they have on-chip read-only caches. Accessing constant cache is faster, but it has only a single port and hence it is beneficial when multiple processor cores load from the same location from the cache. Texture cache has higher
latency than constant cache, but it does not suffer greatly when memory read accesses are irregular and it is also beneficial for accessing data with 2D spatial locality.

![Organization of GPU Architecture](image)

Figure 5.1: Organization of GPU Architecture

The architectural configurations of two popular NVIDIA GPU chips, namely, GeForce 8800 GTX and GeForce GTX 280, are presented in Table 5.1.

### 5.1.2 CUDA Programming Model

Programming GPUs for general-purpose applications is enabled through an easy-to-use C/C++ language interface exposed by the NVIDIA Compute Unified Device Architecture
<table>
<thead>
<tr>
<th>Feature</th>
<th>8800 GTX</th>
<th>GTX 280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiprocessors (SMs)</td>
<td>16</td>
<td>30</td>
</tr>
<tr>
<td>Processor cores (SPs)</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Processor Clock</td>
<td>1.35 GHz</td>
<td>1.296 GHz</td>
</tr>
<tr>
<td>Off-chip Memory Size</td>
<td>768 MB</td>
<td>1 GB</td>
</tr>
<tr>
<td>Off-chip Memory BW</td>
<td>384 bits @ 1.8 GHz</td>
<td>512 bit @ 2.2 GHz</td>
</tr>
<tr>
<td>Off-chip Memory BW</td>
<td>86.4 GBps</td>
<td>141.7 GBps</td>
</tr>
<tr>
<td>Processing units</td>
<td>1 multiply-add unit per each SP and 2 SFUs in each SM</td>
<td>1 multiply-add unit and 1 SFU per each SP</td>
</tr>
<tr>
<td>Peak Performance</td>
<td>388.8 GFLOPS</td>
<td>933.12 GFLOPS</td>
</tr>
</tbody>
</table>

Table 5.1: Architectural configurations of NVIDIA GeForce 8800 and 280 GTX

(CUDA) technology [65]. The CUDA programming model provides an abstraction of the GPU parallel architecture using a minimal set of programming constructs such as hierarchy of threads, hierarchy of memories, and synchronization primitives. A CUDA program comprises of a host program which is run on the CPU or host and a set of CUDA kernels that are launched from the host program on the GPU device. The CUDA kernel is a parallel kernel that is executed on a set of threads. The threads are organized into groups called thread blocks. The threads within a thread block synchronize among themselves through barrier synchronization primitives in CUDA and they communicate through a shared memory space that is available to the thread block. A kernel comprises of a grid of one or more thread blocks. Each thread in a thread block is uniquely identified by its thread id (threadIdx) within its block and each thread block is uniquely identified by its block id (blockIdx). The dimensions of the thread and thread block are specified at the time of launching the kernel, through the identifiers blockDim and gridDim, respectively. The dimensions may be 1, 2 or 3.
Each CUDA thread has access to various memories at different levels in the hierarchy. The threads have a private local memory space and register space. The threads in a thread block share a shared memory space. The GPU DRAM is accessible by all threads in a kernel. Fig. 5.2 illustrates the CUDA execution model.

5.1.3 GPU Execution Model

The GPU computing architecture employs a Single Instruction Multiple Threads (SIMT) model of execution. The threads in a kernel are executed in groups called warps, where a warp is an unit of execution. The scalar SPs within a SM share a single instruction unit and the threads of a warp are executed on the SPs. All the threads of a warp execute the same instruction and each warp has its Program Counter. The SM hardware employs a zero overhead warp scheduling through the CUDA runtime scheduler. Warps whose next instruction
has its operands ready are eligible for execution and eligible warps are selected for execution on a prioritized scheduling policy. The warp scheduling is completely transparent to the CUDA programmer.

The computational resources on a multiprocessor unit, i.e., the shared memory and the register bank, are shared among the active thread blocks on that unit. For example, an application abstracted as a grid of 64 thread blocks can have 4 thread blocks mapped on each of the 16 multiprocessors of the NVIDIA GeForce 8800 GTX. The GeForce 8800 GTX GPU has a 16 KB shared memory space and 8192 registers. If the shared memory usage per thread block is 8 KB or the register usage is 4096, at most 2 thread blocks can be concurrently active on a multiprocessor. When any of the two thread blocks complete execution, another thread block can become active on the multiprocessor.

The architectural constraints of GeForce 8800 GTX and GeForce GTX 280 chips are presented in Table 5.2.

<table>
<thead>
<tr>
<th>Resource</th>
<th>GeForce 8800 GTX</th>
<th>GeForce GTX 280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads per thread block</td>
<td>512</td>
<td>512</td>
</tr>
<tr>
<td>Shared memory per SM</td>
<td>16 KB</td>
<td>16 KB</td>
</tr>
<tr>
<td>32-bit registers</td>
<td>8192</td>
<td>16384</td>
</tr>
<tr>
<td>Active threads per SM</td>
<td>768</td>
<td>1024</td>
</tr>
<tr>
<td>Active thread blocks per SM</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 5.2: Architectural constraints of NVIDIA GeForce 8800 and 280 GTX
5.2 Performance Characterization of GPGPU

In this Section, micro-benchmarks are used to characterize key factors that affect GPGPU performance and the implications for compiler optimization are discussed. The micro-benchmarks were run on a NVIDIA GeForce 8800 GTX GPU device.

5.2.1 Global Memory Access

The off-chip DRAM in the GPU device (i.e., the global memory) has latencies of hundreds of cycles. While maximizing data reuse helps to improve the performance of programs with temporal locality, reducing the latency in accessing data from global memory is critical for good performance.

The cost of global memory access was characterized by measuring the memory read bandwidth achieved for different data sizes, for blocked and cyclic distribution of computation amongst the threads. In the micro-benchmark used for bandwidth measurement, a one-dimensional array of size $M$ (where $M = 16 \times N$) was accessed from global memory by 16 thread blocks (one mapped to each multiprocessor unit), where each thread block was a grid of $T$ threads. Each thread in a thread block accessed $N/T$ elements of the array ($N$ was chosen as a multiple of $T$). Two different access patterns were compared: (1) blocked access, where thread 0 accesses the first $N/T$ elements, thread 1 accesses the next set of $N/T$ elements, ..., and thread $T - 1$ accesses the last $N/T$ elements, and (2) cyclic access, where thread 0 accesses element 0, thread 1 accesses element 1, ..., thread $T - 1$ accesses element $T - 1$, and the threads cyclically repeat the same access pattern. The bandwidth achieved is shown in Table 5.3. Although the threads in both cases accessed the same number of elements from global memory, cyclic access resulted in significantly higher memory bandwidth – up to 68.5GBps, improvement by a factor of 10, compared to blocked access.
<table>
<thead>
<tr>
<th>M</th>
<th>Block (GBps)</th>
<th>Cyclic (GBps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2048</td>
<td>4.11</td>
<td>22.91</td>
</tr>
<tr>
<td>4096</td>
<td>4.78</td>
<td>37.98</td>
</tr>
<tr>
<td>8192</td>
<td>5.11</td>
<td>48.20</td>
</tr>
<tr>
<td>16384</td>
<td>5.34</td>
<td>56.50</td>
</tr>
<tr>
<td>32768</td>
<td>6.43</td>
<td>68.51</td>
</tr>
</tbody>
</table>

Table 5.3: Global memory bandwidth for block and cyclic access patterns

The significant difference in performance of the two versions is due to a hardware optimization – \textit{global memory access coalescing}. Accesses from adjacent threads in a half-warp to adjacent locations (that are aligned to 4, 8, or 16 bytes) in global memory are coalesced into a single contiguous aligned memory access. Interleaved access to global memory by threads in a thread block is essential to exploit this architectural feature.

Figure 5.3: Global memory bandwidth for different access strides across threads
Using cyclic data access by threads, the effect on achieved memory bandwidth was evaluated for different numbers of threads per thread block. In addition, the impact of strided data access on memory performance was evaluated. The stride of access across threads was varied from 1 through 64, and the number of threads per thread block was varied from 32 through 512. The results from this experiment are shown in Figure 5.3. It may be observed that non-unit strides across threads lead to significant degradation in performance due to the absence of global memory coalescing. With non-unit access strides, all accesses are issued individually to memory, resulting in poor performance. With unit access stride, as the number of threads per thread block is increased, an increase in the memory bandwidth is observed, with the maximum bandwidth achieved for 512 threads. This is due to better ability to mask global memory access latencies with increase in the number of warps per multiprocessor.

The significant performance benefits due to coalesced access of memory make it one of the most important optimizations to be enabled by a compiler framework for GPGPUs. Also, the high latency of global memory access highlights the importance of reducing the number of global memory loads/stores.

We discuss more about global memory access coalescing in NVIDIA GPUs. Global memory access coalescing is applicable to memory requests issued by threads belonging to the same half-warp. The constraints for global memory accesses of a half-warp to get coalesced are slightly different for NVIDIA GeForce 8800 GTX and NVIDIA GeForce GTX 280. The global memory can be assumed to be consisting of aligned memory segments. We further base our discussion to memory requests for 32-bit words. In 8800 GTX device, when all 16 words requested by the threads of a half-warp lie within the same 64 byte memory segment and if consecutive threads access consecutive words, then all the
memory requests of the half-warp are coalesced into one memory transaction. But if that access pattern is not followed among the threads of a half-warp, then it results in 16 separate memory requests. However, in GTX 280 device, the access pattern need not be so strict for coalescing to happen. In GTX 280, the hardware detects the number of 128 byte memory segments that hold the 16 words requested by the threads of a half-warp and issues as many memory transactions. There is no restriction on the sequence of access within the threads of a half-warp.

In both GPU devices, when the base address of global memory access requests issued by the threads of a half-warp is aligned to memory segment boundary and the threads access words in sequence, it results in fewer memory transactions. It is a strict requirement for coalescing in GeForce 8800 GTX, however it is beneficial even in GeForce GTX 280.

5.2.2 Shared Memory Access

The shared memory is a fast on-chip software-managed memory space that can be accessed by all threads within a thread block. The shared memory space is divided into equal-sized memory modules called banks, which can be accessed in parallel. In the NVIDIA GeForce 8800 GTX and GTX 280, the shared memory is divided into 16 banks. Successive 32-bit words are assigned to successive banks. Hence, if the shared memory addresses accessed by a half-warp (i.e., the first 16 threads or the next 16 threads of a warp) map to different banks, there are no conflicting accesses, resulting in 16 times the bandwidth of one bank. However if \( n \) threads of a half-warp access the same bank at a time, there is an \( n \)-way bank conflict, resulting in \( n \) sequential accesses to the shared memory. In our further discussion, we refer to the number of simultaneous requests to a bank as degree of bank conflicts. Hence \( k \) degree of bank conflicts means a \( k \)-way bank conflict and 1 degree of
bank conflicts means no bank conflicts (since there is only one simultaneous request to a bank). The bandwidth of shared memory access is inversely proportional to the degree of bank conflicts.

![Graph showing average clock cycles per word for different access strides across threads.](image)

Figure 5.4: Shared memory access time for different access strides across threads

We conducted an experiment to study the effect of bank conflicts, by measuring the shared memory access time for different access strides (strides from 0 to 16) across threads. 32 threads per thread block were used for the experiment and each thread accessed 100 data elements (100 32-bit words). When access stride is 0, all threads access the same word. When access stride is 1, successive threads in a thread block access successive words in shared memory, which fall in different banks. Hence there are no bank conflicts between the thread accesses. When access stride is 2, the first thread accesses a word from bank $i$,
the second thread accesses a word from bank $i+2$ and so on. Thus there is a 2-way conflict, i.e., conflict between the accesses of thread 0 and thread 8, thread 1 and thread 9, \ldots, thread 7 and thread 15. Figure 5.4 shows the observed behavior. There are no bank conflicts when the access stride is odd and hence the observed access time is fastest for odd access strides. From Figure 5.4, we clearly observe that shared memory access time depends on the degree of bank conflicts and access time is almost the same for access strides that lead to the same degree of bank conflicts. From Figure 5.4, we also observe that the access time when all threads access the same word is as fast as that when there are no bank conflicts. This is due to a hardware optimization in the shared memory that enables broadcast of a 32-bit word to several threads simultaneously when servicing one memory read request.

The importance of the fast on-chip shared memory space that is introduced in GPU architectures to improve the memory access performance elucidates the fact that minimizing shared memory bank conflicts is an important optimization to be handled by a compiler framework for GPGPUs.

5.2.3 Degree of Parallelism vs Register Pressure

One of the important optimizations to be performed in the thread-level computation code is to reduce the number of dynamic instructions in the run-time execution. Loop unrolling is one of the techniques that reduces loop overhead and increases the computation per loop iteration. Also, register-level tiling through unroll-and-jam to reduce number of loads/stores per computation is a well known program optimization when there is sufficient reuse in the data accessed. Though loop unrolling reduces dynamic instructions and register tiling reduces the number of loads/stores, they increase register usage. The number of
threads that can be concurrently active in a multiprocessor unit depends on the availability of resources such as shared memory and registers. A thread block of threads can be launched in a multiprocessor unit only when the number of registers required by its threads and the amount of required shared memory are available in the multiprocessor. Clearly, increased register pressure may reduce the active number of threads in the system.

For code for which performance is limited by memory access, having more threads can efficiently mask global memory access latency. Figure 5.3 clearly illustrates the impact of parallelism on the bandwidth of global memory access. Hence a memory-access-bound code requires more threads to efficiently overcome the global memory access latency.

Putting the issues together, a computation that has large global memory access overhead requires higher concurrency and also demands more registers to enable the benefits of loop unrolling such as loop overhead reduction and reduction of number of loads/stores. Hence there is a clear trade-off between number of active concurrent threads and number of registers available for a thread in a thread block to exploit the above benefits. Due to such a tight coupling of GPU resources, an empirical evaluation becomes necessary to select an optimal choice of program parameters such as unroll factors and tile sizes, and system parameters such as number of threads and thread blocks.

Having identified the key performance-influencing characteristics of GPUs, we now discuss the compile-time optimization approaches developed to address these issues, towards the goal of developing a compiler framework for automatic parallelization and performance optimization of affine loop nests on GPGPUs.
5.3 Optimizing Global Memory Access

In this section, we develop an approach for performing program transformations that enable interleaved access to global memory by threads in a thread block which is necessary to facilitate coalesced global memory accesses and thereby improve global memory access performance.

5.3.1 Global Memory Access Coalescing

In GPUs, execution of a program proceeds by distributing the computations across thread blocks and across threads within a thread block. In a thread block, data required for computation can be either accessed directly from global memory or copied to shared memory and then accessed. We focus on the code executed in a thread block to optimize for global memory access. We develop the global memory access optimization approach on top of Pluto [71] automatic transformation framework. As discussed earlier, Pluto optimizes sequences of imperfectly nested loops, simultaneously for parallelism and locality, through tiling transformations. We use Pluto to generate tiling transformations to distribute tiles across thread blocks. For finding tiling hyperplanes to generate tiled code at the level of threads, we modify Pluto to generate program transformations that enable interleaved access to global memory by threads in a thread block - this is necessary to facilitate coalesced global memory accesses that improve global memory access bandwidth.

We first determine array references whose accessed data space have either higher-order reuse or sufficient constant reuse and mark them as candidates that have to be copied from global memory to shared memory for efficient performance. Array references whose accessed data space have no reuse are candidates for direct access from global memory. But
inefficient access to global memory may degrade the performance as illustrated in Section 5.2.1. We find program transformations that can lead to efficient direct global memory access of as many ‘candidate’ array references as possible. If a resulting program transformation does not optimize access of an array reference, then the data accessed by the reference is copied (efficiently) to shared memory.

To enable global memory coalescing for an array reference in a statement, iterations accessing adjacent elements (along the fastest varying dimension) of the array due to the reference have to be executed simultaneously (in time) by distinct virtual processors that are consecutive in processor space. This is enforced by the time schedule adjacency constraint which enforces two statement instances that access adjacent elements of an array to be executed at the time instance. The time schedule adjacency constraint is defined (assuming row major storage of arrays) as:

\[ \forall x_s \in D_s, \forall y_s \in D_s \text{ s.t. } F_{r_{2s}}(x_s) + (0 \ldots 1)^T = F_{r_{2s}}(y_s), \quad \theta_s(x_s) = \theta_s(y_s) \quad (5.1) \]

In addition to the above constraint, iterations accessing adjacent data elements of an array have to be in adjacent space partitions so that they are accessed by adjacent virtual processors. This is enforced by the space partition adjacency constraint which enforces two statement instances that access adjacent elements of an array to be executed by adjacent processors in the processor space. The space partition adjacency constraint is defined as:

\[ \forall x_s \in D_s, \forall y_s \in D_s \text{ s.t. } F_{r_{2s}}(x_s) + (0 \ldots 1)^T = F_{r_{2s}}(y_s), \quad \pi_s(y_s) = \pi_s(x_s) + 1 \quad (5.2) \]

The space adjacency constraint also enforces cyclic distribution of virtual processors to physical processors as block distribution may nullify the effect of optimization achieved by the transformation satisfying space adjacency constraint.
We now explain the procedure used to determine transformations, that enable interleaved global memory access by threads, for code executed in a thread block. In our approach, we solve for a time schedule (for each statement) that preserves all dependencies and satisfies *time schedule adjacency constraint* (Equation 5.1) for all candidate array references whose accessed data space do not have enough reuse in the program. If there does not exist a solution, we try all subsets of those array references and generate time schedules that satisfy the *time schedule adjacency constraint* and potentially generate space partitions that satisfy the *space partition adjacency constraint* (Equation 5.2). Once a \( t \) dimensional time schedule is determined for a statement with \( m \) loops surrounding it, we find a \( m - t \) dimensional space partition mapping such that each of the \( m - t \) mappings are linearly independent of each other and the time schedule, and one of the space mappings (treated as the innermost space partition) satisfies the *space partition adjacency constraint*. If there is no valid space-time transformation satisfying dependencies and adjacency constraints, for all statements, for any non-empty subset of array references considered, then we use Pluto-generated tiling transformation (space-time transformation generated without enforcing adjacency constraints) for each statement. The procedure is summarized in Algorithm 4.

### 5.3.2 Examples

Consider the kernels in Figure 5.5. Array \( a \) in mv and tmv kernels has no reuse and is considered for direct global memory access. Without applying the constraints defined by Equations 5.1 and 5.2, we get the following valid time schedule and space partition mapping for statement \( Q \) in mv kernel and statement \( T \) in tmv kernel.

\[
\theta_Q(x_Q) = j \quad \text{and} \quad \pi_Q(x_Q) = i
\]
Algorithm 4 Finding transformations enabling coalesced global memory access

**Input:** Set of statements - $S$, Iteration Space Polytopes of all statements $I_s, s \in S$, Array references (whose accessed data space do not have reuse) - $\{F_{xyz}\}$, Set of Dependencies - $\mathcal{R}$

1. for all non-empty subsets $G$ of array references do
2. Find a time schedule $\theta$ for each statement $s$ that preserves all dependencies in $\mathcal{R}$ and satisfies time schedule adjacency constraint (5.1) for all references in $G$.
3. for each statement $s$ (with dimensionality of iteration space being $m$ and dimensionality of time schedule being $t$) do
4. Find a space partition $\pi_1$ that is linearly independent to $\theta$ and satisfies space partition adjacency constraint (5.2) for all references in $G$. Mark this space partition as the innermost space partition.
5. Find $m-t-1$ space partitions that are linearly independent to each other and also to $\pi_1$ and $\theta$.
6. if no valid space-time transformation (satisfying adjacency constraints) exists for all statements, for any non-empty subset of array references considered then
7. Use tiling transformation (space-time transformation) generated by Pluto without enforcing adjacency constraints, for each statement.

**Output:** Transformations enabling coalesced global memory access along with marking of references for which copy to shared memory is needed

mv kernel:

```c
for (i=0;i<n;i++) {
    P: x[i]=0;
    for (j=0;j<n;j++)
        Q: x[i]+=a[i][j]*y[j];
}
```

tmv kernel:

```c
for (i=0;i<n;i++) {
    S: x[i]=0;
    for (j=0;j<n;j++)
        T: x[i]+=a[j][i]*y[j];
}
```

Figure 5.5: mv and tmv kernels

$$\theta_T(\vec{x}_T) = j \quad \text{and} \quad \pi_T(\vec{x}_T) = i$$

where $\vec{x}_Q = (i_j)$ and $\vec{x}_T = (j_i)$.

Applying adjacency constraints for the mv kernel (in a system with row major storage) yields no valid transformation. Adjacent global memory access by distinct threads is
possible only across different $j$-loop iterations of an $i$-loop iteration. Hence the time schedule adjacency constraint results in a time schedule $\theta_Q(x_Q) = i$, which does not dismiss all dependencies. Hence there is no valid transformation possible that can enable coalesced global memory access. Hence the transformation (time schedule and space partition mapping) obtained without applying adjacency constraints is used and array $a$ in mv kernel is copied to shared memory and accessed, but not accessed directly from global memory.

On the other hand, applying adjacency constraints for the tmv kernel, yields a time schedule $\theta_T(x_T) = j$ (as adjacent global memory access by distinct threads is possible across different $i$-loop iterations of an $j$-loop iteration) and a space partition mapping $\pi_T(x_T) = i$, which preserve data dependencies, and hence results in a valid transformation that can enable coalesced global memory access.

### 5.3.3 Effective Use of Register and Non-register Memories

The approach not only makes decision on what data needs to be moved to shared memory and what needs to be accessed directly from global memory, but also makes decisions on effectively using register memory and non-register memories such as constant memory, and thereby reduces the number of global memory accesses. Constant memory has an on-chip portion in the form of cache which can be effectively utilized to reduce global memory access. Access to constant memory is useful when a small portion of data is accessed by threads in such a fashion that all threads in a warp access from the same location simultaneously. When threads in a warp access different locations in constant memory, the requests are serialized.

We determine arrays that are read-only and whose access function does not vary with respect to the loop iterators corresponding to the parallel loops that are used for distributing
computation across threads, and consider them as candidates for storing in constant memory. Similarly arrays whose access function varies only with respect to the loop iterators corresponding to the parallel loops are considered as candidates for storing in registers in each thread.

5.3.4 Optimized Copy from Global Memory to Shared Memory

Array references that have sufficient reuse and array references that are marked to be copied to shared memory because of infeasible transformation for coalesced global memory access, have to be efficiently copied from/to shared memory. A detailed discussion on the approach to determine accessed data spaces of array references, automatically allocate storage space in the form of arrays in shared memory, and generate code to move data between global and shared memories was presented in Chapter 3. The loop structure of the data movement code (copy code) is a perfect nest of $n$ loops, where $n$ is the dimensionality of the accessed data space. By using a cyclic distribution of the innermost loop across threads of a warp, we enable interleaved access of global memory by threads.

5.4 Optimizing Shared Memory Access

This section describes our approach to optimize access of on-chip shared memory in GPU multiprocessor units. Following the observation from Section 5.2.2, optimization of shared memory access can be equivalently viewed as minimization of bank conflicts. The strategy to minimize bank conflicts in shared memory access is to pad the arrays copied into shared memory. However, finding a suitable padding factor for an array in shared memory is not trivial. The procedure of finding a padding factor for an array in order to minimize bank conflicts has to consider the effects of padding on all references made to
the array. Padding to minimize bank conflict with respect to one reference might have a negative impact with respect to another reference.

We define a formal relation between the degree of bank conflicts and the access stride across threads in a half warp that determine the degree of bank conflicts and hence the shared memory access bandwidth. With shared memory organized into banks and successive words stored in successive banks in a cyclic pattern, the degree of bank conflicts is given by $GCD(stride\ of\ array\ access\ across\ threads\ of\ a\ half\ warp,\ number\ of\ bank\ modules)$. When the stride of array access across threads is zero, i.e. when all threads access the same word, there is a special hardware optimization in the GPU architecture that enables broadcast of the word to all threads. Hence in that case, the degree of bank conflicts is considered as one as there is only one simultaneous bank request.

We model the cost of accessing a word from a shared memory bank as a linear function of the degree of bank conflicts. Let $C(n)$ be the cost of accessing a word from a shared memory bank when there are $n$ simultaneous requests to the bank (possibly by different threads of a half warp). The cost function is given by

$$C(n) = t_{start} + t_{request} \times n$$  \hspace{1cm} (5.3)

where $t_{start}$ is the startup time to access a bank when there is one or more requests to the bank and $t_{request}$ is the time to service a request. Figure 5.6 shows the trend of the linear shared memory bank access cost function (plotted using data obtained from the experiment described in Section 5.2.2).

The algorithm to find optimal padding factors for arrays in shared memory focuses on the code to be executed in a thread block which has been transformed for global memory coalescing using the approach described in Section 5.3.1. The algorithm has information
regarding arrays that need to be copied into shared memory, as well as the space and time partitions of each statement in the code. For each reference, based on the distribution of innermost space loop across threads (block or cyclic), the distance between data points accessed by successive threads is calculated and it defines the access stride across threads for that reference. For each array in shared memory, the algorithm enumerates all padding factors and finds the optimal one that minimizes the total number of bank conflicts caused by all the references to the array.

### 5.5 Approach for Selecting Optimal Tile sizes and Unroll factors

In this section, we discuss optimization of program parameters such as tile sizes and unroll factors that are closely linked with the choice of system parameters such as number of threads and number of thread blocks used for execution, and the availability of GPU
Algorithm 5 Finding Optimal Padding Factor

**Input:** Input array for which padding is to be determined - \( A \), Number of references to \( A \) in the sub-program - \( N_{ref} \), Original access strides across threads for the \( N_{ref} \) references - \( AS[N_{ref}] \), Number of bank modules - \( NB \), Cost function from Eq. (5.3) - \( C \)

1. \( \text{MinAccessCost} = \infty \)
2. \( \text{OptPadding} = 0 \)
3. for \( pad = 0 \) to \( NB - 1 \) do
4. \( \text{TotalAccessCost} = 0 \)
5. for \( ref = 1 \) to \( N_{ref} \) do
6. Calculate new access stride \( AS_{\text{new}}[ref] \) for reference \( ref \) using original access stride \( AS[ref] \) and padding factor \( pad \)
7. if \( AS_{\text{new}}[ref] = 0 \) then
8. \( \text{BankConflict}[ref] = 1 \)
9. else
10. \( \text{BankConflict}[ref] = \text{GCD}(AS_{\text{new}}[ref], NB) \)
11. \( \text{AccessCost}[ref] = C(\text{BankConflict}[ref]) \)
12. \( \text{TotalAccessCost} += \text{AccessCost}[ref] \)
13. if \( \text{TotalAccessCost} < \text{MinAccessCost} \) then
14. \( \text{OptPadding} = pad \)
15. \( \text{MinAccessCost} = \text{TotalAccessCost} \)

**Output:** Optimal padding factor for \( A \) - \( \text{OptPadding} \)

local resources such as shared memory and registers. The approach we take is to employ a model-driven empirical search on tile sizes and unroll factors.

We perform multiple levels of tiling for exploiting parallelism across thread blocks and threads, and also perform register-level tiling through unroll-and-jam to optimize thread-level code. The first level of tiling is done to exploit parallelism across thread blocks. In GPUs, the size of a tile executing in a thread block at a time instance depends on the amount of shared memory available for execution of the thread block. The second level of tiling within a thread block is done, if needed, to bound shared memory usage within available limits. When the number of iteration points in a loop executed within a thread block is more than the number of threads, one more level of tiling is needed to distribute the computation
across threads. Finally, if there is enough reuse to be exploited, register-level tiling is done to reduce the number of loads/stores from global/shared memory.

It would be ideal to characterize and model tile size determination based on the number of loads/stores between global and shared memory, and the number of loads/stores between shared memory and registers. We now discuss a model to estimate memory traffic expected during the execution of a tile. This is then used to guide an empirical search on tile sizes and unroll factors.

5.5.1 Model to Estimate Memory Traffic

Consider a tile to be executed by a thread block or a thread. The iteration space of statements in the tile is parameterized by the tile sizes of the loops defining the tile. Consider a tile of \( n \) loops with tile sizes being \( t_1, t_2, \ldots, t_n \). Consider \( k \) arrays \( (a_1, a_2, \ldots, a_k) \) being accessed in the tile. Let \( r_i \) be the number of read references and \( w_i \) be the number of write references of array \( a_i \). Let \( \mathcal{F}_{ij1}, \mathcal{F}_{ij2}, \ldots, \mathcal{F}_{ijr_i} \) be the read accesses of array \( a_i \) in the tile and \( \mathcal{G}_{ij1}, \mathcal{G}_{ij2}, \ldots, \mathcal{G}_{ijw_i} \) be the write accesses of array \( a_i \) in the tile. Let \( I \) be the iteration space of the tile parameterized by the tile sizes. Let \( f \) be a function that counts the number of integer points in a polytope given the parameters. Let \( \mathcal{DS}_l \) denote the accessed data space of read references of array \( a_i \). The number of integer points in polytope \( \mathcal{DS}_l \) gives the number of loads due to array \( a_i \). Let \( \mathcal{DS}_s \) denote the accessed data space of write references of array \( a_i \). The number of integer points in \( \mathcal{DS}_s \) gives the number of stores due to array \( a_i \).

The model to estimate memory loads and stores in a tile can be characterized as follows.

\[
\mathcal{DS}_l = \bigcup_{j=1}^{r_i} \mathcal{F}_{ij} I \quad \text{and} \quad \mathcal{DS}_s = \bigcup_{j=1}^{w_i} \mathcal{G}_{ij} I
\]
The number of loads and stores for an array $a_i$ in a tile =

$$f(\mathcal{DS}_{l_i}, t_1, t_2, \ldots, t_n) + f(\mathcal{DS}_{s_i}, t_1, t_2, \ldots, t_n)$$

As discussed in Section 4.1.2, the load and store of each array can be placed at an optimal position in the tiled loop nest. Based on the position, the total number of loads and stores (for the entire tiled iteration space) can be estimated for all arrays. Thus the total memory traffic can be estimated.

### 5.5.2 Model-driven Empirical Search

Using the model discussed above, we can obtain an estimate of memory loads/stores. However, because of the lack of control on the number of registers actually used by NVIDIA CUDA Compiler (NVCC), and because of the tight coupling of the GPU resources, optimal tile sizes and unroll factors cannot be determined by a cost model alone. An empirical search is needed to find an optimal set of tile sizes for the tiled loops and optimal unroll factors for the loops that are unrolled. Hence we employ an empirical search to pick the optimal code variant among various code variants resulting due to different transformations enabling efficient global memory access, different tile sizes at multiple levels, and different unroll factors. The search space due to different choices of tile sizes and unroll factors are pruned with the help of the cost model that estimates memory loads/stores.

The model-guided empirical search procedure that we used is outlined below.

- For each tiled loop structure, determine the register usage $r$ and determine the maximum concurrency ($L$ threads) possible within a multiprocessor. (There is an option in NVCC to generate a low-level object code file called the cubin file that provides information on the amount of shared memory used by a thread block and the number
of registers used by a thread in a thread block). Set the exploration space of number of threads in a thread block to be $T, T/2, T/4$, where $T$ is the nearest multiple of warp size of the GPU device less than $L$ and 512.

- For all valid tile sizes that distribute computation equally among thread blocks and among threads within a thread block, and satisfy shared memory limit constraint, estimate the total number of global memory loads/stores using the model in Section 5.5.1. Discard loop structures for which the number of loads/stores is more than twice that of the loop structure with the lowest number of loads/stores.

- For all selected loop structures, do register-level tiling and explicit unrolling, instrument the register usage and discard those for which register pressure is increased to an extent where concurrency is reduced to less than 25% of maximum possible concurrency.

- Search empirically among the selected code versions by explicitly running them and timing the execution time, and select the best one.

### 5.5.3 Experimental Evaluation for Model-driven Empirical Search

The MM kernel is used to illustrate the steps involved in the model-driven empirical search procedure explained above. A problem size of $4096 \times 4096$ (that was barely able to fit in the GPU DRAM of 8800 GTX) was used.

For the multi-level tiled code generated using the program transformations (without loop unrolling and register-level tiling), the register usage per thread was estimated using cubin as 13, leading to a possibility of 512 concurrent threads. Further experiments were
done for 128, 256 and 512 threads per thread block. The number of thread blocks was varied between 16, 32 and 64.

For various tile sizes that distribute the computation equally among thread blocks and among threads within a thread block, and satisfy the shared memory limit constraint, the total global memory loads varied from the order of $4096^3/2^7$ to $4096^3/2^4$. All code versions which had loads in the order of $4096^3/2^7$ to $4096^3/2^6$ were considered and various combinations of loop unrolling and register-level tiling were performed for the selected code versions. Since the choices of register-level tiling depend on the size of the tile being executed in a thread, the choices were limited. The register usage of each unrolled, register-tiled version was determined, and those versions with excessive register usage (those which
restricted the number of concurrent threads to below 128) were eliminated. Figure 5.7 illustrates the performance of the selected candidates that were run empirically to select the best one. The code version that was selected by the search procedure resulted in a performance of around 97 GFLOPS – compared to vendor-optimized MM kernel performance of around 101 GFLOPS.

5.6 Experimental Results

In this Section, we demonstrate the benefits of the multi-level tiling approach detailed in Section 4.1 for mapping computation in regular programs on to GPUs which are representatives of multi-level parallel systems. We also demonstrate the effectiveness of the compile-time techniques developed for efficient execution on GPUs.

Experiments were conducted on an NVIDIA GeForce 8800 GTX GPU device. The device has 768 MB of DRAM and has 16 multiprocessors (MIMD units) clocked at 675 MHz. Each multiprocessor has 8 processor cores (SIMD units) running at twice the clock frequency of the multiprocessor and has 16 KB of shared (scratchpad) memory. CUDA version 1.0 was used for the experiments. The CUDA code was compiled using the NVIDIA CUDA Compiler (NVCC) to generate the device code that is launched from the CPU (host). The CPU was a 2.13 GHz Intel Core2 Duo processor with 2 MB L2 cache. The GPU device was connected to the CPU through a 16-x PCI Express bus. The host programs were compiled using the icc compiler at -O3 optimization level.

5.6.1 Multi-level Tiling

Experimental study was conducted on two kernels – an MPEG4 Motion Estimation (ME) kernel and a 1-D Jacobi kernel, the former requiring no synchronization across thread
blocks and the latter requiring synchronization across thread blocks. Multi-level tiling was performed on the kernels and data in scratchpad memory was managed using the automatic data management approach.

![Graph showing execution time for various problem sizes]

**Figure 5.8: Execution time of MPEG4 ME for various problem sizes**

We first present experimental results on using scratchpad memory to efficiently access data for computation. Fig. 5.8 and Fig. 5.9 illustrate the benefits of efficient data access using scratchpad memory and also exemplify the high speedup achieved by running the computation on the GPU in contrast to running on the host CPU. The speedup of the implementation utilizing scratchpad memory was 8x for the MPEG4 ME kernel and 10x for the Jacobi kernel over that using only GPU DRAM. The speedup over CPU performance was over 100x for the MPEG4 ME kernel and 15x for the Jacobi kernel. The performance
Figure 5.9: Execution time of 1D Jacobi for various problem sizes

(in GFLOPS) of these kernels is shown in Fig. 5.10 and Fig. 5.11. For the Jacobi kernel, for smaller problem sizes, performance increases as problem size increases since the overhead in launching the kernels in GPUs and reading the results back from GPU DRAM is non-negligible for smaller sizes. For larger problem sizes, performance remains almost constant as problem size is increased. There is a dip in performance after problem size of 32K. This is the point at which an additional level of tiling within a thread block (and hence more synchronization within a thread block) is required due to limited scratch pad memory availability. For the MPEG4 ME kernel, similar to the Jacobi kernel, performance is affected by the GPU kernel launching overhead for smaller problem sizes and performance remains almost constant for larger problem sizes.
Figure 5.10: Performance (GFLOPS) of MPEG4 ME for various problem sizes

The structure of original code and that of multi-level tiled code of the MPEG4 ME kernel are shown in Fig. 4.2 and Fig. 4.3. For various problem sizes, experiments were conducted to analyze the performance of the MPEG4 ME kernel. The results are shown in Fig. 5.12. The number of thread blocks was chosen as 32 and the number of threads as 256. The $i$ and $j$ loops in Fig. 4.2 are the space loops that are tiled across thread blocks and threads, and $k$ and $l$ loops are the time loops. The sizes of tiles that distribute computation across thread blocks were set by dividing the problem size equally (except for boundary) among the thread blocks. The time loops $k$ and $l$ in this kernel iterated over a very small index range (16 in our experiment) and hence the data accessed in the iteration space of the loops did not occupy much space in the scratchpad memory. The tile search algorithm described in Section 4.1.3 found tile sizes of 32, 16, 16 and 16 for $i$, $j$, $k$ and $l$ loops to be
optimal as these tile sizes lead to lower data movement cost by reducing the number of data movements, given the scratchpad memory constraint. From the results shown in Fig. 5.12, it is clear that the tile sizes chosen by the algorithm gave better performance than other tile sizes, for various problem sizes.

The Jacobi-1D kernel has a space loop surrounded by a time loop. The overlapped tiling approach described in [53] was used to modify the tiled code to enable concurrent start of execution on all processes. Multi-level tiling was performed over the modified code. Experiments were run on the kernel for various problem sizes that could completely fit in the total scratchpad memory available in the GPU device. Results are depicted in Fig. 5.13. The number of time iterations was chosen as 4096 and time tile size as 32. The problem size was equally divided (except for boundary) by the number of thread blocks used, to set
the size of the space tile executed per thread block. The number of threads used was 64. For problem sizes that completely fit in the total scratchpad memory available in the device, the number of thread blocks has no constraint. However, since the kernel requires synchronization across all thread blocks, for very high number of thread blocks, the computation done by a thread block is too small to hide the synchronization cost incurred. The same behavior is illustrated by Fig. 5.13. Performance first improves as the number of thread blocks increases, and then decreases when the synchronization cost dominates the time for computation done in a thread block.

For larger problem sizes that have to do be tiled to fit in the scratchpad memory, we set the number of thread blocks to be 128, determined empirically from the experimental results shown in Fig. 5.13, to allow better concurrency and incur lower synchronization
cost. The active scratchpad memory used by a thread block was hence limited to $2^{11}$ bytes or $2^9$ words. The number of threads was set to 64. The tile search algorithm described in Section 4.1.3 gave a space tile size of 256 and time tile size of 32 to be optimal for minimizing the data movement cost between scratchpad memory and global memory. The experiments confirmed the same, as indicated by the results in Fig. 5.14 that show the performance of the kernel for various tile sizes and different problem sizes.

### 5.6.2 Compile-time Transformations for GPUs

Figure 5.15 shows the performance of several kernels — Matrix Vector multiply (mv), Transposed Matrix Vector multiply (tmv), Matrix Vector Transpose (mvt), and Matrix Matrix multiply (mm) — that were optimized using the compile-time techniques developed
in this Chapter. It is to be noted that multi-level tiled code for all kernels was generated using the approach detailed in Section 4.1. The performance comparison was done with the vendor-optimized CUBLAS library (version 1.0) supplied by NVIDIA. The effectiveness of the compile-time optimizations developed in this Chapter is evident from these results: for mv, tmv and mvt kernels, the performance achieved is better than that of the vendor-optimized CUBLAS implementation, and for mm kernel, the performance is close to that of CUBLAS implementation.

The mv and tmv kernels (Figure 5.5) are used to illustrate the benefits of global and shared memory access optimization. Table 5.4 shows the performance of mv kernel implemented using space and time partition mappings, as discussed in Section 5.3.2. Our approach makes a decision to copy the elements of array \( a \) into shared memory though
there is no reuse of the elements copied. An implementation with efficient copy of elements of array \( a \) from global memory to shared memory (column “Non-optimized Shared”) provides an order of magnitude better performance than the version implemented with direct access of \( a \) from global memory (column “Direct Global”). The implementation with copy to shared memory is further enhanced by minimizing the shared memory bank conflicts through effective padding of the shared memory buffer created to hold the elements of array \( a \). A further 2x improvement in performance is achieved (column “Optimized Shared”) due to this shared memory access optimization.

<table>
<thead>
<tr>
<th>N</th>
<th>Direct Global</th>
<th>Optimized Shared</th>
<th>Non-optimized Shared</th>
</tr>
</thead>
<tbody>
<tr>
<td>4K</td>
<td>0.43</td>
<td>13.18</td>
<td>5.61</td>
</tr>
<tr>
<td>5K</td>
<td>0.48</td>
<td>13.87</td>
<td>5.79</td>
</tr>
<tr>
<td>6K</td>
<td>0.35</td>
<td>14.37</td>
<td>6.04</td>
</tr>
<tr>
<td>7K</td>
<td>0.30</td>
<td>13.86</td>
<td>5.78</td>
</tr>
<tr>
<td>8K</td>
<td>0.24</td>
<td>13.63</td>
<td>5.52</td>
</tr>
</tbody>
</table>

Table 5.4: Performance comparison (in GFLOPS) of mv kernel

Table 5.5 shows the performance of the tmv kernel implemented using the space and time partition mappings discussed in Section 5.3.2. When tiling along space loops is done in a blocked fashion to map virtual processors in a space partition mapping to threads, it violates the coalesced memory access constraints and performance degrades (column “Non-optimized Global”). Hence tiling along space loops is done in a cyclic fashion (column “Optimized Global”), as inferred by our approach.

Matrix Vector Transpose (mvt) is a kernel that involves two matrix-vector multiplies, where one matrix is the transpose of the other, and hence it encompasses the computations involved in mv and tmv kernels. Our approach identifies the data space accessed by array
<table>
<thead>
<tr>
<th>N</th>
<th>Non-optimized Global</th>
<th>Optimized Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>4K</td>
<td>4.22</td>
<td>25.21</td>
</tr>
<tr>
<td>5K</td>
<td>3.09</td>
<td>28.90</td>
</tr>
<tr>
<td>6K</td>
<td>3.24</td>
<td>33.47</td>
</tr>
<tr>
<td>7K</td>
<td>3.70</td>
<td>33.58</td>
</tr>
<tr>
<td>8K</td>
<td>4.13</td>
<td>34.93</td>
</tr>
</tbody>
</table>

Table 5.5: Performance comparison (in GFLOPS) of tmv kernel

reference involved in mv kernel computation as a candidate to be copied to shared memory and on the other hand identifies the data space accessed by array reference involved in tmv kernel computation as a candidate for direct global memory access, and results in optimized global memory access.

5.7 Related Work

Prior to the introduction of CUDA [65], GPU programming systems have relied on graphics API-based implementations, which have limited the size and kind of codes that are implementable on GPUs. In addition, CUDA has significantly enhanced programmer productivity by relieving the programmer of the burden of thinking in terms of graphics operations. Previous GPU generations and their APIs had restrictive memory access patterns such as allowing only sequential writes to a linear array. For example, Accelerator [91] does not allow access to an individual element in parallel arrays and operations are performed on all array elements. Brook [20] is a stream-based model that executes its kernel for every element in the stream with restrictions. The GeForce 8800 allows for general addressing of memory by each thread, which supports a much wider variety of algorithms.
With this general addressing, it is important to apply data locality optimizations in order to exploit high bandwidth and hide memory latency.

Traditional GPUs also provided limited cache bandwidth for general purpose applications. Fatahalian et al. [32] mention that low-bandwidth cache designs on GPUs prevent general purpose applications from benefiting from the available computational power. Govindaraju et al. [43] use an analytical cache performance prediction model for GPU-based algorithms. Their results indicate that memory optimization techniques designed for CPU-based algorithms may not directly translate to GPUs.
In the context of getting good performance on graphics applications running on CPUs, two works have developed compiler solutions. Breternitz et al. [19] have developed a compiler to generate efficient code on a CPU for SIMD graphic workloads by extending the base ISA to SSE2. Liao et al. [58] have developed a framework that works with Brook [20] to perform aggressive data and computation transformations. Recently, Ryoo et al. [84, 83] have presented experimental studies on program performance on NVIDIA GPUs using CUDA; they do not use or develop a compiler framework for optimizing applications, but rather perform the optimizations manually. Ryoo et al. [85] have presented performance metrics to prune the optimization search space on a pareto-optimality basis. However, they manually generate the performance metrics data for each application they have studied.

5.8 Conclusions

In this Chapter, critical performance-influencing factors on GPUs were characterized and techniques were developed to address the issues, that include 1) generation of effective program transformations for GPUs that enable efficient global memory access, 2) determination of optimal padding factors for conflict-minimal data access from shared memory, and 3) model-driven empirical optimization approach to optimize values for system and program parameters. The effectiveness of the developed techniques was demonstrated with various kernels.
CHAPTER 6

END-TO-END C-TO-CUDA SYSTEM

CLooG [10, 27] is a powerful open-source state-of-the-art code generator that transforms a polyhedral representation of a program and affine scheduling constraints into concrete loop code. The Pluto compiler enables end-to-end automatic parallelization and locality optimization of affine programs for general-purpose multi-core targets [15, 16, 71]. Using CLooG and adapting Pluto, we develop an end-to-end automatic C-to-CUDA system using the compiler transformations discussed in Chapter 5.

Figure 6.1: The C-to-CUDA Code Generation Framework
6.1 Design of C-to-CUDA Generator

Before providing details on the various transformation aspects in our end-to-end C-to-CUDA code generator, we first outline the general steps involved in source-to-source code generation using a polyhedral compiler framework.

1. The input program is run through a scanner and parser that constructs an abstract syntax tree (AST) for the input program. From the AST, iteration space polytopes and array access functions are extracted.

2. Data dependences are analyzed and dependence polytopes are generated.

3. After analyzing the dependences, affine statement-wise transforms are determined. The affine transforms provide the new lexicographic ordering of the statements in the transformed program.

4. When tiling has to be performed, the affine statement-wise transforms are used as tiling hyperplanes to generate higher-dimensional statements domains (involving supernode iterators and intra-tile iterators).

5. The transformed statement polytopes along with the affine transformations are provided to a polyhedral code generator such as CLooG to generate transformed code.

The GPU architecture represents a multi-level parallel architecture. It has various memory units (with different access properties) that are at different proximity with respect to the chip (on-chip and off-chip) and very different access latencies. We now discuss the various issues that are addressed by our code generation system for generating effective CUDA code along the lines of the code generation process described above. There are several
publicly available polyhedral transformation frameworks and tools. We used the Pluto [71] 
polyhedral parallel tiling infrastructure and CLooG [10, 27], a state-of-the-art polyhedral 
code generator. Our tool chain is shown in Fig. 6.1.

1. One of the key optimizations is to determine effective access pattern for global (off-
chip) memory access. Pluto finds affine transforms that are (1) communication-
optimized, and (2) locality-optimized. At Step 3 of the code generation process 
(outlined above), our framework finds affine transforms that enable global memory 
coalescing in addition to being communication-optimized and locality-optimized.

2. Two levels of parallelism must be extracted to exploit parallelism at the thread block 
level and the thread level for GPUs. At Step 4, we use the affine transforms deter-
mined at Step 3 to find multi-level tiled statement domains and identify and extract 
parallelism.

3. A critical optimization for GPUs is the utilization of on-chip memories. It is benefi-
cial to move data reused in computation from off-chip memory to on-chip memory 
and move it back. At Step 4, our framework generates iteration space polytopes of 
data movement statements using polyhedral techniques, in addition to generating the 
transformed statement domains.

4. At Step 5, we use the CLooG polyhedral code generator to generate the target code 
structure. Suitable input, in the form of description of all statements (computation 
and data movement), together with their iteration spaces (as polytopes) as well as 
the transformations (as scheduling functions) specifying the new execution order for 
each statement instance, is input to the CLooG code generator. The union of all input
iteration space polytopes is scanned by CLooG according to the specified scheduling functions, in order to generate loop nests in the target program that execute the statement instances in this new execution order.

5. After Step 5, the AST of the generated parallel tiled code is post processed to generate compilable CUDA code. The post processing is primarily (1) to introduce thread-centricity in the parallel code, i.e., to add thread identifier and thread block identifier, and (2) add inter-thread and inter-thread-block synchronizations at appropriate execution points.

In the rest of this section, we provide details on the following three key aspects of the C-to-CUDA generator:

1. generation of multi-level tiled parallel code,

2. generation and placement of code to move data between on-chip and off-chip memories, and

3. generation of thread-centric parallel code.

### 6.1.1 Multi-level Parallel Tiled Code Generation

**Tiling Hyperplanes and Tiling Legality Condition.** In order to generate tiled code, Pluto finds affine transforms that satisfy the following tiling legality condition [16] in a multi-statement imperfectly nested program and use them as tiling hyperplanes which constitute the loops in the transformed program:

A set of one-dimensional affine transformation functions (one corresponding to each statement in a imperfectly nested multi-statement program), \{\phi_{s_1}, \phi_{s_2}, \ldots, \phi_{s_n}\}, represents a
valid tiling hyperplane if for each pair of dependent statement instances \((\vec{s}_i, \vec{s}_j)\)

\[
\phi_{s_j}(\vec{s}_j) - \phi_{s_i}(\vec{s}_i) \geq 0
\]

This condition guarantees that any inter- or intra-statement affine dependence is carried in the forward direction along the tiling hyperplane. Hence if a program is transformed using the affine transforms satisfying the above condition, then rectangular tiling is legal in the transformed program.

**Affine transformations for CUDA.** With CUDA, execution of a program involves distributing the computation across thread blocks and across threads within a thread block. For tiling at the outer level (at the level of thread blocks), our framework uses the affine transforms generated by Pluto. For finding tiling hyperplanes to generate tiled code at the inner level (at the level of threads), we modify Pluto to generate program transformations that enable interleaved access to global memory by threads in a thread block - this is necessary to facilitate coalesced global memory accesses that improve global memory access bandwidth. An approach to achieve this was discussed in Section 5.3.1.

**Extracting Parallel Loops.** The affine transformations may or may not result in parallel tile loops that are synchronization-free (*doall* loops). If doall loops exist in the tile space, they are used as parallel loops. However when no synchronization-free parallelism exists, parallel code generation needs additional processing. There may be one or more loops that carry dependences (*doacross* loops). Since the tiling legality condition assures that the dependences are always carried in the forward direction, pipelined parallelism with synchronization can be exploited in such cases.

If \(\{\phi^1, \phi^2, \ldots, \phi^n\}\) represent the doacross loops in the tile space, then the sum \(\phi^1 + \phi^2 + \cdots + \phi^n\) carries all dependences that are carried by each \(\phi^i\), \(1 \leq i \leq n\), and represents a legal
wavefront of tiles such that all tiles in the wavefront are parallel [46]. In other words, the set of loops are transformed (using a unimodular skewing transformation) as follows:

\[
\begin{pmatrix}
\phi'^1 \\
\phi'^2 \\
\phi'^3 \\
\vdots \\
\phi'^n
\end{pmatrix}
= 
\begin{pmatrix}
1 & 1 & \ldots & 1 & 1 \\
1 & 0 & \ldots & 0 & 0 \\
0 & 1 & \ldots & 0 & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & \ldots & 1 & 0
\end{pmatrix}
\begin{pmatrix}
\phi^1 \\
\phi^2 \\
\phi^3 \\
\vdots \\
\phi^n
\end{pmatrix}
\]

This ensures that \( \phi'^1 \) is sequential and \( \phi'^2, \phi'^3, \ldots, \phi'^n \) represent the parallel loops. This is the approach we employ to extract parallel loops at one level. A synchronization call to synchronize across the parallel units has to be placed at each iteration of the sequential loop. Handling the placement of synchronization calls is discussed later in the Section.

Pluto generates parallel code for general purpose multi-core architectures; it generates multi-level tiled code with parallelism only at the outer level. However for multi-level parallel architectures like GPGPUs, parallelism has to be extracted at multiple levels (two levels for CUDA - thread block level and thread level). Algorithm 6 provides details on the approach to generate multi-level tiled transformed statement domains (which are later fed to CLooG for code generation) along with the identification of parallel loops at thread block level and thread level.

**Using CLooG for Multi-level Tiled Code Generation.** As described earlier, CLooG scans a union of statement (iteration space) polyhedra using an optionally provided global lexicographic ordering specified through statement-wise scheduling functions or *scattering functions*, and generates loop nests in the target program that execute the statement instances in the new lexicographic order. CLooG does not include any data dependence information and hence the legality of scanning the statement polyhedra should be guaranteed by the user specifying the scattering functions. In our framework, the statement-wise
affine transforms provided as scattering functions to CLooG ensure effective and correct execution of the transformed program. Tiled code is generated using CLooG by specifying a modified higher dimensional statement domain for each statement and also specifying the scheduling or scattering functions (using the affine statement-wise transforms) to generate the correct ordering of inter-tile and intra-tile loops.

**Algorithm 6 Multi-level Parallel Tiled Code Generation**

**Input:** Set of statements - \( S \), Iteration Space Polytopes of all statements \( \mathcal{D}_s, s \in S \), Statement-wise affine transforms for each level \( k: \phi_{s}^{k_1}, \phi_{s}^{k_2}, \ldots, \phi_{s}^{k_n}, s \in S \), Tile sizes \( t_1, t_2, \ldots, t_n \) for each level

1: for each level do
2:   for each statement \( s \in S \) do
3:     for each transform \( \phi_{s} = C_{s}(\tilde{i}_s) \) do
4:       Increase the statement domain’s dimensionality so that the domain includes the supernode iterators
5:       Add constraints involving supernode iterators \( (\phi T_s) \) and tile sizes that represent a statement instance in a supernode \( t \times \phi T_s \leq C_{s}(\tilde{i}_s) \leq t \times \phi T_s + t - 1 \)
6:     Add scattering functions corresponding to supernodes. (The scattering functions are identity functions involving the supernode iterators)
7:     if level to be parallelized then
8:       if there exists doall loops then
9:         Mark them as parallel
10:     else
11:       Transform the first non-sequential loop \( \phi^i \) as follows: \( \phi^i \leftarrow \phi^i + \phi^{i+1} + \ldots + \phi^n \)
12:     Mark \( \phi^i \) as sequential and remaining subsequent loops in the band as parallel

**Output:** Transformed computation statement domains and scattering functions

### 6.1.2 Data Movement Between Off-chip and On-chip Memories

It is very important to reduce the accesses to off-chip memory and utilize the on-chip memories. Array references that have sufficient data reuse are good candidates to be copied
to shared memory since the repeated accesses would be made in low-latency on-chip memory instead of off-chip memory.

Given a program block or tile (having one or more statements), the data spaces accessed by array references within the block are determined using the iteration space of each statement and the array access function of each reference in each statement (as detailed in Chapter 3). The data spaces accessed by the read and write references of each array are represented as separate polytopes and are then used to determine the size of storage buffer needed to host the required data. The code for data movement is then generated by scanning the data space polytopes using CLooG. The loop structure of the data movement code (copy code) is a perfect nest of $n$ loops, where $n$ is the dimensionality of the accessed data space. By using a cyclic distribution of the innermost loop across threads of a warp, we enable interleaved access of global memory by threads. The data movement statements are of two types: (1) those that move data in to shared memory (further referred to as copy-in statements) and (2) those that move data out of shared memory (further referred to as copy-out statements).

The target code should encompass the data movement statements and computation statements in proper order so that the parallel code results in correct program execution. At the level of thread blocks, the data movement statements are placed such that they respect the following order: copy-in, computation, copy-out. We utilize the scattering functions in CLooG to achieve the proper placement of data movement and computation statements. The scattering functions provide a multi-level multi-dimensional schedule. The basic idea is to introduce an additional ‘constant’ dimension in the original schedule at the level of thread blocks to define the order of statements. Suppose that in the transformed program, the computation and data movement statements are defined at the outer level by
a schedule using the iterators \((c_1, c_2, \ldots, c_n)\). We modify the schedule of the copy-in, computation, copy-out statements as \((c_1, c_2, \ldots, c_n, 0)\), \((c_1, c_2, \ldots, c_n, 1)\), and \((c_1, c_2, \ldots, c_n, 2)\), respectively, to achieve the required order.

The algorithm to generate data movement statement domains and scattering functions to properly place data movement code in the target CUDA code structure is outlined in Algorithm 7.

**Algorithm 7 Generation and Placement of Data Movement Code**

**Input:** Set of statements \(- S\), Transformed Statement Domains of all statements \(D_s, s \in S\) from Algorithm 6, Affine array access functions

1. for each array \(A\) do
2. for all references of the array do
3. Find the data space accessed by the references
4. Partition the set of all data spaces into maximal disjoint sets such that each partition has a subset of data spaces each of which is non-overlapping with any data space in other partitions
5. For each partition, find the convex union of its data spaces and the bounding box of the convex union gives the storage buffer needed for the partition
6. for each statement \(s \in S\) do
7. for all read references of the array do
8. Find the data space accessed by the references and use them as domains of copy-in statements
9. Use “identity function” as scattering function
10. for all write references of the array do
11. Find the data space accessed by the references and use them as domains of copy-out statements
12. Use “identity function” as scattering function
13. Let the number of copy-in and copy-out statements be \(c\) and \(d\), respectively
14. Add a new dimension in all scattering functions (those of copy-in, computation, and copy-out statements) with just a constant value; the constant being 0 to \(c - 1\) for copy-in statements, \(c\) for computation statements, \(c + 1\) to \(c + d\) for copy-out statements

**Output:** Data movement statement domains and updated scattering functions

**Optimizing on-chip memory access.** In addition to handling data movement to the on-chip shared memory, we optimize the access of on-chip shared memory by minimizing
bank conflicts by padding the arrays copied into shared memory with suitable padding factor, as discussed in Section 5.4. Also, we determine the possibility to use on-chip constant memory and registers, as discussed in Section 5.3.3. We determine arrays that are read-only and whose access function does not vary with respect to the loop iterators corresponding to the parallel loops used for distributing computation across threads, and consider them as candidates for storing in constant memory. Similarly, arrays whose access functions vary only with respect to the loop iterators corresponding to the parallel loops are considered as candidates for storing in registers in each thread.

6.1.3 Syntactic Post-processing

The transformed multi-level tiled computation statement domains and data movement statement domains along with the scattering functions (generated by Algorithms 6 and 7) are fed to CLooG to generate multi-level tiled code. Syntactic post processing of the multi-level tiled code generated by CLooG is needed to generate a final compilable CUDA code. The primary tasks of the post processing are (1) to generate thread-centric code and (2) to place synchronization calls for correct parallel execution.

An important aspect of CUDA code generation is thread-centric code generation, i.e. generation of code where the computation is distributed across the threads in the system. A thread in the system is uniquely identified by a combination of its “thread block identifier” and “thread identifier” within the thread block. We take a syntactic approach to introduce thread-centricity in the parallel code generated using the above technique. The CLooG tool has its own AST representation called the CLAST. The CLAST generated for the parallel tiled code is parsed to introduce “thread block and thread identifiers” in the parallel loops (identified in Algorithm 6) such that the parallel tiles at the outer level are cyclically
distributed across the thread blocks and that at the inner level are cyclically distributed across the threads. The data movement code is also parsed to place “thread identifier” in the data movement loops.

CUDA offers a synchronization primitive to synchronize across threads within a thread block, but no built-in synchronization primitives to synchronize across thread blocks. We introduce a primitive through a code segment that uses a “single-writer multiple-reader” technique to achieve synchronization across thread blocks using the global memory space. It is necessary to place barrier synchronizations at each iteration of a sequential loop (if any) that precedes parallel loops, and at the end of data movement loops. It is done syntactically by modifying the CLAST. Algorithm 8 summarizes the CUDA code generation steps after applying Algorithms 6 and 7.

It should be noted that the tile sizes used for tiling are fixed at compile time and provided by the user. The code generated by our framework represents the number of threads and thread blocks as symbolic constants, which the user sets before the actual execution. In the case of wavefront parallelism, the framework restricts the user to set as many thread blocks as the number of streaming multiprocessors in the GPU chip to avoid deadlock during synchronization across thread blocks. Our framework also syntactically inserts an “unroll” pragma - #pragma unroll unroll_factor - which enables the CUDA compiler to perform inner loop unrolling. The approach discussed in Section 5.5 for selecting optimal tile sizes and unroll factors would be integrated into the system in future.
Algorithm 8 Parallel CUDA Code Generation

**Input:** Computation statement domains, Data movement statement domains, Scattering functions

1: Feed the computation and data movement statement domains and scattering functions to CLooG to generate CLAST
2: Parse CLAST to change the lower bounds and loop increments of (outer and inner level) parallel loops to make them thread-centric
3: Parse CLAST to change the lower bounds and loop increments of data movement loops to make them thread-centric
4: Place barrier synchronization at each iteration of sequential loop (if any) that precedes parallel loops, and at the end of data movement loops
5: Print the modified CLAST to generate CUDA code

**Output:** Multi-level parallel tiled CUDA code with data movement

### 6.2 Experimental Results

Experiments were conducted on an NVIDIA GeForce 8800 GTX GPU device that was connected to a 2.13 GHz Intel Core2 Duo processor through a 16-x PCI Express bus. We used CUDA version 2.1 for our experiments.

The multi-core system used for our experiments was a quad-core Intel Core 2 Quad Q6600 CPU clocked at 2.4 GHz (1066 MHz FSB) with a 32 KB L1 D cache, 8MB of L2 cache (4MB shared per core pair), and 2 GB of DDR2-667 RAM, running Linux kernel version 2.6.22 (x86-64). ICC 10.x was the primary compiler used to compile the code on the multi-core system; it was run with -fast -funroll-loops (-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.

We evaluated the effectiveness of the CUDA code generated by our framework using seven benchmarks. Where available, we compared the performance of the automatically generated CUDA code with hand-tuned CUDA code. We also compared the performance of the automatically generated code on the GPU with the performance of C code optimized
for (t1=0; t1<VOLY; t1++) {
    for (t2=0; t2<VOLX; t2++) {
        for (t3=0; t3<NATOMS; t3++) {
            energy[zDim*VOLX*VOLY + t1*VOLX + t2] =
            atoms[3+4*t3]/ ... atoms[2+4*t3] ...
            atoms[1+4*t3] ... atoms[4*t3];
        }
    }
} 

Figure 6.2: Original code structure for Coulombic Potential (cp) benchmark

by icc, on the Intel multi-core CPU. All GPU performance data reported below includes all
data transfer time to move data between the host CPU and the GPU device.

6.2.1 Coulombic Potential (cp)

This benchmark is used for the computation of electric potential in a volume contain-
ing point charges. It is one of the codes in the parboil benchmark suite from UIUC [69].
Fig. 6.3 presents the performance data - performance of the generated CUDA code with
different optimizations is compared with the hand-tuned code from the parboil benchmark
suite and icc optimized C code. The CUDA code generated by our framework performs
better than the optimized version on general-purpose multi-core system. The performance
of the code generated by turning on all optimizations is very close to that of the hand-tuned
code. In addition to extracting “doall” parallelism across threads and thread blocks, the
code has optimized off-chip access in one of the two ways - (1) utilizing shared memory
or (2) utilizing constant memory. Fig. 6.3 shows the performance measurements for both
the cases and it can be seen that the performance when constant memory is used is signif-
icantly higher than that when shared memory is used. This is because the use of constant
memory significantly reduces global memory traffic in comparison to accessing data after moving from global memory to shared memory. Inner loop unrolling was performed using NVIDIA’s `#pragma unroll` option.

![Figure 6.3: Performance of cp benchmark](image)

Figures 6.2 and 6.8 illustrate the CUDA code generation. Fig. 6.2 shows the structure of sequential code (along with the array accesses) for Coulombic Potential (cp) benchmark. Fig. 6.8 shows the structure of two-level tiled parallel code that is thread-centric where the parallelism is across thread blocks at the outer level and across threads at the inner level (Note the modified lower bounds and loop increments of parallel loops). Fig. 6.8 also shows the proper placement of data movement and computation statements.
6.2.2 N-Body Simulation (nbody)

N-body simulation is an important computation that arises in many computational science applications. It approximates the evolution of a system of bodies in which each body continuously interacts with every other body. The CUDA code generated by our framework performs much better than the optimized version on general-purpose multi-core system and performs very comparably to the hand-tuned CUDA code, as illustrated in Table 6.1. The code generated by our framework exploited “doall” parallelism across threads and thread blocks. It effectively moved data from arrays that exhibited data reuse from global memory to shared memory, thereby enabling coalesced global memory access and also reduction in off-chip memory access latency, by exploiting data reuse in on-chip shared memory. Further, inner loop unrolling was performed using NVIDIA’s #pragma unroll option. Fig. 6.4 depicts incremental performance improvement when different optimizations are applied. The importance of shared memory utilization and inner loop unrolling (to reduce loop overhead and dynamic loop instruction count) are illustrated by this benchmark.

The hand-tuned version was taken from the NVIDIA CUDA SDK, the code being based on the article in [55]. The code generated by our framework represents the number of threads and thread blocks as symbolic constants, which the user sets before the actual execution.

<table>
<thead>
<tr>
<th>N</th>
<th>Auto-CUDA</th>
<th>Hand-tuned</th>
<th>icc</th>
</tr>
</thead>
<tbody>
<tr>
<td>2048</td>
<td>129.67</td>
<td>157.34</td>
<td>1.00</td>
</tr>
<tr>
<td>4096</td>
<td>187.41</td>
<td>182.31</td>
<td>1.10</td>
</tr>
<tr>
<td>8192</td>
<td>191.81</td>
<td>188.78</td>
<td>1.42</td>
</tr>
<tr>
<td>16384</td>
<td>192.45</td>
<td>198.43</td>
<td>1.47</td>
</tr>
<tr>
<td>32768</td>
<td>192.91</td>
<td>200.35</td>
<td>1.50</td>
</tr>
</tbody>
</table>

Table 6.1: Performance of nbody benchmark (in GFLOPS)

110
Figure 6.4: Performance variation of nbody benchmark w.r.t Optimizations

6.2.3 MRI kernels

We employed our framework to generate code for two kernels used in Magnetic Resonance Imaging, MRI-Q and MRI-FHD [69]. Both the kernels involve two computational blocks such that data computed in the first computation block is used as “read-only” data in the second computational block. The hand-tuned code from parboil optimizes the two computational blocks independently and executes them as separate GPU kernels. We used our framework to generate two versions of code for each of the two MRI kernels - version (1) in which CUDA code is generated independently for the two computational blocks (first block pre-computes data for second block) and version (2) in which unified CUDA code is generated for both blocks.
<table>
<thead>
<tr>
<th>N</th>
<th>Auto CUDA (2) no unroll</th>
<th>Auto CUDA (2) unroll</th>
<th>Auto CUDA (1) no unroll</th>
<th>Auto CUDA (1) unroll</th>
<th>Hand tuned</th>
<th>icc</th>
</tr>
</thead>
<tbody>
<tr>
<td>32768</td>
<td>87.11</td>
<td>122.19</td>
<td>137.1</td>
<td>176.50</td>
<td>178.98</td>
<td>0.91</td>
</tr>
<tr>
<td>65536</td>
<td>88.27</td>
<td>121.87</td>
<td>141.7</td>
<td>179.32</td>
<td>179.12</td>
<td>1.14</td>
</tr>
<tr>
<td>131072</td>
<td>88.53</td>
<td>123.11</td>
<td>142.3</td>
<td>181.23</td>
<td>179.32</td>
<td>1.14</td>
</tr>
<tr>
<td>262144</td>
<td>89.16</td>
<td>122.12</td>
<td>142.6</td>
<td>183.32</td>
<td>180.91</td>
<td>1.15</td>
</tr>
</tbody>
</table>

Table 6.2: Performance of MRI-Q (in GFLOPS)

<table>
<thead>
<tr>
<th>N</th>
<th>Auto CUDA (2) no unroll</th>
<th>Auto CUDA (2) unroll</th>
<th>Auto CUDA (1) no unroll</th>
<th>Auto CUDA (1) unroll</th>
<th>Hand tuned</th>
<th>icc</th>
</tr>
</thead>
<tbody>
<tr>
<td>32768</td>
<td>57.91</td>
<td>90.91</td>
<td>112.52</td>
<td>142.52</td>
<td>143.11</td>
<td>1.37</td>
</tr>
<tr>
<td>65536</td>
<td>61.27</td>
<td>91.2</td>
<td>116.12</td>
<td>143.15</td>
<td>142.27</td>
<td>1.68</td>
</tr>
<tr>
<td>131072</td>
<td>62.13</td>
<td>91.6</td>
<td>116.22</td>
<td>144.21</td>
<td>144.39</td>
<td>2.19</td>
</tr>
<tr>
<td>262144</td>
<td>62.67</td>
<td>91.52</td>
<td>116.67</td>
<td>142.61</td>
<td>144.43</td>
<td>2.21</td>
</tr>
</tbody>
</table>

Table 6.3: Performance of MRI-FHD (in GFLOPS)

Tables 6.2 and 6.3 summarize the performance measures of the code versions of MRI-Q and MRI-FHD, respectively. The code version (1) generated as two separate GPU kernels outperforms the code version (2) generated as single GPU kernel because of the fact that in version (1) the data precomputed in the first GPU kernel is stored in constant memory and accessed in the second kernel. However both the versions identified various data arrays as candidates for constant memory and thereby optimized off-chip memory access. The code version (1) generated by our framework performs as well as the hand-tuned version.
6.2.4 Stencil Computation Kernels

We used two stencil computation kernels, 2D Jacobi and 2D Finite Difference Time Domain (FDTD). The code generated using our framework performs better than the optimized version on the Intel multi-core system, as illustrated in Figures 6.5 and 6.6. For these two kernels, we were unable to find any hand-tuned CUDA code to compare against. The code generated by our framework exploits parallelism across threads and thread blocks and effectively utilizes shared memory and exploits data reuse. The parallel execution of stencil computations is characterized by synchronization overhead at every time step across the processors. This overhead is particularly costly in GPUs where the thread blocks have to
synchronize using the slow off-chip memory. This is the reason for the lower absolute performance of these kernels on GPUs, relative to the previous benchmarks. The performance of the stencil kernels is very low for smaller problem sizes for the same reason.

6.2.5 Gauss Seidel Successive Over Relaxation

The Gauss Seidel benchmark illustrates the effect of exploiting wavefront or pipelined parallelism on GPUs. We achieve better performance than the optimized version on multicore system, as illustrated in Fig. 6.7. However, the absolute performance is rather low because of (1) low processor utilization during the starting and draining of pipeline and (2) synchronization overhead across thread blocks at every time step.
6.3 Related Work

Recently, Lee et al. [56] developed a compiler framework for automatic translation from OpenMP to CUDA. It handles both regular and irregular programs which are parallelized using OpenMP primitives. Work sharing constructs in OpenMP are translated into distribution of work across threads in CUDA. However it is not clear if and how data is identified for movement between global memory and shared memory.

Very recently, Liu et al. [63] developed a GPU adaptive optimization framework (G-ADAPT) for automatic prediction of near-optimal configuration of parameters that affect GPU performance. They take unoptimized CUDA code as input and traverse an optimization space search to determine optimal parameters to transform the unoptimized input
CUDA code into an optimized CUDA code. Using our framework, a user can automatically generate CUDA code for any arbitrary input affine C code, hand-parallelization of which is very cumbersome in many cases. The user may then use G-ADAPT to further tune the CUDA code generated from our system.

6.4 Conclusions

In this Chapter, we have presented the design of an automatic source-to-source transformation framework that can take an arbitrarily nested affine input C program and generate an efficient CUDA program. Experimental results illustrate the performance improvements achieved using the framework. Planned further work includes incorporation of an iterative/empirical tuning system to search over a space of various performance influencing parameters such as tile sizes, number of thread blocks, number of threads, etc. We also plan to incorporate the dynamic scheduling technique for distributing computation, described in Chapter 4, for achieving enhanced load balanced execution on GPUs.
```c
int by = blockIdx.y;
int bx = blockIdx.x;
int ty = threadIdx.y;
int tx = threadIdx.x;

int t1,t2,t3,t4,t5,t6;
// Parallel loops distributed across thread blocks
// Loops modified syntactically for thread block identifiers
for (t1=by; t1<=floord(VOLY-1,16); t1+=NBLKSY) {
  for (t2=bx; t2<=floord(VOLX-1,16); t2+=NBLKSX) {
    for (t3=0;t3<=NATOMS-1;t3+=256) {
      // Data movement code
      __shared__ float atomsS[1024];
      for (t6=4*t3+THREADY*NTHRDSX+THREADX;
       t6<=min(4*NATOMS-1,4*t3+1023);
       t6+=NTHRDSX*NTHRDSY)
        atomsS[t6-4*t3] = atoms[t6];
      __syncthreads();
      // Parallel loops distributed across threads
      // Loops modified syntactically for thread identifiers
      for (t4=max(0,16*t1)+ty;
       t4<=min(VOLY-1,16*t1+15);t4+=NTHRDSY) {
        for (t5=max(0,16*t2)+tx;
         t5<=min(VOLX-1,16*t2+15);t5+=NTHRDSX) {
          ...
          // Computation code
          for (t6=t3; t6<=min(NATOMS-1,t3+255);
          t6++) {
            energy[zDim*VOLX*VOLY + t4*VOLX + t5] =
            atomsS[3+4*t6-4*t3]/ ... atomsS[2+4*t6-4*t3] ...
            atomsS[1+4*t6-4*t3] ... atomsS[4*t6-4*t3];
          }
        }
      }
    }
  }
}

Figure 6.8: Parallel tiled code structure (with data movement) for cp benchmark
```
CHAPTER 7

CONCLUSIONS AND FUTURE WORK

With the widespread emergence of multi-core processors, the community faces an enormous crisis in developing high-performance applications for modern parallel architectures. The reality today is that existing and new applications must be changed to make them multi-threaded if they are to experience any performance benefits from newer generations of processors because single-core performance has saturated and the increased power of newer chips comes almost entirely from multi-core parallelism. The emergence of GPUs as cost-effective and powerful processors for general-purpose computing has raised great interest in their use for many scientific and engineering applications, but developing high-performance applications for GPUs is even more complicated than programming general-purpose multicore processors. In this dissertation, we have worked towards addressing this broad challenge faced today and developed compile-time and run-time techniques that are aimed at easing the development of high-performance applications for multi-core and many-core systems. Two significant concerns in modern multi-core processors are - (1) to improve memory performance according to the characteristics of the memory access model of the architecture and (2) to exploit multi-core parallelism. We have developed techniques to address these concerns on general purpose multi-core CPUs and GPUs.
7.1 Optimizations for Irregular Applications

It is not only an interesting direction, but also a pressing need for future, to extend the focus of our compile-time and run-time optimizations for irregular and non-structured applications. Various real-time applications involving intensive irregular computations (computations in which data accesses are not regular and may be dynamic (e.g. indirect indexing)) are expected to be critical applications in future. Irregular codes are common in various areas like Computational Fluid Dynamics (CFD), Molecular Biology, Molecular Dynamics, etc. They are widely common in clusters of problems such as those that involve linear algebra operations on sparse matrices (for e.g., a variety of iterative methods for solving linear systems), and partial differential equation (PDE) solvers.

It is an interesting research problem to develop compile-time and run-time optimizations for irregular applications to maximize memory performance and exploit parallel computation power on modern parallel architectures. There are significant challenges to address in such applications in data dependence analysis, finding good program transformations, and generation of transformed code. It would be interesting to explore on an inspector-executor kind of technique to analyze the data access pattern and computation pattern, and optimize the memory access based on the memory access model of the target architecture, through possible techniques like changing the data access order, data reordering (optimizing the data layout), or computation reordering [64].

7.1.1 Overview of Issues for Irregular Applications on GPUs

Some of the prominent issues that would arise while optimizing irregular applications on GPUs are as follows.
• Load imbalance on processor cores: Irregular applications may result in load imbalance on the processor cores in a multiprocessor unit on GPUs. This would affect the SIMD model of execution on the processor cores, as the execution paths of threads (belonging to the same warp) mapped to the processor cores would become divergent.

• Inefficient GPU DRAM memory access: Due to irregular data accesses, there would be inefficient memory access from GPU DRAM (global memory), due to possible non-coalesced memory accesses.

• Inefficient scratchpad memory access: Irregular data accesses may result in complex data access pattern across threads, thereby complicating the task of handling bank conflicts in scratchpads (shared memory).

7.2 Heterogeneous Computing Platforms

Heterogeneous parallel processing platforms that include a mix of multi-core CPUs, GPUs and other processors like the Cell B.E. are becoming more prominent. This has prompted an industry-wide consortium to come together and create the OpenCL language as an extension to C. OpenCL (Open Computing Language) [67] is an open standard for general purpose parallel programming across CPUs, GPUs and other processors, that enables the user to utilize the power of the heterogeneous parallel processors. Although OpenCL presents a through a high-performance, portable abstraction for parallel computing, because of the “low level close-to-the-metal” programming interface, OpenCL programs will be much more complicated than sequential programs (and also more complicated than OpenMP). This obviates the need for developing a compiler/run-time system
that enables automatic/semi-automatic generation of OpenCL code from annotated C programs provided by the user. Our C-to-CUDA system can be naturally extended to develop high-performance applications for GPUs using OpenCL. It is an interesting problem to develop a system for automatic OpenCL generation for heterogeneous processors.


BIBLIOGRAPHY


