Automatic Parallelization of Tiled Stencil Loop Nests on GPUs

by

Peng Di

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN THE SCHOOL

OF

COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF
NEW SOUTH WALES

SCIENTIA

MANU ET MENTE

SYDNEY · AUSTRALIA

Thursday 19th December, 2013

All rights reserved. This work may not be

reproduced in whole or in part, by photocopy

or other means, without the permission of the author.

© Peng Di 2013

PLEASE TYPE

THE UNIVERSITY OF NEW SOUTH WALES
Thesis/Dissertation Sheet

Surname or Family name: **DI**

First name: **PENG** Other name/s:

Abbreviation for degree as given in the University calendar: **PhD**

School: **School of Computer Science and Engineering** Faculty: **Faculty of Engineering**

Title: **Automatic Parallelization of Tiled Stencil Loop Nests on GPUs**

Abstract 350 words maximum: (PLEASE TYPE)

This thesis attempts to design and implement a compiler framework based on the polyhedral model. The compiler automatically parallelizes loop nests; especially stencil kernels, into efficient GPU code by loop tiling transformations which the polyhedral model describes. To enhance parallel performance, we introduce three practically efficient techniques to process different types of loop nests. The experimental results of our compiler framework have demonstrated that these advanced techniques can outperform previous approaches.

Firstly, we aim to find efficient tiling transformations without violating data dependences. How to select a tile's shape and size is an open issue that is performance-critical and influenced by GPU's hardware constraints.

We propose an approach to determine the tile shapes out of consideration for improving two-level parallelism of GPUs. The new approach finds appropriate tiling hyperplanes by embedding parallelism-enhancing constraints into the polyhedral model to maximize intra-tile, i.e., intra-SM parallelism. This improves the load balance among the streaming processors (SPs), which execute a wavefront of loop iterations within a tile. We eliminate parallelism-hindering false dependences to optimize inter-tile, i.e., inter-SM parallelism. This improves the load balance among the streaming multiprocessors (SMs), which execute a wavefront of tiles.

Furthermore, to avoid combinatorial explosion of tile size's configurations, we present a model-driven approach to automating tile size selection that is performance-critical for loop tiling transformations, especially for DOACROSS loop nests. Our tile size selection model accurately estimates the execution times of tiled loop nests running on GPUs. The selected tile sizes lead to the performance results that are close to the best observed for a range of problem sizes tested.

Finally, to address the difficulty and low-performance of parallelizing widely used SOR stencil loop nests, we present a new tiled parallel SOR method, called MLSOR, which admits more efficient data-parallel SIMD execution on GPUs. Unlike the previous two approaches that are dependence-preserving, the basic idea is to algorithmically restructure a stencil kernel based on a non-dependence-preserving parallelization scheme to avoid pipelining for higher parallelism. The new approach can be implemented in compilers through a pattern matching pass to optimize SOR-like DOACROSS loop nests on GPUs.

Declaration relating to disposition of project thesis/dissertation

I hereby grant to the University of New South Wales or its agents the right to archive and to make available my thesis or dissertation in whole or in part in the University libraries in all forms of media, now or here after known, subject to the provisions of the Copyright Act 1968. I retain all property rights, such as patent rights. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

I also authorise University Microfilms to use the 350 word abstract of my thesis in Dissertation Abstracts International (this is applicable to doctoral theses only).

[illegible]	Liang Tang	Dec 19, 2013
Signature	Witness	Date

The University recognises that there may be exceptional circumstances requiring restrictions on copying or conditions on use. Requests for restriction for a period of up to 2 years must be made in writing. Requests for a longer period of restriction may be considered in exceptional circumstances and require the approval of the Dean of Graduate Research.

FOR OFFICE USE ONLY Date of completion of requirements for Award:

THIS SHEET IS TO BE GLUED TO THE INSIDE FRONT COVER OF THE THESIS

COPYRIGHT STATEMENT

'I hereby grant the University of New South Wales or its agents the right to archive and to make available my thesis or dissertation in whole or part in the University libraries in all forms of media, now or here after known, subject to the provisions of the Copyright Act 1968. I retain all proprietary rights, such as patent rights. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.
I also authorise University Microfilms to use the 350 word abstract of my thesis in Dissertation Abstract International (this is applicable to doctoral theses only).
I have either used no substantial portions of copyright material in my thesis or I have obtained permission to use copyright material; where permission has not been granted I have applied/will apply for a partial restriction of the digital copy of my thesis or dissertation.'

Signed ……[illegible]……

Date ……Dec 19,2013……

AUTHENTICITY STATEMENT

'I certify that the Library deposit digital copy is a direct equivalent of the final officially approved version of my thesis. No emendation of content has occurred and if there are any minor variations in formatting, they are the result of the conversion to digital format.'

Signed ……[illegible]……

Date ……Dec 19, 2013……

ORIGINALITY STATEMENT

'I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.'

Signed ………[illegible]…………..............

Date ………Dec 19, 2013…………..............

Abstract

Parallelization of loop nests continues to drive much of the ongoing research in parallel computing. Stencil loop nests are widely used in scientific and engineering applications. Their efficient implementations on distributed systems, such as *graphics processing unit (GPU)*, are becoming increasingly important, as GPUs have recently emerged as powerful fine-grained parallel co-processors for general-purpose computing. However, manual development of high-performance GPU kernels can be time-consuming and error-prone. Among several alternative approaches, automatic parallelization is promising yet challenging. For implementing automation, the polyhedral model is introduced into compilers as a powerful mathematical representation of loop nests that can be used to efficiently parallelize loop nests for GPUs.

This thesis attempts to design and implement a compiler framework based on the polyhedral model. The compiler automatically parallelizes loop nests; especially stencil kernels, into efficient GPU code by loop tiling transformations which the polyhedral model describes. To enhance parallel performance, we introduce three practically efficient techniques to process different types of loop nests. The experimental results of our compiler framework have demonstrated that these advanced techniques can outperform previous approaches.

Firstly, we aim to find efficient tiling transformations without violating data dependences. How to select a tile's shape and size is an open issue that is performance-critical and influenced by GPU's hardware constraints.

We propose an approach to determine the tile shapes out of consideration for improving two-level parallelism of GPUs. The new approach finds appropriate tiling hyperplanes by embedding parallelism-enhancing constraints into the polyhedral model to maximize intra-tile, i.e., intra-SM parallelism. This improves the load balance among the *streaming processors (SPs)*, which execute a wavefront of loop iterations within a tile. We eliminate parallelism-hindering false dependences to optimize inter-tile, i.e., inter-SM parallelism. This improves the load balance among the *streaming multiprocessors (SMs)*, which execute a wavefront of tiles.

Furthermore, to avoid combinatorial explosion of tile size's configurations, we present a model-driven approach to automating tile size selection that is performance-critical for loop tiling transformations, especially for DOACROSS loop nests. Our tile size selection model accurately estimates the execution times of tiled loop nests running on GPUs. The selected tile sizes lead to the performance results that are close to the best observed for a range of problem sizes tested.

In the end, to address the difficulty and low-performance of parallelizing widely used SOR stencil loop nests, we present a new tiled parallel SOR method, called ML-SOR, which admits more efficient data-parallel SIMD execution on GPUs. Unlike the previous two approaches that are dependence-preserving, the basic idea is to algorithmically restructure a stencil kernel based on a non-dependence-preserving parallelization scheme to avoid pipelining for higher parallelism. Despite its relatively slower convergence, the efficiency of our method is demonstrated against the existing method RBSOR by making a better balance between data reuse and parallelism and trading off convergence rate for SIMD parallelism. The new approach can be implemented in compilers through a pattern matching pass to optimize

SOR-like DOACROSS loop nests on GPUs.

Publications

- **Peng Di**, Jingling Xue, Changjun Hu and Jingjing Zhou. A Cache-Effcient Parallel Gauss-Seidel Solver with Alternating Tiling. *In the 15th International Conference on Parallel and Distributed Systems (ICPADS'09)*, pages 244-251, Shenzhen, China, 2009.

- **Peng Di**, Qing Wan, Xuemeng Zhang, Hui Wu and Jingling Xue. Toward Harnessing DOACROSS Parallelism for Multi-GPGPUs. *In the 2010 International Conference on Parallel Processing (ICPP'10)*, pages 40-50, San Diego, USA, 2010.

- **Peng Di** and Jingling Xue. Model-Driven Tile Size Selection for DOACROSS Loops on GPUs. *In the 17th International European Conference on Parallel and Distributed Computing (Euro-Par'11)*, pages 401-412, Bordeaux, France, 2011.

- **Peng Di**, Ding Ye, Yu Su, Yulei Sui and Jingling Xue. Automatic Parallelization of Tiled Loop Nests with Enhanced Fine-Grained Parallelism on GPUs. *In the 2012 International Conference on Parallel Processing (ICPP'12)*, Pittsburgh, USA, 2012.

- **Peng Di**, Hui Wu, Jingling Xue, Feng Wang and Canqun Yang. Parallelizing SOR for GPGPUs Using Alternate Loop Tiling. *Parallel Computing*, volume

38, issues 6-7, pages 310-328, June-July 2012.

- Jilin Zhang, Liting Zhu, Jie Mao, Jian Wan and **Peng Di**. An Efficient Parallel Implementation for Three-dimensional Incompressible Pipe Flow based on SIMPLE. *In the 12th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid'12)*, pages 660-667, Ottawa, Canada, 2012.

Acknowledgements

I would like to gratefully acknowledge the guidance of my supervisor, Prof. Jingling Xue. During my PhD study, He has spent an incredible amount of time and energy on guiding my research. I have especially benefited from his insight, breadth of knowledge, and continuous encouragement.

I would like to thank Xinwei Xie, Qing Wan and Ding Ye for their collaboration on the my work. They provided helpful thoughts towards my research topic.

Finally, I dedicate this thesis to my parents, my wife and my child. Their unconditional love and support during my PhD study help me concentrate on the research work and finalize my PhD on time.

Contents

List of Figures

List of Tables

List of Algorithms

List of Theorems and Definitions

Chapter 1

Introduction

This thesis attempts to design and implement a compiler framework for automatically tiling and parallelizing loop nests, especially stencil kernels, into efficient CUDA code. To achieve high performance, we introduce three practically efficient techniques into the framework to process different types of loop nests in the polyhedral model. Each technique proposes or explores a *novel* approach to solve the corresponding problem. Through implementation in our compiler framework, these advanced techniques can outperform previous approaches.

1.1 Background

Stencil Computations Iterative stencils are important computation-intensive kernels that are widely adopted in scientific and engineering applications. For example, for regular matrices, the widely used Jacobi method is implemented as a stencil loop. In addition, image-processing filters, computational electromagnetic and numerical analyses are also implemented as stencil loops.

The outermost loop in a stencil kernel is called a time step, as stencil programs are often used to simulate the evolution of physical systems over time. At each time step, the inner loop nests operate on a multi-dimensional array, and compute each element of an array as a function of neighboring locations in the array. The choice of the neighboring elements follows a fixed pattern. According to types of inner loops, stencil loops fall into two categories: DOALL loop nests that are dependence-free in the inner loops, and DOACROSS loop nests that exhibit loop-carried dependences in the inner loops (see Chapter 3).

To achieve high performance, inner loop nests of the iterative stencil loops are usually tiled, and each tile is assigned to a processing node for parallel execution. Data locality, communication and parallelism are the main issues when optimizing and parallelizing stencil programs. Due to different characteristics of DOALL and DOACROSS stencils, tiling strategies are different. Unlike DOALL loops, DOACROSS loops must be skewed first to ensure that subsequent tiling transformations preserve the loop-carried dependences. In spite of past efforts on optimizing stencil kernels, how to address a variety of performance factors for different parallel architectures is still challenging.

Polyhedral Model Stencil computations are inherently loop nest programs. Loop transformations and optimizations are important to parallelize and accelerate stencil computations. As a well-studied mathematic framework, the polyhedral model provides a powerful loop representation that reasons about loop transformations by mapping each statement as an integer point in a multi-dimensional space.

With such a representation for each statement and a precise dependence analyzer among these statements, complicated loop transformations can be proved in the completely mathematical machinery, including linear algebra and integer linear programming. Based on these transformations, compilers can generate new parallel loops in reordered execution sequences with improved locality. Therefore, the polyhedral model is practical in compilers as a flexible and powerful method to compose and process transformations.

The polyhedral model is quite expressive to represent a potentially intricate and long sequence of many loop transformations, such as reversal, skewing, interchange, peeling, shifting, fusion, and fission. One of these loop transformations, called *tiling* [124, 126], is a classic and effective optimization for loop nests. Numerous optimizations based on the tiling of iteration spaces have been proposed for improving data locality or exploiting parallelism. The polyhedral model represents a loop tiling transformation by a set of hyperplanes (see Chapter 3). Compilers can utilize such mathematical model to automatically generate tiled parallel code.

However when tiling loop nests by a set of hyperplanes, an important NP-hard issue is the choice of tiling from a huge space of valid solutions [86]. Different tiling hyperplanes significantly influence the performance of the generated parallel code. To enable appropriate tiling hyperplanes for selection, pioneering work has focused on pruning the search space and optimizing hyperplane generation strategies. These strategies are perhaps not appropriate to a new parallel platform, such as *graphics processing unit (GPU)*, as they are strongly related to architectural constraints.

Automatic Parallelization for GPUs As GPUs have recently emerged as powerful fine-grained parallel co-processors for general-purpose computing, GPUs are becoming appropriate parallel architectures for stencils. However, manual development of high-performance *compute unified device architecture (CUDA)* kernels for GPUs can be time-consuming and error-prone.

Among several candidate approaches to address the parallel programming problem, automatic parallelization is promising yet challenging. Automatic parallelization is the process to automatically convert a sequential program to a parallel version which can be executed in parallel. This process does not require any effort of programmers in parallelization. Automatic parallelization is typically implemented at a high level *intermediate representation (IR)* phase, in which the needed programs' information is available. The output is a race-free deterministic program that obtains the same results as the original sequential program.

There are some works to automatically parallelize loop nests by using the polyhedral model. Nevertheless, the performance is afflicted with the design defects of these tools, such as lacking appropriate loop transformations for two levels of fine-grained parallelism; lacking practical and scalable models for complex tile size selection and lacking specific optimization of loop nests with loop-carried dependences.

1.2 Contributions

1.2.1 Compiler Framework for Automatic Tiling

We have implemented our compiler framework for automatically tiling and parallelizing loop nests, especially stencil kernels, into efficient CUDA code. This framework consists of the Clan polyhedral representation extractor, the Candl analyzer for dependences in loop nests, the PLUTO polyhedral parallel tiling infrastructure and the CLooG code generator. The details of the entire tool-chain are shown in Section 3.4. To achieve better performance, we implemented several practically efficient techniques to process different loop nests, such as optimizing two-level parallelism, selecting appropriate tile sizes and generating MLSOR kernels by pattern matching. The former two techniques attempt to find optimized shapes and sizes of tiling transformations respectively. The last technique is designed to improve the certain kind of widely used stencil kernels. Unlike the former two techniques, the latter algorithmically breaks data dependences among iterations.

1.2.2 Tiling with Optimized Two-level Parallelism of GPUs

We firstly improve two levels of wavefront parallelism that are decided by GPUs' hardware features. The new approach finds appropriate tiling hyperplanes by embedding parallelism-enhancing constraints in the polyhedral model to maximize intra-tile, i.e., intra-SM parallelism. This improves the load balance among the *streaming processors (SPs)*, which execute a wavefront of loop iterations within a tile. We eliminate parallelism-hindering false dependences to maximize inter-tile, i.e., inter-SM parallelism. This improves the load balance among the *streaming multiprocessors (SMs)*, which execute a wavefront of tiles.

Our approach has been implemented in our framework and validated using eight benchmarks on two different NVIDIA GPUs (C1060 and C2050). Compared to PLUTO, our approach achieves 2 – 5.5X speedups across the benchmarks. Compared to highly hand-optimized 1-D (1-dimensional) Jacobi (3 points), 2-D Jacobi (5 points), 3-D Jacobi (7 points) and 3-D Jacobi (27 points), our speedups, 1.17X, 1.41X, 0.97X and 0.87X with an average of 1.10X on C1060 and 1.24X, 1.20X, 0.86X and 0.95X with an average of 1.06X on C2050, are competitive.

1.2.3 Model-driven Tile Size Selection

In tiling transformations, tile size selection is performance-critical. Due to loop-carried dependences, tile size selection for DOACROSS loop nests becomes more complex than DOALL loop nests on GPUs. We present a model for estimating the execution times of tiled loop nests, especially for DOACROSS loop nests, running on GPUs and implement it into our compiler framework to automate tile size selection. We evaluate the accuracy of our model using representative 1-D, 2-D and 3-D SOR solvers and show that the tile sizes selected lead to the performances close to the best observed for a range of problem sizes tested.

1.2.4 MLSOR Algorithm for DOACROSS Loop Nests

Widely used SOR stencil loop nests are difficult to be parallelized, owing to the existence of DOACROSS data dependences. We present a new parallel SOR method, called MLSOR, which admits more efficient data-parallel SIMD execution than the traditional red-black SOR (denoted RBSOR) on GPUs. Our approach can be implemented in compilers as a pattern-guided model to significantly enhance the SOR-like DOACROSS loop nests on GPUs.

This solution is obtained non-conventionally, by starting from a K-layer SOR method and then parallelizing it by applying a non-dependence-preserving scheme consisting of a new domain decomposition technique followed by an alternate tiling technique. Despite its relatively slower convergence, our new method outperforms RBSOR by making a better balance between data reuse and parallelism and by trading off convergence rate for SIMD parallelism.

1.3 Thesis Organization

The rest of the thesis is organized as follows.

In Chapter 2, we provide a comprehensive and detailed survey of existing works related to the problems researched in this thesis. In Chapter 3, after providing the brief mathematical background for the polyhedral model and loop transformations, we introduce our compiler framework that performs automatic C-to-CUDA parallelization with optimizing techniques proposed in this thesis.

In Chapters 4, 5 and 6, we separately present the three novel techniques built in our compiler framework to enhance performance of the tiled code. Chapter 4 introduces an approach to find loop tiling hyperplanes with optimized parallelism. This technique determines tile shapes in the loop transformations. Chapter 5 describes a model-driven approach to automating tile size selection for GPUs. This technique determines the tile size in the loop transformations. Chapter 6 presents a new algorithm MLSOR to efficiently optimize common pattern loop nests in stencil computations.

Chapter 7 concludes the thesis and discusses future research.

Chapter 2

Related Work

In this chapter, we explore the literature related to the content studied in this thesis. We start with an overview of stencil computations in Section 2.1, followed by a brief survey of tiling transformations in Section 2.2, which is the approach for parallelizing stencil computations. Then we dedicate Section 2.3 to related works to the polyhedral model, a representation framework of loop transformations, which is implemented in real compilers to automatically parallelize loop nests. Finally, Section 2.4 introduces performance analysis and automatic compilation for GPUs that further optimize GPU code to gain better performance and simplify parallel programming.

2.1 Stencil Computations on GPUs

Iterative stencil loop nests are important kernels of many computations that are widely used in scientific and engineering applications. Depending on whether the inner loops contain loop-carried dependences, stencil loop nests fall into two categories: DOALL loop nests and DOACROSS loop nests. For example, Jacobi and SOR solvers are widely used as an iterative solver arising from the discretization of Partial Differential Equations (PDEs). They are stencil computations nested DOALL and DOACROSS loops respectively, which are introduced in Chapter 3.

DOALL Stencil There are a number of prior efforts on implementing DOALL stencil kernels on GPUs [27, 49, 52, 69, 70, 81, 96, 99, 136].

In the case of Jacobi-like stencil computations with DOALL inner loops, the circular queue method [27] streamed tiled data to hide I/O latency and exploited data locality in the GPU memory hierarchy. In [132], the circular queue method was extended to exploit a new reuse pattern spanning across the multiple time steps in stencil computations so that circular queues can be implemented by both shared memory and registers effectively in a balanced manner. A cached plane sweep algorithm [70] was introduced to put all threads of a block on a 2-D plane and let this plane sweep through the chunk. However, it does not exploit the data reuse among multiple sweeps across a computational domain.

For 3-D stencil kernels, good performance is achieved through multiple code transformations [71] that improve the locality, meanwhile resulting in moderate parallelism and abundant synchronization between tasks. In [49], automated code generation and time tiling optimization were adopted for improving 3-D stencil computations.

Special programming models for stencil computations have also attracted much attention recently. In [136], CUDA programming and on-chip texture memory were used, together with auto-tuning to find the best thread block configuration. A model in [69] was designed for iterative stencil computations running on GPUs. It relied on separating memory and computed requirements to predict ghost zone configurations.

However, they all focused on DOALL loop nests, since DOALL loop nests are amenable to parallel implementation. DOACROSS loop nests have to be reconstructed to be efficiently solvable on parallel computers. Locality enhancement and parallelization are two important optimizations to speed up such iterative computations on GPUs.

DOACROSS Stencil Many previous efforts attempted to design efficient parallel algorithms of the SOR-like DOACROSS loop nests on parallel machines. As a general algorithm, SOR has several parallel versions, including RBSOR [14], multi-color SOR [2, 68] and block-parallel SOR [115], which have been proposed mostly for distributed memory machines. Tang and Xue [91] presented a method for tiling SOR by applying skewing and tiling for distributed memory machines. Goumas et al. [43] later continued this line of investigation by focusing on the parallelization of 2-D iteration spaces that result from the discretization of PDEs. In addition, Huang et al. [51] introduced a code tiling technique for improving the cache performance of PDEs solvers. Michelle et al. [90] presented a parallel Gauss-Seidel (a special case of SOR, when relaxation factor $\omega = 1$) by applying a full sparse tiling technique to improve the cache locality of a program for shared memory machines. Wallin et al. [100] considered temporally tiling Gauss-Seidel with pipelining techniques to improve parallelism on shared memory machines. However, these DOACROSS techniques are not designed for GPU architectures and some features of the GPU are not considered in these early algorithms.

Recently, [64, 99] introduced an efficient "asynchronous" solution for Jacobi and SOR kernels on GPUs without respecting some loop-carried dependences. The basic idea is to algorithmically restructure a stencil kernel based on a non-dependence-preserving parallelization scheme to avoid pipelining for higher parallelism. In contrast, our approach described in Chapter 4 exploits better fine-grained parallelism without violating data dependences, and is applicable to all loop nests with uniform dependences. Moreover, in Chapter 6, we will introduce the other non-dependence-preserving algorithm to efficiently parallelize DOACROSS loop nests on GPUs.

2.2 Loop Tiling Transformations

Loop Tiling Loop tiling is a classic and effective optimization for programs whose execution time is dominated by loops, such as stencil kernels. Numerous optimizations based on the tiling of iteration spaces have been proposed for improving data locality [1, 54, 82, 88, 104, 105, 106, 113, 128, 129] or exploiting parallelism [5, 4, 18, 48, 103, 112, 127].

Earlier studies in [53, 107, 108] formally defined tiling as a loop transformation that divides the iteration space using hyperplanes into parallelepiped tiles and traverses the tiles to cover the iteration space, and proved that given a fixed tile size, the tiled iteration space is a polyhedron. For general cases, the standard tiling solution was described in Xue's book [126], including execution legality analysis and tile distribution strategies.

Tile Size Selection Tile size selection has been studied for decades. Earlier studies focused on matching the matrices of loop transformations to memory systems [1, 107]. Cost model solutions to tile size selection can characterize the performance of tiled loop nests with various tile configurations. These cost models are closely tied to the execution platform architectures and programs.

DOACROSS loop nests must be complicatedly transformed to preserve the loop-carried dependences. Performing tiling transformations allows wavefront parallelism to be exploited both across the tiles and within a tile. Intricate transformations make tile size selection more complicated for DOACROSS loop nests than DOALL loop nests. Thus, it is not practical to rely on the user to pick the right tile sizes to optimize code through improving processor utilization and reducing synchronization overhead. Existing tile size techniques proposed for caches in CPU architectures do not apply. Therefore, we design a model-driven method for tile size selection, which will be described in detail in Chapter 5.

Tiling Hyperplanes Selection When tiling a loop nest by a set of hyperplanes (see Chapter 3), the choice of hyperplanes is an NP-hard problem. To enable appropriate tiling hyperplanes to be selected, some work has focused on pruning the search space and optimizing hyperplane generation strategies [75, 78, 134, 129].

Communication-minimal tiling is one of the most appropriate methods of tiling hyperplanes, because it can significantly reduce volumes of communication that are the key bottleneck of parallel computing on distributed systems. Schreiber and Dongarra were perhaps the first investigating compiler techniques for finding communication-minimal tiling [34]. Lim and Lam's algorithm obtained affine partitions that minimized the order of communication as well as maximized the degree of parallelism [63]. Griebl's approach enabled tiling of the time dimension with a forward communication-only placement [44]. Bondhugula [13] further developed a communication-minimized parallelization framework in which a cost function was introduced to quantify communication volumes and reuse distances [24].

In fact, GPUs execute the inter-tile communication by synchronization of thread blocks. Active threads have to move all data from shared memory into global memory regardless of the data amount used for communication, because the shared memory will be occupied by those ready-to-activate thread blocks. Hence the overhead of synchronization is more influenced by the frequency rather than the size of communication, and the communication-minimal tiling is not the most appropriate tiling strategy for GPUs. In Chapter 4, we illustrate the disadvantages of the communication-minimized tiling on GPUs and present a new tiling strategy to maximize intra-tile parallelism for better performance.

Dependence Elimination Eliminating dependences is a method in restructuring programs to suppress anti and output dependences. This approach breaks loop-carried dependences, but keeping programs' correctness so as to optimize loop transformations, especially loop tiling [17, 62, 66]. Dependence elimination is usually done by array expansion or introducing new temporary arrays and associated copy operations [16, 36, 89, 130].

Compared with these proposed efforts restricted to CPUs, Chapter 4 presents a new dependence elimination method to enhance fine-grained (intra- and inter-SM) parallelism on GPUs, resulting in improved inter-tile parallelism and reduced bank conflicts that happen on shared memory.

2.3 Polyhedral Model

The polyhedral model is a well-studied, powerful mathematical framework to represent loop nests and their transformations, overcoming the limitations of classical, syntax driven models. Many studies have tried to assess a predictive model characterizing the best transformations within this model, mostly to express parallelism [39, 40, 63, 116, 117] or to improve locality [11, 26, 67, 104].

In the past decade, significant researches on the polyhedral model were made on dependence analysis [37, 38, 79] and code generation [45, 56, 61]. These efforts boosted theories of polyhedral model. Recent advances [9, 80, 97, 98] have solved the problem of scalability, so that the polyhedral techniques can be applied to code representative of real applications [19, 41].

Furthermore, the polyhedral model is practically designed as a part of optimizers in many mainstream compilers. Thanks to the flexibility and strength of composing and applying loop transformations, the polyhedral model could replace the standard Loop Nest Optimizer (LNO) pass in compilers, such as GRAPHITE for GCC [77], URUK for Open64 [41] and Polly for LLVM [46]. However, these tools mainly aim to gain better locality for traditional uni-processor architectures.

There are some efforts to automatically parallelize loop nests by using the polyhedral model. PIPS [57] was primarily designed as a C and Fortran source-to-source compilation framework for analyzing and transforming programs. It can be used to generate assembly code by lowering and specializing the representation. Based on PIPS, Par4All [74] generated CUDA and OpenCL code from C code with easy-to-use high-level directives. PLUTO [7] was released as a source-to-source compiler for translating sequential C programs into parallel OpenMP programs. In [8], PLUTO was extended to automatically generating CUDA code for GPUs. Recently, The Polly group has claimed they are preparing to create a preliminary implementation of GPU code generation. With this addition, users can parallelize some perfect loop nests to execute on a heterogeneous platform, composed of CPUs and GPUs. While performing some code optimizations, these tools do not specifically optimize loop nests with loop-carried dependences, as we do.

2.4 Performance Analysis and Compilation for GPUs

GPU Performance Analysis Many hardware limits are exposed for efficiently utilizing the GPU resources. There are two major bottlenecks significantly affecting the productivity in GPU programming: utilization of memory hierarchy and management of parallelism. Therefore, it is important to develop cost models and tuning tools for estimating and understanding the performances affected by complex interactions among the GPU architectural constraints.

To address these two issues, G-ADAPT [65] was introduced as an input-adaptive optimization framework to search and predict the best configuration for different input sizes for GPU programs. Yang et al. [133] presented a compiler framework for optimizing naive GPU kernel functions with effective utilization of GPU memory hierarchy and judicious management of parallelism. Ryoo et al. introduced useful performance metrics for pruning the optimization space by calculating the utilization and efficiency [84], and discussed optimization principles for GPU architectures [83]. Williams et al. [102] proposed a Roofline model that can visualize computing or memory bounded multi-core systems. In [50, 87], Hong et al. presented a MWP-CWP model for estimating the execution time of a program running on GPUs. Baghsorkhi et al. [6] proposed a work flow graph (WFG)-based analytical model to predict the performance of GPU programs. The WFG is an extension of a control flow graph, where nodes represent instructions and arcs represent latencies. A performance model was proposed in [137] to evaluate the execution time of the instruction pipeline, shared memory, and global memory in order to identify the bottlenecks, but without estimating the potential performance benefits caused by bottlenecks.

High-level GPU Programming Although CUDA offers improved programmability for general-purpose computing, programming GPUs using CUDA is still time-consuming and error-prone. Therefore, an important research direction is to implement the automatic source-to-source translation for programs written in high-level programming languages. Section 2.3 presents several automatic parallelization frameworks based on the polyhedral model. All of these tools need easy-to-use directives to assist compilers identify code regions that need be parallelized. Apart from these tools, there are still some efforts to simplify GPU programming by using directive-based languages, generating special API libraries and translating programs written in other existing parallel languages into CUDA kernels.

Brook [15] was introduced to feature the use of a streaming programming model. Accelerator [92] was later developed as a C# API library that uses a data-parallel model based on parallel arrays to program GPUs. CUDA-lite [95], as an enhancement to CUDA, automatically generated optimized code for coalescing memory transaction on GPUs. JCUDA [131] was designed to be a programmer-friendly foreign function interface for invoking CUDA kernels from Java code. Similarly, Dubach et al. presented a new Java compatible language based on Lime that allows an optimizing compiler to generate high-quality GPU code [35]. The hiCUDA project [94] was an academic research for developing a compiler that can translate annotated C programs into CUDA programs. Lee et al. aimed to translate OpenMP programs directly into CUDA kernels for execution on GPUs [60]. This work was extended to provide an abstraction for the CUDA programming model and offer high-level controls over involved parameters and optimizations [59].

In addition, many parallel computing service providers also dedicate their efforts into development of high-level GPU programming standards and commodity tools. As a commercial product, the PGI accelerator model [110] represented a high-level programming paradigm for GPUs, similar to the widely used OpenMP programming model. The IBM X10 language [25] provides abstractions for programming GPUs with a globally shared, partitioned address space and GPU threads communicate and synchronize through shared memory. CAPS Enterprise delivered a directive-based programming model, called HMPP [33]. Then based on HMPP, OpenACC [73] is designed as a new directive-programming standard to simplify parallel programming of heterogeneous CPU/GPU systems, and supported by many providers, including Cray, CAPS, Nvidia, PGI and PathScale.

Cui et al. [20, 21, 22, 23] introduced a pattern-oriented approach that allows domain-specific knowledge to be exploited by the compiler to generate efficient code for multi-core CPUs and GPUs. Some significant performance improvements can be achieved for matrix and stencil computations on GPUs.

Chapter 3

Compiler Framework Based on the Polyhedral Model

In this chapter, we firstly present an overview of the categories of loops, polyhedral model, and introduce notations used throughout this thesis. The main mathematical background on linear algebra and linear programming required to understand the theoretical aspects of this thesis are covered in this chapter. But some fundamental concepts and definitions relating to polyhedron, linear inequalities and loop transformation have been omitted and they can be found in [86, 101, 109, 118, 122, 126]. Finally, we present our compiler framework that automatically translates sequential C code into optimized parallel CUDA kernels.

3.1 Loops Categories

3.1.1 Dependence

Two iterations are said to be *dependent* if they access the same memory location and one of them is a write. There are several kinds of dependences. A *flow* dependence, also known as a data dependence or *true* dependence or *read-after-write (RAW)* dependence, exists when the target iterator's access is a read that depends on the result of a write source iterator. Similarly, if a read precedes a write to the same location, the dependence is called an *anti* dependence or *write-after-read (WAR)* dependence. Both writes the same memory location are known as *output* dependences or *write-after-write (WAW)* dependences. Anti and output dependences are also called *false dependences*. *Input* dependences or *read-after-read (RAR)* dependences are not actually dependences, but they still could be important in characterizing data reuse.

3.1.2 Categories of Loop Nests

Based on the nature of the dependences between iterations, loops have been classified into DOALL loops and DOACROSS loops. In a DOALL *loop*, as shown in Figure 3.1(a), there are no dependences between any pair of iterations of the loop. DOALL loops can be executed in parallel without any synchronization between iterations. Loops with *inter-iteration* or *loop-carried* dependences are called DOACROSS *loops*. We classify DOACROSS loops into three groups: regular DOACROSS loops, irregular DOACROSS loops and DOSEQUENTIAL loops. In a *regular* DOACROSS *loop*, as shown in Figure 3.1(b), dependence distances are constant. In contrast, in an *irregular* DOACROSS *loop* (as illustrated in Figure 3.1(c)), the dependence distance can vary from iteration to iteration. Typically, regular DOACROSS loops are more amenable to parallel execution than irregular ones. This thesis only discusses regular DOACROSS loops and the "DOACROSS" is used to only represent regular DOACROSS loops, since stencil computations only contain regular DOACROSS loops. Finally, when the dependence distance of a regular DOACROSS loop equals to 1, such loop has no parallelism at the iteration level

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

(a) DOALL Loop

```
for (i=1; i<=N; i++) {
   A[i]=A[i-d];
}
```

(b) Regular DOACROSS Loop

```
for (i=1; i<=N; i++) {
   A[i]=A[B[i]];
}
```

(c) Irregular DOACROSS Loop

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

(d) DOSEQUENTIAL Loop

Figure 3.1: Illustration of the different classes of loops.

and hence is referred to as a DOSEQUENTIAL *loop*, as shown in Figure 3.1(d).

A loop nest is composed of several nested loops. If all loops contained by a loop nest are DOALL loops, such loop nest is called a DOALL *loop nest*; otherwise if the nested loops contains one or more DOACROSS loops, it is called a DOACROSS *loop nest*.

3.1.3 Imperfect Loop Nests and Uniform Dependences

A set of nested loops is called a *perfect loop nest* if all statements appearing in the nest appear inside the body of the innermost loop and an *imperfect loop nest* otherwise [119, 121, 125]

Traditionally, the concept of *uniform dependence* is restricted to a set of perfect loop nests. In this thesis, we consider both perfect and imperfect loop nests with uniform dependences. Given loop nests (normalized with all loops in the same depth being associated with the same loop variable), a dependence between two references is *uniform* if their dependence distance is constant. Jacobi and SOR, given in Figure 3.2, are imperfect and perfect loop nests with uniform dependences, respectively.

Unlike the Jacobi in Figure 3.2(a) in which each statement updates array B by using the neighbouring values from the array A, the program of SOR is to update the same array A's values. Therefore, there are loop-carried dependences in the inner-most loop which is a DOACROSS loop. Such loop nest is called a SOR-like loop nest [42].

```
for (j=1;j<=J;j++){ //DOSEQUENTIAL
   for (i=1;i<=I;i++) //DOALL
S0:   B[i]=(A[i-1]+A[i]+A[i+1])/3;
   for (i=1;i<=I;i++) //DOALL
S1:   A[i]=B[i];
}
```

(a) Jacobi: an imperfect loop nest

```
for (j=1;j<=J;j++){ //DOSEQUENTIAL
   for (i=1;i<=I;i++) //DOACROSS
S:    A[i]=(A[i-1]+A[i]+A[i+1])/3;
}
```

(b) SOR: a perfect loop nest

Figure 3.2: Loop nests with uniform dependences.

3.2 Polyhedral Model

This section introduces the polyhedral model, which has been used to represent many kinds of loop transformations.

3.2.1 Iteration Space Representation

The loop nest $\mathcal{L}$ in Figure 3.3 can be presented as an iteration space $\mathcal{I}$ that is a finite set of points in the n-D space $\mathbb{Z}^n$, where $\mathbb{Z}$ is the set of integers. The loop bounds vectors $\vec{l} = (l_1, l_2, ..., l_n)$ and $\vec{u} = (u_1, u_2, ..., u_n)$ are functions of the loops'

```
for(i₁=l₁();i₁<=u₁();i₁++)
    for(i₂=l₂(i₁);i₂<=u₁(i₁);i₂++)
        ...
        for(iₙ=lₙ(i₁,i₂,...,iₙ₋₁);iₙ<=uₙ(i₁,i₂,...,iₙ₋₁);iₙ++)
            Statement S;
            ...
```

Figure 3.3: Loop nests with scalar induction variables

index variables, where

$$l_k = l_k(i_1, i_2, ..., i_{k-1}), \qquad k \in [1, n] \tag{3.1}$$

$$u_k = u_k(i_1, i_2, ..., i_{k-1}), \qquad k \in [1, n] \tag{3.2}$$

Apparently, the iteration space $\mathcal{I}$ can be defined as a convex polyhedron.

$$\mathcal{I} = \{\vec{i_S} \mid \forall k \in [1, n] : l_k \leq i_k \leq u_k\} \tag{3.3}$$

$$\vec{l} \leq \vec{i_S} \leq \vec{u} \tag{3.4}$$

Each point in $\mathcal{I}$ can be considered as a vector $\vec{i_S}$ in an *affine space*. When vectors $\vec{i_1}$ and $\vec{i_2}$ are in a space, this space is an affine space, iff all points lying on the line joining $\vec{i_1}$ and $\vec{i_2}$ belong to the space. The affine space is closed under affine combinations.

The shape of $\mathcal{I}$ is usually surrounded by a set of edges, called *hyperplanes*. A hyperplane in an n-D space is an $(n-1)$-D affine subspace of the n-D space and can be represented by an affine equality. Therefore, the points on one side of hyperplanes in the n dimensions are denoted by an affine inequality. A polyhedron $\mathcal{D}_S$ is a set of points surrounded by finitely many hyperplanes and can be described by a group of affine inequalities.

$$\mathcal{D}_S = \{\vec{i_S} \in \mathbb{Z}^n \mid A\vec{i_S} + \vec{b} \geq 0\} \tag{3.5}$$

```
for (j=1;j<=J;j++)
   for (i=1;i<=I;i++)
S:    A[i]=0.5*(A[i]+A[i+1]);
```

Figure 3.4: Example code

The polyhedral model uses an n-D bounded polyhedron to statement-wisely represent the iteration space $\mathcal{I}$ of a loop nest, in which a statement S is surrounded by n-D nested loops. Using matrix representation, the polyhedron $\mathcal{D}_S$ of a loop nest can be defined as $R_S(\vec{i}_S, \vec{g}, 1)^T \geq 0$, where $\vec{i}_S$ is the iteration vector, $\vec{g}$ is the vector of global parameters that are usually used to define the loop bound, and R_S is a matrix representing loop bound constraints. For instance, the loop nest shown in Figure 3.4 can be represented as follows:

$$\mathcal{D}_S : \quad R_S \begin{pmatrix} \vec{i}_S \\ \vec{g} \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & -1 \\ 0 & -1 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} j \\ i \\ J \\ I \\ 1 \end{pmatrix} \geq 0 \tag{3.6}$$

where $\vec{i}_S = (j, i)^T$ and $\vec{g} = (J, I)^T$.

3.2.2 Affine Transformations

An *affine transformation* or *affine map* is a function between affine spaces. This means that an affine transformation maps each point in one coordinate system into another one by sending points to points, lines to lines, planes to planes, etc.

A 1-D affine transformation for statement S in an n-D loop nest at the d-th level is an affine function defined by

$$\phi_S^d(\vec{i}_S) = (c_1^d, c_2^d, \ldots, c_n^d)\vec{i}_S + c_0^d = (c_1^d, c_2^d, \ldots, c_n^d, c_0^d)\begin{pmatrix}\vec{i}_S \\ 1\end{pmatrix} \tag{3.7}$$

where $c_0^d, c_1^d, c_2^d, \ldots, c_n^d \in \mathbb{Z}$ and $\vec{i}_S \in \mathbb{Z}^n$.

Such transformation for each statement can be interpreted as a partitioning hyperplane with normal $(c_1^d, c_2^d, \ldots, c_n^d)$, which maps statement S to a point in a dimension of the transformed iteration space. A multi-dimensional affine transformation Φ_S can be represented as a set of ordered ϕ_S, described as $\Phi_S = \{\phi_S^1, \phi_S^2, \ldots, \phi_S^m\}$, for each statement S mapped into a point in the m-D transformed space:

$$\Phi_S(\vec{i}_S) = \begin{pmatrix}\phi_S^1(\vec{i}_S) \\ \phi_S^2(\vec{i}_S) \\ \vdots \\ \phi_S^m(\vec{i}_S)\end{pmatrix} = \begin{pmatrix}c_1^1 & c_2^1 & \cdots & c_n^1 \\ c_1^2 & c_2^2 & \cdots & c_n^2 \\ \vdots & \vdots & \vdots & \vdots \\ c_1^m & c_2^m & \cdots & c_n^m\end{pmatrix}\vec{i}_S + \begin{pmatrix}c_0^1 \\ c_0^2 \\ \vdots \\ c_0^m\end{pmatrix} \tag{3.8}$$

3.2.3 Loop Fusion and Fission

When loop transformations cover fusion and fission, Φ_S needs more dimensionality than n to represent partially fused or unfused dimensions. This level k is called a *scalar dimension*, because such a row with ϕ_S^k, its $(c_1^k, c_2^k, \cdots, c_n^k)$ always equals $\vec{0}$, and only c_0^k is a constant. When statements are fused at the k-th level, they have the same value of c_0^k. Unfused sets have an increasing c_0^k. Hence, the level is a scalar dimension if the ϕ_S^k for all statements at this level is a constant function. For example, the two sequential loop nests in Figure 3.5(a) are fused into Figure 3.5(b). Equations 3.9 and 3.10 show the fusion transformations. After fusion, the order of loop indices changes, ij for $S0$ and ji for $S1$. ϕ^1 and ϕ^3 are scalar dimensions that respectively represent the fusion of the first loop i of $S0$ and j of $S1$, and the fission of the second loop j of $S0$ and i of $S1$:

$$\Phi_{S_0}(\vec{i}_{S_0}) = \begin{pmatrix} \phi^1_{S_0}(\vec{i}_{S_0}) \\ \phi^2_{S_0}(\vec{i}_{S_0}) \\ \phi^3_{S_0}(\vec{i}_{S_0}) \\ \phi^4_{S_0}(\vec{i}_{S_0}) \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} j \\ i \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ i \\ 0 \\ j \end{pmatrix} \tag{3.9}$$

$$\Phi_{S_1}(\vec{i}_{S_1}) = \begin{pmatrix} \phi^1_{S_1}(\vec{i}_{S_1}) \\ \phi^2_{S_1}(\vec{i}_{S_1}) \\ \phi^3_{S_1}(\vec{i}_{S_1}) \\ \phi^4_{S_1}(\vec{i}_{S_1}) \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} j \\ i \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ j \\ 1 \\ i \end{pmatrix} \tag{3.10}$$

If loop fusion is not considered, scalar dimensions can be ignored. For clarity of expression, Φ_S in this thesis has generally ignored scalar dimensions, unless otherwise noted.

3.2.4 Dependence Polyhedron

The *data dependence graph (DDG)* $G = (V, E)$ is a directed multi-graph with each vertex representing a statement S and an edge $e_{S_s,S_t} \in E$ from S_s to S_t indicating a dependence between the source and target conflicting accesses in S_s and S_t, respectively. If $\vec{i}_s \in \mathcal{D}_{S_s}$ and $\vec{i}_t \in \mathcal{D}_{S_t}$ are dependent through edge $e_{S_s,S_t} \in E$, we write $\langle \vec{i}_s, \vec{i}_t \rangle \in \mathcal{P}_{e_{S_s,S_t}}$, where $\mathcal{P}_{e_{S_s,S_t}}$ is the *dependence polyhedron* of e_{S_s,S_t}. In the important special case when $\vec{i}_t - \vec{i}_s$ is a constant $\vec{d}$ for all $\langle \vec{i}_s, \vec{i}_t \rangle \in \mathcal{P}_{e_{S_s,S_t}}$, then $\vec{d} = \vec{i}_t - \vec{i}_s$ is known as a *distance vector*. A loop nest is said to have *uniform dependences* if all its dependences can be expressed as distance vectors. In this thesis, we express a dependence using its distance vector.

The loop nest given in Figure 3.4 only has uniform dependences. Table 3.1 lists its five dependences and their dependence polyhedrons. The DDG for this example is shown in Figure 3.6.

```
for (j=0; j<N; j++) {
   for (i=0; i<N; i++) {
S0:   C[i]=0.5*C[i]+0.5*A[i][j];
   }
}
for (j=0; j<N; j++) {
   for (i=0; i<N; i++) {
S1:   D[i]=D[i]+C[j]*B[j][i];
   }
}
```

(a) Original code

```
for (e1=0;e1<N;e1++) {
   for (e2=0;e2<N;e2++) {
S0:   C[e1]=0.5*C[e1]+0.5*A[e1][e2];;
   }
   for (e2=0;e2<N;e2++) {
S1:   D[e2]=D[e2]+C[e1]*B[e1][e2];;
   }
}
```

(b) Fused code

Figure 3.5: An example of polyhedral transformations for loop fusion.

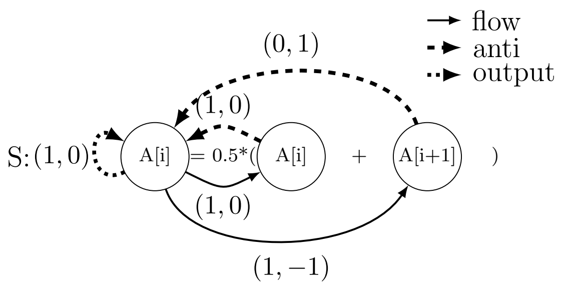


Figure 3.6: DDG for the loop nest given in Figure 3.4 with each dependence labelled by its distance vector.

3.2.5 Precedence Condition

For a loop transformation to respect the program semantics, it has to ensure that the execution order of instances will respect the precedence condition, for each pairs

Dependence	Type	Source $\vec{i_s}$ (j_s, i_s)	Target $\vec{i_t}$ (j_t, i_t)	Distance vector $\vec{d}$	Dependence Polyhedron
e_1: A[i] (LHS) → A[i] (RHS)	flow	(j, i)	$(j+1, i)$	$(1, 0)$	$\mathcal{P}_{e_1}: \begin{array}{l} j_s = j_t - 1, i_s = i_t, \\ 2 \le j_t \le J, 1 \le i_t \le I \end{array}$
e_2: A[i] (LHS) → A[i+1] (RHS)	flow	(j, i)	$(j+1, i-1)$	$(1, -1)$	$\mathcal{P}_{e_2}: \begin{array}{l} j_s = j_t - 1, i_s = i_t + 1, \\ 2 \le j_t \le J, 1 \le i_t \le I - 1 \end{array}$
e_3: A[i+1] (RHS) → A[i] (LHS)	anti	(j, i)	$(j, i+1)$	$(0, 1)$	$\mathcal{P}_{e_3}: \begin{array}{l} j_s = j_t, i_s + 1 = i_t, \\ 1 \le j_t \le J, 1 \le i_t \le I - 1 \end{array}$
e_4: A[i] (RHS) → A[i] (LHS)	anti	(j, i)	$(j+1, i)$	$(1, 0)$	$\mathcal{P}_{e_4}: \begin{array}{l} j_s + 1 = j_t, i_s = i_t, \\ 1 \le j_t \le J - 1, 1 \le i_t \le I \end{array}$
e_5: A[i] (LHS) → A[i] (LHS)	output	(j, i)	$(j+1, i)$	$(1, 0)$	$\mathcal{P}_{e_5}: \begin{array}{l} j_s + 1 = j_t, i_s = i_t, \\ 1 \le j_t \le J - 1, 1 \le i_t \le I \end{array}$

Table 3.1: Dependences in the loop nest given in Figure 3.4.

of instances in dependences.

Theorem 3.1 (Precedence Condition) *Given two statements S_s and S_t, and their dependence polyhedron $\mathcal{P}_{e_{S_s,S_t}}$. Φ_{S_s} and Φ_{S_t} preserve the program semantics if*

$$\Phi_{S_s}(\vec{i_s}) \prec \Phi_{S_t}(\vec{i_t}) \tag{3.11}$$

where $\prec$ denotes *lexicographically smaller.* Let $\vec{x} = (x_1, \cdots, x_m)$ and $\vec{y} = (y_1, \cdots, y_n)$, $\vec{x} \prec \vec{y}$ means $\exists i \in [1, min(m, n)]$, $(x_1, \cdots, x_{i-1}) = (y_1, \cdots, y_{i-1}) \wedge x_i < y_i$.

3.2.6 Parallelization

One loop nest can be parallelized, if its parallel partition does not violate its dependences.

Theorem 3.2 (Parallelism-aware Loop) *Consider a loop nest with DDG $G = (V, E)$. $\Phi_{S_1}, \Phi_{S_2}, \ldots$ are (statement-wise) legal hyperplanes. Let E_k is the set of edges as follows:*

$$\forall l \in [1, k-1], \phi^l_{S_t}(\vec{i_t}) - \phi^l_{S_s}(\vec{i_s}) = 0, \langle \vec{i_s}, \vec{i_t} \rangle \in \mathcal{P}_{e_{S_s,S_t}} \tag{3.12}$$

Such a loop nest is parallelism-aware *and can be parallelized at the k-th level, iff*

$$\forall e_{S_s,S_t} \in E_k, \phi^k_{S_t}(\vec{i_t}) - \phi^k_{S_s}(\vec{i_s}) = 0, \langle \vec{i_s}, \vec{i_t} \rangle \in \mathcal{P}_{e_{S_s,S_t}} \tag{3.13}$$

Here, if $E_k = E$, such parallelism is referred to as synchronization-free parallelism; otherwise data synchronization is needed at the k'-th level at which Equation 3.12 has been broken, i.e. $\exists e_{S_s,S_t} \in E$, $\langle \vec{i_s}, \vec{i_t} \rangle \in \mathcal{P}_{e_{S_s,S_t}}, \exists l \in [1, k'], \phi^l_{S_t}(\vec{i_t}) - \phi^l_{S_s}(\vec{i_s}) > 0$.

For example, Figure 3.7 shows an example of the synchronization-free parallelism. The source and target statements are the same and a dependence $(0, 1)$ exists. If transformation hyperplanes are $\Phi_S(\vec{i_S}) = \left(\begin{smallmatrix}1 & 0\\0 & 1\end{smallmatrix}\right)\left(\begin{smallmatrix}j\\i\end{smallmatrix}\right) + \left(\begin{smallmatrix}0\\0\end{smallmatrix}\right)$, then $\Phi_{S_t}(\vec{i_t}) - \Phi_{S_s}(\vec{i_s}) = \Phi_S(\left(\begin{smallmatrix}0\\1\end{smallmatrix}\right)) = \left(\begin{smallmatrix}0\\1\end{smallmatrix}\right)$. In fact, the new coordinate system mapped by Φ_S is the same with the original one. Apparently, it satisfies Theorem 3.2 at the first level, and the dependence set $E_1 = E$. Such loop nest can be partitioned at the first level and does not need synchronization, shown in Figure 3.9 (left).

```
for (j=1; j<=N; j++) {
   for (i=1; i<=N; i++) {
S:     A[j][i]=A[j][i-1]+0.5*A[j][i];
   }
}
```

Figure 3.7: An example of synchronization-free parallelism.

The dependence distance vectors in Figure 3.8 are $(1, 1)$, $(1, 0)$ and $(1, -1)$. In the transformed space by $\Phi_S(\vec{i_S}) = \left(\begin{smallmatrix}1 & 0\\0 & 1\end{smallmatrix}\right)\left(\begin{smallmatrix}j\\i\end{smallmatrix}\right) + \left(\begin{smallmatrix}0\\0\end{smallmatrix}\right)$, the first level's value of all mapped dependence vectors remain 1, therefore the dependence set E_2 is empty. According to Theorem 3.2, such loop nest can be parallelized at the second level with synchronizing data at the first level, shown in Figure 3.9 (right).

It is illegal to move a non-synchronization-free parallel loop in the outer direction over the k'-th level by loop exchanging, since the dependences satisfied at

```
for (j=1; j<=N; j++) {
   for (i=1; i<=N; i++) {
S:    A[j][i]=0.333*(A[j-1][i-1]+A[j-1][i]+A[j-1][i+1]);
   }
}
```

Figure 3.8: An example of non-synchronization-free parallelism.

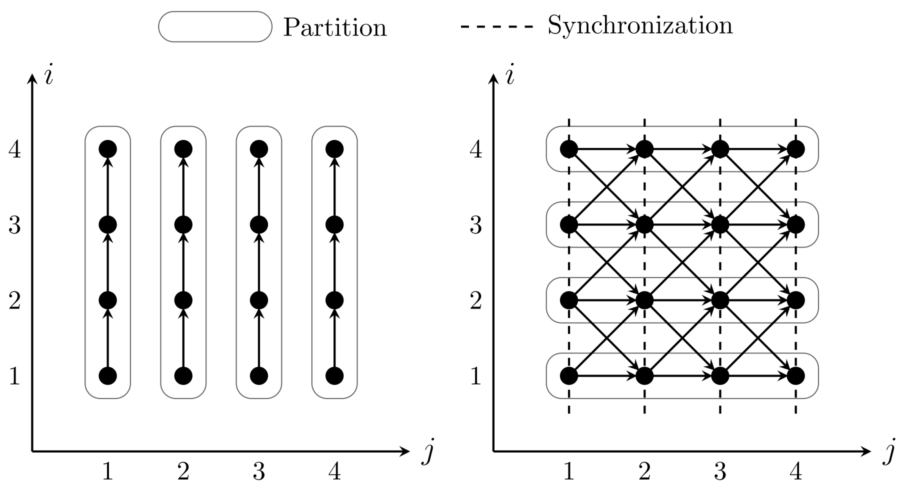


Figure 3.9: Illustration of parallel-aware loop nests.

the parallel loop nest may be violated at the new position of the moved loop nest. However, it is always legal to move the parallel loop further inside.

The loop nests in Figures 3.7 and 3.8 are said to exhibit DOALL *parallelism.* DOACROSS loop nests with loop-carried dependences can be also executed in parallel by appropriate hyperplanes. The set of hyperplanes rearranges DOACROSS loop nests to delay computations with respect to the others by a fixed amount, so that the dependences are laid to outer loop. This method is called *loop skewing*. Figure 3.10 shows an example of DOACROSS loop nest parallelism. In the new 2-D iteration space with the $e1$ and $e2$ coordinate axes mapped by $\Phi_S(\vec{i}_S) = \left(\begin{smallmatrix}1 & 1\\ 0 & 1\end{smallmatrix}\right)\left(\begin{smallmatrix}j\\ i\end{smallmatrix}\right)+\left(\begin{smallmatrix}0\\ 0\end{smallmatrix}\right)$, the dependences distance vectors are mapped from $(0, 1)$ to $(1, 1)$, $(1, 1)$ to $(2, 1)$ and $(1, 0)$ to $(1, 0)$. Successive iterations can start with a delay of one and con-

```
for (j=1; j<=N; j++) { //DOSEQUENTIAL
   for (i=1; i<=N; i++) { //DOACROSS
S:    A[j][i]=0.333*(A[j][i-1]+A[j-1][i]+A[j-1][i-1]);
   }
}
```

Figure 3.10: An example of DOACROSS parallelism.

```
for (e1=2; e1<=N+N; e1++) { //DOSEQUENTIAL
   for (e2=max(1,e1-N); e2<=min(N,e1-1); e2++) { //DOALL
      i=e2;
      j=e1-e2;
S:    A[j][i]=0.333*(A[j][i-1]+A[j-1][i]+A[j-1][i-1]);
   }
}
```

Figure 3.11: Transformed code in Figure 3.10 with hyperplanes $\Phi_S(\vec{i_S}) = \left(\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right)\left(\begin{smallmatrix} j \\ i \end{smallmatrix}\right) + \left(\begin{smallmatrix} 0 \\ 0 \end{smallmatrix}\right)$.

tinue executing iterations for $e1$ in sequence. According to Theorem 3.2, it can be parallelized at the second level with synchronizing data at the first level, shown in Figure 3.12. Similarly, if a DOACROSS loop nest has m independent dimensions, at most $m-1$ of them can be pipelined while the iterations along at least one will be executed in sequence.

3.2.7 Tiling Representation with the Polyhedral Model

3.2.7.1 Tiling

The polyhedral model provides elegant representations for traditional tiling schemes. The polyhedral model uses a set of hyperplanes that partition the iteration space to represent tiling transformations. The steps of hyperplanes are the tile sizes at each level these hyperplanes exist. Tiling of k levels out of an m-D affine space is implemented by mapping the space from $\mathbb{Z}^m$ to $\mathbb{Z}^{m+k}$. Combining

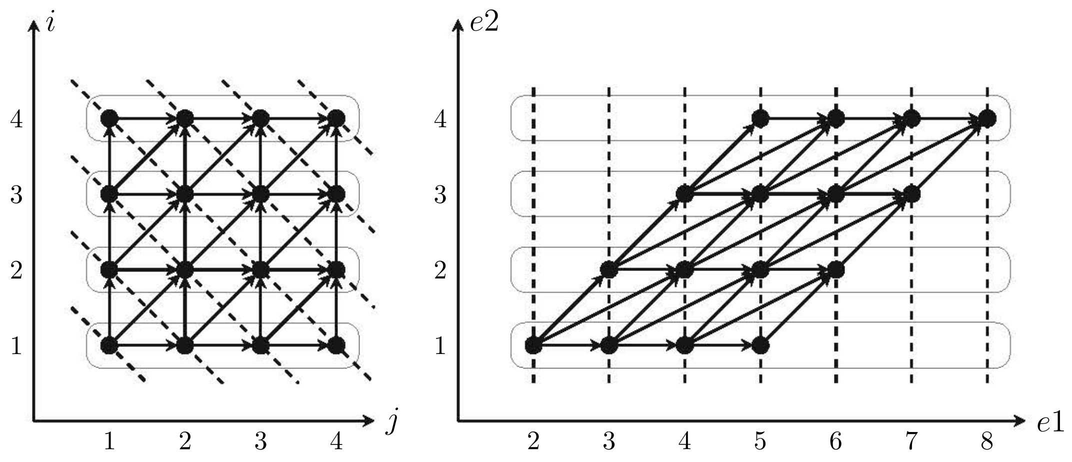


Figure 3.12: Illustration of code in Figure 3.10. The left figure is original iteration space; the other is the new coordinate system mapped by $\Phi_S(\vec{i}_S) = \left(\begin{smallmatrix}1 & 1\\ 0 & 1\end{smallmatrix}\right)\left(\begin{smallmatrix}j\\ i\end{smallmatrix}\right) + \left(\begin{smallmatrix}0\\ 0\end{smallmatrix}\right)$.

with the affine transformation from $\mathbb{Z}^n$ to $\mathbb{Z}^m$ described in Section 3.2.2, the entire transformation maps the space from $\mathbb{Z}^n$ to $\mathbb{Z}^{m+k}$. For clarity of expression, this thesis only considers the case of $k = m$, i.e. mapping the space from $\mathbb{Z}^n$ to $\mathbb{Z}^{2m}$. A rectangular tiling is characterised by the tile size vector $\vec{s} = (s_1, s_2, \cdots, s_m)^T$. If ignoring scalar dimensions, tiling transformations Γ_S are presented as follows [126]:

$$\Gamma_S : \mathbb{Z}^n \mapsto \mathbb{Z}^{2m},\ \Gamma_S(\vec{i}_S) = \begin{pmatrix}\Phi_T(\vec{i}_S)\\ \\ \Phi_S(\vec{i}_S)\end{pmatrix} = \begin{pmatrix}\vec{t}\\ \\ \vec{e}\end{pmatrix} = \begin{pmatrix}\lfloor\frac{\Phi_S(\vec{i}_S)}{\vec{s}}\rfloor\\ \\ \Phi_S(\vec{i}_S)\end{pmatrix} \tag{3.14}$$

where $\Phi_T(\vec{i}_S) = \vec{t} = \lfloor\frac{\Phi_S(\vec{i}_S)}{\vec{s}}\rfloor$ identifies the index of tile that contains the point $\vec{i}_S$, and $\vec{e} = \Phi_S(\vec{i}_S)$ simply remembers the point mapped to the tile $\vec{t}$ by Φ_S. The floor operators $\lfloor\ \rfloor$, when extended to vectors, are component-wise. Thus, if $\vec{x} = (x_1, x_2, \cdots, x_n)$ is a vector, then $\lfloor\vec{x}\rfloor = (\lfloor x_1\rfloor, \lfloor x_2\rfloor, \cdots, \lfloor x_n\rfloor)$.

At this point, a $2m$-D loop nest is created such that the first m loops (called *tile loops* $\mathcal{T}$) enumerate the tiles and the inner m loops (called *element loops*) enumerate the iterations within a tile. The tiled iteration space $\mathcal{I}'$, is the image of the iteration space $\mathcal{I}$:

$$\mathcal{I}' = \{\Gamma_S(\vec{i}_S) \mid \vec{i}_S \in \mathcal{I}\} = \left\{ (\vec{t}, \vec{e}) \;\middle|\; \begin{array}{c} \vec{t} = \lfloor \frac{\Phi_S(\vec{i}_S)}{\vec{s}} \rfloor, \vec{e} = \Phi_S(\vec{i}_S), \vec{i}_S \in \mathcal{I} \\ \vec{s} \circ \vec{t} \leq \vec{e} \leq \vec{s} \circ \vec{t} + \vec{s} - \vec{1} \end{array} \right\} \tag{3.15}$$

where if $\vec{x} = (x_1, x_2, \cdots, x_n)$ and $\vec{y} = (y_1, y_2, \cdots, y_n)$, operator $\circ$ is defined as $\vec{x} \circ \vec{y} = (x_1 y_1, x_2 y_2, \cdots, x_n y_n)$.

Legality of Tiling To preserve the dependences in a given loop nest, a legal tiling hyperplanes Φ_S must satisfy all its dependences [12, 120, 123, 126].

Theorem 3.3 (Legality of Tiling) *Consider a loop nest with DDG $G = (V, E)$. $\Phi_{S_1}, \Phi_{S_2}, \ldots$ are legal (statement-wise) tiling hyperplanes iff*

$$\forall e_{S_s,S_t} \in E, \forall k \in [1, m], \phi^k_{S_t}(\vec{i}_t) - \phi^k_{S_s}(\vec{i}_s) \geq 0, \langle \vec{i}_s, \vec{i}_t \rangle \in \mathcal{P}_{e_{S_s,S_t}} \tag{3.16}$$

Figure 3.15 shows the legal and illegal tiling of code in Figure 3.4 with the dependence vectors $(0, 1)$, $(1, 0)$ and $(1, -1)$. Tiling hyperplanes $\Phi_S(\vec{i}_S) = \left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)\left(\begin{smallmatrix} j \\ i \end{smallmatrix}\right) + \left(\begin{smallmatrix} 0 \\ 0 \end{smallmatrix}\right)$ are illegal, because the new dependence vector $(1, -1)$ mapped from $(1, -1)$ does not satisfy Theorem 3.3. By contrast, tiling hyperplanes $\Phi_S(\vec{i}_S) = \left(\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right)\left(\begin{smallmatrix} j \\ i \end{smallmatrix}\right) + \left(\begin{smallmatrix} 0 \\ 0 \end{smallmatrix}\right)$ are legal.

3.2.7.2 Inter- and Intra-tile Wavefront Parallelism

In many cases, the pipelining execution at tiling level is necessary for preserving dependences. Figure 3.14 shows the tiled loops in Figure 3.10. Nevertheless, due to the inter-tile and intra-tile dependences, this code cannot be directly parallelized. Thus, if one or more of the tile loops $\mathcal{T}$ have dependences along them, a transformation Θ_T is necessary in $\mathcal{T}$ to schedule tiles for exposing parallelism across the tiles.

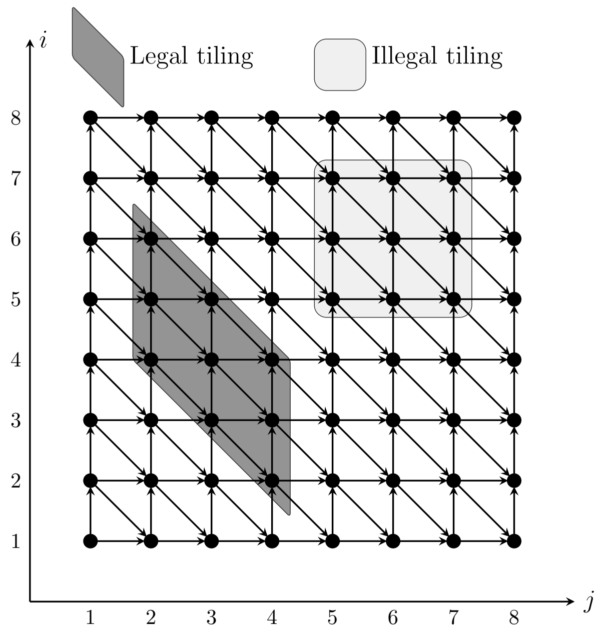


Figure 3.13: Legal and illegal tiling of code in Figure 3.4

Theorem 3.3 makes dependence components non-negative along each of the tiling hyperplanes, therefore a simple wavefront transformation Θ_T is adopted to expose inter-tile parallelism [12]:

$$\Theta_T(\vec{t}) = \Theta_T(\Phi_T(\vec{i}_S)) = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & 1 \end{pmatrix}_{m\times m} \begin{pmatrix} t_1 \\ t_2 \\ \vdots \\ t_m \end{pmatrix} \tag{3.17}$$

Due to Theorem 3.1 and Theorem 3.3, for m-D legal (statement-wise) tiling hyperplanes $\Phi_{S_1}, \Phi_{S_2}, \ldots$, there exists

$$\sum_{k=1}^{m}(\phi^k_{S_t}(\vec{i}_t) - \phi^k_{S_s}(\vec{i}_s)) \geq 1, \langle \vec{i}_s, \vec{i}_t \rangle \in \mathcal{P}_{e_{S_s,S_t}} \tag{3.18}$$

The m-D outer tiling transformation $\Phi_T = \{\phi^1_T, \phi^2_T, \ldots, \phi^m_T\}$ of Γ_S still satisfies Equation 3.18, because tiling transformations do not modify the dependences' direction. According to Theorem 3.2, the transformed tile space can be non-synchronization-free parallelized at the second level. For example, Figure 3.16

shows the tiling transformations of Figure 3.10 with the tile space wavefront transformation Θ_T. The transformed code can be parallelized at the level t_2' and the data synchronization occurs at the level t_1'.

```
/*---outer inter-tile loop nest---*/
for (t1=0;t1<=floord(N,2);t1++){//DOSEQUENTIAL
  for (t2=0;t2<=floord(N,2);t2++){//DOSEQUENTIAL
    /*+++inner intra-tile loop nest+++*/
    for (e1=max(1,2*t1);e1<=min(N,2*t1+1);e1++){//DOSEQUENTIAL
      for (e2=max(1,2*t2);e2<=min(N,2*t2+1);e2++){//DOSEQUENTIAL
S:        A[e1][e2]=A[e1][e2-1]+A[e1-1][e2]+A[e1-1][e2-1];
      }
    }
    /*++++++++++++++++++++++++++++++++*/
  }
}
```

Figure 3.14: Tiled loops of code in Figure 3.10 with tiling hyperplane $\Phi_S(\vec{i}_S) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} j \\ i \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$. and $\vec{s} = (2, 2)^T$

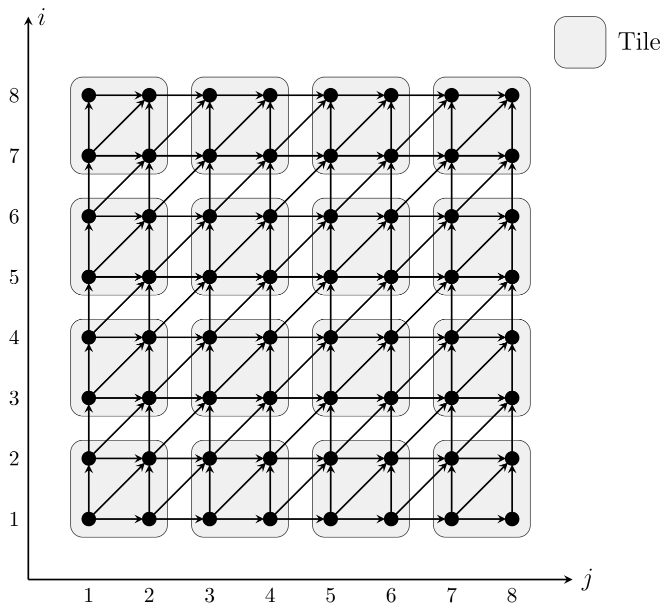


Figure 3.15: Illustration of code in Figure 3.14, with the tiles executed sequentially and the points in a tile also executed sequentially point-wise.

```
/*---outer inter-tile loop nest---*/
for (t1'=0;t1'<=N;t1'++){//DOSEQUENTIAL
  lb1=max(0,ceil(2*t1'-N,2));
  ub1=min(floor(N,2),t1');
  for (t2'=lb1; t2'<=ub1; t2'++){//DOALL
    /*+++inner intra-tile loop nest+++*/
    lb2=max(max(max(2,2*t1'),2*t2'+1),2*t1'-2*t2'+1);
    ub2=min(min(min(2*N,2*t1'+2),2*t2'+N+1),2*t1'-2*t2'+N+1);
    for (e1'=lb2;e1'<=ub2;e1'++){//DOSEQUENTIAL
      lb3=max(max(max(1,2*t2'),e1'-N),-2*t1'+2*t2'+e1'-1);
      ub3=min(min(min(N,2*t2'+1),e1'-1),-2*t1'+2*t2'+e1');
      for (e2'=lb3;e2'<=ub3;e2'++){//DOALL
        j=e1'-e2'-(2*(t1'-t2')-1);
        i=e2'-(2*(t2')-1);
S:      A[j][i]=A[j][i-1]+A[j-1][i]+A[j-1][i-1];
      }
    }
    /*++++++++++++++++++++++++++++++++*/
  }
}
```

Figure 3.16: Parallelizing tiled code of Figure 3.10 with the tiling hyperplane $\Phi_S(\vec{i}_S) = \left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right)\left(\begin{smallmatrix}j\\i\end{smallmatrix}\right) + \left(\begin{smallmatrix}0\\0\end{smallmatrix}\right)$, $\vec{s} = (2,2)^T$ and $\Theta_T = \Theta_S = \left(\begin{smallmatrix}1&1\\0&1\end{smallmatrix}\right)$

Similar to inter-tile parallelism, when the points in one tile are mapped to different processors, if the intra-tile loops do not satisfy Theorem 3.2, the intra-tile wavefront transformation Θ_S is applied to intra-tile parallelism. For clarity of statement, let $\Phi_S^{\text{inter}}(\vec{i}_S) = \Theta_T(\Phi_T(\vec{i}_S))$ and $\Phi_S^{\text{intra}}(\vec{i}_S) = \Theta_S(\Phi_S(\vec{i}_S))$, the new parallel tiling transformation is defined as follows:

$$\Gamma_S' : \mathbb{Z}^n \mapsto \mathbb{Z}^{2m}, \Gamma_S'(\vec{i}_S) = \begin{pmatrix} \Phi_S^{\text{inter}}(\vec{i}_S) \\ \Phi_S^{\text{intra}}(\vec{i}_S) \end{pmatrix} = \begin{pmatrix} \vec{t} \\ \vec{e} \end{pmatrix} = \begin{pmatrix} \Theta_T(\vec{t}) \\ \Theta_S(\vec{e}) \end{pmatrix} \tag{3.19}$$

In fact, introducing the wavefront transformation Θ_S may not be the best method to expose intra-tile parallelism. A more feasible and efficient way is to make Φ_S directly satisfy Theorem 3.2. We further discuss this issue and propose our method to enhance intra-tile parallelism in Chapter 4.

In addition, due to the wavefronting scheduling caused by loop-carried dependences, the tile size directly impacts the intra- and inter-tile parallelism that is performance-critical. This issue will be further discussed in Chapter 5.

3.3 Mapping Tiled Loop Nests to GPUs

In this Section, we firstly provide an overview of the GPU architectures, and then introduce how to map tiled loop nests to GPUs.

3.3.1 GPU Architectures

In this thesis, we mainly use the NVIDIA Tesla C1060 and C2050 GPU computing system as the basis for our investigation.

Memory Hierarchy Each GPU has many *streaming multiprocessors (SMs)* with each SM containing some *streaming processors (SPs)* to form a large set of processor cores. Every GPU has an off-chip memory, called *global memory* or *device memory*, but its access bandwidth can be easily saturated. Due to very high latency of global memory accessing, several on-chip memories are available to exploit data reuse so as to lessen an application's demand for off-chip memory bandwidth and reduce expensive off-chip memory traffic. In particular, each SM has a user/compiler-managed *shared memory* for data reuse or sharing among threads and 32-bit registers partitioned among threads. For read-only data, the *constant* and *texture cache memories* can significantly reduce memory latency. In C2050, shared memory can be further partly partitioned as *L1 cache* that is used to reduce latency of register spill and so on.

Two-level Parallelism A GPU, with a scalable array of SMs that comprises a number of SPs, allows a large number of fine-grained threads to cooperate in solving large-scale applications. In the NVIDIA CUDA programming model [72], an execution of a CUDA kernel launches a set of *thread blocks* and each thread block can contain hundreds of *threads*, with every 32 threads grouped into a *warp*. SPs on one SM execute a warp of 32 threads at a time; and all threads in one block are not able to be executed on two or more SMs at a time. Since threads from different blocks can be executed on SMs concurrently, there exist two levels of fine-grained parallelism for a running kernel: inter-SM and intra-SM (i.e., among the SPs in the same SM).

Bank Conflicts *Bank conflicts* occur when the threads in one warp access different shared memory addresses in the same bank. The threads with bank conflicts are forced to access shared memory sequentially. According to [72], if the threads in one warp access a sequence of continuous shared memory addresses (i.e., exhibit stride-1 assesses), there will be no bank conflicts. If the accesses are made in stride-2, for example, two-way bank conflicts will occur. Then a pair of threads must each wait for the other to finish, which produces a huge latency thereby reducing the performance.

Global Memory Access Coalescing In addition, reducing the latency in accessing data from global memory is crucial for good performance. Due to a hardware optimization known as *global memory access coalescing*, accesses from adjacent threads in a warp to adjacent locations are coalesced into a single contiguous aligned memory access. Thus, the significant performance benefits due to coalesced accesses should be leveraged by compiler optimizations.

```
/*---outer inter-tile loop nest---*/
for (t1'=0;t1'<=N;t1'++){//DOSEQUENTIAL
  lb1=max(0,ceil(2*t1'-N,2));
  ub1=min(floor(N,2),t1');
  for (t2'=lb1+blockIdx.x; t2'<=ub1; t2'+=gridDim.x){//DOALL
    /*+++inner intra-tile loop nest+++*/
    /*---Code for shared memory coalesced loads---*/
    __syncthreads();//① barrier for the loads
    lb2=max(max(max(2,2*t1'),2*t2'+1),2*t1'-2*t2'+1);
    ub2=min(min(min(2*N,2*t1'+2),2*t2'+N+1),2*t1'-2*t2'+N+1);
    for (e1'=lb2;e1'<=ub2;e1'++){//DOSEQUENTIAL
      lb3=max(max(max(1,2*t2'),e1'-N),-2*t1'+2*t2'+e1'-1);
      ub3=min(min(min(N,2*t2'+1),e1'-1),-2*t1'+2*t2'+e1');
      for (e2'=lb3+threadIdx.x;e2'<=ub3;e2'+=blockDim.x){//DOALL
        j=e1'-e2'-(2*(t1'-t2')-1);
        i=e2'-(2*(t2')-1);
S:      A[j][i]=A[j][i-1]+A[j-1][i]+A[j-1][i-1];
      }
      __syncthreads();//② barrier for each intra-tile wavefront
    }
    /*---Code for shared memory coalesced stores---*/
    __syncthreads();//③ barrier for the stores
    /*+++++++++++++++++++++++++++++++++*/
  }
  /* Code for synchronizing blocks */
  __syncblocks();//④ barrier for each inter-tile wavefront
}
```

Figure 3.17: Tiled CUDA code of Figure 3.16

3.3.2 Parallelizing Tiled Loop Nests on GPUs

We describe a scheme for mapping sequential tiled loop nests to CUDA code on GPUs. Our scheme is the same as that supported by PLUTO [8] except tiles are mapped to thread blocks in a different way in order to achieve better load balance.

Tiles are mapped to thread blocks and individual loop iterations in a tile are mapped to the threads in a block. All inter-tile wavefronts are executed sequentially to satisfy inter-tile (or inter-block) dependences. Hence, the `syncblocks` macro as introduced at ④. The tiles in an $(m-1)$-D inter-tile wavefront are distributed over CUDA's grid of thread blocks of size `gridDim.x` × `gridDim.y` cyclically along two of the $m-1$ dimensions of the wavefront. This can cause load imbalance for large tiles since a wavefront has irregular boundaries. The tile coordinates in such a wavefront are "linearized" much like how the subscripts of a multi-dimensional array are. Then the tiles are mapped to a 1-D grid of thread blocks of size `gridDim.x` cyclically to achieve better load balance (with `gridDim.y`=1 always). As a result, all thread blocks in an inter-tile wavefront can be potentially executed in parallel but the tiles with in a block are always executed sequentially.

The loop iterations in a tile are distributed to a 3-D thread block of size `blockDim.x` × `blockDim.y` × `blockDim.z` cyclically. To improve memory coalescing, all data read by a tile are first loaded from device memory to shared memory at ① and all those written in a tile are stored back to device memory at ③. Like inter-tile wavefronts, all intra-tile wavefronts are executed sequentially. Hence, the `syncthreads` instruction is introduced at ②. The iterations in an intra-tile wavefront that are assigned to different threads can execute in parallel.

3.4 Compiler Framework

We have implemented our compiler techniques using a combination of the Clan polyhedral representation extractor, the Candl analyzer for dependences in loop, the PLUTO's polyhedral parallel tiling infrastructure and the CLooG code generator. Figure 3.18 shows the entire tool-chain that automatically translates sequential C loop nests into CUDA kernels. Our techniques are used in the modules highlighted by the dotted rectangular boxes with numbers that indicate the chapter in which the techniques are presented.

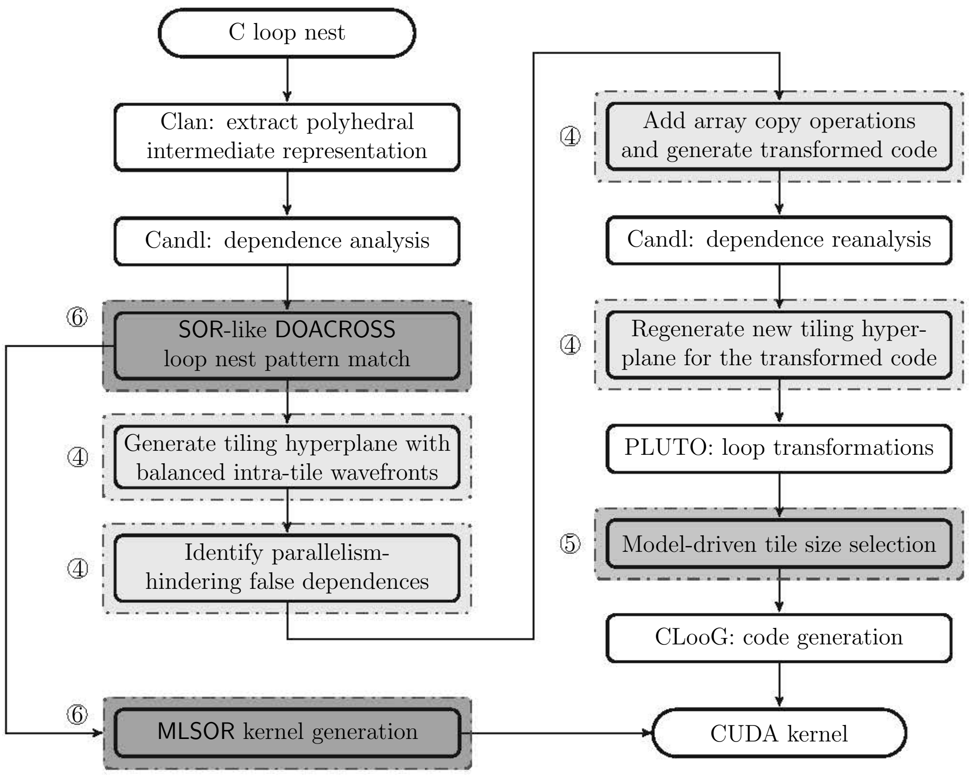

C loop nest
Clan: extract polyhedral intermediate representation
Candl: dependence analysis
⑥
SOR-like DOACROSS loop nest pattern match
④
Generate tiling hyperplane with balanced intra-tile wavefronts
④
Identify parallelism-hindering false dependences
⑥
MLSOR kernel generation
④
Add array copy operations and generate transformed code
Candl: dependence reanalysis
④
Regenerate new tiling hyperplane for the transformed code
PLUTO: loop transformations
⑤
Model-driven tile size selection
CLooG: code generation
CUDA kernel

Figure 3.18: The C-to-CUDA compiler framework.

We firstly use Clan [10] to translate some particular parts of high level program into the corresponding polyhedral representation and then introduce Candl as the dependence scanner and tester to compute dependences in loop nests. Accurate dependence analyses assist the compiler to determine whether the program is a SOR-like DOACROSS loop nest. If so, the MLSOR code generation pass will be called to directly translate programs into CUDA kernels, otherwise the compiler will enter the loop transformation passes that start with finding tiling hyperplanes. If parallelism-hindering false dependences exist, the compiler tries to eliminate them by array copy described in [130], and then regenerates the tiling hyperplanes and performs the subsequent loop transformations. At this point, the tile shape with optimized intra- and inter-tile parallelism is determined. Next, through model-driven tile size selection, the compiler can precisely estimate the execution time of loop nests with different tile sizes and thread organizations. In this pass, efficient tile sizes are chosen for CUDA kernel generation. Furthermore, coalescing data movement is seriously considered to relieve the burden of bandwidth [8]. Finally, we use CLooG with the extension to generate CUDA kernels from the tiling hyperplanes selected [9].

Chapter 4

Loop Tiling with Optimized Two-level Parallelism

In this chapter, we present our solution to maximize two-level parallelism of GPUs. The key techniques implemented in our compiler framework are to find appropriate tiling hyperplanes for intra-tile parallelism and eliminate parallelism-hindering false dependences for inter-tile parallelism. The research presented in this chapter was published in [32].

4.1 Introduction

An important open problem faced by the polyhedral model is how to synthesize a sequence of affine transformations from a huge space of valid solutions. To execute a tiled loop nest efficiently on a cluster of CPU nodes, one well-known communication-minimal approach is to find appropriate tiling hyperplanes so that the inter-tile communication is minimized [13, 120, 123]. Although a significant body of prior work focuses on optimizing hyperplane generation strategies and tiling heuristics [3, 44, 47, 58, 63, 75, 78], these existing solutions, once directly deployed for GPUs, lead to poor performance for some applications.

In GPUs, the degree of parallelism across the SMs and among the SPs in an SM is the key to improving performance [6, 50, 83]. To parallelize a tiled loop nest on GPUs, the wavefronts within a tile are pipelined for parallel execution to exploit intra-SM parallelism and the wavefronts across the tiles are pipelined for parallel execution to exploit inter-SM parallelism. By applying directly the CPUs-oriented communication-minimal approach [13, 120, 123, 126] to GPUs, severe load imbalance may occur due to pipeline fill-up and drain delay. In this work, we overcome this limitation by employing two new parallelism-exposing transformations for GPUs. This chapter makes the following contributions:

- We introduce an affine tiling framework that performs automatic C-to-CUDA parallelization from loop nests with uniform dependences. Our framework aims to maximize two levels of wavefront parallelism by reducing pipeline fill-up and drain delay: intra-tile, i.e., intra-SM parallelism by finding appropriate tiling hyperplanes and inter-tile, i.e., inter-SM parallelism by eliminating parallelism-hindering false dependences.

- We have compared our framework with PLUTO, an existing C-to-CUDA compiler [8], using eight stencil kernels and some hand-tuned code on two NVIDIA GPUs, C1060 and C2050. Compared to PLUTO, our speedups are 2 – 5.5X for these benchmarks. Compared to highly hand-optimized 1-D Jacobi (3 points), 2-D Jacobi (5 points), 3-D Jacobi (7 points) and 3-D Jacobi (27 points), our speedups, 1.17X, 1.41X, 0.97X and 0.87X with an average of 1.10X on C1060 and 1.24X, 1.20X, 0.86X and 0.95X with an average of 1.06X on C2050, are considered to be competitive.

4.2 Two-level Parallelism Transformations

In this section, we introduce our framework to tiling and parallelizing loop nests with uniform dependences to exploit both intra- and inter-SM parallelism on GPUs. In Section 4.2.1, we review the communication-minimal tiling transformations built for a cluster of CPU nodes [13, 123, 126] and see how they may lead to poor performance on GPUs due to long pipeline fill-up and drain delay. We overcome such load imbalance in two ways. In Section 4.2.2, we improve intra-tile parallelism by finding better tiling hyperplanes in the polyhedral model. In Section 4.2.3, we improve inter-tile parallelism by eliminating parallelism-hindering false dependences.

4.2.1 Communication-minimal Transformations

To preserve the dependences in a given loop nest, a legal tiling must satisfy Theorem 3.3.

Consider the example in Figure 3.4 with the five dependences listed in Table 3.1. To construct its tiling hyperplanes $\Phi_S \in \mathbb{Z}^{2\times 2}$, we apply Theorem 3.3 to the example. There are many valid solutions with different degrees of parallelism, which will have varying degrees of impact on performance. For the purposes of minimizing the inter-tile communication, $\Phi_S = \left(\begin{smallmatrix}1 & 0\\ 1 & 1\end{smallmatrix}\right)\left(\begin{smallmatrix}j\\ i\end{smallmatrix}\right) + \left(\begin{smallmatrix}0\\ 0\end{smallmatrix}\right)$, which is illustrated in Figure 4.1, is recommended [13, 123, 126]. In the figure, different instances of statement S are represented by dots. In the larger tile on the right, the solid/dashed/dotted arrows denote flow/anti/output dependences between loop iterations. In the center, the three dependences between tiles are also depicted (without distinguishing flow from false dependences).

Given this tiling transformation, we can map tiles to thread blocks and loop iterations in a tile to the threads in a block by exploiting two levels of wavefront

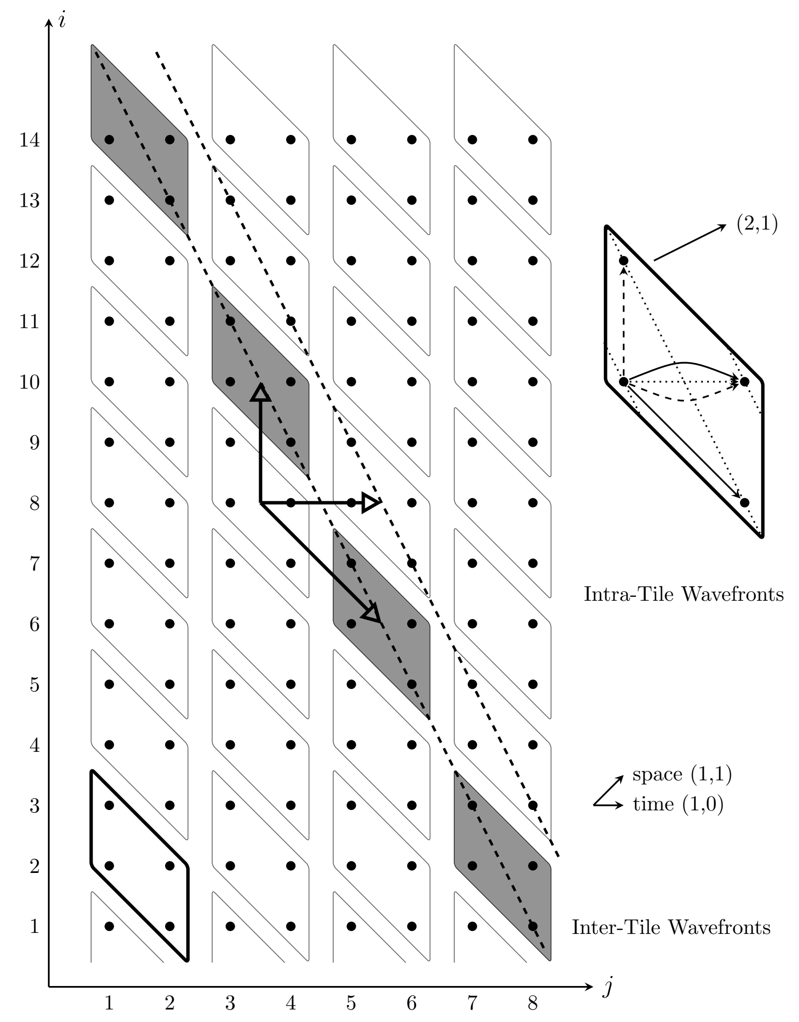


Figure 4.1: Iteration space tiling of the loop nest in Figure 3.4 with $\Phi_S(\vec{i_S}) = \left(\begin{smallmatrix}1 & 0\\ 1 & 1\end{smallmatrix}\right)\left(\begin{smallmatrix}j\\ i\end{smallmatrix}\right) + \left(\begin{smallmatrix}0\\ 0\end{smallmatrix}\right)$. The tile size used is 2×2.

parallelism. In the CUDA code given in Figure 4.2, the first two loops enumerate all the tiles and the inner two the loop iterations within a tile.

To satisfy the three inter-tile data dependences, the tiles are grouped into wavefronts, which are pipelined for parallel execution along the direction $(1, 1)$ in the iteration space representing the tiles. One wavefront is highlighted with its tiles depicted in gray. To exploit inter-tile parallelism, different wavefronts are executed sequentially in a pipelined manner while the tiles in the same wavefront can be executed in parallel on different SMs. Therefore, the SMs are synchronized every

```
/*---outer inter-tile loop nest---*/
for(t1'=0;t1'<=floor(2*T+I,256);t1'++){ //DOSEQUENTIAL
  lb1=max(ceil(t1',2),ceil(256*t1'-T,256));
  ub1=min(floor(T+I,256),floor(256*t1'+I+255,512),t1');
  for(t2'=lb1+blockIdx.x;t2'<=ub1;t2'+=gridDim.x){ //DOALL
    /*+++++++++++++++++++++++++++++++++*/
    /*---inner intra-tile loop nest---*/
    /*---Code for shared memory coalesced loads---*/
    __syncthreads();// barrier for the loads
    lb2=max(3,256*t1',256*t2'+1,512*t2'-I,512*t1'-512*t2'+1);
    ub2=min(2*T+I,256*t1'+510,512*t2'+509,256*t2'+T+255,512*t1'-512*t2'+I+510)
        ;
    for(e1'=lb2;e1'<=ub2;e1'++){ //DOSEQUENTIAL
      lb3=max(ceil(e1'+1,2),256*t2',e1'-T,-256*t1'+256*t2'+e1'-255);
      ub3=min(floor(e1'+I,2),256*t2'+255,e1'-1,-256*t1'+256*t2'+e1');
      for(e2'=lb3+threadIdx.x;e2'<=ub3;e2'+=blockDim.x){ //DOALL
S:        A[-e1'+2*e2']=0.5*(A[-e1'+2*e2']+A[-e1'+2*e2'+1]);
      }
      __syncthreads();
    }
    /*---Code for shared memory coalesced stores---*/
    __syncthreads();// barrier for the stores
    /*+++++++++++++++++++++++++++++++++*/
  }
  /* Code for synchronizing blocks */
  __syncblocks();// barrier for each inter-tile wavefront
}
```

Figure 4.2: CUDA kernel generated for the code in Figure 3.4 using the communication-minimal tiling hyperplanes in Figure 4.1.

time after a wavefront has been executed.

Like inter-tile wavefronts, the loop iterations in a tile are also scheduled to exploit intra-tile wavefront parallelism, as illustrated in the right part of Figure 4.1. To satisfy the five intra-tile dependences, the wavefronts in a tile are pipelined along $(2, 1)$ and the loop iterations in the same wavefront can be executed in parallel in different threads on the SPs (in an SM). Due to coalesced loads and stores used, these threads are synchronized at the end of each wavefront.

However, minimizing inter-tile communication alone often leads to long pipeline fill-up and drain delay for both intra- and inter-tile wavefronts, as can be visualized in Figure 4.1. The resulting load imbalance, which may be insignificant for a cluster of CPUs, can be problematic for GPUs that rely on massive fine-grained parallelism to achieve high performance. To alleviate this problem, Section 4.2.2 focuses on improving intra-tile parallelism while Section 4.2.3 focuses on improving inter-tile parallelism.

4.2.2 Improving Intra-tile Parallelism

We improve intra-tile parallelism by embedding parallelism-enhancing constraints in the polyhedral model so that better tiling hyperplanes can be found.

Definition 4.1 (Balanced Intra-tile Wavefronts) *The wavefronts for a statement S in a tile are* balanced *if they are pipelined along the normal of a tiling hyperplane in Φ_S.*

In practice, this implies that statement S will be executed the same number of times in all its intra-tile wavefronts.

Theorem 4.1 (Tiling Hyperplanes with Balanced Intra-tile Wavefronts) *Consider a loop nest with DDG $G = (V, E)$. Let $\Phi_{S_1}, \Phi_{S_2}, \ldots$ be its legal tiling hyperplanes. If there exists a $k \in [1, m]$ such that*

$$\phi_S^k(\vec{i}_t) - \phi_S^k(\vec{i}_s) \geq 1, \langle \vec{i}_s, \vec{i}_t \rangle \in \mathcal{P}_{e_{S,S}} \tag{4.1}$$

for all self dependences $e_{S,S} \in E$, then the intra-tile wavefronts for every statement S are balanced.

PROOF. Follows from Theorem 3.3 and Definition 4.1. □

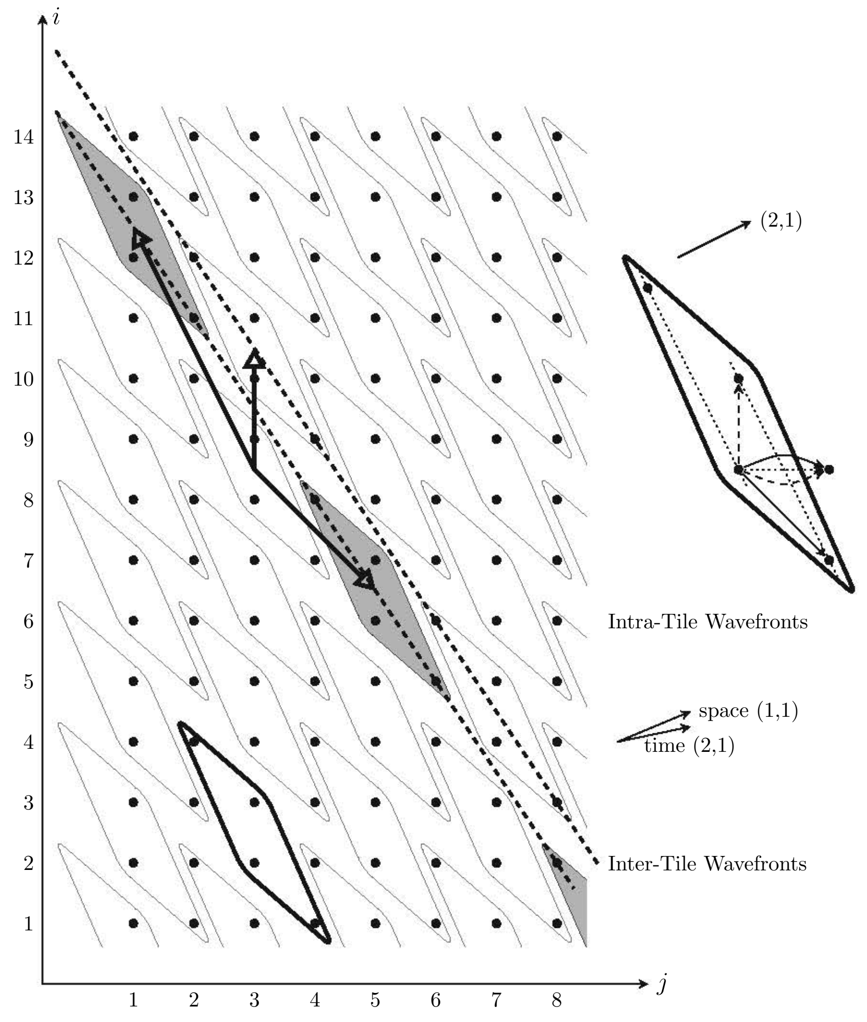


Figure 4.3: Iteration space tiling of the loop nest in Figure 3.4 with $\Phi_S^{\text{intra}}(\vec{i}_S) = \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} j \\ i \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$. The tile size used is 2×2.

By combining the constraints given in Equation 4.1 with those given in Equation 3.16, $\Phi_S^{\text{intra}} = \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} j \\ i \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ is found. As illustrated in Figure 4.3, statement S is executed the same number of times in all intra-tile wavefronts. Therefore, the SPs executing S in a tile are sufficiently utilized with perfect load balance.

As shown in Algorithm 4.1, the tiling hyperplanes can be found by applying the techniques described in [13, 123, 126] except that both Equations 3.16 and 4.1 must now be taken into account. This ensures that a loop nest is tiled not

```
A0[I+1]=A[I+1];
for (j=1;j<=J;j++){
   for (i=2;i<=I;i++){
S0:    A0[i]=A[i];
   }
   for (i=1;i<=I;i++){
S1:    A[i]=0.5*(A[i]+A0[i+1]);
   }
}
```

Figure 4.4: Transformed code by eliminating anti dependence $(0, 1)$ via array copying for the loop nest given in Figure 3.4.

only to incur the least inter-tile communication but also to exhibit balanced intra-tile wavefronts for pipelined execution. By setting $k = 1$ in Equation 4.1, we will generate a set of loops to execute a statement in a tile so that the outermost one iterates over its wavefronts and the remaining loops over its different instances. In GPUs, each thread block is organized as a 3-D array of threads. In our current implementation, only the innermost three loops in a loop nest are tiled.

4.2.3 Improving Inter-tile Parallelism

By comparing Figure 4.1 obtained with $\Phi_S = \left(\begin{smallmatrix} 1 & 0 \\ 1 & 1 \end{smallmatrix}\right)\left(\begin{smallmatrix} j \\ i \end{smallmatrix}\right) + \left(\begin{smallmatrix} 0 \\ 0 \end{smallmatrix}\right)$ and Figure 4.3 obtained with $\Phi_S^{\text{intra}} = \left(\begin{smallmatrix} 2 & 1 \\ 1 & 1 \end{smallmatrix}\right)\left(\begin{smallmatrix} j \\ i \end{smallmatrix}\right) + \left(\begin{smallmatrix} 0 \\ 0 \end{smallmatrix}\right)$, we find that the anti dependence $(0, 1)$ affects the shapes of tiles used. In particular, the first hyperplane has changed from $(1, 0)$ to $(2, 1)$ in order to achieve balanced intra-tile wavefronts when $k = 1$ (Theorem 4.1). Unfortunately, the price paid is that the tiles become more slanted, reducing the degree of inter-tile parallelism exposed, which can be easily observed by comparing the inter-tile wavefronts in both cases.

In this section, we describe how to eliminate some false (anti and output) dependences in a loop nest by introducing array copy operations. Once they are

Algorithm 4.1: Generating tiling hyperplanes

```
Input: Data dependence graph $G = (V, E)$
Output: Tiling hyperplanes $\Phi_{S_1}, \Phi_{S_2}, \ldots$
1  for each $e_{S_s,S_t} \in E$ do
2      Build legality constraints Equation 3.16 and eliminate all Farkes multipliers ;
3      Add constraints for achieving balanced intra-tile wavefronts Equation 4.1 by setting
       $k = 1$ and eliminate all Farkes multipliers;
4      Build bounding function constraints [13] and eliminate all Farkes multipliers;
5      Aggregate constraints from Step 2, Step 3 and Step 4 into constraints set $C_e$;
6  end for
7  Let $C = \bigcup_{e \in E} C_e$;
8  Let $E_c = \emptyset$;
9  while $H_S^{\perp} \neq 0 \vee E_c \neq E$ do
10     Iteratively compute statement $S$'s independent lexicographic minimal solutions $H_S$ to
       $C$ with linear independence constraints [13] ;
11     if no solutions were found then
12         Remove dependences between two strongly-connected components in the DDG;
13     end if
14     Compute $E_c$: dependences satisfied by solutions of Step 10;
15     Let $C = \bigcup_{e \in E - E_c} C_e$;
16     Compute the $H_S$'s sub-space orthogonal $H_S^{\perp} = I - H_S^T (H_S H_S^T)^{-1} H_S$ for judging null
       space [76];
17 end while
```

eliminated in a loop nest, the transformed loop nest can be parallelized with tiles of more “regular” shapes, resulting in improved inter-tile parallelism and reduced bank conflicts. At the same time, balanced intra-tile wavefronts can still be retained.

Definition 4.2 (Parallelism-hindering False Dependence) *Consider a loop nest with $G = (V, E)$ as its DDG. A false dependence $e \in E$ is said to be a*

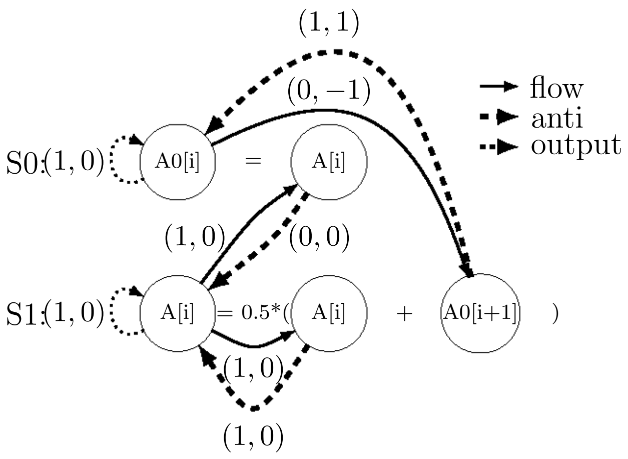


Figure 4.5: DDG for the loop nest given in Figure 4.4 with each dependence labelled by its distance vector.

parallelism-hindering false dependence *if some constraints generated for e according to Equations 3.16 and 4.1 are* not *redundant with respect to the set of constraints generated for the dependences in* $E - \{e\}$ *generated according to Equation 3.16 and Equation 4.1.*

Consider our example with its five dependences listed in Table 3.1 and illustrated in Figures 4.1 and 4.3. There are three false dependences. The output dependence $(1, 0)$ and the anti dependence $(1, 0)$ are redundant due to the existence of the flow dependence $(1, 0)$. However, the anti dependence $(0, 1)$ is a parallelism-hindering false dependence. This false dependence should be eliminated to improve the load balance during the parallel execution of inter-tile wavefronts.

Following [130], we eliminate the anti dependence $(0, 1)$ by introducing a temporal array A0 to perform some array copy operations. The resulting code is given in Figure 4.4. To ensure that the anti dependence is not violated, some values of A[i] are copied into A0[i] so that they can be read later when needed. In the transformed code, there are a total of eight dependences as depicted in Figure 4.5.

There are now two statements, $S0$ and $S1$. To find tiling hyperplanes Φ_{S0}

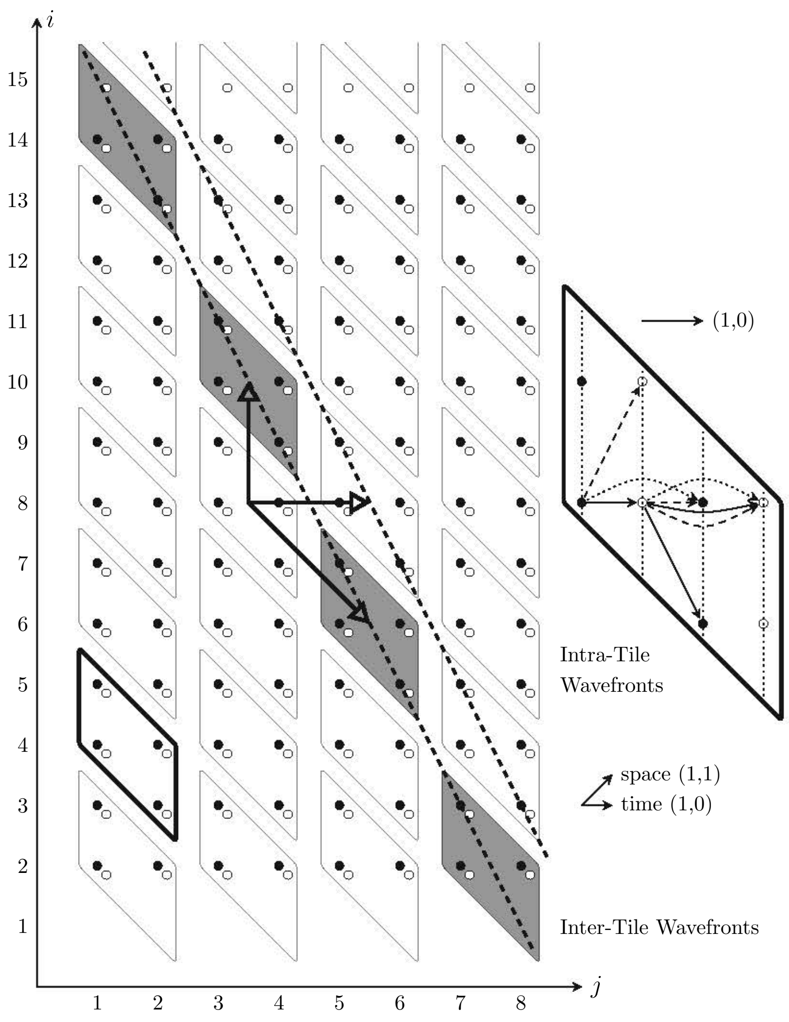


Figure 4.6: Iteration space tiling of the loop nest in Figure 4.4 using $\Phi_{S0} = \left(\begin{smallmatrix}1&0\\1&1\end{smallmatrix}\right)\left(\begin{smallmatrix}j\\i\end{smallmatrix}\right) + \left(\begin{smallmatrix}0\\0\end{smallmatrix}\right)$ and $\Phi_{S1} = \left(\begin{smallmatrix}1&0\\1&1\end{smallmatrix}\right)\left(\begin{smallmatrix}j\\i\end{smallmatrix}\right) + \left(\begin{smallmatrix}0\\1\end{smallmatrix}\right)$. The tile size used is 2×2. The eight dependences depicted inside the larger tile on the right are from Figure 4.5.

and Φ_{S1} with balanced intra-tile wavefronts for both statements, we solve Equations 3.16 and 4.1 for all the eight dependences in the transformed loop nest to obtain: $\Phi_{S0} = \left(\begin{smallmatrix}1&0\\1&1\end{smallmatrix}\right)\left(\begin{smallmatrix}j\\i\end{smallmatrix}\right) + \left(\begin{smallmatrix}0\\0\end{smallmatrix}\right)$ and $\Phi_{S1} = \left(\begin{smallmatrix}1&0\\1&1\end{smallmatrix}\right)\left(\begin{smallmatrix}j\\i\end{smallmatrix}\right) + \left(\begin{smallmatrix}0\\1\end{smallmatrix}\right)$. As a result, the transformed loop nest is tiled as shown in Figure 4.6. By comparing with Figure 4.3, we find that the tiles are now more regularly shaped, yielding better inter-tile parallelism. In addition, the intra-tile wavefronts are also balanced.

Furthermore, making tile shapes regular also tends to reduce bank conflicts in-

```
A0[I+1]=A[I+1];
/*---outer inter-tile loop nest---*/
for(t1'=0;t1'<=floor(2*T+I,256);t1'++){ //DOSEQUENTIAL
  for(t2'=max(ceil(t1',2),ceil(256*t1'-T,256))+blockIdx.x;
      t2'<=min(floor(T+I,256),floor(256*t1'+I+255,512),t1');t2'+=gridDim.x){
         //DOALL
    /*+++++++++++++++++++++++++++++++++*/
    /*---inner intra-tile loop nest---*/
    /*---Code for shared memory coalesced loads---*/
    __syncthreads();// barrier for the loads
    for(e1'=max(1,256*t1'-256*t2',256*t2'-I);e1'<=min(T,256*t2'+254,256*t1'-256*t2'
        +255);e1'++){ //DOSEQUENTIAL
      for(e2'=max(256*t2'+1,e1'+2)+threadIdx.x;e2'<=min(256*t2'+256,e1'+I);e2'+=
          blockDim.x) //DOALL
S0:       A0[-e1'+e2']=A[-e1'+e2'];
      __syncthreads();
      for(e2'=max(256*t2'+1,e1'+2)+threadIdx.x;e2'<=min(256*t2'+256,e1'+I+1);e2'
          +=blockDim.x) //DOALL
S1:       A[-e1'+e2'-1]=0.5*(A[-e1'+e2'-1]+A0[-e1'+e2']);
      __syncthreads();
    }
    /*---Code for shared memory coalesced stores---*/
    __syncthreads();// barrier for the stores
    /*+++++++++++++++++++++++++++++++++*/
  }
  /* Code for synchronizing blocks */
  __syncblocks();// barrier for each inter-tile wavefront
}
```

Figure 4.7: CUDA kernel generated for the code in Figure 4.4 using the tiling hyperplanes in Figure 4.6.

curred during the execution of a tile. For loop nests with uniform dependences, the number of such bank conflicts can be easily estimated. This fact can be exploited in tile size selection to tune for better tiling hyperplanes (see Chapter 5). Consider our example again. In Figures 4.1 and 4.3, where the anti dependence $(0, 1)$ is still presented, the wavefronts in a tile are pipelined along $(2, 1)$. As a result, each

Algorithm 4.2: Eliminating parallelism-hindering false dependences

```
    Input: A loop nest $\mathcal{L}$ with uniform dependences
    Output: Parallelism-hindering false dependences set $E_p$ of $\mathcal{L}$
1   Set parallelism-hindering false dependences set $E_p = \emptyset$;
2   Compute the solutions $H_S$ by Algorithm 4.1;
3   for each false dependence $e_f \in E$ do
4       Let $E^{'} = E - e_f$;
5       Compute the solutions $H_S^{'}$ based on $E^{'}$;
6       if $H_S^{'} \neq H_S$ then
7           $E_p = E_p \cup e_f$;
8       end if
9   end for
10  Transform $\mathcal{L}$ into $\mathcal{L}'$ by eliminating these false dependences using array copy
    operations [130];
```

wavefront can be identified by a line $i = -2j + c$ for some c. As the values of i are discontinuous with a gap of 2, stride-2 memory accesses are made. When one warp executes a wavefront, two-way bank conflicts will occur. In Figure 4.6, where the anti dependence $(0, 1)$ has been removed, the memory accesses in each wavefront are now stride-1. As a result, no bank conflicts can occur.

The CUDA code generated using the tiling hyperplanes illustrated in Figure 4.6 is given in Figure 4.7. There are two statements $S0$ and $S1$. The wavefronts for each statement in each tile are pipelined for parallel execution. Both are synchronized to satisfy their inter-statement dependences.

As shown in Algorithm 4.2, once parallelism-hindering false dependences in a loop nest are identified, we can apply the techniques described in [130] to eliminate them. This allows the transformed loop nest to be parallelized using tiling hyperplanes with better inter-tile parallelism.

4.3 Experiments

4.3.1 Implementation

We have implemented our compiler techniques using the compiler framework introduced in Section 3.4. Our techniques are used in the modules highlighted by the dotted rectangular boxes with ④.

We identify parallelism-hindering false dependences in a loop nest after Candl's dependence analysis. We then generate the transformed loop nest without these false dependences by using the techniques described in [130]. When searching for tiling hyperplanes with balanced intra-tile wavefronts and performing subsequent loop transformations, we make use of PLUTO's polyhedral implementation. Finally, after tile size selection that is introduced in the following chapter, we use CLooG to generate CUDA code from the tiling hyperplanes and size selected.

4.3.2 Result

Benchmark	Max Loop Depth	Innermost DOACROSS Loop?	Perfectly Nested?	Our Approach		Input Problem Size
				False Dependence Elimination	Balanced Intra-Tile Wavefronts	
1-D Jacobi-3 (3 points)	2			✓	✓	65536*65536
2-D Jacobi-5 (5 points)	3			✓	✓	1000*4096*4096
3-D Jacobi-7 (7 points)	4			✓	✓	256*256*256*256
3-D Jacobi-27 (27 points)	4			✓	✓	256*256*256*256
1-D SOR-3 (3 points)	2	✓	✓		✓	65536*65536
2-D SOR-5 (5 points)	3	✓	✓		✓	1000*4096*4096
2-D FDTD	3			✓	✓	1000*4096*4096
2-D Heat (7 points)	3			✓	✓	1000*4096*4096

Table 4.1: Benchmarks and their characteristics.

We have selected eight benchmarks on stencil computations, as listed in Table 4.1. These include 1-D, 2-D and 3-D Jacobi solvers from the Rodinia benchmark suite and the Circular Queue toolkit, 2-D FDTD as well as 1-D and 2-D SOR solvers from Polybench, and the 2-D Heat solver from the Chombo toolkit. For the two 3-D Jacobi solvers, only their innermost three loops are tiled and parallelized. Our evaluation with small benchmarks is preliminary. In future work, more extensive benchmarking will be conducted.

We have carried out our experiments on two NVIDIA GPUs, C1060 and C2050. For each benchmark, the problem size used is given in Table 4.1. For all the benchmarks, we have succeeded in finding tiling hyperplanes with balanced intra-tile wavefronts. For all except the two SOR solvers, some parallelism-hindering false dependences have been eliminated to expose more inter-tile parallelism.

We evaluate our framework by comparing with PLUTO [8], an existing C-to-CUDA translator, and some hand-optimized CUDA code. We compare our framework and PLUTO across all the eight benchmarks to highlight the performance advantages of the two new optimizations introduced for GPUs in this chapter. We compare the CUDA code generated for the four Jacobi solvers with their highly hand-tuned CUDA kernels to demonstrate further the strengths and limitations of our compiler optimizations.

4.3.2.1 Compared with PLUTO

Both our framework and PLUTO generate tiling hyperplanes for loop nests automatically. In both cases, the best tile sizes for tiling hyperplanes are determined empirically by using a cost model from [30].

Figure 4.8 shows the speedups achieved by our framework over PLUTO. There are two configurations tested in our framework, depending on how the two optimizations listed in Table 4.1 are used. "Intra" means that the optimization for achieving balanced intra-wavefronts is turned on. "Intra+Inter" means that in addition to the "intra" optimization, the "inter" optimization for eliminating parallelism-hindering false dependences to achieve better inter-tile parallelism is also turned on. There are no such false dependences to eliminate in the two SOR solvers. Therefore, the speedups in the two configurations in either benchmark are identical. The "Inter" optimization, if used alone, is not very beneficial. Figure 4.9 shows the speedups on C2050.

Our experimental results demonstrate the performance advantages of the "intra" and "inter" optimizations. When the "intra" optimization alone is used, the speedups range from 2.05X – 5.41X with an average of 3.32X on C1060 and from 2.07X – 5.43X with an average of 3.27X on C2050. As the intra-tile wavefronts are balanced, the degree of intra-SM parallelism achieved in our framework is higher.

When the "inter" optimization is also turned on, there are performance improvements across all the six benchmarks (with some parallelism-hindering false dependences to remove) on both C1060 and C2050. The speedups range from 3.10X – 5.52X with an average of 3.98X on C1060 and from 3.20X – 5.48X with an average of 4.07X on C2050. By eliminating some false dependences, tiles with more regular shapes can be used. This optimization has two benefits. First, better inter-tile parallelism is achieved. Second, the number of bank conflicts incurred during the execution of a tile is reduced. On the other hand, this optimization also has its associated costs. False dependences are eliminated by introducing new temporary arrays and new copy operations on these arrays. Such code rewriting can increase the number of instructions executed and inter-statement synchronization operations used. Overall, the benefits outweigh the costs.

4.3.2.2 Compared with Hand-tuned Code

We compare the CUDA code generated by our framework for the four Jacobi benchmarks with the CUDA code manually obtained using the Circular Queue approach

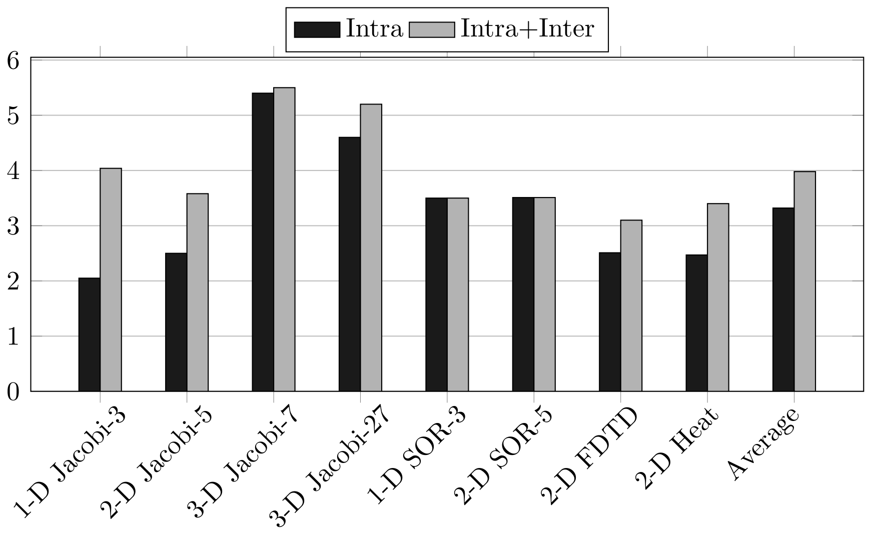


Figure 4.8: Speedups of our framework (with the two optimization configurations) over PLUTO on C1060.

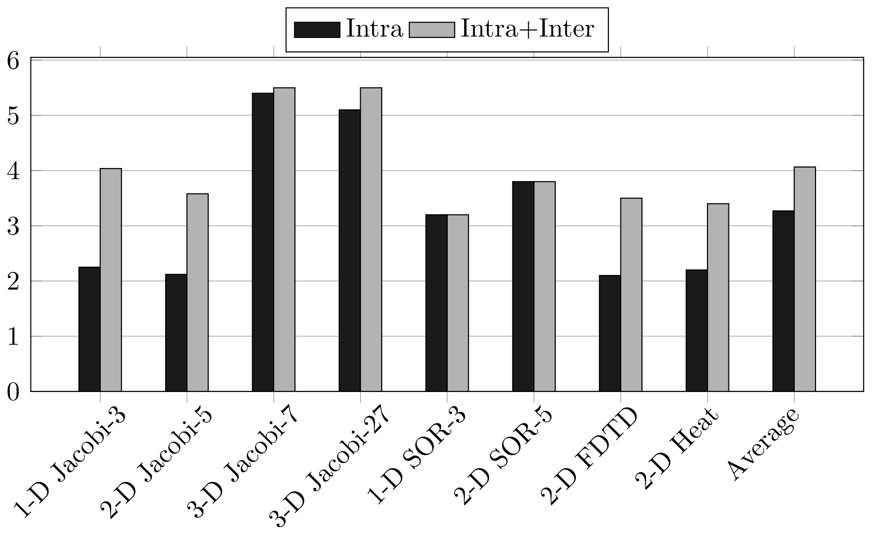


Figure 4.9: Speedups of our framework (with the two optimization configurations) over PLUTO on C2050.

[27]. Circular Queue improves data locality by streaming tiled data. It strives to exploit the maximum amount of parallelism among the computations executed during every iteration of the outermost (time) loop but ignores the data reuse across its different iterations. Taking 2-D Jacobi (5 points) as an example, Circular Queue loads data line by line with 5 times reuse of each non-border point (four neighbour-

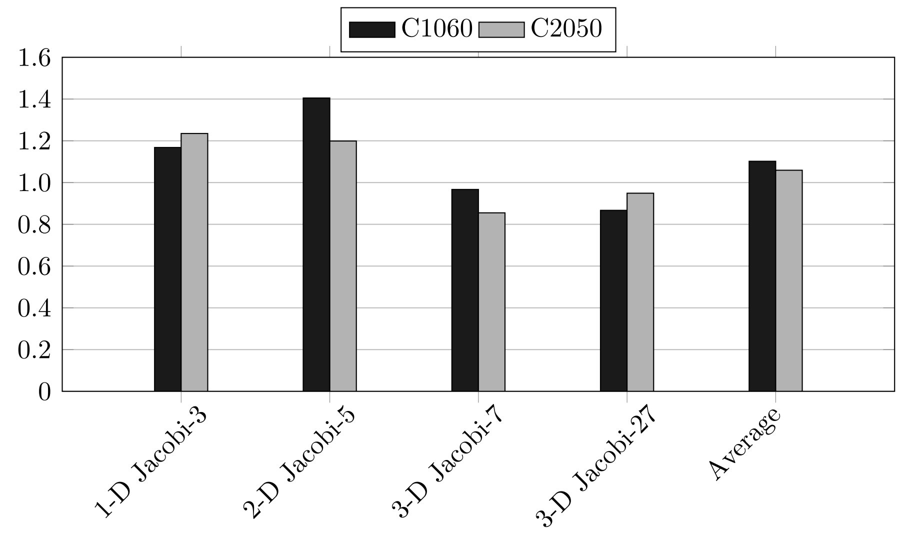


Figure 4.10: Speedups over Circular Queue for Jacobi.

ing points and itself). Unlike Circular Queue, our approach trades off parallelism for data reuse due to pipelined execution of wavefronts. If 2-D Jacobi (5 points) on our scheme is iterated three times along the time dimension, each non-border point is reused 15 times. Since tiles are skewed, the reuse of the points on the borders may be lower than non-border points. In the meanwhile skewed tiles impact intra- and inter-tile parallelism. The method of predicting execution time is introduced in Chapter 5, which illustrates how to make a tradeoff between parallelism and data reuse.

As shown in Figure 4.10, our approach is competitive. For 1-D Jacobi (3 points), 2-D Jacobi (5 points), 3-D Jacobi (7 points) and 3-D Jacobi (27 points), our speedups are 1.17X, 1.41X, 0.97X and 0.87X with an average of 1.10X on C1060 and 1.24X, 1.20X, 0.86X and 0.95X with an average of 1.06X on C2050. For the 1-D and 2-D solvers, we achieve better performance due to better data reuse exploited. For the two 3-D solvers, only the innermost three loops are parallelized. Our CUDA kernels are similar to the ones generated by Circular Queue. Circular Queue delivers better performance due to some hand optimizations. For example, simpler loop bounds are used, resulting in fewer instructions to be executed. The amount of data transferred between device memory and shared memory is meticulously calculated. Finally, memory accesses are better coalesced.

4.4 Conclusion

In this chapter, we present a novel tiling technique that contains generating tiling with hyperplanes balanced intra-tile parallelism and eliminating parallelism-hindering false dependences. These techniques are developed based on a dependence-preserving parallelization scheme, and implemented in our automatic code generating framework. Our experimental results show that our approach is promising. Compared to highly hand-optimized 1-D Jacobi (3 points), 2-D Jacobi (5 points), 3-D Jacobi (7 points) and 3-D Jacobi (27 points), our speedups, 1.17X, 1.41X, 0.97X and 0.87X with an average of 1.10X on C1060 and 1.24X, 1.20X, 0.86X and 0.95X with an average of 1.06X on C2050, are considered to be competitive. Compared to existing automatic compiler PLUTO, our speedups are 2 – 5.5X for eight benchmarks.

Chapter 5

Model-driven Tile Size Selection

This chapter presents a model-driven approach to automating tile size selection. Our results show that our framework can find the tile sizes for these representative loop nests, especially DOACROSS loop nests, to achieve performances close to the best observed for a range of problem sizes tested. The research presented in this chapter was published in [30].

5.1 Introduction

DOALL loop nests are tiled to exploit DOALL parallelism and data locality on GPUs. Unlike DOALL loop nests, DOACROSS loop nests must be skewed first to ensure that the subsequent tiling transformations preserve the loop-carried dependences. Furthermore, performing skewing and tiling allows wavefront parallelism to be exploited both across the tiles and within a tile. Tile size selection is more complex for DOACROSS than DOALL loop nests due to parallelism-inhibiting loop-carried dependences and intricate interactions among the GPU architectural constraints, that may impact the inter- and intra-tile parallelism. As discovered in this work regarding the best tile granularity used, parallelizing DOACROSS loop nests requires much coarser parallelism than DOALL loop nests. Thus, it is not practical to rely on the user to pick the right tile sizes to optimize code through improving processor utilization and reducing synchronization overhead. Existing tile size techniques proposed for caches in CPU architectures do not apply [51, 135].

This chapter makes the following contributions:

- We present (for the first time) a model for estimating the execution times of tiled DOACROSS loop nests running on GPUs (Section 5.3);
- We introduce a model-driven framework to automate tile size selection for tiled DOACROSS loop nests running on GPUs (Section 5.4);
- We evaluate the accuracy of our model using representative 1-D, 2-D and 3-D SOR solvers and show that the tile sizes selected lead to the performances close to the best observed for a range of problem sizes tested (Section 5.5).

5.2 Parallelization of DOACROSS loop nests on GPUs

In this section, we briefly describe a scheme for mapping sequential DOACROSS loop nests to CUDA code on GPUs. Our illustrating example is a 1-D SOR-like solver shown in Figure 3.2(b).

5.2.1 Loop Transformations

According to Equation 3.19, parallelizing an n-D DOACROSS loop nest consists of mapping it into a $2m$-D loop nest by $\Gamma'_S : \mathbb{Z}^n \mapsto \mathbb{Z}^{2m}$.

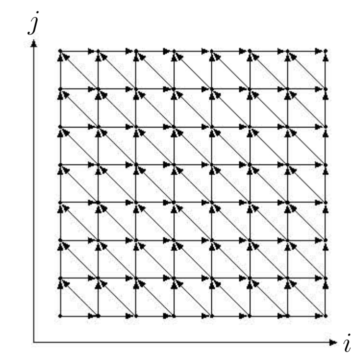
$$\Theta_S(\vec{e}) = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & 1 \end{pmatrix}_{m\times m} \begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_m \end{pmatrix}, \Theta_T(\vec{t}) = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & 1 \end{pmatrix}_{m\times m} \begin{pmatrix} t_1 \\ t_2 \\ \vdots \\ t_m \end{pmatrix} \quad (5.1)$$

The mapping process for the 1-D SOR solver in Figure 3.2(b) is illustrated in Figure 5.1. The mapping Γ' [126] is realized by composing a loop skewing Φ_S, a loop tiling $\Phi_T = \lfloor \frac{\Phi_S(\vec{i}_S)}{\vec{s}} \rfloor$ and another loop wavefront scheduling Θ_T and Θ_S. First, the iteration space of $\mathcal{I}$ is skewed by a unimodular transformation Φ_S. Second, the skewed iteration space is tiled into m-D rectangles of size $s_1 \times \cdots \times s_m$ by Φ_T. Φ_S is chosen to guarantee the legality of tiling so that all loop-carried dependences in $\mathcal{L}$ are preserved by Theorem 3.3. Finally, another skewing transformation Θ is applied to the iteration spaces of both sets of loop nests to expose wavefront parallelism across the tiles and within a tile. In either loop nest, the first loop is sequential and the remaining $m-1$ loops are DOALL. We will speak of inter-tile wavefronts and intra-tile wavefronts (as shown in Figure 5.1).

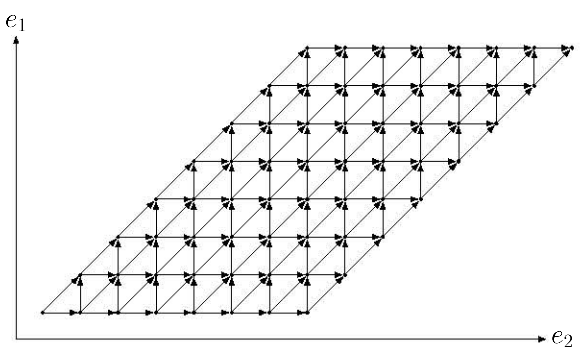
5.2.2 Mapping to GPUs

Figure 5.2 gives the CUDA code for the 1-D SOR solver parallelized as shown in Figure 5.1 using the mapping strategy described in Section 3.3.2. Tiles are mapped to thread blocks and individual loop iterations in a tile are mapped to the threads in a block. All inter-tile wavefronts are executed sequentially to satisfy inter-tile dependences with `syncblocks` synchronization. 2-D tiles are re-mapped to a 1-D grid of thread blocks of size `gridDim.x` cyclically to achieve better load balance.

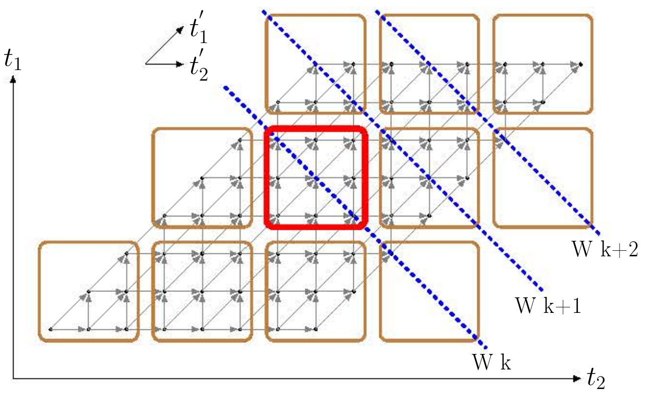
The loop iterations in a tile are distributed to a 3-D thread block of size `blockDim.x` $\times$ `blockDim.y` $\times$ `blockDim.z` cyclically. Coalescing access de-



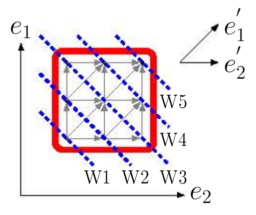
(a) Orignal iteration space



(b) Skewed iteration space with $\Phi_S(\vec{i}_S) = \binom{e_1}{e_2} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \binom{j}{i} + \binom{0}{0}$



(c) Tiling and inter-tile wavefront



(d) Intra-tile wavefront

Figure 5.1: Exploiting wavefront parallelism for 1-D SOR on GPUs.

vice memory followed by `syncthreads` synchronization ① and ③ is necessary to optimize utilization of bandwidth. The iterations in an intra-tile wavefront that are assigned to different threads can execute in parallel with data synchronization at ②.

5.3 Execution Time Modeling

We model the execution time of a tiled DOACROSS loop nest running on GPUs in terms of some performance-impacting parameters. This model is used in Section 5.4 to automate tile size selection. Initially, we assume that all tiles (including those at

```
/*---outer inter-tile loop nest---*/
for($t_1'$=$L_{t_1'}$;$t_1'$<$U_{t_1'}$;$t_1'$++) { //DOSEQUENTIAL
  for($t_2'$=$L_{t_2'}$($t_1'$)+blockIdx.x;$t_2'$<$U_{t_2'}$($t_2'$);$t_2'$+=gridDim.x) { //DOALL
    /*+++inner intra-tile loop nest+++*/
    /*---Code for shared memory coalesced loads---*/
    __syncthreads();//① barrier for the loads
    for($e_1'$=$L_{e_1'}$($t_1'$,$t_2'$);$e_1'$<$U_{e_1'}$($t_1'$,$t_2'$);$e_1'$++) { //DOSEQUENTIAL
      for($e_2'$=$L_{e_2'}$($t_1'$,$t_2'$,$e_1'$)+threadIdx.x;$e_2'$<$U_{e_2'}$($t_1'$,$t_2'$,$e_1'$);$e_2'$+=blockDim.x) { //
          DOALL
        i2=h($t_1'$,$t_2'$,$e_1'$,$e_2'$);
        A[i2]=(A[i2-1]+A[i2]+A[i2+1])/3;
      }
      __syncthreads();//② barrier for each intra-tile wavefront
    }
    /*---Code for shared memory coalesced stores---*/
    __syncthreads();//③ barrier for the stores
    /*++++++++++++++++++++++++++++++++*/
  }
  /* Code for synchronizing blocks */
  __syncblocks();//④ barrier for each inter-tile wavefront
}
```

Figure 5.2: CUDA code for 1-D SOR on GPUs.

the borders) are full. We consider first intra-tile execution (Section 5.3.1) and then inter-tile execution (Section 5.3.2). In Section 5.3.3, we describe how to estimate the parameters used. In Section 5.3.4, we discuss briefly how to mitigate the effects of border tiles on performance.

5.3.1 Intra-tile Execution

This section focuses on estimating the execution time, $T_{\mathcal{TILE}}$, for a single (full) tile, denoted $\mathcal{TILE}$. As shown in Equation 3.19 and Figures 5.1 and 5.2, the loop iterations in a tile are indexed by $(e_1', \dots, e_m')$, where e_1' enumerates all intra-wavefronts within the tile. As illustrated in Figure 5.1, $T_{\mathcal{TILE}}$, which can be broken down into the time on loading the input data at ①, the time on executing the tile, and the time on storing the results back ③, is approximated by:

$$T_{\mathcal{TILE}} \;=\; \sum_{e_1^{'}=L_{e_1^{'}}}^{U_{e_1^{'}}} T_{e_1^{'}} + T_{mem} + 2\sigma_{thd} \tag{5.2}$$

The first term $\sum T_{e_1^{'}}$ is the computation cost of $\mathcal{TILE}$ estimated as a sum of the execution times $T_{e_1^{'}}$ of all its intra-tile wavefronts with $e_1^{'}$ ranging over these wavefronts starting from the smallest given by the lower bound $L_{e_1^{'}}$ of loop $e_1^{'}$ to the largest given by the upper bound $U_{e_1^{'}}$ of loop $e_1^{'}$ along dimension $e_1^{'}$. The second term T_{mem} denotes the memory latency consumed by the memory accesses at the code before ① and the code before ③. The last term $2\sigma_{thd}$ denotes the overhead of the two `syncthreads` at ① and ③, where σ_{thd} is dependent on the number of threads used, i.e., `blockDim.x` × `blockDim.y` × `blockDim.z`.

The execution time $T_{e_1^{'}}$ of the intra-tile wavefront indexed by $e_1^{'}$ is given by:

$$T_{e_1^{'}} \;=\; \alpha \times G_{e_1^{'}} + \beta \times H_{e_1^{'}} + \sigma_{thd} \tag{5.3}$$

GPUs execute instructions with warps as units of execution and hide memory latency through interleaving of thread blocks. In the scheme shown in Figure 5.2, the warps are never idle when executing a wavefront as all memory accesses happen before and after the execution of $\mathcal{TILE}$. Thus, the first term represents the workload for computing the wavefront indexed by $e_1^{'}$, which is estimated to be proportional to $G_{e_1^{'}}$, the number of 32-thread warps executed at the wavefront. In addition, the first term implicitly considers the effects of bank conflicts on the execution time of $\mathcal{TILE}$. However, the same $G_{e_1^{'}}$ may result when the number of loop iterations, $H_{e_1^{'}}$, in the wavefront indexed by $e_1^{'}$ varies (due to division by 32). To accommodate its impact on performance, $T_{e_1^{'}}$ is fine-tuned by including the second term $\beta \times H_{e_1^{'}}$, which attempts to differentiate the effects of varying $H_{e_1^{'}}$ values on performance.

Note that $G_{e_1'}$ and $H_{e_1'}$ may vary from wavefront to wavefront as shown in Figure 5.1(d). Given an intra-tile wavefront, both can be precisely calculated. The last term σ_{thd} is the overhead of `syncthreads` at ③.

By substituting $T^i_{e_1'}$ in (5.3) into (5.2), we obtain:

$$T_{\mathcal{TILE}} = \sum_{e_1'=L_{e_1'}}^{U_{e_1'}} (\alpha \times G_{e_1'} + \beta \times H_{e_1'} + \sigma_{thd}) + T_{mem} + 2\sigma_{thd} \qquad (5.4)$$

which is illustrated graphically in Figure 5.3. As highlighted, $G_{e_1'}$ varies across the wavefronts with less work being done at the beginning and end of the computation process for $\mathcal{TILE}$ when its wavefronts are executed.

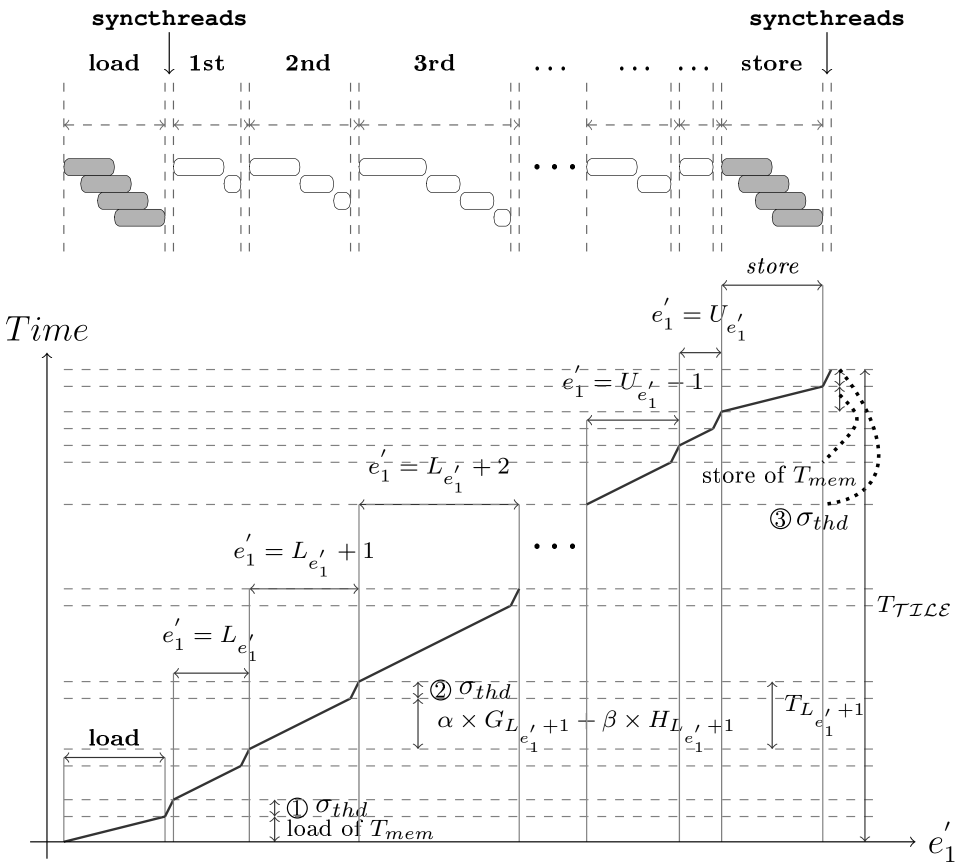


Figure 5.3: An illustration of the intra-tile wavefronts of $\mathcal{TILE}$ according to Equation 5.4.

The memory access latency T_{mem} can be estimated by:

$$T_{mem} \quad = \quad \gamma \times N_{mem} \tag{5.5}$$

where N_{mem} denotes the number of loads and stores made in $\mathcal{TILE}$. Note that N_{mem} is not related to memory coalescing since that would make its estimation dependent on the actual data layout at run time. This simple approximation seems to be adequate as T_{mem} is not a dominant term in (5.2) for the following reasons combined. First, DOACROSS loop nests are usually not bandwidth-bound. Second, optimal tile sizes found are large in order to exploit two levels of wavefront parallelism. Finally, the memory accesses performed by warps can overlap. We will return to this issue briefly at the end of Section 5.5.

By substituting T_{mem} given in (5.5) into (5.4), simplifying and letting

$$L_{\mathcal{TILE}} \quad = \quad s_1 \times \cdots \times s_m = \sum_{e_1'=L_{e_1'}}^{U_{e_1'}} H_{e_1'} \tag{5.6}$$

we obtain the following estimated execution time of $\mathcal{TILE}$:

$$T_{\mathcal{TILE}} \quad = \quad \alpha \times \sum_{e_1'=L_{e_1'}}^{U_{e_1'}} G_{e_1'} + (U_{e_1'} - L_{e_1'} + 3) \times \sigma_{thd} + \beta \times L_{\mathcal{TILE}} + \gamma \times N_{mem} \tag{5.7}$$

The second term indicates that there are $U_{e_1'} - L_{e_1'} + 1$ `syncthreads` instructions executed at ② inside $\mathcal{TILE}$ and two at ① and ③, as shown in Figure 5.2.

5.3.2 Inter-tile Execution

A DOACROSS loop nest is parallelized into a CUDA kernel. The execution time, T_{total}, for the kernel, i.e., for its inter-tile wavefronts is estimated by:

$$T_{total} = \sum_{t_1'=L_{t_1'}}^{U_{t_1'}} T_{t_1'} + \sigma_{ker} \tag{5.8}$$

The first term is the computation cost of all tiles in the kernel estimated as a sum of the execution times $T_{t_1^{'}}$ of all its inter-tile wavefronts with $t_1^{'}$ starting from the lower bound $L_{t_1^{'}}$ of loop $t_1^{'}$ to the upper bound $U_{t_1^{'}}$ of loop $t_1^{'}$ along dimension $t_1^{'}$. The second term σ_{ker} is the kernel startup cost.

Thus, $U_{t_1^{'}} - L_{t_1^{'}} + 1$ is the number of tiles contained in the inter-tile wavefront indexed by $t_1^{'}$. If all P *SMs* execute simultaneously up to B thread blocks each, then the number of tiles, denoted $I_{t_1^{'}}$, contained in a thread block is:

$$I_{t_1^{'}} = \lceil \frac{U_{t_1^{'}} - L_{t_1^{'}} + 1}{B \times P} \rceil = \lceil \frac{U_{t_1^{'}} - L_{t_1^{'}} + 1}{\texttt{gridDim.x}} \rceil \tag{5.9}$$

where B is decided by the GPUs architectural constraints and kernel code according to the CUDA programming guide as demonstrated in Table 5.1.

The execution time $T_{t_1^{'}}$ is determined by the slowest among the P SMs with the other SMs idle waiting at the `syncblocks` macro at ④. As a result, we have:

$$T_{t_1^{'}} = \sum_{i=1}^{I_{t_1^{'}}} T_{t_1^{'}}^{i} + \sigma_{blk} \tag{5.10}$$

$$T_{t_1^{'}}^{i} = \begin{cases} B \times T_{\mathcal{TILE}} & 1 \le i \le I_{t_1^{'}} - 1 \\ \left\lceil \frac{(U_{t_1^{'}} - L_{t_1^{'}} + 1)\%(B \times P)}{P} \right\rceil \times T_{\mathcal{TILE}} & i = I_{t_1^{'}} \end{cases} \tag{5.11}$$

where $T_{t_1^{'}}^{i}$ is the execution time of the i-th tiles in all B thread blocks by the slowest SM and σ_{blk} is the overhead of `syncblocks` at ④ (to be measured below).

5.3.3 Parameter Estimation

We determine the six parameters used in T_{total} given in (5.8) for a tiled loop nest $\mathcal{L}$: σ_{ker}, σ_{thd}, σ_{blk}, α, β and γ. We do so for a given thread organization (determined by `gridDim` and `blockDim`) so that its tile size selection can be automated (Section 5.4). For the purposes of this work on NVIDIA GPUs, there are at

Description	Name
Warp Size (H)	$W = 32$
Max Number of Active Warps per SM (H)	$W_{SM} = 32$
Max Number of Active Threads per SM (H)	$T_{SM} = 1024$
Max Number of Active Blocks per SM (H)	$B_{SM} = 8$
Shared Memory per SM (H)	$S_{SM} = 16KB$
Number of 32-bit Registers per SM (H)	$R_{SM} = 16K$
Threads per Thread Block (S)	K
Register Usage per Thread Block (S)	R_{TB}
Shared Memory Usage per Thread Block (S)	S_{TB}
Warps per Thread Block (B)	$W_{TB} = \lceil \frac{K}{W} \rceil$
Thread Blocks Limited by Warps (B)	$B_W = \min(B_{SM}, \lfloor \frac{W_{SM}}{W_{TB}} \rfloor)$
Thread Blocks Limited by Registers (B)	$B_R = \lfloor \frac{R_{SM}}{R_{TB}} \rfloor$
Thread Blocks Limited by Shared Memory (B)	$B_S = \lfloor \frac{S_{SM}}{S_{TB}} \rfloor$
Thread Blocks (B)	$B = \min(B_W, B_R, B_S)$

Table 5.1: Determining B for an NVIDIA Tesla C1060 GPU. An item in Column 1 that depends on hardware, kernel code or both is indicated with an H, S or B appropriately.

most 16 B_{max} different thread organizations because (1) there are B_{max} different 1-D grid layouts with `gridDim.x` $= B \times P$, where $B \leq B_{max} \leq B_{SM} = 8$ as shown in Table 5.1, and (2) the number of threads per block, i.e., `blockDim.x` $\times$ `blockDim.y` $\times$ `blockDim.z` is one of the 16 possibilities contained in $\{32, 64, \ldots, 512\}$.

5.3.3.1 Architectural Parameters: σ_{ker}, σ_{thd} and σ_{blk}

These overheads are small (relative to the execution time of a loop nest $\mathcal{L}$) and are measured for a GPU architecture as follows. First of all, σ_{ker} is the startup overhead of the kernel for $\mathcal{L}$, which can be obtained through running an empty version of the kernel (with the computations in $\mathcal{L}$ removed) for a given thread organization. In fact, as $\sigma_{ker} \ll T_{total}$, treating it as a small constant for all thread organizations does not affect in practical terms how the relative performances of $\mathcal{L}$ are ranked for all combinations of thread organizations and tile sizes used. As for `syncthreads` executed at ①, ② and ③, it is lightweight on NVIDIA GPUs. Its overhead σ_{thd} depends on the number of threads per block, i.e., `blockDim.x` $\times$ `blockDim.y` $\times$ `blockDim.z` and is measured as in [111]. There are only 16 cases to consider as `blockDim.x` $\times$ `blockDim.y` $\times$ `blockDim.z` is a multiple of 32 ranging from 32 to 512. Finally, the `syncblocks` macro is invoked at the end of each inter-tile wavefront at ④. Its overhead σ_{blk}, which is higher than σ_{thd}, depends mainly on the number of thread blocks contained in an inter-tile wavefront, i.e., `gridDim.x` $= B \times P$. The effects of different `blockDim.x` $\times$ `blockDim.y` $\times$ `blockDim.z` values on σ_{blk} are negligible. As $B \leq B_{max} \leq B_{SM} = 8$, `syncblocks` is measured as in [114] for a few, i.e., up to B_{max} different `gridDim.x` values.

5.3.3.2 Program-dependent Parameters: α, β and γ

Once the values of σ_{ker}, σ_{thd} and σ_{blk} are determined, the given loop nest $\mathcal{L}$ is simplified to possess one inter-tile wavefront with exactly $B \times P$ thread blocks consisting of only full tiles. This ensures that all P SMs have exactly the same workload so that these three program-dependent parameters can be accurately measured.

The three parameters are found for each of up to $16B_{max}$ different thread organizations as mentioned earlier (where $B_{max} \leq 8$). In each case, the simplified loop nest $\mathcal{L}$ is executed for a total of m times, each with a different tile size. Let T_i be the execution time corresponding to the tile size s_i used. Given a tile size s_i, all parameters in T_{total} except α, β and γ are now known. We can find the values of α, β and γ by performing a linear curve fitting using the least-square method for T_{total} (by calling MATLAB library) with respect to the m execution times, $T_1, \ldots, T_m$, obtained.

5.3.4 Border Tiles

A border tile may execute faster than a full tile. If the i-th inter-tile wavefront that induces $T^i_{t'_1}$ in (5.10) contains non-full border tiles, then $T^i_{t'_1}$ may be longer than the actual execution time of the wavefront. We can improve this inaccuracy with an estimate of $0.5 \times T_{\mathcal{TILE}}$ as the execution time of a border tile $\mathcal{TILE}$ by assuming that the average size of border tiles is half of a full tile.

5.4 Model-driven Tile Size Selection

Given the estimated execution time of T_{total} in (5.8) for a tiled loop nest $\mathcal{L}$ as input, we employ an "educated" search to find automatically and efficiently an optimal tile size $\vec{s} = (s_1, \dots, s_m)$ for $\mathcal{L}$ and an associated thread organization, determined by `gridDim` and `blockDim`, used for realizing the optimal tiling. In this chapter, a *tile layout* is determined by a tile size and a thread organization.

5.4.1 The Algorithm

We use two kinds of constraints to prune the search space:

Tile Size Constraint The tile size, i.e., $L_{\mathcal{TILE}} = s_1 \times \cdots \times s_m$ is bounded from below by a data reuse rate $D = \frac{L_{\mathcal{TILE}}}{N_{mem}}$ (where N_{mem} is introduced in (5.5)) and from above by the size of shared memory. For DOACROSS loop nests, large tile sizes lead to higher data reuse rates. Thus, D must be larger than an empirical minimum threshold to ensure better intra-tile data locality.

`blockDim` Constraint In NVIDIA GPUs, `blockDim.x` $\times$ `blockDim.y` $\times$ `blockDim.z` represents the number of threads per block. According to [111], the SP performance usually suffers with too many or too few threads.

Algorithm 5.1: Algorithm for automating tile size selection on a GPU with P *SMs*.

```
Input: A transformed loop nest $\mathcal{L}$
Output: The best tile size $\vec{s}_{best}$ and thread organization gridDim and blockDim
1  Compute the register usage per thread, $R_T$, using any tile size and thread organization;
2  $T_{best}$ is initialized to $\infty$;
3  for each tile size $\vec{s} = (s_1, \ldots, s_m)$ that satisfies the tile size constraint do
4      Set $S_{TB}$ (shared memory usage per block) as the shared memory usage per tile ;
5      for each $t = (blockDim.x, blockDim.y, blockDim.z)$ that satisfies the
       blockDim constraint do
6          Let $R_{TB} = R_T \times$ blockDim.x $\times$ blockDim.y $\times$ blockDim.z;
7          Let $B = \min(B_W, B_R, B_S)$, where $B_W$, $B_R$ and $B_S$ are computed in Table 5.1;
8          Evaluate $T_{total}$ given in (5.8) for the current tile size $\vec{s}$ and the current thread
           organization specified by gridDim $= B \times P$ and blockDim $= t$ ;
9          if $T_{total} < T_{best}$ then
10             $T_{best} = T_{total}$ ;
11             Record $\vec{s}_{best} = \vec{s}$;
12             Record gridDim.x $= B \times P$ and blockdDim $= t$;
13         end if
14     end for
15 end for
```

Furthermore, blockDim.x $\times$ blockDim.y $\times$ blockDim.z must be no smaller than the number of iterations contained in the largest intra-tile wavefront to ensure that every thread has some work to do. Thus, some small and large values of blockDim.x $\times$ blockDim.y $\times$ blockDim.z can be ignored.

Our algorithm for automating tile size selection for $\mathcal{L}$ is outlined in Algorithm 5.1. Recall that as shown in Figure 5.2, all tiles in a thread block are executed sequentially. Thus, for every type of resource listed in Table 5.1, the amount consumed by a block is calculated on a per-tile basis. The basic idea is to perform an "educated" search when going through all tile layouts to find the one with the smallest execution time T_{total}. In line 1, the register usage per thread, denoted R_T, is measured independently of tile layouts used. This is because in each case the same code as shown in Figure 5.2 is compiled for each thread by NVIDIA's `nvcc` compiler. Finding R_T this ways speeds up the process for calculating R_{TB} in line 6. Similarly, in line 4, S_{TB} does not depend on `blockDim`. Once R_{TB} and S_{TB} are known, B_W, B_R and B_S are computed in line 7 as per Table 5.1.

5.4.2 Implementation

We have implemented our tile size selection technique as a part of our compiler, as shown in Figure 3.18. Our tile size selection module is invoked the dotted rectangular boxes with ⑤.

5.5 Experiments

We use three representative DOACROSS kernels, 1-D (3-point), 2-D (5-point) and 3-D (7-point) SOR solvers, to demonstrate the accuracy and efficiency of our tile size selection framework on an NVIDIA Tesla C1060 GPU (c.f. Table 5.1). Four problem sizes are discussed for each kernel, representing 12 different optimization problems for which best tile layouts (tile size/`blockDim` combinations) are solved.

5.5.1 Accuracy

It is impractical to measure the accuracy of our tile size selection framework for a kernel by comparing the actual execution time of the best tile layout found with the execution times of all tile layouts. For example, 1-D SOR has over 10^8 tile layouts (assuming 16KB shared memory and 16KB register), taking over three years to execute if one tile layout costs one second to run.

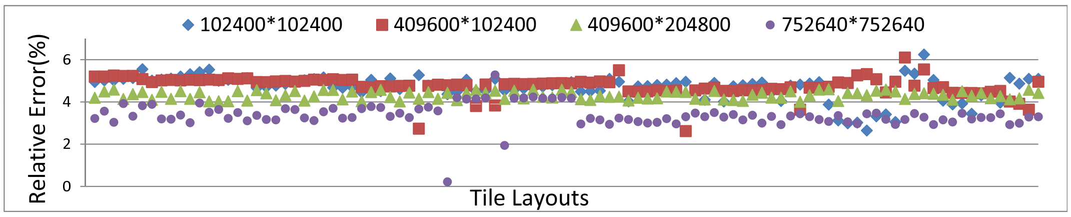


Figure 5.4: 1-D SOR: relative errors for 100 tile layouts in each of the four problem sizes.

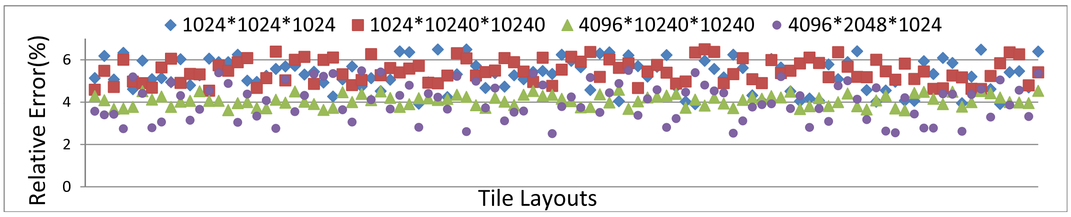


Figure 5.5: 2-D SOR: relative errors for 100 tile layouts in each of the four problem sizes.

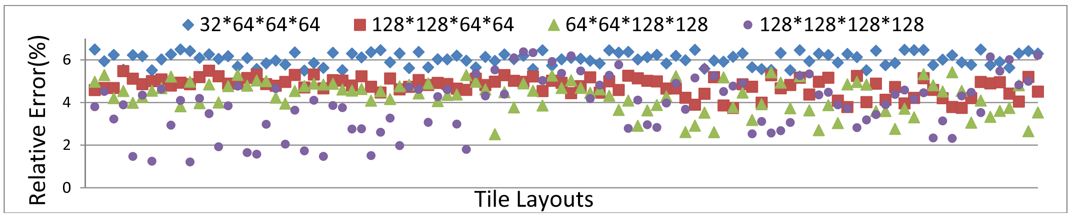


Figure 5.6: 3-D SOR: relative errors for 100 tile layouts in each of the four problem sizes.

We have decided to evaluate this work empirically as is often done in automated performance tuning. For each of the 12 optimization problems discussed here, we have randomly sampled 1000 different tile layouts. The largest relative error (between the estimated execution time T_{total} and actual execution time) is observed to be within 6.5%. To see this graphically, the relative errors of 100 sampled tile layouts for each optimization problem are plotted in Figures 5.4 – 5.6. Let us look at the actual performance gap between the tile layout found by us and the best-performing one in each case. Let us consider a generic optimization problem O.

Let T^s_{total} and R_s be the estimated and actual execution times of any tile layout $\vec{s}$ for O with the relative error being e_s. In particular, let T^{opt}_{total} and R_{opt} be the estimated and actual execution times of the the best tile layout *opt* predicted for O with the relative error being e_{opt}. The (worst) performance gap between *opt* and the best-performing one s is bounded by $(\frac{1+e_s}{1+e_{opt}} - 1) \times 100\%$, when $R_{opt} - R_s = T^{opt}_{total}/(1+e_{opt}) - T^s_{total}/(1+e_s)$, i.e., e_s is the largest, where $R_{opt} > R_s$. (Note that $(1+e_{opt}) \neq 0$ and $(1+e_s) \neq 0$ as $|e_{opt}|$ and $|e_s|$ are no larger than 6.5%.) Based on our sampled tile layouts, the performance gaps are found to be 5.29%, 0.51%, 2.10% and 4.92% for the four problem sizes of 1-D SOR (displayed from left to right in Figure 5.4), 4.33%, 1.31%, 5.14% and 2.01% for the four problem sizes of 2-D SOR (Figure 5.5), and 0.66%, 2.28%, 6.04% and 1.30% for the four problem sizes of 3-D SOR (Figure 5.6).

5.5.2 Search Time

Our algorithm is efficient in finding the best tile layout for a loop nest (on an Intel Xeon 2.0 GHz CPU). When tiling an m-D loop nest that represents an $(m-1)$-D SOR solver with a tile size $\vec{s} = (s_1, \ldots, s_m)$, s_1 represents the time dimension and $s_2, \ldots, s_m$ represent the $m-1$ spatial dimensions for the underlying mesh. Due to loop skewing, the worksets of different time slices of a tile are also skewed [51]. Thus, the data reuse rates of a tile for the 1-D, 2-D and 3-D SOR solvers are expressed as a function of $\vec{s}$ and are bounded from above by s_2, $\frac{s_2 s_3}{s_2+s_3}$ $\frac{s_2 s_3 s_4}{s_2 s_3 + s_2 s_4 + s_3 s_4}$, respectively, when $s_1 \to \infty$. For the 1-D SOR solver, the data reuse rate induces a tile size constraint: $\frac{L_{\mathcal{TILE}}}{N_{mem}} \geq 300$, where the threshold 300 is empirically set (Section 5.4.1). The search time is 238 secs over a search space of 3×10^6 tile layouts. For the 2-D SOR solver, the data reuse rate gives rise to $\frac{L_{\mathcal{TILE}}}{N_{mem}} \geq 6$. The search time is 369 secs over a search space of 3.2×10^6 tile layouts. For 3-D SOR, the data reuse rate imposes $\frac{L_{\mathcal{TILE}}}{N_{mem}} \geq 1$. The search time is 503 secs over a search space of 4.7×10^6 tile layouts.

5.5.3 Discussions

In our model, β is usually negative. This is because α accounts for the effects of bank conflicts in a warp on performance assuming that the warp is full (with 32 loop iterations). As a result, β is used to counteract the effects of non-existing bank conflicts in a partial warp on performance. This approximation may introduce inaccuracy into our model. In addition, the effects of coalesced loads and stores on performance are also approximate, without, for example, considering the actual data layout. Since DOACROSS kernels are not bandwidth-bound, the role that γ plays is not as significant as α and β.

5.6 Conclusion

It is difficult to find optimal tile sizes for tiled DOACROSS loop nests on GPU algebraically (due to the nonlinear nature of the optimization problem) or by manual tuning (due to large search spaces). In this chapter, we present a cost model for estimating the execution time of a DOACROSS loop nest tiled for GPUs. Based on this model, we present an algorithmic framework to automate tile size selection for a DOACROSS loop nest. The performances achieved with selected tile sizes are close to the best observed for the three SOR solvers tested.

Chapter 6

Efficient Parallelization of DOACROSS Stencils on GPUs

In this chapter, we present a new parallel SOR method that admits more efficient data-parallel SIMD execution than RBSOR on GPUs. Our solution is obtained non-conventionally, by starting from a K-layer SOR method and then parallelizing it by applying a non-dependence-preserving scheme consisting of a new domain decomposition technique followed by a loop tiling technique called alternate tiling. Despite its relatively slower convergence, our new method outperforms RBSOR by making a better balance between data reuse and parallelism and by trading off convergence rate for SIMD parallelism. Our experimental results highlight the importance of synergy between domain experts, compiler optimizations and performance tuning in maximizing the performance of SOR-like DOACROSS loop nests on GPUs. The new approach can be implemented in a pattern-guided compiler to optimize SOR-like DOACROSS loop nests on GPUs. The research presented in this chapter was published in [28, 29, 31].

6.1 Introduction

Gauss-Seidel and SOR, which are widely used smoothers in multigrid methods, are difficult to parallelize, particularly on GPUs. In these iterative PDE solvers, each calculation at an iteration depends on some previous results from the same iteration. As a result, the presence of such parallelism-inhibiting dependences makes it fundamentally difficult to create a massive number of fine-grained threads on GPUs to execute the computations in a DOACROSS loop nest concurrently on a massive number of processor cores.

In this chapter, we focus on iterative PDE solvers to establish optimization principles and strategies for their efficient mapping to GPUs. We have chosen to parallelize the 2-D symmetric SOR method on GPUs for two reasons. First, this method represents one of the most important iterative solvers for large systems of linear equations with a massive amount of data parallelism to be harnessed. Second, the underlying loop nest exhibits representative loop-carried dependences with both temporal and spatial data reuse to be captured.

In summary, this chapter makes the following contributions:

- We present a new parallel SOR method that admits efficient data-parallel SIMD execution in GPUs. We have obtained this solution in a non-conventional manner. The starting point is a K-layer SOR solver that performs K forward SOR sweeps and K backward SOR sweeps alternately. This sequential method is then parallelized by applying a non-dependence-preserving scheme consisting of a new domain decomposition technique followed by a generalized loop tiling called alternate tiling to the two sweep directions alternately.

- Our new method outperforms RBSOR on both one and multiple GPUs by making a better balance between data reuse and parallelism and by trading-off convergence rate for SIMD parallelism.

- Our experimental results demonstrate that certain DOACROSS loop nests can be accelerated further on GPUs if they are algorithmically restructured to be more amendable to GPU parallelization, judiciously optimized, and carefully tuned by a performance-tuning tool. This highlights the importance of synergy between domain experts, compiler optimizations and performance tuning in maximizing the performance of many applications, particularly SOR-like DOACROSS loop nests, on GPUs.

6.2 Limits of GPU Architectures

In this work, we use an NVIDIA Tesla S1070 GPU computing system as the basis for our investigation. It has four C1060 GPUs. Each C1060 GPU has 30 SMs with each SM containing eight SPs or processor cores, running at 1.3GHz. Each SP can perform one FMAD (two ops) and one FMUL (one op) for three single-precision FLOPs per cycle. With 240 SPs in total, C1060 has a single-precision peak performance of 936 GFLOPS (30SMs $\times$ 8SPs $\times$ $(2+1)$ $\times$ 1.3GHz). With four times as many SPs, S1070 can deliver 3744 GFLOPS (936GFLOPS $\times$ 4GPUs) of single-precision peak performance.

Every C1060 GPU has 102 GB/s bandwidth to its 4GB global memory. This amount of bandwidth can be easily saturated with computational resources supporting nearly 936 GFLOPS of performance. In addition, a global memory access has very high latency (400 – 600 cycles). In particular, each SM has a user/compiler-managed 16KB shared memory for data reuse or sharing among threads and 16,384 32-bit registers partitioned among threads. In S1070, data

Resource	Limits	
	C1060	C2050
Number of Threads per Block	512	1024
Number of Active Threads per SM	1024	1536
Number of Active Blocks per SM	8	8
Shared Memory per SM	16KB	48KB/16KB
Number of 32-bit Registers per SM	16K	32K

Table 6.1: CUDA constraints on C1060 and C2050.

exchange among its four C1060 GPUs (i.e., inter-GPU communication) is accomplished through the host (via PCI-e).

In addition to Tesla S1070-400, we have also evaluated this work using an NVIDIA Tesla S2050 Fermi system. The S2050 computing system consists of four C2050 GPUs. Its 448 cores are organized in 14 SMs of 32 cores each. Each SM has 64 KB of L1 cache and shared memory (64kB split 16/48 or 48/16 between cache and shared memory) and 768 KB of L2 cache. Its single-precision peak performance reaches 1030GFLOPS.

In the NVIDIA CUDA programming model [72], GPU threads are organized hierarchically into thread blocks and warps that have different synchronization mechanisms (see Section 3.3). While optimizing a CUDA kernel, shared memory bank conflicts and global memory access coalescing are performance-critical and should be leveraged by compilers.

Table 6.1 lists some architectural constraints imposed on a CUDA program [72]. Due to their complex interactions, it can be difficult to accurately predict the effects of compiler optimizations on the performance of a kernel. Unpublished details about GPU architectures further exacerbate the problem. There is often a tradeoff between the performance of individual threads and the TLP (thread-level parallelism) among all threads [7, 60, 84].

6.3 The 2-D SOR Iterative Solver

Partial Differential Equations (PDEs) are widely used in scientific and engineering applications. Iterative methods are faster than direct methods in solving a large system of linear equations and are thus often used. Three well-known iterative methods are Jacobi, Gauss-Seidel and SOR. In this section, we algorithmically analyze the parallel method of SOR.

Many applications involve boundary value problems that require solving diffusion equations. Consider a 2-D case:

$$\Delta u = \frac{\partial^2 u}{\partial i^2} + \frac{\partial^2 u}{\partial j^2} \tag{6.1}$$

where $\Omega = [0,1] \times [0,1] \in \mathbb{R}^2$ is bounded with $\partial\Omega$ as its boundary. The domain size is $N_1 \times N_2$. By using $u_{i,j}$ to denote the finite difference approximation of u at grid point (i,j), we obtain the following five-point approximation of (1):

$$4u_{i,j} - u_{i-1,j} - u_{i,j-1} - u_{i+1,j} - u_{i,j+1} = 0 \tag{6.2}$$

where $i = 1, \ldots, N_1$ and $j = 1, \ldots, N_2$. The boundary condition is set to be $\partial\Omega = 0$ in the normal manner.

Such a system of equations is often solved using an iterative solver. The Jacobi method updates all grid points at an iteration, say, k using their previous values obtained at iteration $k-1$:

$$u_{i,j}^{k} = \frac{1}{4} \times (u_{i-1,j}^{k-1} + u_{i,j-1}^{k-1} + u_{i+1,j}^{k-1} + u_{i,j+1}^{k-1}) \tag{6.3}$$

For the SOR method, the computation of $u_{i.j}^{k}$ uses the values of $u_{i-1,j}^{k}$ and $u_{i,j-1}^{k}$ that have already been computed at iteration k and the old values of $u_{i,j}^{k-1}$, $u_{i+1,j}^{k-1}$ and $u_{i,j+1}^{k-1}$ from iteration $k-1$:

$$u_{i,j}^{k} = (1-\omega) \times u_{i,j}^{k-1} + \frac{\omega}{4} \times (u_{i-1,j}^{k} + u_{i,j-1}^{k} + u_{i+1,j}^{k-1} + u_{i,j+1}^{k-1}) \tag{6.4}$$

The Gauss-Seidel method is a special case of SOR when $\omega = 1$.

A 2-D iterative solver is typically implemented using a 3D loop nest, where the inner two loops i and j enumerate all grid points in the $i - j$ plane, i.e., domain and the outermost loop k performs multiple sweeps, i.e., iterations across the $i - j$ plane. The set of all points (k, i, j) is known as the *iteration space* of the loop nest.

The Jacobi method is inherently parallel since all points can be computed at the same time. However, it is often not used due to its slow convergence and high memory usage. The SOR and Gauss-Seidel methods are known to be inherently sequential in their original forms. With an appropriate choice of the relaxation factor ω, SOR converges faster than Gauss-Seidel.

The symmetric SOR combines two SOR sweeps together in such a way that the resulting iteration matrix is similar to a symmetric matrix. In other words, SOR is a forward sweep performed using Equation (6.4) followed by a backward sweep according to:

$$u_{i,j}^{k} = (1 - \omega) \times u_{i,j}^{k-1} + \frac{\omega}{4} \times (u_{i-1,j}^{k-1} + u_{i,j-1}^{k-1} + u_{i+1,j}^{k} + u_{i,j+1}^{k}) \tag{6.5}$$

In this chapter, we consider to apply Equations (6.4) and (6.5) alternately as illustrated in Figure 6.1, resulting in what is referred to here as a multi-layer SOR method. There are K layers since every sweeping direction is repeated K times and the method is symmetric due to the five-point stencil discretization used. In the *forward sweep*, Equation (6.4) is applied for K iterations, starting at the left and bottom boundaries of the domain and moving towards the right and top boundaries at each iteration. In the *backward sweep*, Equation (6.5) is applied also K times with the sweeping direction being reversed.

The multi-layer SOR method, MLSOR, which is guaranteed to converge [85], enables us to develop a new data-reuse-effective and data-parallel implementation for GPU architectures below.

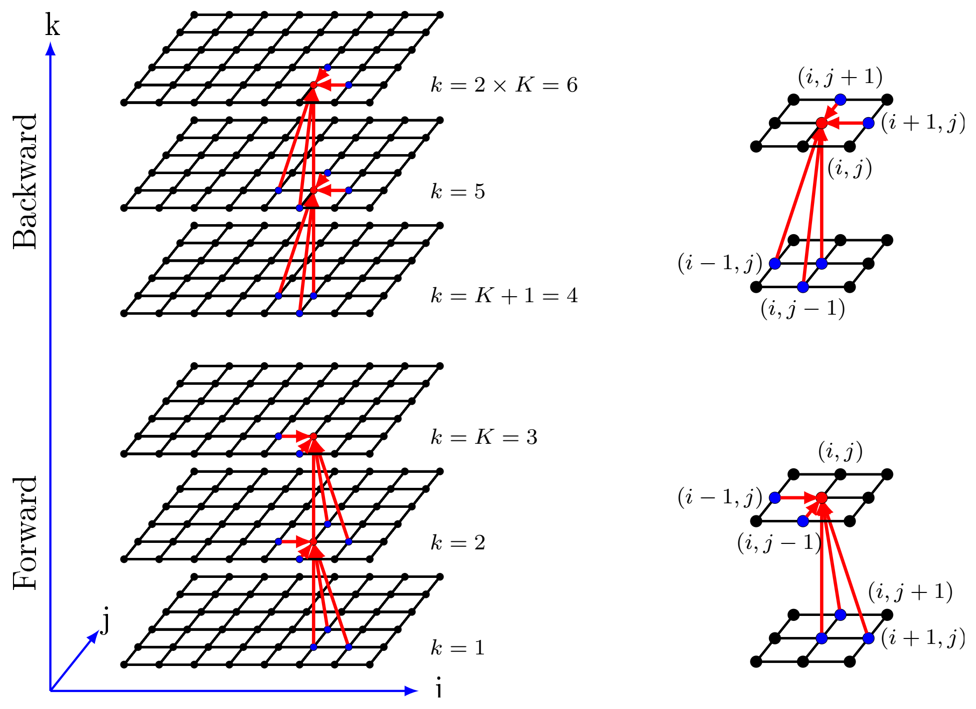


Figure 6.1: A sequential multi-layer symmetric five-point SOR method (MLSOR) ($K = 3$). The five dependences at each point in the forward and backward sweeps are directly derived from Equations (6.4) and (6.5), respectively.

6.4 A Parallel MLSOR Method for GPUs

Due to its simplicity and good performance, RBSOR has often been a popular choice not only for distributed memory machines, but recently, also for GPUs [131]. RBSOR divides a domain of grid points into a chessboard of red and black points. Due to the absence of data dependences between red and black points, the Jacobi method (using SOR) is applied to update the points of one color simultaneously using the previous values computed at the points of the other color. This high degree of fine-grained parallelism makes RBSOR amenable to data-parallel execution on GPUs. On the other hand, RBSOR does not respect the data dependences in the original SOR, resulting in some slightly slower convergence rates under some inputs. In addition, RBSOR exhibits less data reuse (due to red-black ordering) and may suffer from high inter-GPU communication overhead.

In this section, we describe a new parallel multi-layer SOR algorithm, also denoted MLSOR (as we will refer to its sequential version as the sequential MLSOR henceforth), for GPU architectures in order to strike a better balance between fine-grained parallelism and data reuse than RBSOR. Like RBSOR, MLSOR is developed using a non-dependence-preserving parallelization scheme as follows. First, a new domain decomposition technique is applied to enable simultaneous point updating using fine-grained threads (Section 6.4.1). Second, a generalized loop tiling called *alternate tiling*, which tiles the two sweep directions alternately, is applied so that the resulting tiled code exhibits the same degree of intra-kernel data reuse but better inter-kernel data reuse than traditional loop tiling (Section 6.4.2). Although existing NVIDIA GPU architectures cannot exploit inter-kernel data reuse, other stream processors such as Imagine [55] and AMD GPUs can. Despite this, applying alternate tiling to NVIDIA GPUs is still beneficial as it improves the convergence rate as discussed in Section 6.5.1. Third, a new tile scheduling scheme is introduced to ensure that all SPs in one GPU can start executing their subdomains at the same time, resulting in significantly improved SIMD parallelism at the expense of some slightly slower convergence rates than the sequential SOR under some inputs (Section 6.4.3). Finally, the tile scheduling scheme works directly for multiple GPUs with the inter-GPU communication cost being kept to a minimum by overlapping computation and communication (Section 6.4.4).

6.4.1 Domain Decomposition

Traditionally, the domain of an SOR solver is partitioned disjointly so that a processor computes all the points in its allotted subdomain in every SOR iteration. So the domain is meant to be the mesh, i.e., the $i - j$ plane for a 2-D SOR solver. In MLSOR, however, the sub-mesh allocated to an SP changes as the iteration pro-

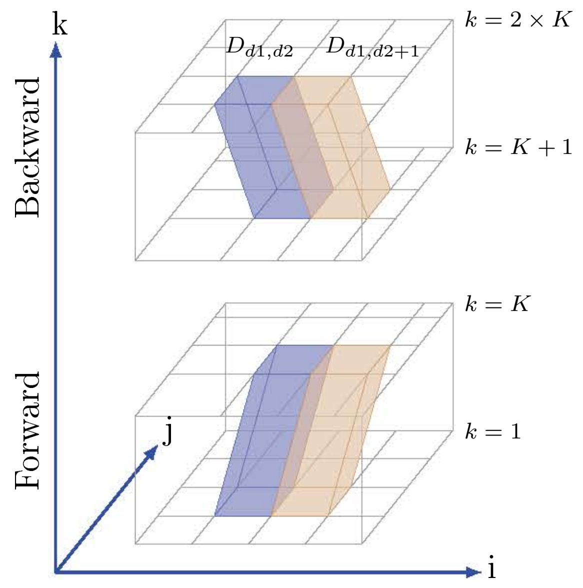


Figure 6.2: Domain decomposition for two alternate sweeps (with K layers in each sweep).

ceeds, causing adjacent sub-meshes to overlap at their boundaries. To avoid any confusion, the domain of an MLSOR loop nest is meant to be its 3D iteration space. As a result, domain decomposition divides the iteration space into 3D rectangular boxes (i.e., parallelepipeds).

The two sweep directions are partitioned as shown in Figure 6.2. We describe only the technique used for a K-layer forward sweep since it is mirrored by a backward sweep. There are two reasons behind this somewhat non-conventional partitioning approach. First, together with our alternate tiling, this partitioning approach allows different subdomains to be executed in parallel with the inter-subdomain communication kept to a minimum, as discussed in Section 6.4.3. Second, as is clear in Section 6.5.1, better data reuse and convergence rate can be obtained.

Let a 2-D mesh of size $N_1 \times N_2$ be partitioned a sub-mesh of size $P \times P$. We use square sub-meshes because as discussed in Section 6.4.3, K can be maximized so that SOR can be applied more often. For simplicity, it is assumed that P divides

N_1 and N_2. Consider a K-layer forward sweep starting from $k = k_f$ and ending at $k = k_f + K - 1$. It is partitioned into $P \times P$ blocks so that its intersection with k_f layer is divided into $P \times P$ rectangles of size $N_1/P \times N_2/P$. This is achieved with the following subdomain cutting planes across the K-layers:

$$\begin{aligned} i &= s \times \tfrac{N_1}{P} + k - k_f + 1, \quad s = 1, \ldots, P-1 \\ j &= t \times \tfrac{N_2}{P} + k - k_f + 1, \quad t = 1, \ldots, P-1 \end{aligned} \tag{6.6}$$

whose normals are $(1, -1, 0)$ and $(1, 0, -1)$, respectively.

All non-border subdomains are 3D rectangles (parallelepipeds) of size $K \times \frac{N_1}{P} \times \frac{N_2}{P}$. The cutting hyperplanes near the borders of the mesh are so chosen that all subdomains (border or non-border) have roughly the same number of grid points. Let D_{d_1,d_2} be a non-border subdomain located at (d_1, d_2), where $0 \leqslant d_1 < P$ and $0 \leqslant d_2 < P$. Let $D^k_{d_1,d_2}$ be its k-th layer. Then we have:

$$D^k_{d_1,d_2} = D^{k-1}_{d_1,d_2} + (1,1,1) \tag{6.7}$$

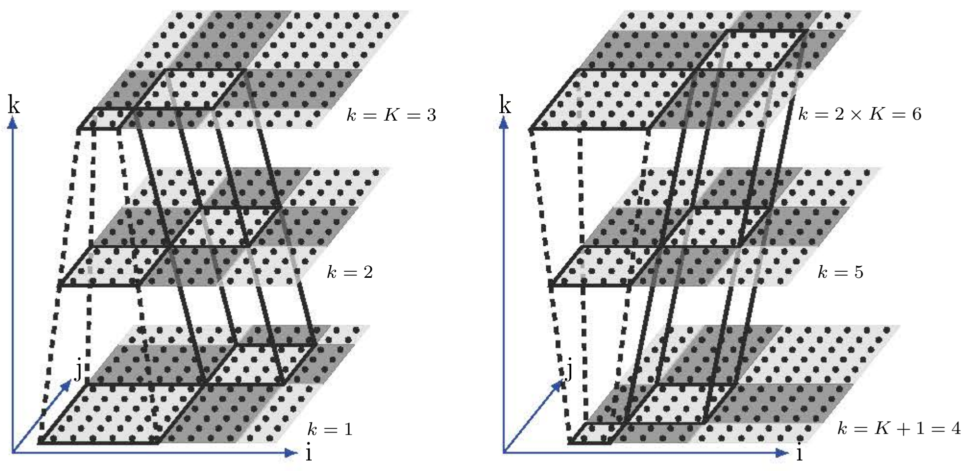
where

$$D^{k-1}_{d_1,d_2} + (1,1,1) = \{(k,i,j) + (1,1,1) \mid (k,i,j) \in D^{k-1}_{d_1,d_2}\} \tag{6.8}$$

In this case, every layer $D^k_{d_1,d_2}$ in the subdomain D_{d_1,d_2} is a *translate* of the layer $D^{k-1}_{d_1,d_2}$ below, i.e., drifts away from the coordinate origin, along $(1, 1, 1)$ as shown in Figure 6.2.

6.4.2 Alternate Tiling

We tile our sequential MLSOR by using alternate tiling, a form of generalized loop tiling. As can be observed from Equations (6.4) and (6.5) and is also illustrated in Figure 6.1, there exists temporal reuse across all three dimensions in the 3D iteration space of the MLSOR loop nest. Specifically, each grid point is accessed



(a) A forward sweep (b) A backward sweep

Figure 6.3: Tiling of a subdomain D_{d_1,d_2} in a forward sweep and a backward sweep shown in Figure 6.2 into $m \times m = 3 \times 3 = 9$ tiles of height $K = 3$. In each case, all $m^2 - 4m + 4$ tiles in the middle are full while the rest are border tiles.

five times during a sweep across the $i - j$ plane, once by itself and four times by its neighbours, and also accessed multiple times during multiple sweeps. To capture such temporal reuse, all three dimensions must be tiled. Due to the existence of data dependencies in both forward and backward sweeps (Figure 6.1), it is illegal to simply tile its iteration space by using rectangular boxes.

Figure 6.3 illustrates loop tiling being applied to a subdomain D_{d_1,d_2} in both a forward sweep and a backward sweep shown in Figure 6.2. As illustrated in Figure 6.1, the sweeping direction used for updating the K layers in a forward sweep is reversed in a backward sweep. Thus, the four *non-self* dependencies are also reversed. This results in loop tiling being applied to two sweeping directions alternately, a generalization of traditional loop tiling [125, 126] that tiles the entire iteration space uniformly. The main reason for tiling a subdomain this way is to ensure that the subdomains can be executed in parallel as discussed shortly. In addition, all border tiles are chosen to have different sizes so that they have all the same amount of work. This ensures load balancing among fine-grained threads.

For reasons of symmetry, we explain only how to tile a K-layer subdomain obtained for a forward sweep, where Equation (6.4) is repeated K times across the $i-j$ plane. In general, a K-layer subdomain is divided into $m \times m$ tiles identified by their tile indices. Let T_{t_1,t_2} be the tile located at (t_1, t_2). Let $K \times M \times M$ be the size of a full tile. Let (k_0, i_0, j_0) be the lexicographically largest point of the bottom-left tile in the subdomain. The subdomain is divided into $m \times m$ tiles by using the following set of $2m-2$ hyperplanes:

$$\begin{aligned} k_0 &\leqslant k < k_0 + K \\ i &= (s-1) \times M + k_0 - k + i_0 + 1, \quad s = 1, \ldots, m-1 \\ j &= (t-1) \times M + k_0 - k + j_0 + 1, \quad t = 1, \ldots, m-1 \end{aligned} \tag{6.9}$$

whose normals are $(1,1,0)$ and $(1,0,1)$, respectively.

There are $(m-2) \times (m-2)$ full (i.e., non-border) rectangular tiles of size $K \times M \times M$ in the center and $m \times m - (m-2) \times (m-2) = 4(m-1)$ border tiles. In Figure 6.3, only the middle one is full while all the rest are border tiles.

- **Full Tiles.** Let $T^k_{t_1,t_2}$ be the set of points in the k-th layer of T_{t_1,t_2}. If T_{t_1,t_2} is a a full tile, then we have:

$$T^k_{t_1,t_2} = T^{k-1}_{t_1,t_2} + (1,-1,-1) \tag{6.10}$$

  Thus, every layer in T_{t_1,t_2} is a *translate* of the layer below along the direction $(1,-1,-1)$, as can be visualized using the middle tile in Figure 6.3.

- **Border Tiles.** A border tile is a boundary tile that is a hexahedron but not a rectangular box as shown in Figure 6.3.

6.4.3 Parallelization for One GPU

We now explain the rationale behind our non-conventional domain decomposition and tiling techniques. Our parallelization strategy is simple. All K-layer sweeps are executed sequentially, bottom-up. Each K-layer sweep is executed concurrently by all SPs in a GPU. In order to enable all SPs to start executing at the same time, the tiles with the same tile index from all subdomains are executed in parallel by the same kernel. The points in a tile are executed sequentially by one thread. Note that in our parallelization scheme, among the $m \times m$ tiles in a subdomain, the tiles in $\{T_{i,j} \mid 0 < i, j < m\}$ (e.g., the tiles $T_{1,1}$, $T_{1,2}$, $T_{2,1}$ and $T_{2,2}$ when $m = 3$ in Figure 6.3) do not have parallelism-inhibiting inter-subdomain dependences and can thus be combined into a larger tile. This avoids unnecessary kernel startup overhead. In Section 6.5.1.1, we discuss how to apply shared memory reduction to deal with large tiles.

Therefore, in our implementation, every subdomain in a K-layer sweep is partitioned into 2×2 tiles. There are a total of four kernels executing a K-layer sweep. All tiles with the same tile index (t_1, t_2) from different subdomains form a grid executed by the same kernel, denoted $\mathcal{K}_{t_1,t_2}$. For example, suppose that the two adjacent subdomains D_{d_1,d_2} and D_{d_1+1,d_2} in a forward sweep (highlighted in Figure 6.2) are of the size $K \times 8 \times 8$. They are each divided into four tiles $\mathcal{T}_{1,1}$, $\mathcal{T}_{1,2}$, $\mathcal{T}_{2,1}$ and $\mathcal{T}_{2,2}$ as illustrated in Figure 6.4. How their sizes are chosen is discussed below.

The four kernels $\mathcal{K}_{1,1}$, $\mathcal{K}_{1,2}$, $\mathcal{K}_{2,1}$ and $\mathcal{K}_{2,2}$ are executed lexicographically in terms of their kernel indices, as illustrated in Figure 6.5. The points in the same tile are also executed lexicographically. Like RBSOR, our parallelization scheme does not respect all data dependences in the original SOR. However, the convergence is guaranteed but at somewhat slower rates under some inputs [93, 138]. To (more than) offset a drop in the convergence rate, all subdomains in a K-layer sweep can now be executed in parallel. This represents a good tradeoff for data-parallel GPU computing, one of the key findings worthy being emphasised in this chapter.

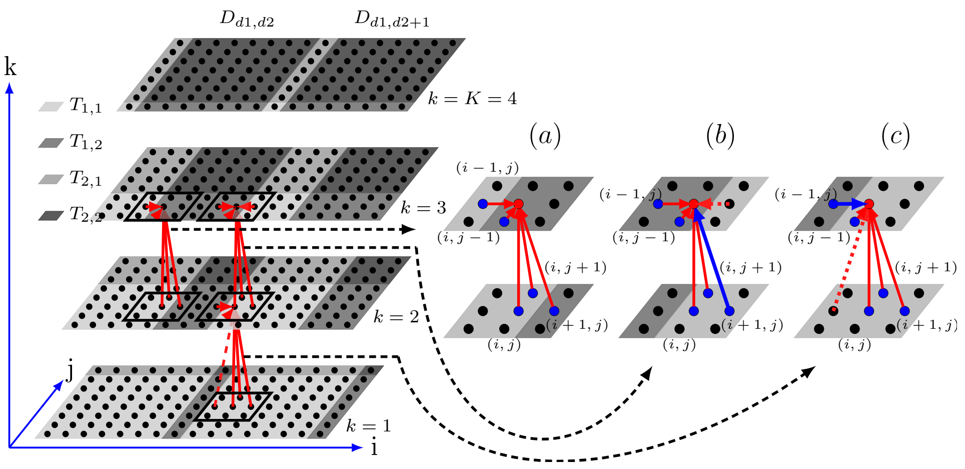


Figure 6.4: Enforcement of the data dependences illustrated for the two subdomains illustrated in Figures 6.2 and 6.3 in a forward sweep. The meanings of the dashed dependence shown in Part (b) (Part(c)) is referred to in Case 2(a) (Case 2(b)) discussed in Section 6.4.3.

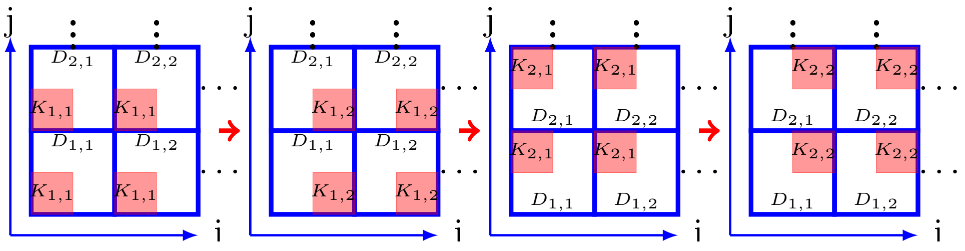


Figure 6.5: Sequential execution of four kernels, $\mathcal{K}_{1,1}$, $\mathcal{K}_{1,2}$, $\mathcal{K}_{2,1}$ and $\mathcal{K}_{2,2}$ in a forward sweep (with each subdomain divided into 2×2 tiles).

Our parallel MLSOR algorithm is guaranteed to converge following a similar line of reasoning as in [93, 138]. However, to accelerate the convergence rate, the bottom-left tile $\mathcal{T}_{1,1}$ is made as large as possible. As shown in Figure 6.4, the size of $\mathcal{T}_{1,1}$ shrinks by the increasing k, and computation stops when there is no point left in $\mathcal{T}_{1,1}$. This ensures that the height K in a K-layer sweep is the largest possible to maximize the chances for SOR to be applied. Therefore, if a subdomain has the size $K \times P \times P$, $\mathcal{T}_{1,1}$ is chosen to have $(P-1) \times (P-1)$ at the bottom layer. This ensures that the largest $K = \lfloor P/2 \rfloor$ is used. The sizes of the other three tiles $\mathcal{T}_{1,2}$, $\mathcal{T}_{2,1}$ and $\mathcal{T}_{2,2}$ are then determined accordingly by the tile-cutting hyperplanes given in Equation (6.9) as illustrated in Figure 6.4.

Figure 6.6 gives the pseudo code for kernel $\mathcal{K}_{1,1}$ in a forward sweep. A given mesh of size $\texttt{W} \times \texttt{L}$ is partitioned into $\frac{\texttt{W}}{\texttt{xgrid}} \times \frac{\texttt{L}}{\texttt{ygrid}}$ grids, where `xgrid` and `ygrid` are derived in lines 10 and 11. More than one grid may be necessary when one single grid is not large enough to cover the entire mesh. Therefore, all such grids are processed sequentially in lines 14 and 15. When each grid is executed, there are altogether `gridDim.x`$\times$`gridDim.y` thread blocks available in CUDA. In our parallelization scheme, each thread block consists of `blockdDim.x` $\times$ `blockDim.y` tiles with each tile being executed by one thread. The execution of a thread block proceeds in three steps. First, all threads in the block fetch the data required by the block from device memory to shared memory (lines 16 – 18). Next, these threads execute their allotted tiles in parallel (lines 19 – 26). Finally, these threads write the results back from shared memory to device memory (lines 27 – 29). The code blocks for data loads and stores are omitted as both are done in the standard manner with or without global memory coalescing. The effects of memory coalescing on the performance of MLSOR are discussed in Section 6.5.1.1.

Below we examine how the data dependences in the original SOR are dealt with and how tile sizes are determined. Consider the five data dependences depicted in Figure 6.1 for a K-layer forward sweep. There are two cases depending on whether they are intra- or inter-subdomain dependences:

1. **Intra-Subdomain Dependences.** There are two subcases depending whether these are intra- or inter-tile dependences. Intra-tile dependences are satisfied since the points in a tile are executed lexicographically. Inter-tile dependences are satisfied for the four tiles $\mathcal{T}_{1,1}$, $\mathcal{T}_{1,2}$, $\mathcal{T}_{2,1}$ and $\mathcal{T}_{2,2}$ in a subdomain since their corresponding kernels $\mathcal{K}_{1,1}$, $\mathcal{K}_{1,2}$, $\mathcal{K}_{2,1}$ and $\mathcal{K}_{2,2}$ are executed in that order (i.e., lexicographically in terms of tile indices). For the five dependences illustrated for $\mathcal{T}_{2,1}$ in D_{d_1,d_2} in Figure 6.4(a), the three from

```
// W*L is the mesh (problem) size
// K is the number of layers in a sweep
// K*P*P is the size of subdomains
// K*M*M is the size of tiles (where M=P-1 as per Section 4.3)
__global__ void MLSOR(float (*data)[W]) {
  int xIndex=blockDim.x*blockIdx.x+threadIdx.x;
  int yIndex=blockDim.y*blockIdx.y+threadIdx.y;
  int xblock=blockDim.x*P;
  int yblock=blockDim.y*P;
  int xgrid=gridDim.x*xblock;
  int ygrid=gridDim.y*yblock;
  __shared__ float local[blockDim.y*M+2][blockDim.x*M+2];
  int jj,ii,j,i,k;
  for (jj=1; jj<L-1; jj+=ygrid) {
    for (ii=1; ii<W-1; ii+=xgrid) {
// Loads data for this thread block from device to shared memory
      code for loading data[][] into local[][]
      __syncthreads();
// Performs MLSOR
      for(k=0;k<K;k++)
        for(j=(jj||yIndex)?k:0;j<M-k;j++)//identify edge subdomain
          for(i=(ii||xIndex)?k:0;i<M-k;i++)
            if(jj+blockIdx.y*yblock+P*threadIdx.y+j<L-1&&
              ii+blockIdx.x*xblock+P*threadIdx.x+i<W-1)
              Computing local[1+P*threadIdx.y+j][1+P*threadIdx.x+i]
      __syncthreads();
// Stores data for this thread block from shared to device memory
      code for storing data from local[][] into data[][]
      __syncthreads();
    }
  }
  // Checks convergence
  ...
}
```

Figure 6.6: Pseud code of $\mathcal{K}_{1,1}$ for MLSOR in a forward sweep.

$\mathcal{T}_{1,1}$ to $\mathcal{T}_{2,1}$ are satisfied since $\mathcal{T}_{1,1}$ is computed earlier than $\mathcal{T}_{2,1}$.

2. **Inter-Subdomain Dependences.** There are two subcases:

(a) **Top and Right Border Tiles: $\mathcal{T}_{1,2}$, $\mathcal{T}_{2,1}$ and $\mathcal{T}_{2,2}$.** If a point (k, i, j) in such a border tile of a subdomain depends on $(k-1, i, j)$, $(k-1, i+1, j)$ or $(k-1, i, j+1)$ computed in a bottom or left border tile of an adjacent subdomain (in Case 2(b)), the dependence is satisfied since the dependent point must have already been computed. Such is the case for the three dependences from $\mathcal{T}_{1,1}$ in D_{d_1+1,d_2} to $\mathcal{T}_{2,1}$ in D_{d_1,d_2} in Figure 6.4(b). This explains why the slanted hyperplanes in Figure 6.3 are used in our loop tiling. In particular, the most up-to-date value of a dependent point is always used. In the sequential SOR, the value used at grid point $(i+1, j)$ of $\mathcal{T}_{1,1}$ in D_{d_1+1,d_2} is computed at $(k-1, i+1, j)$ (along the dependence depicted with an blue arrow). In the parallel version, the value is fetched from $(k, i+1, j)$ (along the dashed dependence).

(b) **Bottom and Left Border Tiles: $\mathcal{T}_{1,1}$, $\mathcal{T}_{1,2}$ and $\mathcal{T}_{2,1}$.** As the opposite of Case 2(a), the situation is reversed except that the dependence from $(k-1, i, j)$ to (k, i, j) is always confined to the same subdomain. If a point (k, i, j) in such a border tile of a subdomain D depends on $(k, i-1, j)$ or $(k, i, j-1)$ computed in a top or right border tile of an adjacent subdomain D', then D' has not been executed yet. In this case, the most up-to-date value $(k-1, i-1, j)$ or $(k-1, i, j-1)$ already computed in D is used instead. The existence of such value is guaranteed due to the use of slanted hyperplanes in domain decomposition shown in Figure 6.2. This is illustrated in Figure 6.4(c). Point (k, i, j) of $\mathcal{T}_{1,1}$ in D_{d_1+1,d_2} requires the value of $(k, i-1, j)$ of $\mathcal{T}_{2,1}$ in D_{d_1,d_2} (along the dependence depicted with an blue arrow), which is computed after (k, i, j). Thus, the most up-to-date value $(k-1, i-1, j)$ of $\mathcal{T}_{1,1}$ in D_{d_1+1,d_2} (along the dashed dependence) is used.

There are four kernels used altogether in our parallelization scheme. When each kernel is executed, all inter-kernel dependences are satisfied by fetching the dependent data from global memory.

6.4.4 Parallelization for Multiple GPUs

The basic idea is to partition the mesh of a solver across the multiple GPUs and apply MLSOR to the sub-mesh allocated to a GPU. With one single GPU, kernels $\mathcal{K}_{1,2}$ and $\mathcal{K}_{2,1}$ do not have inter-kernel data dependences and can thus be combined with somewhat improved data reuse and reduced kernel startup overhead. However, this kernel fusion increases the frequency of inter-GPU communication in multiple GPUs, leading to reduced performance. Thus, the two kernels run sequentially in our experiments.

6.5 Experimental Evaluation on S1070

We demonstrate the performance advantages of MLSOR over RBSOR using NVIDIA S1070 GPUs. SOR-like PDEs can also be parallelized by the PLUTO compiler [7, 8]. However, the performance of SOR parallelized this way 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" according to the authors. Therefore, pattern-guided compiler is necessary to accelerate such applications. The framework is introduced in Chapter 3.

We present our results in two sections, with Section 6.5.1 focused on one GPU and Section 6.5.2 on multiple GPUs.

6.5.1 Results and Analysis for One GPU

In this section, we present and analyze the performance results and various tradeoffs that need to be made by a compiler for executing MLSOR and RBSOR on a single-GPU Tesla C1060. We focus more on MLSOR and touch on RBSOR briefly. For SOR, once the data operated by a thread are loaded into a buffer in the shared memory, there are few bank conflicts incurred. So we will not talk about bank conflicts any further.

Table 6.2 compares the convergence rates of SOR, MLSOR and RBSOR for three different inputs. The convergence rate of MLSOR depends on the subdomain size used. MLSOR is designed to trade off its convergence rate for data parallelism, as demonstrated below, and may converge more slowly than RBSOR. For all experimental results presented on one GPU in this section, the input data set used in Column 2, i.e., "Input 1" of this table is used. For the results presented in Section 6.5.2 on multiple GPUs, different inputs are used for different problem sizes.

We are now ready to explain the two reasons for applying our alternate tiling to the multi-layer SOR. First, better convergence is achieved than if a single sweeping direction is used as shown in Table 6.2. Second, inter-kernel data reuse can be exploited in some stream processors even though this is not presently possible for NVIDIA GPUs. As shown in Figure 6.2, any pair of mirrored subdomains in two adjacent sweeps access the same set of points. So MLSOR is expected to achieve much higher performance if such inter-kernel reuse can materialize.

6.5.1.1 MLSOR

We focus on two different subdomain sizes when $K \times P \times P = 2\times4\times4$ and $K \times P \times P = 4\times8\times8$. In each case, K is the largest possible to obtain the fastest convergence

Algorithm		Number of Iterations to Converge		
Input		1	2	3
Tolerance Error		0.001	0.001	0.000001
SOR		64	67	10214
RBSOR		78	70	10567
MLSOR	$2 \times 4 \times 4$	112 (139)	105 (120)	12769 (13215)
	$4 \times 8 \times 8$	96 (111)	91 (92)	11463 (12043)
	$8 \times 16 \times 16$	76 (82)	71 (85)	10735 (10997)
	$16 \times 32 \times 32$	71 (81)	64 (70)	10447 (10853)

Table 6.2: Convergence rates for three inputs of size 8192×8192 with the given tolerance errors shown. For MLSOR, different subdomain sizes lead to different convergence rates. The rates inside the brackets are obtained when only the forward sweeping direction is used.

as discussed in Section 6.4.3 and the best temporal reuse. For each subdomain size, we consider four different optimizations depending on whether loop unrolling and global memory coalescing are used or not. We first present the performance results of MLSOR and then analyze these results. We also discuss various tradeoffs along the way, highlighting the importance of compiler optimizations, performance modeling and performance tuning for improving the performance of DOACROSS loop nests.

Performance Figure 6.7 shows the execution times of MLSOR with respect to varying number of threads per thread block. Some performance bars are missing since in those configurations the 16KB shared memory (cf. Table 6.1) is not big enough to hold the data used by all the threads in a single thread block. We observe that the performance of MLSOR is sensitive to subdomain size. The effect of any optimization (or combination) on performance is non-linear due to complex interactions among various GPU architectural constraints (cf. Table 6.1).

Resource Usage and Performance Estimates We analyze the results of Figure 6.7 by making use of the resource usage information for kernel $\mathcal{K}_{1,1}$ from Ta-

Subdomain Size	Points/ Thread	Threads/ Block	Shared Memory/ Thread Block (bytes)	Registers/Thread		W_{TB}	B_{SM}	#Active Threads	Performance (GFLOPS)		Bandwidth (GB/s)	
				No Unrolling	Unrolling				No Unrolling	Unrolling	No Unrolling	Unrolling
$2 \times 4 \times 4$	3×3	4	404	16	13	0.125	40	160	99.7	82.2	46.9	78.7
		8	724			0.25	22	176				
		16	1364			0.5	12	192				
		32	2644			1	6	192				
		64	5204			2	3	192				
		128	10324			4	1	128				
$4 \times 8 \times 8$	7×7	4	1260	17	59	0.125	13	52	108.1	93.9	26.9	46.6
		8	2412			0.25	6	48				
		16	4716			0.5	3	48				
		32	9324			1	1	32				
		64	Out of Shared Memory									
		128										

Table 6.3: Resource usage of MLSOR (insensitive to coalescing) for kernel $\mathcal{K}_{1,1}$ under the configurations defined by the first three columns (and also illustrated in Figure 6.7).

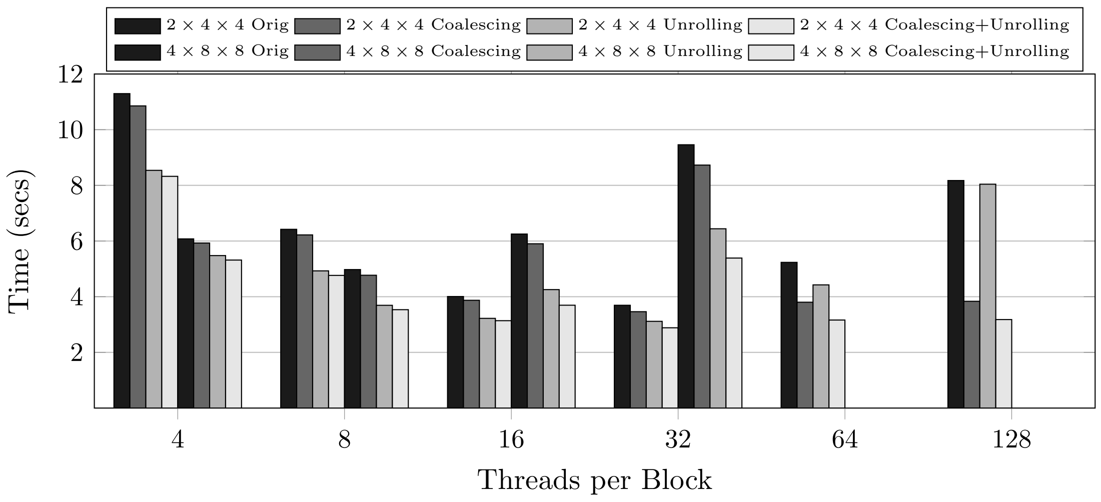


Figure 6.7: Execution times of MLSOR. In each case, Orig means that neither unrolling nor coalescing is performed.

ble 6.3. Memory coalescing does not appear in the table since it is immaterial to the statistics collected. A similar trend is observed if one of the other three kernels is used. For subdomain sizes $2 \times 4 \times 4$ and $4 \times 8 \times 8$, the points/thread values for $\mathcal{K}_{1,1}$ are 3×3 and 7×7, respectively, as shown in Figure 6.3. For each configuration identified by the first three columns together, a number of statistical data are listed. Columns 4 – 6 are self-explanatory. By compiling CUDA code with the `-cubin` flag, we could get some understanding about on-chip memory usage. In Column 7, W_{TB} is the number of warps in a thread block, which is determined by dividing the number of threads in a thread block by 32. In Column 8, B_{SM} is the number of thread blocks assigned to each SM. It is usually determined by shared memory and register usage (Columns 4 – 6). As indicated in Table 6.1, C1060 has 16KB shared memory and 16,384 registers per SM. Consider the configuration when the subdomain size is $2 \times 4 \times 4$ and threads/block is 8. Every thread block needs 724B shared memory. So the maximum number of simultaneously active blocks in one SM is $16KB \div 724B = 22$ (Column 8). The number of registers required by 22 threads block is 16 registers/thread $\times$ 8 threads/block $\times$ 22 active blocks $=$ 2816. Since this number is less than 16384, 22 blocks can be assigned to the same SM. Otherwise, B_{SM} is decided by register usage. The bottleneck then shifts from shared memory to registers. In a special case, although there are enough registers and shared memory to execute more blocks, the number of active threads may exceed the maximum value 1024 available per SM (cf. Table 6.1). B_{SM} has to change to satisfy this constraint. For this configuration, the largest number of active threads is 176 only (Column 9).

Finally, let us look at the last four columns in Table 6.3, which give performance and bandwidth estimates for kernel $\mathcal{K}_{1,1}$. Tesla C1060 is capable of issuing 240SPs $\times$ 1.3GHz $=$ 312 billion operations per second. When all the SPs are fully occupied, which is achievable in an application that has many threads, does not have many synchronization operations, and does not stress memory bandwidth. In this situation, for example, if 10% of a program instruction mix are fused FMAD and FMUL which can be done each GPU cycle, then its single-precision performance can be at most $3 \times 10\%\text{FP} \times 312 = 93.6\text{GFLOPS}$. We can obtain kernel assemble instructions through the `-ptx` flag. In Columns 10 and 11, the GFLOP estimates for $\mathcal{K}_{1,1}$ are given for both unrolled and non-unrolled cases, which will be further discussed in Section 6.5.1.1.

Another potential bottleneck is global memory bandwidth. If 5% of code are loads from off-chip memory, required bandwidth is 240SPs $\times$ 5%instructions $\times$ 4B/instruction $\times$ 1.3GHz $=$ 62.4GB/s. This value is estimated average bandwidth for running threads. It is possible that all threads simultaneously load data, thus the latency of accessing to global memory still exists. However, if this value without any global memory optimization is more than 102 GB/s, which is Tesla C1060's off-chip bandwidth, a lot of time will be spent on waiting for data transport and the bandwidth is likely to be bottleneck. In the last two columns, the bandwidth estimates for $\mathcal{K}_{1,1}$ are given for unrolled and non-unrolled cases.

Effects of Unrolling and Coalescing on Performance Full loop unrolling often achieves the best performance for MLSOR and is thus applied to obtain the results given in Figure 6.7. Unrolling improves data parallelism by removing branch instructions, and consequently, reduces significantly the dynamic number of instructions executed. With full unrolling, the MLSOR performance always improves as shown in Figure 6.7 although the GFLOP estimates have dropped as listed in Table 6.3. In addition, unrolling also affects register usage. In the case of $2 \times 4 \times 4$, the number of registers per thread has dropped from 16 to 13. For $4 \times 8 \times 8$, the register requirement increases noticeably from 17 to 59. Register usage reduction has also been observed in [83, 84] and is likely caused by eliminating loop induction variables and max operators that appear in loop bounds. Since the mechanism by which the CUDA runtime performs scheduling and register allocation is not visible to us, it is difficult to discuss further the existence of this non-uniform behavior. Although unrolling usually increases register pressure, the increase is small relative to the total number of registers available per SM (at least for MLSOR). Thus, full loop unrolling accelerates the MLSOR performance by reducing the dynamic instruction count executed.

Two strategies for reducing the negative impact of bandwidth on performance are to improve data reuse and reduce global memory access. The bandwidth is used most efficiently when the simultaneous memory accesses by threads in a half-warp can be coalesced into a single memory transaction. The memory system may be able to combine these into a single memory accessing request. Even if the average required bandwidth in our experiments is less than 102 GB/s and thus not the key bottleneck, the impact of coalescing on performance is still noticeable. As shown in Figure 6.7, coalescing always improves performance. In particular, when $B_{SM} = 1$

in Table 6.3, the performance benefit of coalescing is maximized. In this case, when threads in the unique block assigned to an SM stall on a load instruction, there exists no other blocks that can be scheduled by the SM to overlap computation and communication.

Correlating Configurations' Relative Performances We find from Figure 6.7 that it is difficult to establish a certain relation between an MLSOR configuration and its execution time. We use the two performance metrics from [84] to provide some rough estimates for the relative performance results of certain configurations. Both are meaningful only if global memory bandwidth is not the performance bottleneck. In addition, both metrics are established based on the PTX instructions (rather than actual machine instructions) of a kernel assuming that the PTX code correlates well with the actual performance of the kernel (at run time).

The Efficiency metric indicates the overall efficiency of a configuration in terms of the total number of instructions that must be executed before the kernel finishes:

$$\mathsf{Efficiency} = \frac{1}{Instr \times Threads} \tag{6.11}$$

where $Instr$ derived from the PTX code of a kernel estimates the number of dynamic instructions executed per thread and $Threads$ is the number of threads created by the kernel. In this work, this metric also reflects well the impact of data reuse on performance. When the tile size increases, $Threads$ decreases sharply. A thread will have more work to do. In addition, the convergence will be accelerated as shown in Table 6.2, resulting in a better efficiency.

The Utilization metric is about the utilization of the compute resources on a GPU by considering how often a warp may wait and the amount of work available

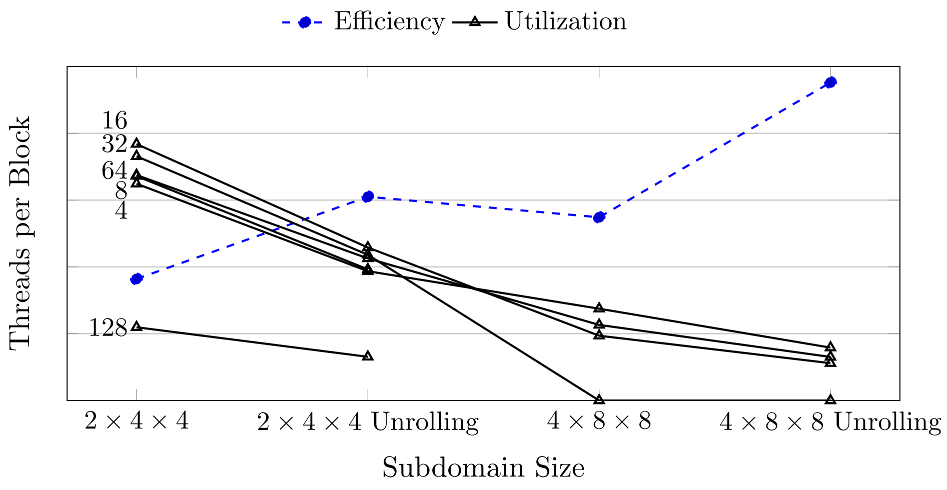


Figure 6.8: Efficiency and Utilization of MLSOR (insensitive to coalescing) of kernel $\mathcal{K}_{1,1}$ for all the configurations given in Figure 6.7.

(from other warps) when it does:

$$\text{Utilization} \quad = \quad \frac{Instr}{Regions}\left[\frac{W_{TB}-1}{2} + (B_{SM}-1) \times W_{TB}\right] \tag{6.12}$$

where *Regions* is the number of dynamic instruction intervals delimited by *blocking instructions* or the start/end of the kernel. Long latency instructions, such as texture memory operations and synchronization instructions, are considered as blocking instructions. $\frac{Instr}{Regions}$ gives the average number of non-blocking instructions per interval. The quantity within the brackets indicates the number of independent warps in an SM. The first term is the number of other currently executing warps in the same thread block. Dividing by two is for computing average possibility, half warps still need to execute. The second item is the number of warps in other thread blocks assigned to the same SM. When the degree of parallelism is low, the value of Utilization is small.

Figure 6.8 plots Efficiency and Utilization of $\mathcal{K}_{1,1}$ for all configurations in Figure 6.7. A solid curve represents Utilization as a function of different configurations for a fixed threads/block as marked. The single dashed curve represents Efficiency for all possible values of threads/block since their efficiency curves are identical.

It should be pointed out that these two metrics are not suitable for the configuration with only 4 threads per block. The full computing capability of an SM consisting of eight SPs is not fully utilized. Otherwise, there are good correlations between the Efficiency and Utilization metrics in Figure 6.8 and the actual performance results in Figure 6.7. Below we use these two metrics to analyze the effects of thread granularity and loop unrolling on performance.

First of all, for both unrolled and non-unrolled code, Utilization drops when thread granularity increases. This implies that the number of active and ready threads is cut down because larger threads consume more resources. In limited resource situations, shared memory and register usage affect the throughput of an SM. However, since an increase in Efficiency counteracts a decrease in the throughput caused by reduced Utilization, MLSOR is not always slower when $4 \times 8 \times 8$ is used, since larger threads, i.e., subdomains lead to faster convergence rates as shown in Table 6.2.

Next, for the same subdomain size with and without unrolling, the improved Efficiency due to unrolling is entirely attributed to a reduction in $Instr$ since $Threads$ remains unchanged. Moreover, unrolling does not alter the number of memory accesses. So $Regions$ should not change remarkably. According to Table 6.3, unrolling usually does not reduce resource usage. As a result, Utilization worsens as $instr$ decreases. However, the gain from improved Efficiency seems to more than offset the loss caused by worsened Utilization here. So unrolling is always beneficial for MLSOR.

Furthermore, looking at the combined effects of thread granularity and unrolling on performance, MLSOR has better Efficiency and Utilization under "$2 \times 4 \times 4$ Unrolling" than "$4 \times 8 \times 8$" (without unrolling). This means that a decrease in $Instr$ due to unrolling affects the two metrics more than an increase in thread

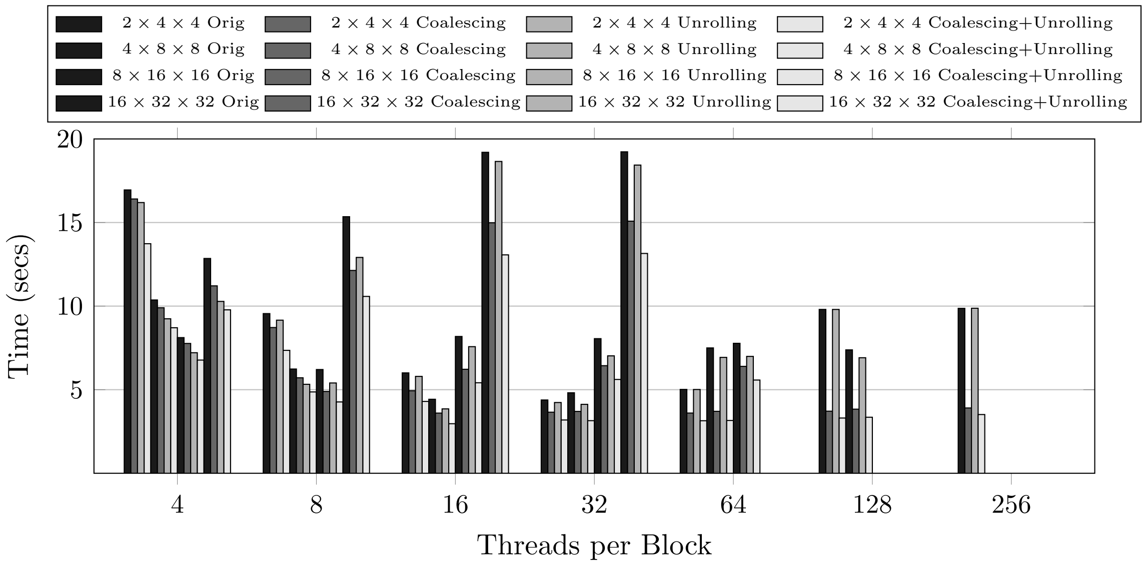


Figure 6.9: Execution times of MLSOR with shared memory reduction.

granularity, i.e., subdomain size. Consequently, "2 × 4 × 4 Unrolling" results in better Efficiency than "4 × 8 × 8", which has better Efficiency than "2 × 4 × 4". This analysis correlates well with the results given in Figure 6.7, where MLSOR runs faster under "2 × 4 × 4 Unrolling" except for the pathological "4 threads/block" case.

Finally, the weights of Efficiency and Utilization are unpredictable, and the optimal configuration may need to balance both metrics. This is consistent with the observation made in [84], highlighting the importance of using a tuning tool for efficient solution space exploration.

Shared Memory Reduction From Table 6.3, we see that the bottleneck of MLSOR is shared memory. This work highlights the importance for future GPU architectures to aggressively exploit both intra- and inter-kernel data reuse in scientific and engineering applications to boost the performance of DOACROSS loop nests. Given the 16KB shared memory, the scarce resource must be effectively utilized with some shared memory reduction technique. One solution is to undo

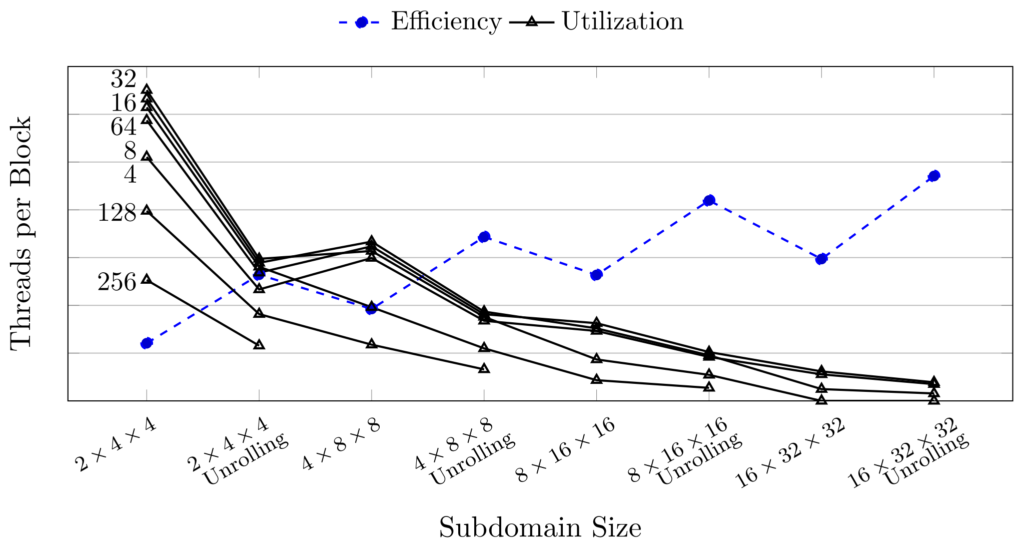


Figure 6.10: Efficiency and Utilization of MLSOR (insensitive to coalescing) of kernel $\mathcal{K}_{1,1}$ with shared memory reduction.

the merge of the tiles in $\{T_{i,j} \mid 0 < i, j < m\}$ into $\mathcal{T}_{1,1}$ so as to execute the smaller tiles in separate kernels. But using smaller tiles leads to smaller tile height K as discussed in Section 6.4.3, resulting in slower convergence and poorer data reuse. To avoid these problems and to facilitate memory coalescing, a different solution is used. When a row of points in a tile, i.e., $\mathcal{T}_{1,1}$, $\mathcal{T}_{1,2}$, $\mathcal{T}_{2,1}$ or $\mathcal{T}_{2,2}$ are computed, only this row and its two adjacent rows are kept in the shared memory. This solution also sacrifices some temporal reuse but allows memory coalescing to be realized more effectively.

With the shared memory reduction technique being applied, Figures 6.9 and 6.10 are now given as the analogues of Figures 6.7 and 6.8, respectively. Some observations are in order. First, the performance of MLSOR drops slightly with shared memory reduction in most but not all configurations. Second, MLSOR did not compile before for $4 \times 8 \times 8$ when the threads/block is 64 or 128 due to lack of shared memory but compiles now. Third, two larger subdomain sizes $8 \times 16 \times 16$ and $16 \times 32 \times 32$ are now included and can compile except for a few large threads/block values.

Table 6.4 is an analogue of Table 6.3 for Figure 6.9. The GFLOPS and bandwidth values corresponding to the "$2 \times 4 \times 4$" and "$4 \times 8 \times 8$" rows in Table 6.4 are "121.0 70.5 58.0 123.9" and "124.5 70.1 62.1 101.3", respectively. Memory reduction is applied no matter whether loop unrolling is used or not. In either case, with memory reduction, the required bandwidths are higher but the GFLOPS values do not change as much.

Again there are good correlations between Figure 6.9 and Figure 6.10. In particular, there is an important point worth being restated. Although "$4 \times 8 \times 8$" results in lower Efficiency than "$2 \times 4 \times 4$ Unrolling", MLSOR is a better performer under "$4 \times 8 \times 8$" when the threads/block is 8 and 16 due to higher Utilization. When the threads/block increases to 32, the effect of a decrease in Efficiency on performance is larger than that of an increase in Utilization, MLSOR is slower under "$4 \times 8 \times 8$". However, the situation is different in 128 threads/block. MLSOR performs better under "$4 \times 8 \times 8$" even though its Efficiency and Utilization values are both lower. As mentioned earlier, the required bandwidth of "$2 \times 4 \times 4$ Unrolling" is 123.9 GB/s, which is beyond the maximum 102 GB/s available in Tesla C1060, while the required bandwidth of "$4 \times 8 \times 8$" is 101.3GB/s. Without coalescing, the GPU may stall on waiting for accessing to global memory. As a result, the execution time under "$2 \times 4 \times 4$ Unrolling" is prolonged. With coalescing, however, the overall memory time is reduced. Thus, MLSOR performs slightly better under "$2 \times 4 \times 4$ Unrolling + Coalescing" than "$4 \times 8 \times 8$".

With shared memory reduction, the number of active threads can sometimes increase by nearly 87.5%. Thus, finer data partitioning reduces an application's demand for resources and increases its degree of parallelism. However, it may affect negatively other architectural constraints, i.e. by saturating the bandwidth. Again

Subdomain Size	Points/ Thread	Threads/ Block	Shared Memory/ Thread Block (bytes)	Registers/Thread		W_{TB}	B_{SM}	#Active Threads	Performance (GFLOPS)		Bandwidth (GB/s)	
				No Unrolling	Unrolling				No Unrolling	Unrolling	No Unrolling	Unrolling
$2 \times 4 \times 4$	4×3	4	268	27	26	0.125	61	244	121.0	70.5	58.0	123.9
		8	460			0.25	35	280				
		16	844			0.5	19	304				
		32	1612			1	10	320				
		64	3148			2	5	320				
		128	6220			4	2	256				
		256	12364			8	1	256				
$4 \times 8 \times 8$	7×7	4	468	27	26	0.125	35	140	124.5	70.1	62.1	101.3
		8	852			0.25	19	152				
		16	1620			0.5	10	160				
		32	3156			1	5	160				
		64	6228			2	2	128				
		128	12372			4	1	128				
$8 \times 16 \times 16$	15×15	4	868	27	26	0.125	18	72	125.2	70.7	58.0	92.2
		8	1636			0.25	10	80				
		16	3172			0.5	5	80				
		32	6244			1	2	64				
		64	12388			2	1	64				
$16 \times 32 \times 32$	31×31	4	1668	25	25	0.125	9	36	125.2	71.6	55.8	88.4
		8	3204			0.25	5	40				
		16	6276			0.5	2	32				
		32	12420			1	1	32				

Table 6.4: Resource usage of MLSOR (insensitive to coalescing) for kernel $\mathcal{K}_{1,1}$ under the configurations defined by the first three columns (and also illustrated in Figure 6.9).

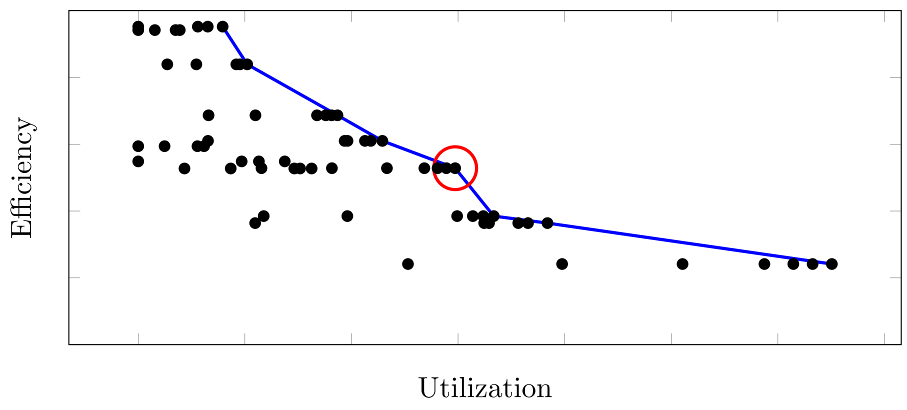


Figure 6.11: Searching for optimal solutions by performance metrics (illustrated using Efficiency and Utilization for kernel $\mathcal{K}_{1,1}$). The best configuration is highlighted by a circle.

the optimal configuration requires a balanced tradeoff to be made.

Summary of Performance Metrics Figure 6.11 plots the two metric values for all configurations given in Figures 6.7 and 6.9. The maximum metric value along each axis has been normalized to one for comparison purposes. In general, the best performance should come from configurations with both high Efficiency and Utilization although their weights are difficult to valuate [84]. Thus, one desires configurations located towards the upper right corner of the graph. The points connected by the line have higher opportunity to get better performance than others. The circled point for “$4 \times 8 \times 8$ Coalescing+Unrolling” using 16 threads per block with shared memory reduction is the best performer.

6.5.1.2 RBSOR

We have implemented RBSOR taken from the Java Grande benchmark suite for CUDA following [131]. The performance results are displayed in Figure 6.12. RB-SOR exhibits the same performance with 1×1 points/thread when threads/block ranges from 64 to 512. By examining resource usage, we find that the bottleneck in

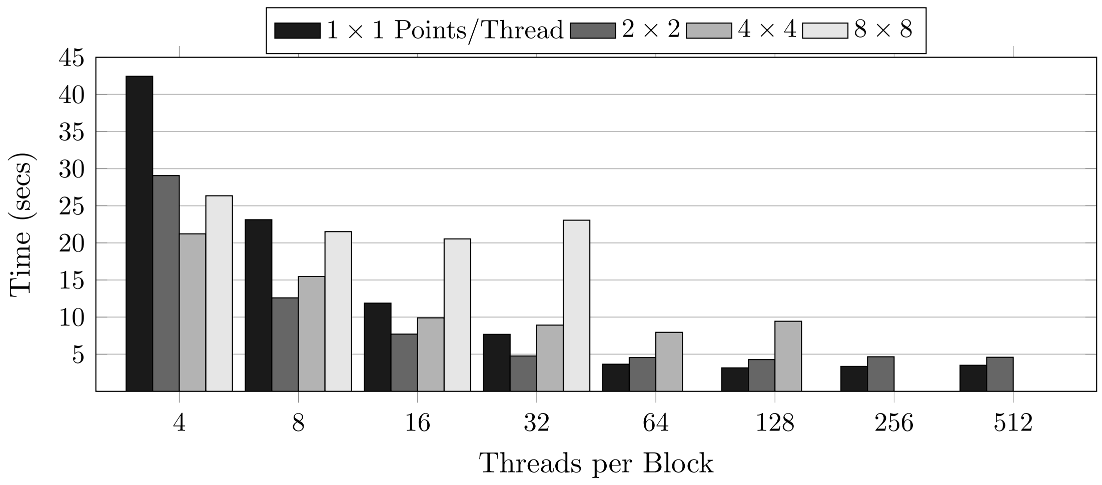


Figure 6.12: Execution times of RBSOR.

this case is neither shared memory nor registers. With 64 or more threads per block, the number of threads to be launched per SM exceeds 1024 (cf. Table 6.1). Thus, the bottleneck is instruction issue. Therefore, fine-grained parallelism often gives rise to good performance on GPU architectures. But the overall performance can be constrained by architectural constraints, such as the number of active threads allowed, if the data reuse is not adequately exploited.

For RBSOR, the Efficiency and Utilization metrics do not appear to be sufficient in explaining its performance results. We have made an attempt to understand its performance trend through experimentation, analysis and also consulting [131]. RBSOR seems to run at its full speed at 1×1 points/thread with 128 threads/block. Both RBSOR and MLSOR are compared in detail below.

6.5.2 Results and Analysis for Multiple GPUs

We compare MLSOR and RBSOR on a Tesla S1070 computing system consisting of four GPUs. We evaluate MLSOR and RBSOR using their configurations giving rise to the best single-GPU performances. These may not be the absolute best for a multi-GPU setting but seem to be a good choice for stencil-based computations.

MLSOR uses a subdomain of size $2 \times 4 \times 4$ with 32 threads/block with shared memory reduction while RBSOR's configuration is 1×1 points/thread with 128 threads/block.

The host is an Intel Xeon Quad-core CPU running at 2.66GHz. The mesh of an SOR solver is distributed block-wise along one dimension to the four GPUs. We need to use a series of CPU threads to schedule and manage the execution of the sub-meshes allocated to the GPUs. In our experiments, four CPU threads are created to run on four CPUs concurrently. Each CPU thread is associated with an individual GPU. It is responsible for distributing the required data in the sub-mesh to the device memory of its associated GPU, scheduling kernel execution on it, and communicating the boundary data of sub-meshes with the other GPUs indirectly via the host.

Figure 6.13 shows the speedups of MLSOR over RBSOR. Overall, MLSOR performs better with increasingly larger problem size and more GPUs. However, the performance increases are not linear. We analyze this phenomenon by separating the inter-GPU communication cost from the computation cost during a program execution. All device-to-device copies are asynchronous. The associated idle times are not stable in different runs of the same program. Thus, the inter-GPU communication time of a program is measured as an average of 10 program runs. Figure 6.15 shows the inter-GPU communication overhead increases of RBSOR over MLSOR as the problem size increases on more and more GPUs. Figure 6.14 replots Figure 6.13 with the inter-GPU communication costs being annihilated. Now, the computation speedup of MLSOR over RBSOR increases more smoothly than before as the problem size increases across the multiple GPUs.

Given a problem size, MLSOR and RBSOR incur about the same amount of inter-GPU communication. The difference lies in the frequency of communication.

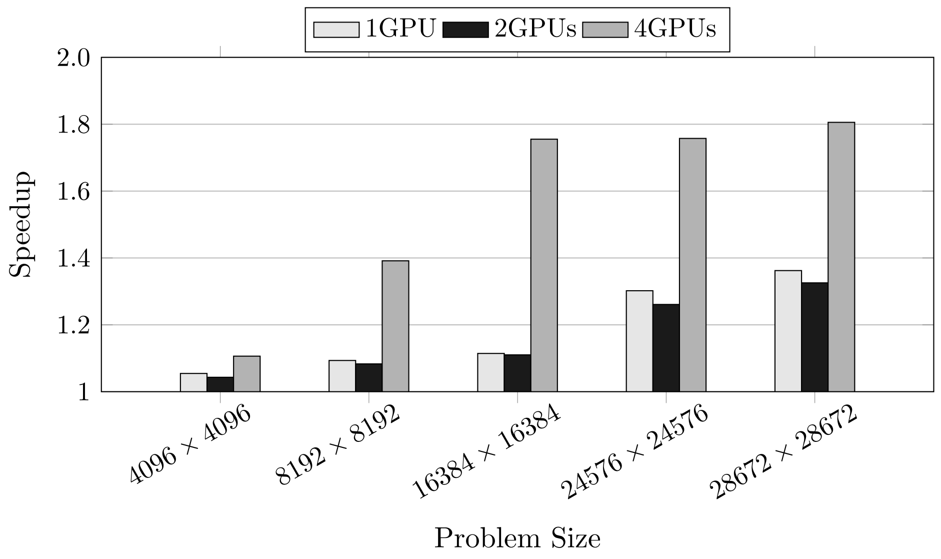


Figure 6.13: Speedups of MLSOR over RBSOR on S1070.

For MLSOR, the inter-GPU communication occurs when $\mathcal{K}_{1,2}$ and $\mathcal{K}_{2,2}$ run to completion. Note that if $\mathcal{K}_{1,2}$ and $\mathcal{K}_{2,1}$ were merged (as discussed Section 6.4.4), the inter-GPU communication would occur when every kernel completes, causing a 50% increase in communication frequency. However, RBSOR communicates four times as many as MLSOR. As thread granularity increases, the frequency of communication incurred by MLSOR decreases. However, due to the idle times elapsed in inter-GPU communication, performance fluctuations are expected across the problem sizes.

6.6 Experimental Evaluation on C2050

In this section, we show MLSOR performance on Tesla S2050 Fermi system. C2050 has 64KB on-chip memory which is can be configured as 48KB of shared memory with 16KB of L1 cache or as 16KB of shared memory with 48KB of L1 cache. According to analysis above, in most cases, the bottleneck of MLSOR is lack of

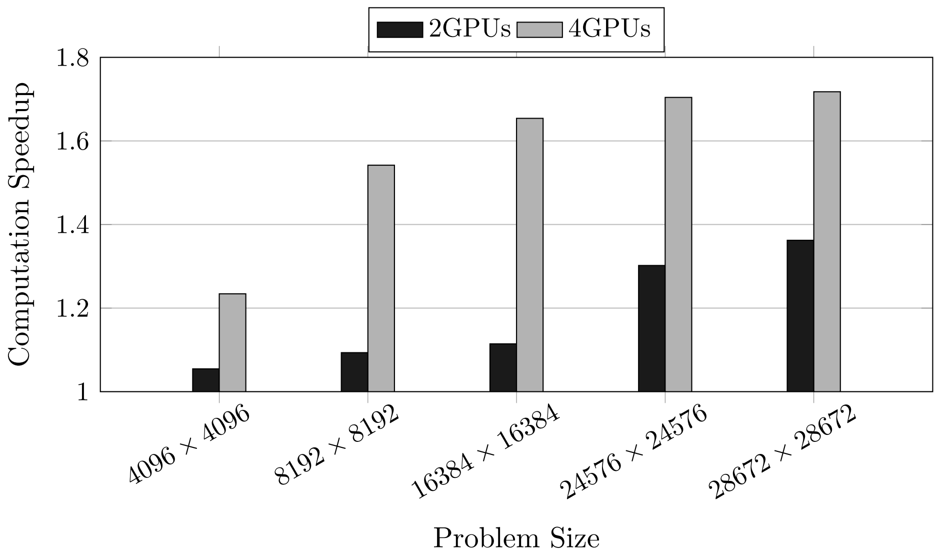


Figure 6.14: Computation speedups of MLSOR over RBSOR on S1070 (with zero communication overhead assumed).

shared memory. Moreover, MLSOR loads and stores data between global memory and shared memory once for each tile. There are not over frequent I/O operation, thereby large L1 cache allocation not benefiting improving performance, particularly when using coalescing optimization. As a result, our experiments use CU_FUNC_CACHE_PREFER_SHARED to configure larger shared memory which guarantees activate more threads.

Through single C2050 experiment, we choose a subdomain of size $2 \times 4 \times 4$ with 512 threads/block as optimal configuration. When problem size is 28672×28672 which exceeds global memory size of C2050, it cannot be executed on single GPU. Because S2050 has enough shared memory to activate more threads, the latency of switching warps can be hidden when pipeline is full, and aggregate throughput of dependent SP more easily saturates. Similar with result on S1070, the computation speedup without communication overhead more smoothly increases than the total performance.

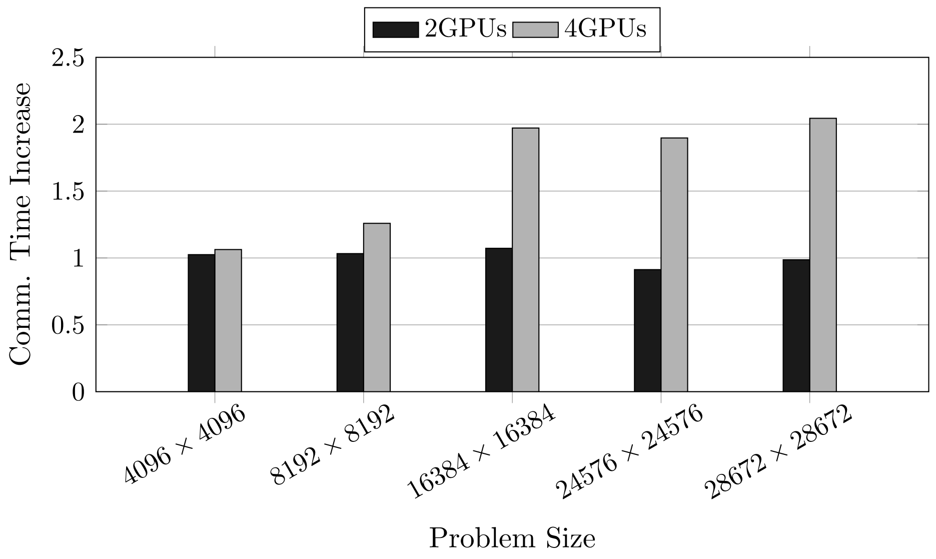


Figure 6.15: Inter-GPU communication overhead increases of RBSOR over MLSOR on S1070.

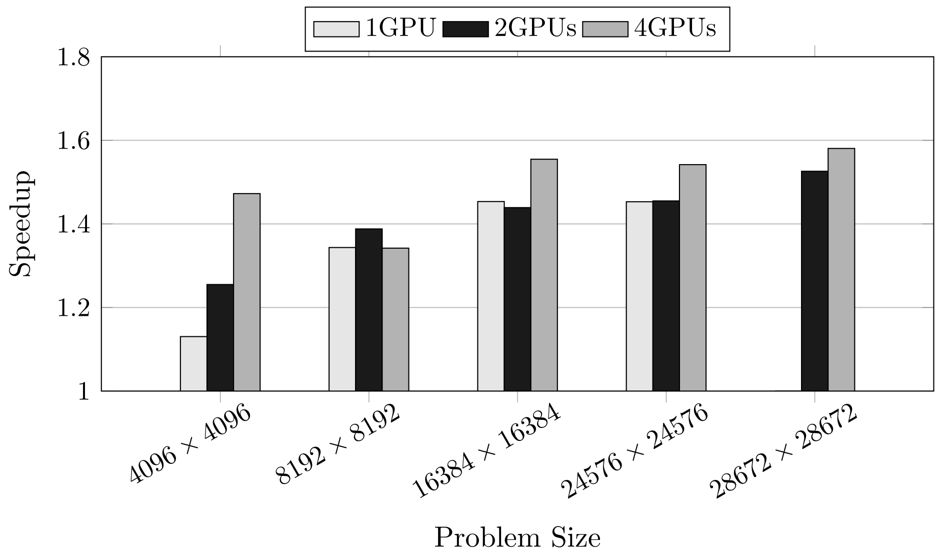


Figure 6.16: Speedups of MLSOR over RBSOR on S2050.

6.7 Conclusion

We have presented a new parallel SOR solver, which is developed based on a non-dependence-preserving parallelization scheme, and demonstrated its performance advantages over RBSOR. Our experimental results, validated by detailed analysis, show that many DOACROSS loop nests may be potentially accelerated if different

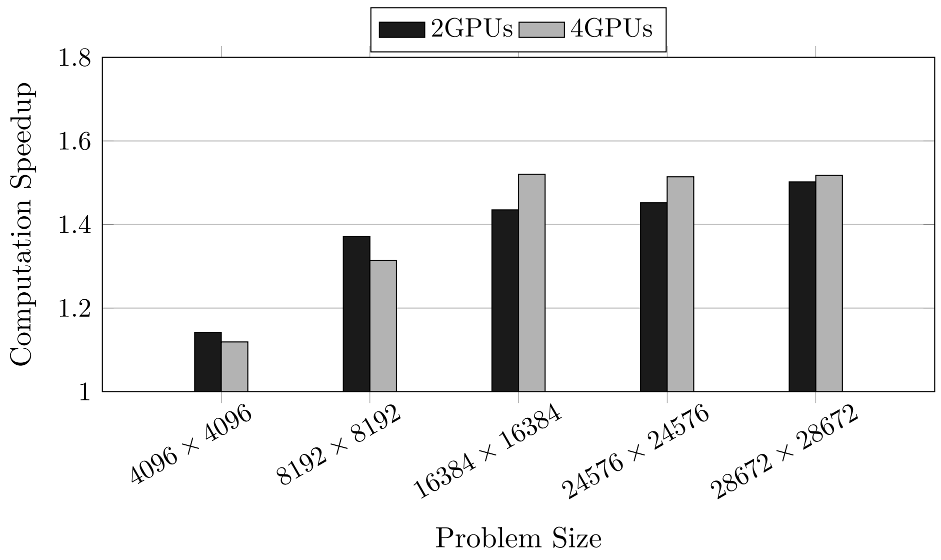


Figure 6.17: Computation speedups of MLSOR over RBSOR on S2050 (with zero communication overhead assumed).

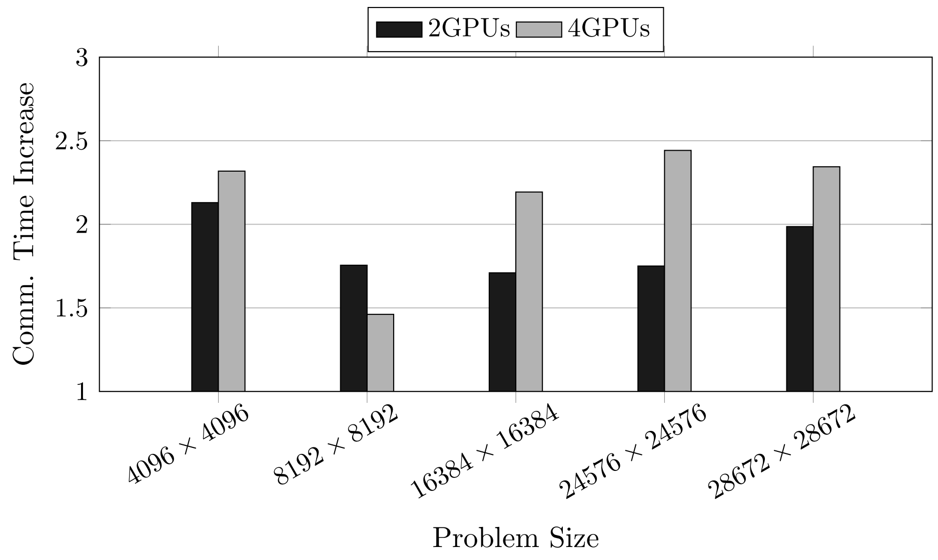


Figure 6.18: Inter-GPU communication overhead increases of RBSOR over MLSOR on S2050.

algorithms that are more amenable to GPU computing can be developed. The importance of synergy between domain experts, compiler optimizations and performance tuning is highlighted in maximizing application performance.

DOACROSS loop nests are difficult to run efficiently on GPUs since their cross-iteration dependences pose a major obstacle to the exploitation of the fine-grained parallelism in these loop nests. We hope that our experience on parallelizing the SOR method can be useful for others to establish optimization principles and strategies for accelerating the performance of DOACROSS loop nests on GPU-accelerated computing systems.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This thesis focuses on implementing a C-to-CUDA compiler framework based on the polyhedral model. It automatically parallelizes sequential C loop nests, especially stencil kernels, into optimized parallel CUDA code for GPUs. To improve the performance, three novel and efficient techniques are introduced in our compiler framework. The experimental results demonstrate that these advances can outperform the existing algorithms.

To improve two levels of wavefront parallelism for loop nests running on GPUs, Chapter 4 presents a novel tiling technique based on a dependence-preserving parallelization scheme. The new approach embeds parallelism-enhancing constraints in the polyhedral model to maximize intra-tile parallelism, and eliminates parallelism-hindering false dependences to maximize inter-tile parallelism. This technique is implemented in our automatic code generation framework to optimize tile shapes of loop tiling by finding appropriate tiling hyperplanes with improved parallelism.

Chapter 5 presents a model-driven approach to automating tile size selection that is performance-critical, particularly for DOACROSS loop nests. Our cost model can accurately estimate the execution times of the tiled loop nests running on GPUs. The selected tile sizes lead to the performance results close to the best observed for a range of problem sizes tested. The approach enables our compiler to determine tile sizes of loop tiling, so that loop nests can be efficiently parallelized for GPUs.

To facilitate parallelizing SOR-like DOACROSS loop nests, Chapter 6 presents a new parallel SOR method that admits more efficient data-parallel SIMD execution than the traditional RBSOR on GPUs. Our new method outperforms RBSOR not only by making a better balance between data reuse and parallelism, but also by trading off convergence rate for SIMD parallelism. The new approach can be implemented in a pattern-guided compiler to make SOR-like DOACROSS loop nests more amendable to GPU parallelization.

7.2 Future Work

During the past decade, significant researches on the polyhedral model have been applied to code representative of real applications. The polyhedral model is practically designed as a part of optimizer in many mainstream compilers. Due to its flexibility and strength of composing and applying loop transformations, the polyhedral model could replace the standard Loop Nest Optimizer (LNO) pass in compilers, such as GRAPHITE for GCC, URUK for Open64 and Polly for LLVM.

Our framework described in Chapter 3 can only handle some polyhedral-friendly loop nests that cannot contain any conditional branches and unknown loop bounds [78]. In order to overcome this limitation, Polly and the NVVM library can be used to assist for the implementation of the improved loop tiling transformations presented in Chapters 4 and 5. Polly is a polyhedral optimizer for the LLVM compiler built on top of LLVM IR. It analyzes and optimizes the memory access pattern of a program, to facilitate cache locality optimizations and automatic thread-level parallelization. The NVVM library, a target-specific subset of the LLVM IR, has recently been released by NVIDIA to translate LLVM IR into GPU executable code via the PTX language. By leveraging the Polly and the NVVM library, our work proposed in this thesis can be integrated into the LLVM compiler. In such way, the existing analyses and optimizations of the LLVM compiler can be utilized to convert as many as loop nests into polyhedral representations. Once it is done, the user can use it to parallelize myriad sequential applications to explore GPUs' potential without requiring any extra programming effort.

Moreover, our pattern-guided approach proposed in this thesis is generic, but not limited to one special application such as SOR-like loop nests. There are some new and interesting research opportunities. First, numerous patterns widely used for loop nests need to be systemically classified and extracted to simplify the complexity of parallelization. For example, a considerable amount of prior work [64, 99] has been conducted on the theories of asynchronous iteration algorithms with the benefit of reducing the synchronization overhead and communication cost on distributed systems. Such kind of parallel algorithms has substantial potential to accelerate similar applications for GPUs as an available pattern. Second, it is practical but challenging to extend our pattern-guided approach to heterogeneous systems. For example, by carefully designing the appropriate strategies of task scheduling, the approach is also applicable to exposing the parallelism of a CPU/GPU architecture. Last but not least, since extracting various patterns from large programs is costly and difficult, devising easy-to-use program directives to resolve different patterns is crucial and useful in practice.

Bibliography

[1] W. Abu-Sufah, D. J. Kuck, and D. H. Lawrie. On the performance enhancement of paging systems through program analysis and transformations. *IEEE Transactions on Computers*, 30(5):341–356, May 1981.

[2] L. Adams and J. Ortega. A multi-color SOR method for parallel computation. In *ICPP*, pages 53–56, 1982.

[3] M. Amini, F. Coelho, F. Irigoin, and R. Keryell. Static compilation analysis for host-accelerator communication optimization. In *LCPC*, 2011.

[4] R. Andonov and S. Balev. Optimal semi-oblique tiling. *IEEE Transactions on Parallel and Distributed Systems*, 14:960, 2001.

[5] R. Andonov and S. Rajopadhye. Optimal orthogonal tiling of 2-D iterations. *Journal of Parallel and Distributed Computing*, 45(2):159–165, Sept. 1997.

[6] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W. W. Hwu. An adaptive performance modeling tool for GPU architectures. In *PPoPP*, pages 105–114, 2010.

[7] M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for GPGPUs. In *ICS*, pages 225–234, 2008.

[8] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA Code Generation for Affine. In *CC*, pages 244–263, 2010.

[9] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT*, pages 7–16, 2004.

[10] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting polyhedral loop transformations to work. In *LCPC*, pages 209–225, october 2003.

[11] C. Bastoul and P. Feautrier. Improving data locality by chunking. In *CC*, pages 320–334, 2003.

[12] U. Bondhugula. *Effective Automatic Parallelization and Locality Optimization Using The Polyhedral Model.* PhD thesis, Ohio State University, 2010.

[13] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *CC*, pages 132–146, 2008.

[14] S. H. Brill and G. F. Pinder. A blockred-black SOR method for a two-dimensional parabolic equation using Hermite collocation. *The Mathematics of Finite Elements and Applications*, 1997.

[15] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *SIGGRAPH*, pages 777–786, 2004.

[16] P. Calland, A. Darte, Y. Robert, and F. Vivien. On the removal of anti and output dependences. *International Journal of Parallel Programming*, 26(2):285–312, 1998.

[17] P.-Y. Calland, A. Darte, Y. Robert, and F. Vivien. Plugging anti and output dependence removal techniques into loop parallelization algorithm. *Parallel Computing*, 23(1C2):251–266, 1997.

[18] S. Chen and J. Xue. Partitioning and scheduling loops on nows. *Computer Communications*, 22(11):1017–1033, 1999.

[19] A. Cohen, M. Sigler, S. Girbal, O. Temam, D. Parello, and N. Vasilache. Facilitating the search for compositions of program transformations. In *ICS*, pages 151–160, 2005.

[20] H. Cui, L. Wang, J. Xue, Y. Yang, and X. Feng. Automatic library generation for BLAS3 on GPUs. In *IPDPS*, pages 255–265, 2011.

[21] H. Cui, J. Xue, L. Wang, Y. Yang, X. Feng, and D. Fan. Extendable pattern-oriented optimization directives. In *CGO*, pages 107–118, 2011.

[22] H. Cui, J. Xue, L. Wang, Y. Yang, X. Feng, and D. Fan. Extendable pattern-oriented optimization directives. *ACM Transactions on Architecture and Code Optimization*, 9(3):14:1–14:37, Oct. 2012.

[23] H. Cui, Q. Yi, J. Xue, and X. Feng. Layout-oblivious compiler optimization for matrix computations. *ACM Transactions on Architecture and Code Optimization*, 9(4):35:1–35:20, Jan. 2013.

[24] H. Cui, Q. Yi, J. Xue, L. Wang, Y. Yang, and X. Feng. A highly parallel reuse distance analysis algorithm on GPUs. In *IPDPS*, pages 1080–1092, 2012.

[25] D. Cunningham, R. Bordawekar, and V. Saraswat. GPU programming in a high level language: compiling X10 to CUDA. In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, pages 8:1–8:10, 2011.

[26] A. Darte, R. Schreiber, and G. Villard. Lattice-based memory allocation. In *CASES*, page 298, 2003.

[27] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC*, pages 1–12, 2008.

[28] P. Di, Q. Wan, X. Zhang, H. Wu, and J. Xue. Toward harnessing DOACROSS parallelism for multi-GPGPUs. In *ICPP*, pages 40–50, 2010.

[29] P. Di, H. Wu, J. Xue, F. Wang, and C. Yang. Parallelizing SOR for GPGPUs using alternate loop tiling. *Parallel Computing*, 38(6-7):310–328, 2012.

[30] P. Di and J. Xue. Model-driven tile size selection for DOACROSS loops on GPUs. In *Euro-Par*, pages 401–412, 2011.

[31] P. Di, J. Xue, C. Hu, and J. Zhou. A cache-efficient parallel gauss-seidel solver with alternating tiling. In *ICPADS*, pages 244–251, 2009.

[32] P. Di, D. Ye, Y. Su, Y. Sui, and J. Xue. Automatic parallelization of tiled loop nests with enhanced fine-grained parallelism on GPUs. In *ICPP*, pages 350–359, 2012.

[33] R. Dolbeau, S. Bihan, and F. Bodin. HMPP: A hybrid multi-core parallel programming environment. In *First Workshop on General Purpose Processing on Graphics Processing Units*, pages 1–5, 2007.

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

[35] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink. Compiling a high-level language for GPUs: (via language support for architectures and compilers). In *PLDI*, pages 1–12, 2012.

[36] P. Feautrier. Array expansion. In *ICS*, pages 429–441, 1988.

[37] P. Feautrier. Parametric integer programming. *RAIRO Recherche Operationnelle*, 22:243–268, 1988.

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

[39] P. Feautrier. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. *International Journal of Parallel Programming*, 21(5), 1992.

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

[41] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, June 2006.

[42] G. Golub, X. Wu, and J.-Y. Yuan. SOR-like methods for augmented systems. *BIT Numerical Mathematics*, 41(1):71–85, 2001.

[43] G. Goumas, N. Drosinos, V. Karakasis, and N. Koziris. Coarse-grain parallel execution for 2-dimensional PDE problems. In *IPDPS*, pages 1–8, 2007.

[44] M. Griebl. *Automatic Parallelization of Loop Programs for Distributed Memory Architectures.* PhD thesis, University of Passau, 2004.

[45] M. Griebl, C. Lengauer, and S. Wetzel. Code generation in the polytope model. In *PACT*, pages 106–111, 1998.

[46] T. Grosser, H. Zheng, R. Aloor, A. Simburger, A. Groblinger, and L.-N. Pouchet. Polly-Polyhedral optimization in LLVM . In *IMPACT*, Apr. 2011.

[47] A. Hartono, M. Baskaran, C. Bastoul, A. Cohen, S. Krishnamoorthy, B. Norris, J. Ramanujam, and P. Sadayappan. Parametric multi-level tiling of imperfectly nested loops. In *ICS*, pages 147–157, 2009.

[48] K. Högstedt, L. Carter, and J. Ferrante. Selecting tile shape for minimal execution time. In *SPAA*, pages 201–211, 1999.

[49] J. Holewinski. High-performance code generation for stencil computations on GPU architectures. In *ICS*, pages 311–320, 2012.

[50] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *ISCA*, pages 152–163, 2009.

[51] Q. Huang, J. Xue, and X. Vera. Code tiling for improving the cache performance of PDE solvers. In *ICPP*, pages 615–625, 2003.

[52] X. Huo, V. Ravi, W. Ma, and G. Agrawal. An execution strategy and optimized runtime support for parallelizing irregular reductions on modern GPUs. In *ICS*, pages 2–11, 2011.

[53] F. Irigoin and R. Triolet. Supernode partitioning. In *POPL*, pages 319–329, 1988.

[54] M. Jiménez, J. M. Llabería, and A. Fernández. Register tiling in nonrectangular iteration spaces. *ACM Transactions on Programming Languages and Systems*, 24(4):409–453, July 2002.

[55] U. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany. The imagine stream processor. In *ICCD*, pages 282–288, 2002.

[56] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *FRONTIERS*, pages 332–341, 1995.

[57] R. Keryell, R. Keryell, C. Ancourt, B. Creusillet, F. Coelho, P. Jouvelot, and F. Irigoin. PIPS: a workbench for building interprocedural parallelizers, compilers and optimizers. Technical report, Mines ParisTech, 1996.

[58] D. Kim, L. Renganarayanan, D. Rostron, S. Rajopadhye, and M. Strout. Multi-level tiling: M for the price of one. In *SC*, pages 1–12, 2007.

[59] S. Lee and R. Eigenmann. OpenMPC : Extended OpenMP programming and tuning for GPUs. In *SC*, pages 1–11, 2010.

[60] S. Lee, S. Min, and R. Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *PPoPP*, volume 44, pages 101–110, 2009.

[61] C. Lengauer. Loop parallelization in the polytope model. *CONCUR*, pages 1–19, 1993.

[62] Z. Li. Array privatization for parallel execution of loops. In *ICS*, pages 313–322, 1992.

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

[64] L. Liu and Z. Li. Improving parallelism and locality with asynchronous algorithms. In *PPoPP*, pages 213–222, 2010.

[65] Y. Liu, E. Z. Zhang, and X. Shen. A cross-input adaptive framework for GPU programs optimization. In *IPDPS*, pages 16–19, 2009.

[66] D. Maydan, S. Amarsinghe, and M. Lam. Data dependence and data-flow analysis of arrays. In *LCPC*, pages 434–448, 1993.

[67] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[68] R. G. Melhem and K. V. S. Ramarao. Multicolor reordering of sparse matrices resulting from irregular grids. *ACM Transactions on Mathematical Software*, 14(2):117–138, 1988.

[69] J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In *ICS*, pages 256–265, 2009.

[70] P. Micikevicius. 3D finite difference computation on GPUs using CUDA. In *GPGPU-2*, pages 79–84, 2009.

[71] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. In *SC*, pages 1–13, 2010.

[72] NVIDIA. NVIDIA CUDA programming guide 5.0, 2013.

[73] OpenACC. http://www.openacc.org/, 2013.

[74] Par4All. http://www.par4all.org, 2013.

[75] E. Park, L. Pouchet, J. Cavazos, A. Cohen, and P. Sadayappan. Predictive modeling in a polyhedral optimization space. In *CGO*, pages 119–129, 2011.

[76] R. Penrose. A generalized inverse for matrices. *Mathematical Proceedings of the Cambridge Philosophical Society*, 51(03):406–413, 1955.

[77] S. Pop, A. Cohen, C. Bastoul, S. Girbal, P. Jouvelot, G. Silber, and N. Vasilache. GRAPHITE: Loop optimizations based on the polyhedral model for GCC. In *Proceedings of the 4th GCC Developper's Summit*, pages 179–198, June 2006.

[78] L. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache. Loop transformations: convexity, pruning and optimization. In *POPL*, volume 46, pages 549–562, 2011.

[79] W. Pugh and E. Rosser. Iteration space slicing for locality. In *LCPC*, pages 164–184, 2000.

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

[81] S. M. F. Rahman, Q. Yi, and A. Qasem. Understanding stencil code performance on multicore architectures. In *CF*, pages 30:1–30:10, 2011.

[82] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for nonshared memory machines. In *Supercomputing*, pages 111–120, 1991.

[83] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP*, pages 73–82, 2008.

[84] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu. Program optimization space pruning for a multithreaded GPU. In *CGO*, pages 195–204, 2008.

[85] V. Saul'yev. *Integration of Equations of Parabolic Type Equation by the Method of Net.* Pergamon Press, 1964.

[86] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley and Sons, 1986.

[87] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc. A performance analysis framework for identifying potential benefits in GPGPU applications. In *PPoPP*, volume 47, page 11, Sept. 2012.

[88] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *PLDI*, volume 34, pages 215–228, 1999.

[89] M. Strout. *Performance Transformations for Irregular Applications.* PhD thesis, University of California, San Diego, 2003.

[90] M. M. Strout, L. Carter, J. Ferrante, and B. Kreaseck. Sparse tiling for stationary iterative methods. *International Journal of High Performance Computing Applications*, 18(1):95–113, 2004.

[91] P. Tang and J. Xue. Generating efficient tiled code for distributed memory machines. *Parallel Computing*, 26(11):1369–1410, 2000.

[92] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. In *ASPLOS*, pages 325–335, 2006.

[93] R. Tavakoli and P. Davami. New stable group explicit finite difference method for solution of diffusion equation. *Applied Mathematics and Computation*, 181(2):1379–1386, 2006.

[94] D. Tianyi and S. Tarek. hiCUDA: a high-level directive-based language for GPU programming. In *Proceedings of 2nd Workshop on General Purpose Processing*, pages 1–10, 2009.

[95] S. Ueng, M. Lathara, S. S. Baghsorkhi, and W. W. Hwu. CUDA-Lite: Reducing GPU programming complexity. In *Languages and Compilers for Parallel Computing*, pages 1–15. Springer-Verlag, Berlin, Heidelberg, 2008.

[96] D. Unat and X. Cai. Mint: realizing CUDA performance in 3D stencil methods with annotated C. In *ICS*, pages 214–224, 2011.

[97] N. Vasilache, C. Bastoul, and A. Cohen. Polyhedral code generation in the real world. In *CC*, pages 185–201, 2006.

[98] N. Vasilache, C. Bastoul, A. Cohen, and S. Girbal. Violated dependence analysis. In *ICS*, pages 335–344, 2006.

[99] S. Venkatasubramanian and R. W. Vuduc. Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems. In *ICS*, pages 244–255, 2009.

[100] D. Wallin, H. Löf, E. Hagersten, and S. Holmgren. Multigrid and Gauss-Seidel smoothers revisited: parallelization on chip multiprocessors. In *ICS*, pages 145–155, 2006.

[101] D. K. Wilde. A library for doing polyhedral operations. Technical report, IRISA, 1993.

[102] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52:65–76, 2009.

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

[104] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *PLDI*, pages 30–44, 1991.

[105] M. E. Wolf, D. E. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. In *MICRO*, pages 274–286, 1996.

[106] W. Wolf and M. Kandemir. Memory system optimization of embedded software. *Proceedings of the IEEE*, 91(1):165–182, Jan. 2003.

[107] M. Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, 1989.

[108] M. Wolfe. More iteration space tiling. In *Supercomputing*, pages 655–664, 1989.

[109] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishing Company, 1995.

[110] M. Wolfe. Implementing the PGI Accelerator model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 43–50, 2010.

[111] H. Wong, M. Papadopoulou, M. Sadooghialvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *ISPASS*, pages 235–246, 2010.

[112] D. Wonnacott. Time skewing for parallel computers. In *LCPC*, pages 477–480, 2000.

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

[114] S. Xiao and W. Feng. Inter-block GPU communication via fast barrier synchronization. In *IPDPS*, pages 1–12, 2010.

[115] D. Xie. A new block parallel SOR method and its analysis. *SIAM Journal on Mathematical Analysis*, 27(5):1513–1533, 2006.

[116] J. Xue. An algorithm to automate non-unimodular transformations of loop nests. In *SPDP*, pages 512–521, 1993.

[117] J. Xue. Automating non-unimodular loop transformations for massive parallelism. *Parallel Computing*, 20(5):711–728, 1994.

[118] J. Xue. Constructing DO loops for non-convex iteration spaces in compiling for parallel machines. In *IPPS*, pages 364–368, 1995.

[119] J. Xue. Affine-by-statement transformations of imperfectly nested loops. In *IPPS*, pages 34–38, 1996.

[120] J. Xue. Communication-minimal tiling of uniform dependence loops. In *LCPC*, pages 330–349, 1996.

[121] J. Xue. Generalising the unimodular approach to restructure imperfectly nested loops. *Parallel Processing Letters*, 6(3):401–414, 1996.

[122] J. Xue. Transformations of nested loops with non-convex iteration spaces. *Parallel Computing*, 22(3):339–368, 1996.

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

[124] J. Xue. On tiling as a loop transformation. *Parallel Processing Letters*, 7(4):409–424, 1997.

[125] J. Xue. Unimodular transformations of non-perfectly nested loops. *Parallel Computing*, 22(12):1621–1645, 1997.

[126] J. Xue. *Loop Tiling for Parallelism.* Kluwer Academic Publishers, 2000.

[127] J. Xue and W. Cai. Time-minimal tiling when rise is larger than zero. *Parallel Comput.*, 28:915–939, June 2002.

[128] J. Xue and C.-H. Huang. Reuse-driven tiling for improving data locality. *International Journal of Parallel Programming*, 26(6):671–696, 1998.

[129] J. Xue and Q. Huang. Reuse-driven tiling for data locality. In *LCPC*, pages 16–33, 1997.

[130] J. Xue and Q. Huang. Enabling loop fusion and tiling for cache performance by fixing fusion-preventing data dependences. In *ICPP*, pages 107–115, 2005.

[131] Y. Yan, M. Grossman, and V. Sarkar. JCUDA: A programmer-friendly interface for accelerating Java programs with CUDA. In *Euro-Par*, pages 887–899, 2009.

[132] Y. Yang, H. Cui, X. Feng, and J. Xue. A hybrid circular queue method for iterative stencil computations on GPUs. *Journal of Computer Science and Technology*, 27(1):57–74, 2012.

[133] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU compiler for memory optimization and parallelism management. In *PLDI*, pages 86–97, 2010.

[134] Q. Yi. POET: A scripting language for applying parameterized source-to-source program. *Software: Practice and Experience*, 42(6):675–706, 2012.

[135] T. Yuki, L. Renganarayanan, S. Rajopadhye, C. Anderson, A. E. Eichenberger, and K. O'Brien. Automatic creation of tile size selection models. In *CGO*, pages 190–199, 2010.

[136] Y. Zhang. Auto-generation and auto-tuning of 3D stencil codes on GPU clusters. In *CGO*, pages 155–164, 2012.

[137] Y. Zhang and J. Owens. A quantitative performance analysis model for GPU architectures. In *HPCA*, pages 382–393, 2011.

[138] H. Zhao. A fast sweeping method for eikonal equations. *Mathematics of Computation*, 74:603–627, 2005.