TILING OPTIMIZATIONS FOR STENCIL COMPUTATIONS

BY

XING ZHOU

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Doctoral Committee:

Professor David Padua, Chair, Co-Director of Research
Research Assistant Professor María J. Garzarán, Co-Director of Research
Professor William Gropp
Professor Wen-Mei Hwu
Doctor Robert Kuhn, Intel

Abstract

This thesis studies the techniques of tiling optimizations for stencil programs. Traditionally, research on tiling optimizations mainly focuses on tessellating tiling, atomic tiles and regular tile shapes. This thesis studies several novel tiling techniques which are out of the scope of traditional research. In order to represent a general tiling scheme uniformly, a unified tiling representation framework is introduced. With the unified tiling representation, three tiling techniques are studied. The first tiling technique is Hierarchical Overlapped Tiling, based on the idea of reducing communication overhead by introducing redundant computations. Hierarchical Overlapped Tiling also applies the idea of hierarchical tiling to take advantage of hardware hierarchy, so that the additional overhead introduced by redundant computations can be minimized. The second tiling technique is called Conjugate-Trapezoid Tiling, which schedules the computations and communications within a tile in an interleaving way in order to overlap the computation time and communication latency. Conjugate-Trapezoid Tiling forms a pipeline of computations and communications, hence the communication latency can be hidden. Third, this thesis studies the tile shape selection problem for hierarchical tiling. It is concluded that optimal tile shape selection for hierarchical tiling is a multidimensional, nonlinear, bi-level programming problem. Experimental results show that the irregular tile shapes selected by solving the optimization problem have the potential to outperform intuitive tiling shapes.

Acknowledgements

The thesis dissertation marks the end of a long and eventful journey. I would like to acknowledge all the friends and family for their support along the way. This work would not have been possible without their support.

First, I would like to express my heartfelt gratitude to my advisors, Professor David Padua and Professor María Garzarán, for their patience, encouragement, and immense knowledge. Their guidance helped me in all the time of research and writing of this thesis. This thesis would certainly not have existed without their support. I would also like to thank committee members Professor William Gropp, Professor Wen-Mei Hwu, and Doctor Robert Kuhn for their insightful comments and suggestions.

I am very grateful to my fellow group members and other friends in Computer Science Department, for the stimulating discussions, enjoyable experience of working together, and all the fun we have had in the last four years.

I thank my parents, who gave me the best education and built my personality.

Last but not the least, I would like to thank my wife, who always stands with me during times of stress and joy.

Table of Contents

List of Figures . . . . **vii**

List of Tables . . . . **x**

Chapter 1 Introduction **1**
1.1 Background 1
1.1.1 Stencil Computations 1
1.1.2 Loop Tiling 2
1.1.3 Hierarchical Tiling 3
1.2 Contributions 4
1.3 Thesis Organization 5

Chapter 2 Prerequisites **7**
2.1 Unified Tiling Representation Framework 7
2.1.1 Iteration Space Representation 7
2.1.2 Tiling Representation with the Polyhedral Model 11
2.1.3 Unified Tiling Representation Framework 13
2.1.4 Dependences after Tiling 16
2.1.5 Schedule of Tiles 18
2.1.6 Tiling Transformation Representation 19
2.1.7 Hierarchical Tiling 20
2.2 Implementation Using the Omega Library 20
2.2.1 Set and Mapping 20
2.2.2 Code Generation 21

Chapter 3 Hierarchical Overlapped Tiling **23**
3.1 Introduce Redundant Computation Through Overlapped Tiling 23
3.2 Hierarchical Overlapped Tiling 25
3.3 Analytical Modeling 28
3.3.1 Overlapped Tiling 29
3.3.2 Hierarchical Overlapped Tiling 32
3.4 Unified Tiling Representation 35
3.5 Evaluation 39
3.5.1 Implementation 39

3.5.2 Environment Setup 40
3.5.3 Benchmarks 40
3.5.4 Experimental Results 41
3.5.5 Compilation Overhead 47
3.6 Conclusion 48

Chapter 4 Hiding Communication Latency with Conjugate-Trapezoid Tiling 49
4.1 Hiding Communication Latency 49
4.2 Conjugate-Trapezoid Tiling 50
4.2.1 Definitions 50
4.2.2 1-dimensional Stencil 53
4.2.3 2-dimensional Stencil 56
4.2.4 Higher Dimensions 61
4.3 Performance Model 63
4.3.1 Definitions 64
4.3.2 Performance Model 66
4.3.3 Optimization 68
4.4 Unified Tiling Representation 71
4.5 Evaluation 74
4.5.1 Target Platform 74
4.5.2 Implementation 76
4.5.3 Benchmark 76
4.5.4 Experimental Result 77
4.6 Conclusion 82

Chapter 5 Tile Shape Selection for Hierarchical Tiling 83
5.1 Problem Definition 85
5.2 Constraints on Tile Shape Selection 87
5.3 Execution Model 90
5.4 Calculate the Longest Dependent Path 92
5.4.1 Calculate $L(T_k, \mathbb{1}_n)$ 94
5.4.2 Calculate $L(E_l, \mathbb{1}_n)$ 96
5.4.3 Summary 100
5.5 Automatic Tile Shape Selection 100
5.6 Unified Tiling Representation 102
5.7 Evaluation 103
5.7.1 Environment Setup 103
5.7.2 Performance 104
5.7.3 Tile Shape 105
5.7.4 Model Accuracy 108
5.8 Conclusion 109

Chapter 6 Related Work **111**

Chapter 7 Conclusions **115**

Bibliography **117**

List of Figures

1.1 Hierarchical tiling. 4

2.1 n-depth loop nest 8
2.2 The edge vectors (thick arrows) of iteration spaces. 9
2.3 Valid and invalid tile shapes for tessellating tiling with atomic tiles. 11
2.4 The effect of tiling transformation. 16
2.5 Overlapped Tiling. Each tile is trapezoid-shaped. The grey triangles are the overlapped areas between tiles. 16
2.6 Split Tiling. There are two shapes of tiles: triangle and trapezoid. The neighboring tiles of different shapes consist a "super" tile (in the dashed box) which is the basic repetition unit. 17
2.7 Generated loop nest code for the integer tuple set S_1 22
2.8 Parallel code with OpenMP and MPI 22

3.1 A simple tiling example for parallel loops 24
3.2 Overlapped tiling of K loops 26
3.3 Comparison of overlapped tiling and hierarchical overlapped tiling on a 4-way/8-core multicore system, where each processor contains 2 cores on chip that share the last level cache. 27
3.4 A sequence of K parallel loops 28
3.5 Equivalent loop nest to the K consecutive parallel loops 35
3.6 Framework of the experimental system. 39
3.7 Speedup of traditional tiling, overlapped tiling and hierarchical overlapped tiling over the original OpenCL code. 42
3.8 Evaluating the performance overlapped tiling and hierarchical overlapped tiling by scaling fusion depth K. 43
3.9 Fusion depth K' for the second level of hierarchical overlapped tiling. 46
3.10 Speedup of hierarchical overlapped tiling over plain overlapped tiling with different input sizes. The horizontal axis is the input size for each benchmark. 47
3.11 Compile time for overlapped tiling and hierarchical overlapped tiling, normalized to the compile time of the original OpenCL code. 48

4.1 1-dimensional Jacobi code with MPI. 50
4.2 An ISL with a d-dimensional stencil. The total depth of the loop nest is $d+1$. 51
4.3 Dependence vectors of a 5-point stencil 52

4.4 Tiling 1-dimensional Jacobi code. Thin arrows represent the dependence vectors. Thick arrows show the dependence between tiles. w and h are the parameters decided by users to determine the size of each tile. 53
4.5 Subtiling and delay 56
4.6 Sequential 2-dimensional Jacobi code. 56
4.7 Tiling with 2-dimensional stencils. The thick arrows in (b) represent the dependence between tiles. 59
4.8 Subtiling with 2-dimensional stencils. 60
4.9 Schedule of subtiles for each time step.$\forall j$, S_j and R_j are the corresponding *send* and *receive* pair. 61
4.10 Projections of subtiles with 2-dimensional stencils. 61
4.11 Slice of a 4-dimensional hyper-parallelepiped tile. 62
4.12 Number of messages to be sent for 3-dimensional Jacobi. Each arrow stands for a message to be sent to an neighboring tile. The arrows with the same direction means that the destination of the messages are the same. 67
4.13 Merge communication operations. The *send* or *receive* operations with in the same dashed box can be merged. Compared to Figure 4.9-(b), the *send* or *receive* operations denoted by the dashed arrows are moved (delayed or brought forward) to the operations denoted by corresponding thick arrows. . 69
4.14 Balanced multi-threading with four processors. The white bars stand for computation time, and the grey bars stand for communication operation overhead. 70
4.15 Speedup with different values of h'. The baseline is the performance achieved by tiling with no communication overlap. The figure shows the performance with h' up to 6 for 2-dimensional stencils and h' up to 4 for 3-dimensional stencils, because the trend shows that there would be no additional performance gains with a larger h'. 78
4.16 Communication overhead with different h'. 80
4.17 Input size sensitivity. 82

5.1 Different tile shape choices for hierarchical tiling. Arrows are the dependences between tiles. Dashed lines represent the tiles that can be executed in parallel in a wavefront schedule. The numbers stand for the order of the wavefronts . 84
5.2 Constraints of tile shape imposed by dependences. 87
5.3 Inter-tile dependences (thicker arraows) generated by original dependence vector $\vec{d_k}$. 89
5.4 The dependent pathes (P and P') within an iteration space. The length of a dependent path is the number of iterations on the path. If any path contains $\vec{d_2}$ (P'), it is always possible to replace $\vec{d_2}$ with $\vec{d_1}$ and $\vec{d_0}$ on the same position, which resulting a longer path. This means the longest path (P) must only contain $\vec{d_0}$ and $\vec{d_1}$. 92
5.5 Determine the length of the longest dependent path within the iteration space of tile T_k. Each $\vec{p}_1$ stands for the coordinate of the point. 96

5.6 Construct path P according to a given path P'. The lengthes of P and P' are approximately equal. 99
5.7 The dependence vectors in the iteration space of stencil computation programs. 104
5.8 Hierarchical tiling performance for 2-dimensional iteration space on GPU and cluster platforms. The horizontal axis is the size of the iteration space. The vertical axis is the speedup over *Wavefront* or *Diamond*. 107
5.9 Hierarchical tiling performance for 3-dimensional iteration space on GPU and cluster platforms. The horizontal axis is the size of the iteration space. The vertical axis is the speedup over *Wavefront*. 108
5.10 Comparison between the real execution time and the ideal execution time for 1-D Gauss-Seidel. The left axis is for real execution time and the axis on the right is for ideal execution time. 109

List of Tables

3.1 Representation of Overlapped Tiling 38
3.2 ISL Benchmarks 40
3.3 Loop Fusion depth K used in experiments. 45

4.1 Representation of Conjugate-Trapezoid Tiling 75
4.2 Benchmarks 76
4.3 Different stencils of the benchmarks 77

5.1 The Unified Tiling Representation 103
5.2 Common tiling schemes: *Wavefront*, *Diamond* and *Skewing*. 105
5.3 Tile shape selection for 1-D Gauss-Seidel and Jacobi. 106
5.4 Tile shape selection for 2-D Gauss-Seidel on GPU (1920×1920×1920). . . . 108

Chapter 1

Introduction

1.1 Background

1.1.1 Stencil Computations

Iterative stencil loops (ISLs) are an important kernel of many computations. For example, for certain regular matrices, the kernel the widely used Jacobi method is a stencil loop; and image processing filters are also implemented as stencil loops. Iterations of the outermost loop of ISLs are called *time step*s, as programs with ISLs are often used to simulate the evolution of physical systems over time. At each time step, the inner loop nest of the ISL operates on a multidimensional array, and computes each element of an array as a function of neighboring locations in the array. The choice of the neighboring elements follows a fixed pattern or *stencil*.

To achieve high performance, the inner loop nests of ISLs are usually tiled and each tile is assigned to a processing node for parallel execution. Data locality and communication are the main issues when optimizing ISLs. Data locality influences the execution time within each computing node. When running on distributed memory systems, the performance of these ISLs may be hindered by inter-node communication. Limited by the performance of the interconnection network, the cost of communication grows with the scale of the system increases. If the ISL is parallelized for a shared memory system, the inter-tile communication is implicit, usually controlled with synchronization operations. Even though the communication overhead of shared memory systems is usually much lower than that of distributed memory systems, it may still cause performance degradation.

1.1.2 Loop Tiling

Loop tiling is an effective optimization to improve performance of multiply-nested loops, which are usually the most time-consuming parts of computationally intensive programs. Numerous techniques for the tiling of iteration spaces have been proposed. The goal of tiling is to improve data locality [39, 42, 1, 2, 36, 14, 33, 38, 41, 45, 48], or contribute to the scheduling of parallel computation [40, 43, 6, 18, 44, 7, 47]. The performance resulting from the use of a tiling scheme is a function of (1) locality, (2) the amount of parallelism exposed, and (3) the communication/synchronization overhead. The size and shape of the tiles, and the scheduling strategy are the main factors that impact locality, parallelism, and communication cost of the code after tiling. Research on tiling techniques usually try to improve one or more factors that impact performance. Locality was the first focus of research in these studies of tiling, but as the importance of parallel computing increased, mechanisms to optimize parallelism organization and communication with tiling techniques gained attention.

Existing tiling transformations mainly relies on *tessellating tiling* and scheduling operations in tiles in *atomic*. In tessellating tiling there is no overlap between tiles. When scheduling operations of tiles atomically, inter-tile communication or synchronization only happen before computation starts or after all the computation within a tile completes. The above two restrictions simplifies the study of loop tiling techniques. Elegant representation frameworks, such as the polyhedral model [5], have been built to facilitate research on tiling schemes under these restrictions. However, by ignoring such restrictions it is possible to taka advantage of the optimization opportunities of general tiling schemes. For example, because of data dependences, inter-tile communication is inevitable with tessellating tiling; and, with atomic tiles execution, schedule of tiles is restricted and often results in load imbalances. In order to exploit the optimization opportunities of general tiling schemes, a new representation framework is necessary.

1.1.3 Hierarchical Tiling

Most massively parallel systems today are organized hierarchically. Different levels in the hierarchy may have different organizations and follow different memory models. Consider a computer cluster consisting of nodes connected by a network. This forms the first level of the hierarchy. At the next level, each node is typically an SMP machine. Modern accelerator devices are also organized hierarchically. An NVIDIA GPGPU contains a number of Multi-Processors (MPs). The MPs have uniform access to a global memory. Each MP consists of several Stream-Processors (SPs), which share the MP-private shared memory.

Hierarchy-aware optimizations are usually necessary to unleash the potential of hierarchically organized systems. In order to make better use of hardware hierarchies, loop nests should also be tiled hierarchically to fit the organization of the target machine. Figure 1.1 gives an example of hierarchical tiling for a cluster. In top-down order, (1) the iteration space of the loop nest is tiled, and each tile is mapped to a node of the cluster. (2) On each node, the iteration space is further partitioned into smaller tiles, each of which is assigned to a processor in the node. Hierarchical tiling can also be done bottom-up by partitioning the original iteration space into tiles that are assigned to the lowest level of hardware and then grouping tiles into larger ones for the higher levels of hardware.

Hierarchical tiling increases the complexity of the study on tiling transformation, because each level of tiling has its own choice tile size/shape and scheduling strategy, and the decisions

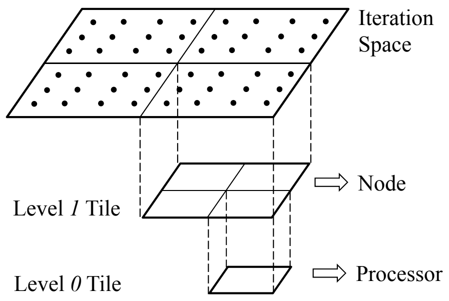


Figure 1.1: Hierarchical tiling.

of tiling scheme at different levels interfere with each other. In order to achieve global optimality for hierarchical tiling, different levels of tiling must be considered in an integral model.

1.2 Contributions

This thesis makes several contributions in the area of tiling for stencil computations. First, it introduces a unified representation framework for tiling transformations, which is able to describe tiling techniques that cannot be represented by the traditional polyhedral model, such as non-tessellating tiling and scheduling with non-atomic execution of tiles, can be defined uniformly with this new representation framework.

Second, this thesis proposes two novel tiling schemes. The first is *Hierarchical Overlapped Tiling*. This is an extension of the existing idea of overlapped tiling [26], which introduces redundant computation to eliminate inter-tile communication. Overlapped tiling is a non-tessellating tiling scheme. It is a useful transformation to reduce communication overhead, but it may also introduce a significant amount of redundant computations. Based on this observation, Hierarchical Overlapped Tiling is designed to solve this problem. Hierarchical Overlapped Tiling trades off redundant computation for reduced communication overhead in a more balanced manner, and thus has the potential to provide higher performance. An analytic model is built to analyze the performance of both Overlapped Tiling and Hierarchical

Overlapped Tiling. The second tiling scheme is called *Conjugate-Trapezoid Tiling*. This tiling scheme also aims at reducing the inter-tile communication. Instead of introducing redundant computation to eliminate inter-tile communication, Conjugate-Trapezoid Tiling schedules the computations and communications within a tile in an interleaving order to overlap the computation and communication. Conjugate-Trapezoid Tiling pipelines computation and communication. The operations in tiles of Conjugate-Trapezoid Tiling are not scheduled assuming that they are atomic; instead, computation and communication within a tile are interleaved with each other. A performance model is used to determine the optimal pipeline parameters of Conjugate-Trapezoid Tiling. The efficiency of both tiling schemes is evaluated using several codes which implement stencil computations.

Third, this thesis studies the tile shape selection problem for hierarchical tiling. It builds an analytic model to analyze the execution time of the tiled loop nest as a function of the tile shape at each level of a hierarchy. The model is not tied to any specific scheduling scheme of tiles, but only focuses on the essence effect on parallelism exposure for all possible tile shape choices. It is concluded that optimal tile shape selection for hierarchical tiling is a multidimensional, nonlinear, bi-level programming problem. An experimental automatic system is implemented, which uses a simulated annealing algorithm to find a near-optimal tile shape choice according the analytic model. Experimental results show that the tiling scheme with automatically chosen tile shapes has the potential to outperform intuitive tiling shapes.

1.3 Thesis Organization

The rest of the thesis is organized as follows: Chapter 2 introduces a new tiling representation framework for general schemes and a code generator implementation based on this framework.In Chapter 3 Hierarchical Overlapped Tiling is introduced building upon the existing idea of overlapped tiling, and an analytic model is built to study the trade off between communication and redundant computation. In Chapter 4 a novel tiling scheme called Conjugate-Trapezoid Tiling is designed, and a performance model is used to determine the optimal pipeline parameters. Chapter 5 studies the tile shape selection problem for hierarchical tiling. Chapter 6 discusses related work, and Chapter 7 concludes this thesis.

Chapter 2

Prerequisites

2.1 Unified Tiling Representation Framework

This chapter first introduces the polyhedral model [5], which has been used in the past to represent of tiling transformations. Based on the observation that the polyhedral model is only able to represent traditional tiling techniques, a more general tiling representation framework is presented. This representation accommodates several non-traditional tiling schemes including non-tessellating tiling and scheduling strategies that do not assume atomic tile operations.

2.1.1 Iteration Space Representation

Consider the loop nest in Figure 2.1-(a). whose vector form is shown in Figure 2.1-(b). Iteration space I is a finite set of points in the n-dimensional space $\mathbb{Z}^n$. The loop bounds $lb_0, lb_1, ...lb_{n-1}$ and $ub_0, ub_1, ...ub_{n-1}$ are functions of the index variables of outer loops as follows:

$$lb_k = lb_k(i_0, i_1, ..., i_{k-1}),$$
$$ub_k = ub_k(i_0, i_1, ..., i_{k-1}), \qquad k = 0, 1, ..n-1.$$

The set of iterations in I is defined by $lb_0, lb_1, ...lb_{n-1}$ and $ub_0, ub_1, ...ub_{n-1}$:

$$I = \{\vec{i} = (i_0, i_1, ..., i_{n-1}) | lb_k(i_0, i_1, ..., i_{k-1}) \leq i_k < ub_k(i_0, i_1, ..., i_{k-1})\}.$$

In the rest of this thesis, $lb_k(i_0, i_1, ..., i_{k-1})$ and $ub_k(i_0, i_1, ..., i_{k-1})$ are restricted to be affine functions of $i_0, i_1, ..., i_{k-1}$.

```
for (int  i_0 = lb_0(); i_0 < ub_0(); ++i_0)
   for (int  i_1 = lb_1(i_0); i_1 < ub_1(i_0); ++i_1)
      ...
         for (int  i_{n-1} = lb_{n-1}(i_0, i_1, ..., i_{n-2}); i_{n-1} < ub_{n-1}(i_0, i_1, ..., i_{n-2}); ++i_{n-1})
             << loop body >>
```

(a) Loop nest with scalar induction variables

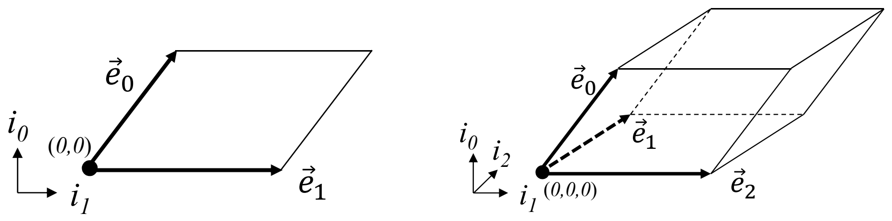
```
for (i⃗ = (i_0, i_1, ..., i_{n-1}) ∈ I)
             << loop body >>
```

(b) The representation with vector induction variables

Figure 2.1: n-depth loop nest

In practice, the shape of I is usually an n-dimensional hyper-parallelepiped. Any set of n edges of the hyper-parallelepiped which share a common vertex can be used as the basis of the n-dimensional iteration space; the common vertex of these edges is the origin. Figure 2.2 gives two examples of the edge vectors that define 2 dimensional and 3 dimensional iteration spaces.

Assume that $\vec{e}_k = (e_{k,0}, e_{k,1}, ..., e_{k,n-1})$, $k = 0, 1, ..., n-1$ are the n edge vectors used as



(a) 2 dimensional space (b) 3 dimensional space

Figure 2.2: The edge vectors (thick arrows) of iteration spaces.

the basis of the hyper-parallelepiped shaped iteration space I. The *basis matrix* of I is:

$$E = \begin{pmatrix} \vec{e}_0 \\ \vec{e}_1 \\ ... \\ \vec{e}_{n-1} \end{pmatrix}.$$

Define the function $span(E)$ as follows:

$$span(E) = \{\vec{i}: \quad \vec{i} \in \mathbb{Z}^n, \quad \exists \vec{a} = (a_0, a_1, ..., a_{n-1}),$$
$$\vec{0}_n = (0, 0, ..., 0) \leq \vec{a} \leq (1, 1, ..., 1) = \vec{1}_n, \quad \vec{i} = \vec{a} \cdot E\}.$$

Clearly $I = span(E)$, and the total number iterations in I is $|I| = |det(E)|$ (because $|det(E)|$ is the volume of the parallelepiped).

Dependences are the partial order that must be enforced between iterations to obtain correct results. If there are no direct or indirect dependences between two iterations, these two iterations can be scheduled either concurrently or in any sequential order. Otherwise, the iteration at the source of the dependence must finish before the iteration at the destination of the dependence starts.

Dependences can be represented by dependence vectors. A dependence vector $\vec{d} =$

$(d_0, d_1, ..., d_{n-1})$ indicates that any iteration $\vec{i}$ must finish before iteration $\vec{i} + \vec{d}$. A valid dependence vector d must satisfy the following condition:

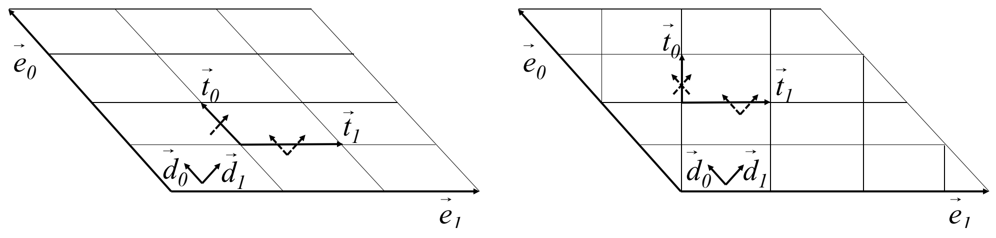
$$\begin{aligned} & if \;\; \forall k = 0, 1, ..., j, \;\; d_k = 0, \;\; and \;\; d_{j+1} \neq 0, \;\; then \\ & d_{j+1} > 0 \end{aligned} \tag{2.1}$$

Assume that a loop with n levels of nesting has m dependence vectors $\vec{d_0}, \vec{d_1}, ..., \vec{d}_{m-1}$, which define the *dependence matrix*:

$$D = \begin{pmatrix} \vec{d_0} \\ \vec{d_1} \\ ... \\ \vec{d}_{m-1} \end{pmatrix}.$$

Without loss of generality, this thesis assumes that $m \geq n$, where m is the number of dependence vectors, and n is the number of loops in the loop nest. It is also assumed that there are n dependence vectors which are linearly independent. Otherwise, it must be possible to make at least one loop fully permutable through a sequence of affine transformations, so it is possible to move the permutable loop either to outmost or innermost, and then it is only needed to focus on the other $n - 1$ loops and the iteration space can be studied as a lower-dimensional one.

The dependences impose constraints on possible tiling transformations. In order to be valid, the new loop nest after tiling must preserve the dependences of the original loop nest. If the computation operating in a tile is ”atomic” - which means inter-tile communication (inter-tile dependence) only happens before inner-tile computation starts and/or after inner-tile computation is finished - a valid tiling scheme requires that it must be possible to topologically sort all the tiles. Otherwise, the neighboring tiles would have a cycle of



(a) Valid tile shape (b) Invalid tile shape

Figure 2.3: Valid and invalid tile shapes for tessellating tiling with atomic tiles.

dependences and it would be impossible to schedule the computations.

Figure 2.3 gives two examples of the constraints on tiling imposed by dependences for tessellating tiling with atomic tiles. The tile shape shown in Figure 2.3-(a) is valid, because dependence vectors only cross the hyper-plane in one direction, which means that at each of its boundaries, the tile has either in-bound dependences or out-bound dependences. However, the tile shape in Figure 2.3-(b) is invalid, since there are both in-bound and out-bound dependences crossing the boundary hyper-planes of tiles. This means that a tile and one of its neighboring tile along the horizontal direction are in a dependence cycle. On the other hand, if not restricted to tessellating tiles or scheduling assuming atomic operations in tiles, there would be more freedom in selecting the tiling schemes. One example is that overlapped tiling, in which the tiles are not tessellating, is able to eliminate inter-tile dependences by introducing redundant computation.

2.1.2 Tiling Representation with the Polyhedral Model

The polyhedral model is the traditional representation of tiling transformations used in the past. The polyhedral model uses a set of hyperplanes that partition the iteration space to represent the tiling transformation. These hyperplanes are defined by a matrix H, where each row represents the normal vector of one of the tiling hyperplanes:

$$H = \begin{pmatrix} \vec{h}_0 \\ \vec{h}_1 \\ \ldots \\ \vec{h}_{n-1} \end{pmatrix}.$$

$\vec{h}_k$ is the normal vector of not just of one, but a set of tiling hyperplanes. The direction $\vec{h}_k$ determines an infinite set of parallel hyperplanes. In addition, this thesis assumes the length of each $\vec{h}_k$ is the distance between tiling hyperplanes. So H can fully determine the shape of tiles under the assumption that the hyperplane that intersects the first iteration is the first hyperplane.

As discussed in Section 2.1.1, dependences constraint tiling transformations. A tiling scheme is valid if there exists a valid total ordering of the tiles. If tiles are required to be scheduled assuming atomic operations in tiles, this constraint is equivalent to saying that no any pair of tiles are dependent on each other. The polyhedral model represents the validity condition as follows [20]:

$$H \cdot D^T \geq 0 \tag{2.2}$$

The polyhedral model provides elegant representations for traditional tiling schemes. However, the polyhedral model has several shortcomings if it is used to study more general tiling techniques. First, using a set of tiling hyperplanes to represent tile shapes implies tessellating tiling, which means that there must be no overlap between tiles. The polyhedral model cannot represent non-tessellating tiling. Second, Equation 2.2 assumes that the scheduling of tiles must follow the order defined by the directions of the normal vectors of tiling hyperplanes. This means that with the polyhedral model, the tile shape and scheduling strategy are determined by H at the same time. So the polyhedral model lacks the flexibility to design different schedule constraints in order to optimize parallelism exposure. Finally, the polyhedral model is not good for studying hierarchical tiling techniques, because the iteration space within each tile does not have a unified representation of the original iteration space so that recursive analysis cannot be simply applied.

2.1.3 Unified Tiling Representation Framework

Based on the observation that the traditional polyhedral model is not good for studying general tiling techniques, a unified representation framework of tiling transformations is introduced in this section.

In this unified representation framework, a base *tile* $\mathbb{T}$ is a set of points in $\mathbb{Z}^n$. Without lost of generality, this thesis assumes that the size of the base tile $\mathbb{T}$ is smaller than the size of the iteration space I that the tiling is applied to: $|\mathbb{T}| < |I|$. Since the iteration space, say I, is also a set of points in $\mathbb{Z}^n$, tile T is also a smaller iteration space. If the tiles are n-dimensional hyper-parallelepiped where $\vec{t_k} = (t_{k,0}, t_{k,1}, ..., t_{k,n-1})$, $k = 0, 1, ..., n-1$ are the n "basis" edges of the hyper-parallelepiped tile, then the base tile $\mathbb{T}$ can be described by the *tiling matrix* T:

$$T = \begin{pmatrix} \vec{t_0} \\ \vec{t_1} \\ ... \\ \vec{t}_{n-1} \end{pmatrix}.$$

For each non-boundary tile, the set of the iterations with in the tile is $span(T)$. And the total number of iterations is $|det(T)|$.

Tiling is a map from $\mathbb{Z}^n$ to $\mathbb{Z}^{2n}$: $I \rightarrow [I' : J] = \{(\vec{i'} : \vec{j})\}$, in which $\vec{i'} = (i'_0, i'_1, ..., i'_{n-1})$ is an index vector for each tile, and $\vec{j} = (j_0, j_1, ..., j_{n-1})$ is an iteration index that falls within tile $\vec{i'}$ (notice that $\vec{j}$ is a global iteration index instead of local index within the tile). If every tile has the same shape as $\mathbb{T}$, given an iteration space I, *tiling* is done by replicating tile $\mathbb{T}$ in a repeated pattern until all the points (iterations) in I are covered by the replications of $\mathbb{T}$.

Let $\mathbb{R}$ denote the repetition operator applied to the index vector of an iteration $\vec{i}$. In the rest of the thesis, repetition patterns are restricted to be translation transformations in the n-dimensional space. Let R denotes the following translation matrix:

$$R = \begin{pmatrix} \vec{r}_0 \\ \vec{r}_1 \\ ... \\ \vec{r}_{n-1} \end{pmatrix}.$$

Then a repetition pattern $\mathbb{R}$ can be defined as follows:

$$\mathbb{R}(\vec{i}, \vec{i'}) = \vec{i} + \vec{i'} \cdot \begin{pmatrix} \vec{r}_0 \\ \vec{r}_1 \\ ... \\ \vec{r}_{n-1} \end{pmatrix} = \vec{i} + \vec{i'} \cdot R \tag{2.3}$$

Since the matrix R represents the repetition pattern of tiling, R is called the *repetition matrix*. Given a base tile $\mathbb{T}$, each $\vec{i'}$ generates a repetition of the tile $\mathbb{T}$ though $\mathbb{R}$, denoted as $\mathbb{T}(\vec{i'})$:

$$\mathbb{T}(\vec{i'}) = \{\mathbb{R}(\vec{i}, \vec{i'}) | \vec{i} \in \mathbb{T}\} \tag{2.4}$$

The tiling can be formally defined as follows:

$$I = \{\vec{i}\} \quad \rightarrow \quad \{(\vec{i'} : \vec{j}) \mid \forall\ \vec{i'}, \vec{j} \quad \mathbb{T}(\vec{i'}) \cap I \neq \emptyset\ \wedge\ \vec{j} \in \mathbb{T}(\vec{i'})\ \wedge\ \vec{j} \in I\} \tag{2.5}$$

The first constraint in Equation 2.5, $\mathbb{T}(\vec{i'}) \cap I \neq \emptyset$, defines another iteration space I', which is a set of $\vec{i'}$, as follows:

$$I' = \{\vec{i'} \mid \mathbb{T}(\vec{i'}) \cap I \neq \emptyset\} \tag{2.6}$$

And the original iteration space I must be covered by the union of all repetitions of the tile:

$$I \subseteq \bigcup_{\forall \vec{i'} \in I'} \mathbb{T}(\vec{i'}) \tag{2.7}$$

Note that the above definition of tiling does not require that the tiles partition the iteration space. If $\exists \vec{i'}_1, \vec{i'}_2, \mathbb{T}(\vec{i'}_1) \cap \mathbb{T}(\vec{i'}_2) \neq \emptyset$, the iterations in the intersection set are redundant computations introduced by tiling compared to the original loop nest. Tiling with redundant computations can still be legal if the loop nest after tiling produces the same result. Overlapped tiling [26, 30, 35, 49] is an example of tiling schemes with redundant computations which has been studied before. Figure 2.5 gives an example of overlapped tiling. In most tiling schemes studied in existing research are *tessellating* tiling, which means the tile repetitions are disjoint:

$$\forall \vec{i'}_1, \vec{i'}_2 \quad \vec{i'}_1 \neq \vec{i'}_2, \quad \mathbb{T}(\vec{i'}_1) \cap \mathbb{T}(\vec{i'}_2) = \emptyset$$

For tessellating tiling, for each $\vec{i} \in I$, there is a unique $(\vec{i'} : \vec{j})$ after tiling corresponding to it, and vice versa. Then $I' = \{\vec{i'}\}$ forms an iteration space, where each $\mathbb{T}(\vec{i'})$ is an iteration. Figure 2.4 shows the effect of tiling transformation.

The representation can handle the case where there are several but finite number of tile shapes, as long as there is a single "super" tile which groups neighboring tiles of different shapes, and the whole iteration space is covered by repetitions of the "super" tile. Therefore the tiling definition above can still be used to analyze the tiling scheme with finite number

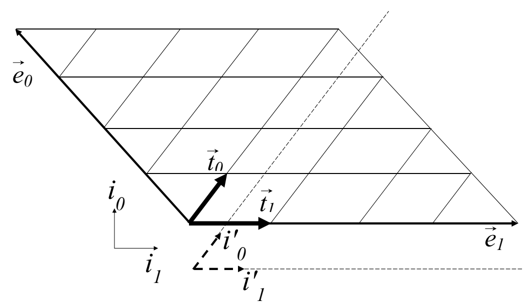


Figure 2.4: The effect of tiling transformation.

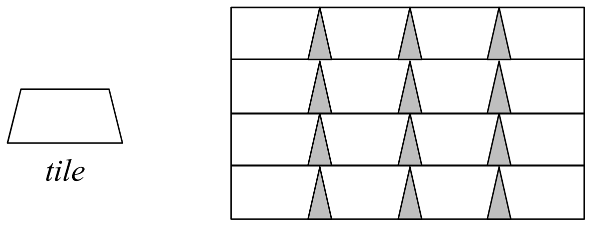


Figure 2.5: Overlapped Tiling. Each tile is trapezoid-shaped. The grey triangles are the overlapped areas between tiles.

of tile shapes, but tile T must represent the "super" tile. Split tiling [26, 37] is an example of tiling schemes with more than one tile shapes. As shown in Figure 2.6, there are two shapes of tiles: triangle and trapezoid. The neighboring tiles of different shapes consist a "super" tile (in the dashed box) which is the basic repetition unit.

2.1.4 Dependences after Tiling

As mentioned above, $I' = \{\vec{i'}\}$ is an iteration space with each iteration containing all the iterations in $\mathbb{T}(\vec{i'})$. The dependences in the original iteration space I impose new dependences in I'. Let $\vec{d'}_0$, $\vec{d'}_1$, ..., $\vec{d'}_{m'-1}$ are the m' resulting dependences vectors in I' and D' is the

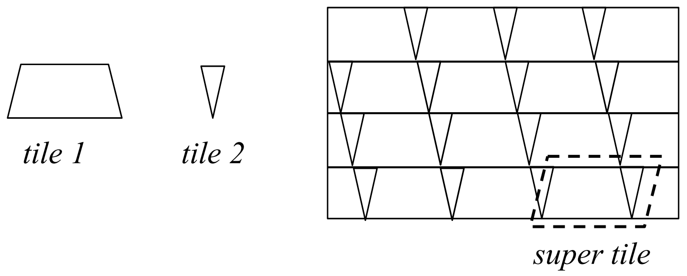


Figure 2.6: Split Tiling. There are two shapes of tiles: triangle and trapezoid. The neighboring tiles of different shapes consist a "super" tile (in the dashed box) which is the basic repetition unit.

dependence matrix:

$$D' = \begin{pmatrix} \vec{d'}_0 \\ \vec{d'}_1 \\ ... \\ \vec{d'}_{m-1} \end{pmatrix}.$$

Note that the execution order of tiles is not necessarily the lexicographical order of $\vec{i'}$. The execution order can be changed by different scheduling schemes as discussed in Section 2.1.5, a dependence vector $\vec{d'}_j$ $(0 \leq j < m)$ does not have to satisfy the condition defined in Equation 2.1.

In order to simplify the problem, the rest of the thesis assumes that the tile chosen for any tiling scheme is always large enough, so that for an n-dimensional iteration space, each tile only has up to $3^n - 1$ neighboring tiles, and inter-tile dependences only exist between neighboring tiles. This means that each component in any dependence vector $\vec{d'}_j$, $0 \leq j < m'$, can only be 0, 1, or -1:

$$\vec{d'}_j = (d'_{j,0}, d'_{j,1}, ..., d'_{j,n-1}), \qquad d'_{j,k} = 0, 1, -1, \quad 0 \leq k < n$$

2.1.5 Schedule of Tiles

Tiles must be scheduled in a way that the inter-tile dependences are preserved. The scheduling of tiles is a reorder of the $\mathbb{T}(\vec{i'})$s. The schedule can be formally defined as a mapping $\mathbb{S}$ from each $\vec{i'}$ to $\vec{i^o}$:

$$\vec{i^o} = \mathbb{S}(\vec{i'})$$

After tiling, each tile $\mathbb{T}(\vec{i'})$ will be executed in a lexicographical order of $\vec{i^o}$. In order to be a legal scheduling, $\vec{i^o}$ must lexicographically conform to all inter-tile dependences: if $\mathbb{S}(\vec{i'}_1) = \vec{i^o}_1$ and $\mathbb{S}(\vec{i'}_2) = \vec{i^o}_2$, and $\vec{i^o}_1 \preceq \vec{i^o}_2$, there must be no direct or indirect inter-tile dependence from the tile $\mathbb{T}(\vec{i'}_2)$ to tile $\mathbb{T}(\vec{i'}_1)$. Note that $\mathbb{S}$ is an n-to-1 mapping instead of 1-to-1 mapping. If $\exists \vec{i'}_1, \vec{i'}_2$, $\vec{i'}_1 \neq \vec{i'}_2$, $\mathbb{S}(\vec{i'}_1) = \mathbb{S}(\vec{i'}_2)$, this means that there is no direct or indirect dependence between $\mathbb{T}(\vec{i'}_1)$ and $\mathbb{T}(\vec{i'}_2)$, and these two tiles can be scheduled in parallel. Define $\mathbb{S}^{-1}$ as follows:

$$\mathbb{S}^{-1}(\vec{i^o}) = \{\vec{i'} | \mathbb{S}(\vec{i'}) = \vec{i^o}\}$$

Then $|\mathbb{S}^{-1}(\vec{i^o})|$ is the amount of parallelism on each step exposed by the scheduling $\mathbb{S}$.

If the scheduling mapping is affine, $\mathcal{S}$ can be described by an $n \times s$ ($s \leq n$) matrix S as follows:

$$\vec{i^o} = \mathbb{S}(\vec{i'}) = \vec{i'} \cdot S$$

S is called the *scheduling matrix*. Whether $\mathbb{S}$ or S is a valid scheduling mapping depends on the dependence matrix D' in the tiled iteration space I'.

Valid Schedule

The lexicographical order of $\vec{i^o} = \mathbb{S}(\vec{i'})$ defines the execution order of tile $\vec{i'}$. The execution order must not violate the dependences between tiles. Given $\vec{i'}_1$ and $\vec{i'}_2 = \vec{i'}_1 + \vec{d'}_j$, $\mathbb{T}(\vec{i'}_1)$

must be executed before $\mathbb{T}(\vec{i'}_2)$. If $\mathbb{S}(\vec{i'}_1) = \vec{i}_1{}^{\mathrm{o}}$ and $\mathbb{S}(\vec{i'}_2) = \vec{i}_2{}^{\mathrm{o}}$, there must be

$$\vec{i}_2{}^{\mathrm{o}} \succ \vec{i}_1{}^{\mathrm{o}}, \quad \mathit{or} \quad \vec{i}_2{}^{\mathrm{o}} - \vec{i}_1{}^{\mathrm{o}} = \mathbb{S}(\vec{i'}_1 + \vec{d'}_j) - \mathbb{S}(\vec{i'}_1) \succ \vec{0}$$

Assume $\mathbb{S}$ is an affine transformation, then $\mathbb{S}(\vec{i'}_1 + \vec{d'}_j) - \mathbb{S}(\vec{i'}_1) = \mathbb{S}(\vec{d'}_j)$. So the condition of valid schedule can be defined as follows:

$$\forall \vec{d'}_j, \quad \mathbb{S}(\vec{d'}_j) \succ \vec{0} \tag{2.8}$$

If scheduling matrix S exists, $\vec{d^{\mathrm{o}}}_j = \mathbb{S}(\vec{d'}_j) = \vec{d'}_j \cdot S$ must satisfy the condition in Equation 2.8.

In addition, because for any $\vec{i^{\mathrm{o}}}$, all tiles $\vec{i'} \in \mathbb{S}^{-1}(\vec{i^{\mathrm{o}}})$ can be scheduled in parallel, there must be no dependence among them:

$$\begin{aligned} &\forall \vec{i'} \in \mathbb{S}^{-1}(\vec{i^{\mathrm{o}}}), \\ &\forall \vec{d'}_j, \;\; \mathbb{S}(\vec{i'} + \vec{d'}_j) \notin \mathbb{S}^{-1}(\vec{i^{\mathrm{o}}}). \end{aligned} \tag{2.9}$$

Equation 2.8 and 2.9 are the two conditions that a valid schedule $\mathbb{S}$ must satisfy.

2.1.6 Tiling Transformation Representation

To sum up the discussion above, given an iteration space I and its dependence matrix D, a specific tiling transformation is determined by tile shape $\mathbb{T}$ (or tiling matrix T), repetition pattern $\mathbb{R}$ (or repetition matrix R) and scheduling mapping $\mathbb{S}$ (or scheduling matrix S). The choices of $\mathbb{T}(T)$, $\mathbb{R}(R)$ and $\mathbb{S}(S)$ must conform the constraints imposed by D.

2.1.7 Hierarchical Tiling

After tiling, either each tile $\mathbb{T}$ or I' can be considered as a new iteration space and can be applied tiling transformation recursively. This means tiling transformations can be done hierarchically. Each level of tiling can be defined and analyzed with corresponding $\mathbb{T}(T)$, $\mathbb{R}(R)$ and $\mathbb{S}(S)$ as a single level of tiling independently.

There are two approaches to do hierarchical tiling. First, do tiling for I and then apply tiling the each tile $\mathbb{T}$; in this way the whole process would produce higher level tiles first then produce lower level tiles. This is called the *top-down* approach. On the other hand, if the iteration space I' is tiled after tiling the original I, the whole process would produce lower level tiles first then higher level tiles, which is called the *bottom-up* approach.

2.2 Implementation Using the Omega Library

In order to automate the process of tiling, an experimental implementation using Presburger formulas [25] has been developed. The manipulation of Presburger formulas is done by the Omega Library [22] automatically.

2.2.1 Set and Mapping

The key concepts of tiling transformation are sets and mappings in the n-dimensional integer space. As discussed in last section, for a given tiling transformation, the iteration space I and tile shape $\mathbb{T}$ are sets of integer vectors, and the repetition pattern $\mathbb{R}$ and scheduling mapping $\mathbb{S}$ are mappings in iteration space. In the experimental implementation, *set*s and *mapping*s in the n-dimensional integer space are represented by *integer tuple set*s and *integer tuple relation*s, respectively. These two are the objects used in the Omega Library.

In the Omega Library, an n-dimensional *integer tuple* $\vec{x} = [x_0, x_1, ..., x_{n-1}]$ is a vector of n integers in $\mathbb{Z}^n$. An *integer tuple set* is a set of integer tuples. Constraints in the form of equations and inequalities can be used to describe sets of integer tuples. For example, the set $S_0 = \{[1,1],[1,2],...,[1,N]\}$ can be represented as $\{[i,j] : i = 1 \wedge 1 \leq j \leq N\}$.

In this thesis it is assumed that the arithmetic expressions in the equations and inequalities are affine and the terms are integers. Logical operators $\neg$, $\wedge$ and $\vee$, and the existential and universal quantifiers $\exists$ and $\forall$ are also needed to describe the set of integer tuples. The representations are known as Presburger formulas. In the experimental implementation, these expressions of Presburger formulas are manipulated using the Omega Library.

*Integer tuple relation*s with rules are described by Presburger formulas, too. For example the relation $R = \{[i,j] \to [x,y] : i-1 \leq x \leq i \wedge j-1 \leq y \leq j+1\}$ when applied to S_0 yields the following set:

$$R(S_0) = \{[x,y] : 0 \leq x \leq 1 \wedge 0 \leq y \leq N+1\}$$

Note that given a relation R the size and shape of the original set S and that of $R(S)$ for a relation R can be different. The union $\cup$ of two relations R_1 and R_2 is defined as:

$$R_1 \cup R_2 = R, if\ \forall S, R_1(S) \cup R_2(S) = R(S)$$

2.2.2 Code Generation

The Omega Library provides the functionality of generating loop nest code in C for a given integer tuple set, which can be used in the automated implementation of tiling transformation schemes. For example, for integer tuple set $S_1 = \{[i,j] : 0 \leq i < 100 \wedge 0 \leq j < 1000\}$, the generated code loop nest is as follows:

The loop iterations of the generated loop nest code are exactly the integer tuples in the given integer tuple set in lexicographic order. The loop nest generated by the Omega Library is sequential code. In order to express parallelism, the experimental implementation

```
for(int i = 0; i < 100; ++i)
    for(int j = 0; j < 1000; ++j)
        << loop body >>
```

Figure 2.7: Generated loop nest code for the integer tuple set S_1

provides the functionality to generate parallel loops for different programming environment. Figure 2.8-(a) and (b) show the generated OpenMP and MPI codes if the outmost loop is parallelized.

```
#pragma omp parallel for
for(int i = 0; i < 100; ++i)
    for(int j = 0; j < 1000; ++j)
        << loop body >>
```

(a) OpenMP code

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
int i = rank; // for(int i = 0; i < 100; ++i)
    for(int j = 0; j < 1000; ++j)
        << loop body >>
```

(b) MPI code

Figure 2.8: Parallel code with OpenMP and MPI

Chapter 3

Hierarchical Overlapped Tiling

3.1 Introduce Redundant Computation Through Overlapped Tiling

Consider the code in Figure 3.1-(a) where two parallel loops are executed in a shared memory machine. Although this figure shows a natural representation of the computation, the pair of loops may cause unnecessary cache misses, depending on how they are scheduled. If the loops are scheduled naively, e.g., dynamic scheduling, the second loop will likely incur frequent cache misses. To increase locality, and also coarsen the granularity of the parallel tasks, the programmer can tile and fuse the loops, as shown in Figure 3.1-(b). The resulting code requires an explicit barrier to guarantee correctness, because of the data dependences between neighboring tiles of iterations (during iteration t of the outer loop, j consumes data produced by adjacent tiles of loop i, namely tiles $t-1$ and $t+1$ or just one of them at the boundaries). Notice that locality would improve if the same task executes the corresponding i and j tiles in the code of Figure 3.1-(b). However, good locality is only possible if array A can be kept in cache memory when the execution moves from the first to the second loop. If, however, the array A is larger than the total cache of the processors executing the loops, the traditional loop fusion and tiling transformation applied in Figure 3.1-(b) will not benefit from locality, because all the iterations of the i loop must complete before the j loop executes. Besides the difficulties for achieving locality of naive tiling, the parallelization transformation may do a suboptimal job because of the barrier introduced. On some architectures, barriers are expensive synchronization operations, and could additionally cause load imbalance. Furthermore, because of the barrier the transformation from Figure 3.1-(a) to Figure 3.1-(b) is not possible in some languages, such as OpenMP and OpenCL [23], which do not allow global barriers inside data parallel constructs.

```
parallel for(int i = 0 : N-1)
    A[i] = ...;
parallel for(int j = 0 : N-1)
    ... = A[j-1] + A[j] + A[j+1];
```

(a) Original loops

```
parallel for(int t = 0 : N/T)
    for(int i = t*T; i < min(N, (t+1)*T; i++)
        A[i] = ...;
    BARRIER;
    for(int j = t*T; j < min(N, (t+1)*T; j++)
        ... = A[j-1] + A[j] + A[j+1];
}
```

(b) Traditional tiling and fusion

```
parallel for(int t = 0 : N/T) {
    for(int i = max(0,t*T-1);
            i < min(N,(t+1)*T+1); i++)
        A[i] = ...;
    for(int j = t*T; j<min(N,(t+1)*T); j++)
        ... = A[j-1] + A[j] + A[j+1];
}
```

(c) Overlapped tiling

Figure 3.1: A simple tiling example for parallel loops

To remove the synchronization and enhance locality, the code can be transformed into the form shown in Figure 3.1-(c). In this case, each iteration of the outer loop t produces all the data it needs so that its iterations (which correspond to tiles) are independent from each other. This is achieved because each iteration performs redundant computation. The result is a code without the BARRIER and with increased locality.

In the example in Figure 3.1-(c) loop i produces $A[max(0, t*T-1) : min(N, (t+1)*T)]$ in each iteration of the outer loop, that is, $T+2$ elements, 2 more than the number of elements of A computed by loop i in Figure 3.1-(b). In total the N/T executions of loop i in Figure 3.1-(c) produce $\frac{T+2}{T} * N$ elements, so this loop performs $\frac{T+2}{T} * N - N = \frac{2*N}{T}$ more iterations than the corresponding loop of Figure 3.1-(b). The transformation leading to a loop of the form of Figure 3.1-(c) is called *overlapped tiling*.

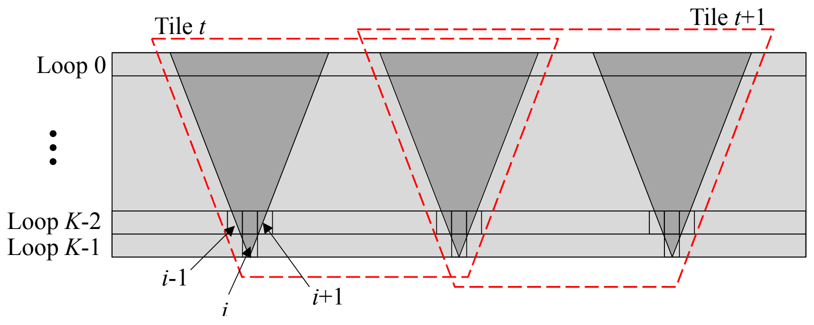
Figure 3.2-(a) shows a code snippet which represents a typical stencil computation. Pairs of consecutive executions of the inner loop form a pattern similar to that of the two inner loops in Figure 3.1-(a). If applying overlapped tiling repetitively and fuse all K executions of the inner loop, it is possible to execute the outer loop without using any barrier. The number of consecutive loops fused is the *depth* of the transformation. In this example, the fusion *depth* is K. The total amount of redundant computation usually grows with the value of *depth*. Figure 3.2-(b) shows the area of overlap. The triangles that bracket each tile represent the redundant computation.

3.2 Hierarchical Overlapped Tiling

The basic idea of overlapped tiling is the trade-off between redundant computation and synchronization overhead. The profitability of overlapped tiling transformation over traditional tiling transformation depends on that the amount of redundant computation introduced should be smaller than total synchronization overhead. However, synchronization overhead is highly related to the performance of the communication channels on the specific target platform, e.g., global memory, bus and shared cache. This means that the efficiency of overlapped tiling is sensitive to the hardware layer which tiles are mapped to. While overlapped

```
for(int k = 0 ; k < K; k++) {
    parallel for(int i = 0 : N-1)
        B[i] = A[i-1] + A[i] + A[i+1];
    swap(A, B);
}
```

(a) Code snippet of K consecutive loops in a stencil code

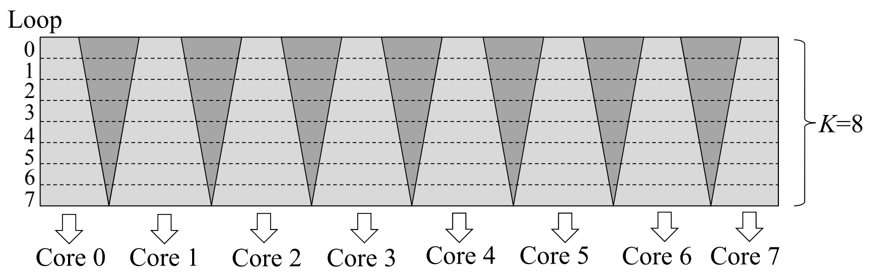


(b) Overlapped tiling

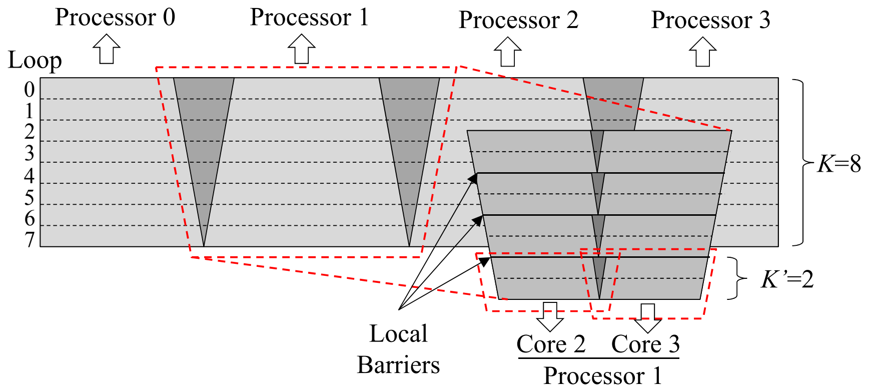
Figure 3.2: Overlapped tiling of K loops

tiling removes synchronization and enhances locality, it does not take into account the hierarchical organization of today's machines. Furthermore, it could suffer from substantial amount of redundant computation. As the fusion *depth* increases, more synchronizations can be removed, but there is also an increase in the total amount of redundant computation (the triangle-shaped areas in Figure 3.2-(b)). Hence, the use of *hierarchical overlapped tiling* is proposed to balance communication overhead and redundant computation.

Figure 3.3 contrasts overlapped tiling with hierarchical overlapped tiling. The example in the figure assumes 8 consecutive loops executing on a 4-way/8-core multicore system where the two cores on each processor share the last level cache. Figure 3.3-(a) illustrates overlapped tiling across the eight cores while Figure 3.3-(b) illustrates hierarchical overlapped tiling. In Figure 3.3-(b), overlapped tiling is first applied across the four processors. Within each processor, pairs of consecutive loops are fused. This forces a local barrier between each pair of loops ($loop_0$ and $loop_1$, $loop_2$ and $loop_3$, and so on). This barrier, however, only synchronizes the two cores on each processor and therefore its cost should be relatively low.



(a) Overlapped tiling



(b) 2-level hierarchical overlapped tiling

Figure 3.3: Comparison of overlapped tiling and hierarchical overlapped tiling on a 4-way/8-core multicore system, where each processor contains 2 cores on chip that share the last level cache.

Within each processor, overlapped tiling is applied to enable the parallel execution of pairs of loops across the two cores without the need for a barrier. Compared to overlapped tiling, the total amount of redundant computation (the shadowed triangle areas between different processors and the smaller shadowed triangles between neighboring cores) caused by the 2 levels of tiling is much smaller. This reduction of the redundant computation is the main source of the performance benefit of hierarchical overlapped tiling over plain overlapped tiling.

3.3 Analytical Modeling

This section gives a quantitative analysis of overlapped tiling and hierarchical overlapped tiling.

Consider K consecutive parallel loops $loop_0$, $loop_1$, ..., $loop_{K-1}$. Assume that the k-th loop $loop_k$ ($0 \leq k < K$) has the form shown Figure 3.4.

```
parallel for(int $\vec{i} = [i_0, i_1, ..., i_{n-1}] \in I_0$) { // $loop_0$
      ...
}
parallel for(int $\vec{i} = [i_0, i_1, ..., i_{n-1}] \in I_1$) { // $loop_1$
      ...
}
      ...
parallel for(int $\vec{i} = [i_0, i_1, ..., i_{n-1}] \in I_k$) { // $loop_k$
      ... = $A_k^0[\vec{f}_k^0(\vec{i})]$;
      ... = $A_k^1[\vec{f}_k^1(\vec{i})]$;
                ...
      ... = $A_k^{L_k-1}[\vec{f}_k^{L_k-1}(\vec{i})]$;
     $B_k^0[\vec{g}_k^0(\vec{i})]$ = ...;
     $B_k^1[\vec{g}_k^1(\vec{i})]$ = ...;
                ...
     $B_k^{M_k-1}[\vec{g}_k^{M_k-1}(\vec{i})]$ = ...;
}
      ...
parallel for(int $\vec{i} = [i_0, i_1, ..., i_{n-1}] \in I_{K-1}$){// $loop_{K-1}$
      ...
}
```

Figure 3.4: A sequence of K parallel loops

Without loss of generality, this thesis assumes that the body of $loop_k$ reads from L_k n'-dimensional arrays A_k^0, A_k^1, ..., $A_k^{L_k-1}$ and writes to M arrays B_k^0, B_k^1,...,$B_k^{M_k-1}$ which are also n'-dimensional. This thesis also assumes that no A array overlaps with a B array. To simplify discussion it is assumed that $A_k^0 = A_k^1 = ... = A_k^{L_k-1} = A_k$ and $B_k^0 = B_k^1 = ... = B_k^{M_k-1} = B_k$. There are L_k references to A_k on the RHS of the first L_k assignment statements in the body of the loop: $A_k[\vec{f}_k^0(\vec{i})]$, $A_k[\vec{f}_k^1(\vec{i})]$, ..., $A_k[\vec{f}_k^{L_k-1}(\vec{i})]$, with $\vec{f}_k^l : \mathbb{Z}^n \rightarrow \mathbb{Z}^{n'}, 0 \leq l < L_k$. There are M_k references to elements of B_k on the LHS of the last M_k statements: $B_k[\vec{g}_k^0(\vec{i})]$, $B_k[\vec{g}_k^m(\vec{i})]$,

..., $B_k[\vec{g}_k^{M_k-1}(\vec{i})]$, with $\vec{g}_k^m : \mathbb{Z}^n \to \mathbb{Z}^{n'}, 0 \leq m < M_k$.

There are 3 sets that must be computed for *loop*$_k$: I_k, the iteration space; R_k, the set of subscripts of A_k; and W_k the set of subscripts of B_k:

$$R_k = \{\vec{r}\} = \bigcup_{l=0}^{L_k-1} \{[r_0, r_1, ..., r_{n'-1}] : \vec{r} = \vec{f}_k^l(\vec{i}) \wedge \vec{i} \in I_k\}$$
$$W_k = \{\vec{w}\} = \bigcup_{m=0}^{M_k-1} \{[w_0, w_1, ..., w_{n'-1}] : \vec{w} = \vec{g}_k^m(\vec{i}) \wedge \vec{i} \in I_k\}$$

C_k is defined the *consuming relation* from I_k to R_k, $C_k(I_k) = R_k$, and P_k is for the *producing relation*, as the relation from W_k to I_k, $P_k(W_k) = I_k$. C_k and P_k represent the access pattern of the loop body. C_k and P_k are used to describe the different tiling transformations. I_k, R_k, W_k, C_k and P_k are related as follows:

$$R_k = C_k(I_k), \qquad I_k = P_k(W_k) \;\; \textit{or} \;\; W_k = P_k^{-1}(I_k)$$

3.3.1 Overlapped Tiling

Performing loop fusion and tiling for a sequence of loops of the form shown in Figure 3.4 is equivalent to finding Q partitions (tiles) I_k^0, I_k^1, ..., I_k^{Q-1} of the iteration space I_k of each *loop*$_k$. Similar to the definition of R_k and W_k discussed in the last subsection, R_k^q is defined as the set of array element indices of A for tile I_k^q, and W_k^q denotes the set of array elements indices of B for tile I_k^q. So,

$$R_k^q = C_k(I_k^q), \qquad I_k^q = P_k(W_k^q) \;\; \textit{or} \;\; W_k^q = P_k^{-1}(I_k^q)$$

Traditional loop fusion and tiling, such as that used in the code of Figure 3.1-(b), produce tessellating tiles. Because the tiles are a partition of the iteration space:

$$I_k^{q1} \cap I_k^{q2} = \phi, \quad q1 \neq q2$$

However, with overlapped tiling the sum of the size of the tiles I_k^q can be larger than the size I_k. The amount of redundant computation RC_k performed by $loop_k$ is:

$$RC_k = |I_k^0| + |I_k^1| + ... + |I_k^{Q-1}| - |I_k| \geq 0$$

In traditional loop fusion and tiling, the tiles of the different loops can be unrelated thanks to the barriers. However, in overlapped tiling the data read in an iteration tile must be produced within the same tile to eliminate the need of synchronization. To simplify the discussion, Assume that the data flow in the sequence of fused loops $loop_0$, $loop_1$, ... $loop_{K-1}$ forms a linear chain. This means that $A_{k+1}^j = B_k^j$ for all k. Under this assumption, it is necessary that $R_{k+1}^q \subseteq W_k^q$ to avoid unnecessary work. Let $R_{k+1}^q = W_k^q$. Hence the corresponding tiles I_{k+1}^q and I_k^q of neighboring loops are related as follows:

$$R_{k+1}^q = C_{k+1}(I_{k+1}^q) = P_k^{-1}(I_k^q) = W_k^q \quad \Rightarrow$$

$$I_k^q = P_k(W_k^q) = P_k(R_{k+1}^q) = P_k(C_{k+1}(I_{k+1}^q)) \tag{3.1}$$

Equation 3.1 states that the tiles of $loop_k$ are determined by the tiles of $loop_{k+1}$. Equation 3.1 provides the procedure to perform overlapped tiling: given an arbitrary partition into tiles of loop $loop_{K-1}$ (the last loop in the sequence), the tiles of all previous loops $loop_k$ ($0 \leq k < K$) can be determined iteratively. Furthermore, all the corresponding tiles from the different loops are fused to build the new loop body.

Consider the code in Figure 3.2-(a). After unrolling, there will be a sequence of K loops. Since every $loop_k$ in Figure 3.3-(a) is the same,

$$
\begin{aligned}
I_0 = I_1 = ... = I_{K-1} = I = \{[i] : 0 \le i < N\} \\
C_0 = C_1 = ... = C_{K-1} = C = \{[i \to x] : i - 1 \le x \le i + 1\} \\
P_0 = P_1 = ... = P_{K-1} = P = \{[x \to i] : i = x\}
\end{aligned}
$$

Initially, assign the following tiling partition for the last loop $loop_{K-1}$:

$$I^q_{K-1} = \{[i] : q \times N/Q \le i < (q+1) \times N/Q\}$$

which evenly partitions the iteration space, I_{K-1}, so that $|I^q_{K-1}| = N/Q$.

Next, the previous loops can be tiled using Equation 3.1 (to simplify the discussion, the boundaries are ignored):

$$
\begin{aligned}
|I^q_{K-2}| &= |P_{K-2}(C_{K-1}(I^q_{K-1}))| = |P(C(I^q_{K-1}))| \\
&= |\{[i] : q \times N/Q - 1 \le i < (q+1) \times N/Q + 1\}| \\
&= N/Q + 2 = |I^q_{K-1}| + 2 \\
|I^q_{K-3}| &= |P(C(I^q_{K-2}))| = N/Q + 4 = |I^q_{K-2}| + 2 \\
&... \\
|I^q_0| &= |P(C(I^q_1))| = N/Q + 2 \times (K-1) = |I^q_1| + 2 \qquad (3.2)
\end{aligned}
$$

Therefore:

$$|I_k^q| = |P(C(I_{k+1}^q))| = N/Q + 2 \times (K - 1 - k)$$
$$RC_k = (\sum_{q=0}^{Q-1} |I_k^q|) - |I_k| = 2 \times Q \times (K - 1 - k)$$

The total amount of redundant computation RC is defined as the sum of the redundant computation of each $loop_k$:

$$RC = \sum_{k=0}^{K-1} RC_k = Q \times K \times (K - 1) \tag{3.3}$$

RC is a monotonic function as the number of tiles Q and the number of loops to fuse K. According to Equation 3.3, for the example shown in Figure 3.2-(a), the trend is that the amount of redundant computation RC increases with both Q and K. Although this observation is derived from the specific example, it is easy to see that the trend is true in general. Compared with traditional loop fusion and tiling, where synchronization is necessary, overlapped tiling saves the overhead of $K - 1$ barriers. Suppose the average overhead of each barrier is t_s, and the average computation time for each iteration in the original code is t_c, the overhead difference between overlapped tiling over traditional tiling is:

$$\Delta Overhead = t_s \times (K - 1) - t_c \times RC/Q \tag{3.4}$$

3.3.2 Hierarchical Overlapped Tiling

According to the analysis above, coarse grain tiles (and thus small number of tiles) reduce the amount of redundant computation in overlapped tiling. However, too few tiles would reduce the amount of parallelism. Similarly, reducing the number of fused loops also reduces the amount of redundant computation, but at the expense of the additional synchronization required between loops.

Hierarchical overlapped tiling should be done according to the memory hierarchy of the target machine. Consider a multicore system with N_p processors, where each processor has N_c cores sharing the last level cache. It is expected that the average overhead of synchronizing the processors sharing a cache (t'_s) will be significantly smaller than that of synchronizing cores on different processors (t_s) that need to communicate through main memory and/or bus.

$$t_s/t'_s \gg 1 \tag{3.5}$$

Based on the above observation, it is possible to proceed as follows: first, perform coarse-grain tiling for processors; then perform overlapped tiling within the tile that is assigned to each processor, and generate sub-tiles for each core of the processor. During this 2-level tiling, the parameters for overlapped tiling are determined separately for each level.

For simplicity, assume that the total number of tiles, Q, generated by the plain overlapped tiling discussed in the previous subsection is equal to the number of cores: $Q = N_p \times N_c$. During the first level tiling of the hierarchical overlapped tiling, only N_p tiles are generated, so the total amount of redundant computation introduced in this level is:

$$RC_1 = RC(N_p, K) = N_p \times K \times (K - 1)$$

After the first level tiling, each processor q is assigned a coarse-grain tile: I^q_0, I^q_1, ..., I^q_{K-1} ($0 \leq q < N_p$), where the tiles assigned to each processor is implemented as a sequence of inner-loops. Then, overlapped tiling can be applied within each tile. However, since the sub-tiles are to be mapped to cores of the same processor, it is expected that the overhead of synchronization of cores within the same processor, t'_s will be significantly smaller than the synchronization between cores across processors, t_s. As a result, for each tile in the second level of tiling the number of fused loops (fusion *depth*) can be smaller. Although this increases the amount of synchronization, it also reduces the amount of redundant computation. Suppose the second level of tiling only fuses K' consecutive loops each time and $K' < K$, then the amount of redundant computation for each processor in the second level tiling would be:

$$RC_2 = RC(N_c, K') = N_c \times K' \times (K' - 1) \tag{3.6}$$

Therefore, the total amount of redundant computation of 2-level overlapped tiling is:

$$\begin{aligned}
RC' &= RC_1 + N_p \times RC_2 \\
&= N_p \times K \times (K-1) + N_p \times N_c \times K' \times (K'-1) \\
&= \frac{Q}{N_c} \times K \times (K-1) + Q \times K' \times (K'-1) \\
&= Q \times K \times (K-1) \times (\frac{1}{N_c} + \frac{K' \times (K'-1)}{K \times (K-1)}) \\
&\approx RC \times (1/N_c + (K'/K)^2)
\end{aligned}$$

Furthermore, additional K/K' local synchronization operations are introduced during the second level of tiling. This adds an extra latency of $t'_s \times K/K'$. Hence the overhead difference between 2-level overlapped tiling and traditional tiling is:

$$\begin{aligned}
\Delta Overhead' &= t_s \times (K-1) - t'_s \times K/K' - t_c \times RC'/Q \\
&\approx t_s \times (K-1) - t'_s \times \frac{K}{K'} - \frac{t_c}{Q} \times RC \times (\frac{1}{N_c} + (\frac{K'}{K})^2) \\
&= t_s \times (K - 1 - \frac{t'_s}{t_s} \times \frac{K}{K'}) - \frac{t_c}{Q} \times RC \times (\frac{1}{N_c} + (\frac{K'}{K})^2)
\end{aligned}$$

As mentioned before, t'_s is much smaller than t_s (Equation 3.5). Thus, if K' is appropriately chosen, $\frac{t'_s}{t_s} \times \frac{K}{K'}$ can still be much smaller than 1. Then $\Delta Overhead'$ can be made larger than $\Delta Overhead$ as defined in 3.4, which shows the potential benefit of hierarchical overlapped tiling.

3.4 Unified Tiling Representation

This section uses the unified tiling representation defined in Section 2.1 to describe the overlapped tiling and hierarchical overlapped tiling discussed in this Chapter.

The model and analysis of overlapped tiling above in this chapter are for a sequence of consecutive parallel loops, which is the paradigm of streaming applications. If every parallel loop in Figure 3.4 has the same form, that is, I_0=I_1=...=I_k=...=I_{K-1} and $\forall k, j$, $L_k = L = M_k = M$, $\vec{f}_k^j = \vec{f}^j$, $\vec{g}_k^j = \vec{g}^j$, the consecutive parallel loops in figure 3.4 become equivalent to the following loop nest:

```
1  for (int $k=0$; $k<K$; $++k$) {
2      parallel for(int $\vec{i} = [i_0, i_1, ..., i_{n-1}] \in I_0$) {
3          ... = $A^0[\vec{f}^0(\vec{i})]$;
4          ... = $A^1[\vec{f}^1(\vec{i})]$;
5                ...
6          ... = $A^{L-1}[\vec{f}^{L-1}(\vec{i})]$;
7          $B^0[\vec{g}^0(\vec{i})]$ = ...;
8          $B^1[\vec{g}^1(\vec{i})]$ = ...;
9                ...
10         $B^{M-1}[\vec{g}^{M-1}(\vec{i})]$ = ...;
11     }
12     swap($A$, $B$);
13 }
```

Figure 3.5: Equivalent loop nest to the K consecutive parallel loops

The loop nest in Figure 3.5 forms an $(n+1)$-dimensional iteration space: $[0 : K-1] \times I_0$, denoted as I. The iteration space does not necessarily have the shape of an $(n+1)$-dimensional parallelepiped, e.g., if the shape of I_0 is irregular, so the corresponding basis matrix E may not exist. Because of Line 12, each iteration of the outmost k loop depends on the previous iterations. The dependence vectors should have the form of $(1, ...)$. So the tiling problem has the iteration space I and the following dependence matrix D (each row of D is a dependence vector):

$$D = \begin{pmatrix} \vec{d}_0 \\ \vec{d}_1 \\ ... \\ \vec{d}_{m-1} \end{pmatrix} = \begin{pmatrix} 1, ... \\ 1, ... \\ ... \\ 1, ... \end{pmatrix}.$$

Each row in D determines a instance of producing or consuming relation.

Each tile is consisted by the I_0^q, I_1^q, ..., I_{K-2}^q, I_{K-1}^q, $I_k^q = P(C(I_{k+1}^q))$, where K is the parameter that affects performance. So tile shape $\mathbb{T}$ is:

$$\mathbb{T} \;=\; \bigcup_{k=0}^{K-1} \{(k : \vec{i}) \mid \vec{i} \in I_k^q\}, \qquad\qquad I_k^q = P(C(I_{k+1}^q))$$

Because $I_k^q = P(C(I_{k+1}^q))$, to determine $\mathbb{T}$ it is only necessary to choose the tile in the last step. So I_{K-1}^q, which is the tile in the last step, determines the size and shape of the tiles in other steps. Usually the shape of I_{K-1}^q is chosen to be an n-dimensional rectangle parallelepiped. For example, let $I_{K-1}^q = \mathit{span}(W)$, in which

$$W = \begin{pmatrix} w_0, 0, 0, ..., 0 \\ 0, w_1, 0, ..., 0 \\ ... \\ 0, 0, 0, ..., w_{n-1} \end{pmatrix}. \tag{3.7}$$

Then I_{K-1}^q becomes an n-dimensional rectangle parallelepiped with w_0, w_1, ...,w_{n-1} being the width in each dimension. If $n = 1$, the shape of $\mathbb{T}$ becomes a trapezoid with top width w_0 and height K. For $n = 2$, the shape usually becomes a bounded pyramid. Since $\mathbb{T}$ does not have a hyper-parallelepiped shape, it is not possible to use a tiling matrix T to describe the shape of tiles.

Overlapped tiling allows overlap between tile. According Equation 2.7, the union of all the tiles must contain the original iteration space. For a trapezoid-shaped or pyramid shaped tile, the top part I^q_{K-1} is the narrowest. So the choice of repetition pattern must guarantee that there is no gap between tiles even on the narrowest part of tiles. If $I^q_{K-1} = span(W)$ as defined in Equation 3.7, the repetition matrix is shown as follows:

$$R = \begin{pmatrix} K, 0, 0, 0, ..., 0 \\ 0, w_0, 0, 0, ..., 0 \\ 0, 0, w_1, 0, ..., 0 \\ ... \\ 0, 0, 0, 0, ..., w_{n-1} \end{pmatrix} = \begin{pmatrix} K, 0 \\ 0, W \end{pmatrix}.$$

In the tiled iteration space I', because of the redundant computation introduced, there is no dependence between tiles in the same step (i.e., there is no "horizontal" dependence). Therefore, every dependence vector between tiles must have the following form:

$$\vec{d'} = (d_0, d_1, ..., d_{n-1}), \quad d_0 = 1, \quad d_k = 0, \pm 1, \quad k = 1, 2, ..., n-1$$

The dependence matrix in I' must have the following form:

$$D' = \begin{pmatrix} \vec{d'}_0 \\ \vec{d'}_1 \\ ... \\ \vec{d'}_{m-1} \end{pmatrix} = \begin{pmatrix} 1, * \\ 1, * \\ ... \\ 1, * \end{pmatrix}.$$

Hence a scheduling mapping, $\mathbb{S}$, that maximizes parallelism can expose n degrees of parallelism. That is, all the tiles within the same step can be executed in parallel. Then the scheduling matrix S is

$$\begin{aligned} S &= \begin{pmatrix} 1 \\ 0 \\ ... \\ 0 \end{pmatrix}, \\ \mathbb{S}(\vec{i'}) &= \vec{i'} \cdot S = (i'_0, i'_1, ..., i'_n) \cdot S = i'_0. \end{aligned}$$

The effect of overlapped tiling makes the above schedule $\mathbb{S}$ valid. After overlapped tiling, the first component in each row of the dependence matrix D' is still 1, which is the same form as the original D. Therefore, the same degree of parallelism can be exposed for coarse grain tiles.

To sum up, Table 3.1 shows the unified representation of overlapped tiling. Since hierar-

Input	$I, D = \begin{pmatrix} 1,* \\ 1,* \\ ... \\ 1,* \end{pmatrix}$
D'	$D' = \begin{pmatrix} 1,* \\ 1,* \\ ... \\ 1,* \end{pmatrix}.$
$\mathbb{T}$	$W = \begin{pmatrix} w_0, 0, 0, ..., 0 \\ 0, w_1, 0, ..., 0 \\ ... \\ 0, 0, 0, ..., w_{n-1} \end{pmatrix}, \quad I^q_{K-1} = span(W)$ $I^q_k = P(C(I^q_{k+1})), \quad \mathbb{T} = \bigcup_{k=0}^{K-1}[k : I^q_k].$
$\mathbb{R}$	$R = \begin{pmatrix} K, 0 \\ 0, W \end{pmatrix}.$
$\mathbb{S}$	$S = \begin{pmatrix} 1 \\ 0 \\ ... \\ 0 \end{pmatrix}.$

Table 3.1: Representation of Overlapped Tiling

chical overlapped tiling is the recursion of overlapped tiling in the top-down approach, the representation of tiling in each level is the same as Table 3.1, except that the input iteration space I is the tile $\mathbb{T}$ of the higher level tiling.

3.5 Evaluation

3.5.1 Implementation

An experimental system is implemented to evaluate the efficiency of overlapped tiling and hierarchical overlapped tiling. A diagram representing the experimental system is shown in Figure 3.6. The dashed arrows represent offline data flow while solid arrows represent runtime data flow. The system contains three major components: a delayed compilation mechanism, an offline analyzer and the optimizer. The offline analyzer is implemented as a pass of Cetus [21]. The optimizer is a source-to-source translator which reads the original kernel code and generates OpenCL code, which is fed to the OpenCL runtime. Both, the offline analyzer and the optimizer use the Omega library [22] to perform the integer tuple space computations. In addition, the analyzer uses the Omega Library to generate loops from the integer tuple sets.

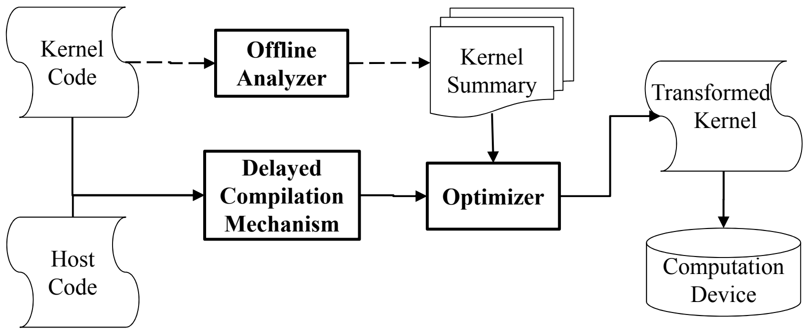


Figure 3.6: Framework of the experimental system.

3.5.2 Environment Setup

The efficiency of overlapped tiling and hierarchical overlapped tiling is evaluated on an SMP workstation with 4 Intel Xeon L7555 processors running at 1.87GHz. Each processor has 8 cores, sharing a 24MB unified L3 cache on chip. Each core contains a 256KB private L2 cache and 32KB L1 D-cache. SMT is disabled for each core, so there is one hardware thread per core.

3.5.3 Benchmarks

Eight benchmarks are evaluated: 1D/2D/3D-Jacobi, PathFinder, Poisson, Biharmonic, HotSpot and Cell. The first 3 benchmarks (1D/2D/3D-Jacobi) are Jacobi iterations for synthetic linear systems; PathFinder uses dynamic programming to find a minimum weighted path; Poisson is a numerical solver of the poisson equation, calculating the Laplace operator [4] over a 2D grid with the 5-point stencil. Biharmonic is the numerical PDE solver calculating the Biharmonic operator [4] over a 2D grid with a 13-point stencil. HotSpot implements a chip temperature estimation model[19]. Cell [3] is a 3D game of life. For each application, the operation in the body of the stencil loop is implemented as an OpenCL kernel. The inputs of the benchmarks are listed in Table 4.2.

	Data Dimension	**Problem Size**	**Points of Stencil**
1D-Jacobi	1	64K	3
2D-Jacobi	2	256x256	9
3D-Jacobi	3	64x64x64	27
PathFinder	1	100K	3
Poisson	2	256x256	5
Biharmonic	2	256x256	13
HotSpot	2	512x512	9
Cell	3	60x60x60	27

Table 3.2: ISL Benchmarks

For some stencil programs, such as Jacobi, the number of steps of the outer loop depends on a convergence test. This requires a synchronization between kernel code and host code that prevents loop fusion. To enable the overlapped and hierarchical overlapped tiling optimizations the code is modified so that the convergence test (and as result the synchronization) only occurs every 1024 iterations. The total iterations number for the benchmarks without convergence test is 16,384.

3.5.4 Experimental Results

Performance Overview

Figure 4.15 shows the performance speedup of traditional tiling, overlapped tiling and hierarchical overlapped tiling relative to the original OpenCL code. Since OpenCL only supports 2 levels of work item organization, a 2-level hierarchical overlapped tiling is used for each benchmark. For overlapped tiling and hierarchical overlapped tiling, the figure shows the speedup for the best value of the loop fusion depth K, as shown in Table 3.3 (and discussed in the next Section). In general, the performance of hierarchical overlapped tiling is always better than that of plain overlapped tiling, and overlapped tiling is always better than that of traditional tiling, with the exception of Cell and 3D-Jacobi. For Cell and 3D-Jacobi, the performance curves of the three tiling transformations are very similar. The reason is that their main data structure is 3-dimensional and the amount of redundant computation grows quartically with the fusion depth K. Therefore, there are not many opportunities for overlapped tiling and hierarchical overlapped tiling. In experiments, overlapped tiling and the 2-level hierarchical overlapped tiling achieves an average speedup of 18% and 37% over traditional tiling, respectively.

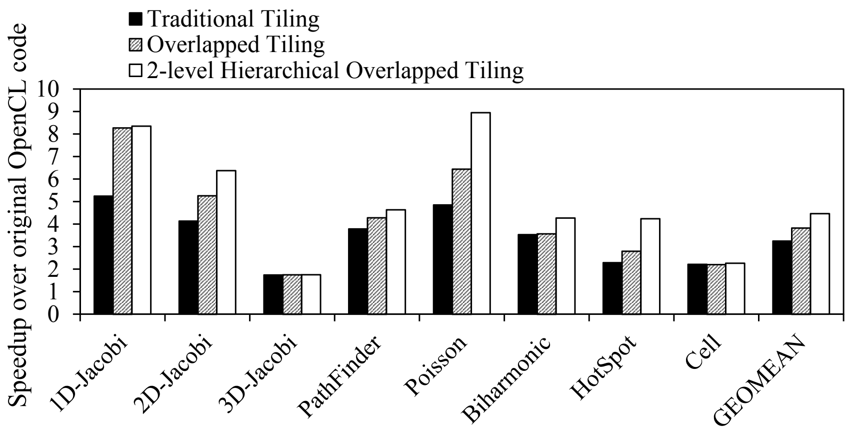


Figure 3.7: Speedup of traditional tiling, overlapped tiling and hierarchical overlapped tiling over the original OpenCL code.

Parameter Sensitivity

Loop Fusion Depth for the First Level of Tiling. Figure 3.8 shows the performance of overlapped and hierarchical overlapped tiling as the value of the loop fusion depth K changes. For overlapped tiling, there is only one value of K; for 2-level hierarchical overlapped tiling, K is the loop fusion depth for the first level of tiling, while K' is the value of loop fusion depth for the second level of tiling (K' is kept constant at 2 for the experiments in Figure 3.8). As discussed in Subsection 3.3.1 and 3.3.2, K, determines the amount of redundant computation introduced by overlapped and hierarchical overlapped tiling, and thus the overall speedup over traditional tiling.

In Figure 3.8, lines a, b and c show the speedup of traditional tiling, overlapped tiling, and 2-level hierarchical overlapped tiling over the original OpenCL code, respectively. In addition, by manually modifing the overlapped tiling code to remove the redundant computation (hence, there are race conditions and the results of the executed code are not guaranteed to be correct), the performance is shown by Line d. OpenCL standard does not support a global barrier across work item groups, which is required for traditional tiling (See Figure 3.1-(b)); thus, the barriers used for traditional tiling (Line a in Figure 3.8) are barriers

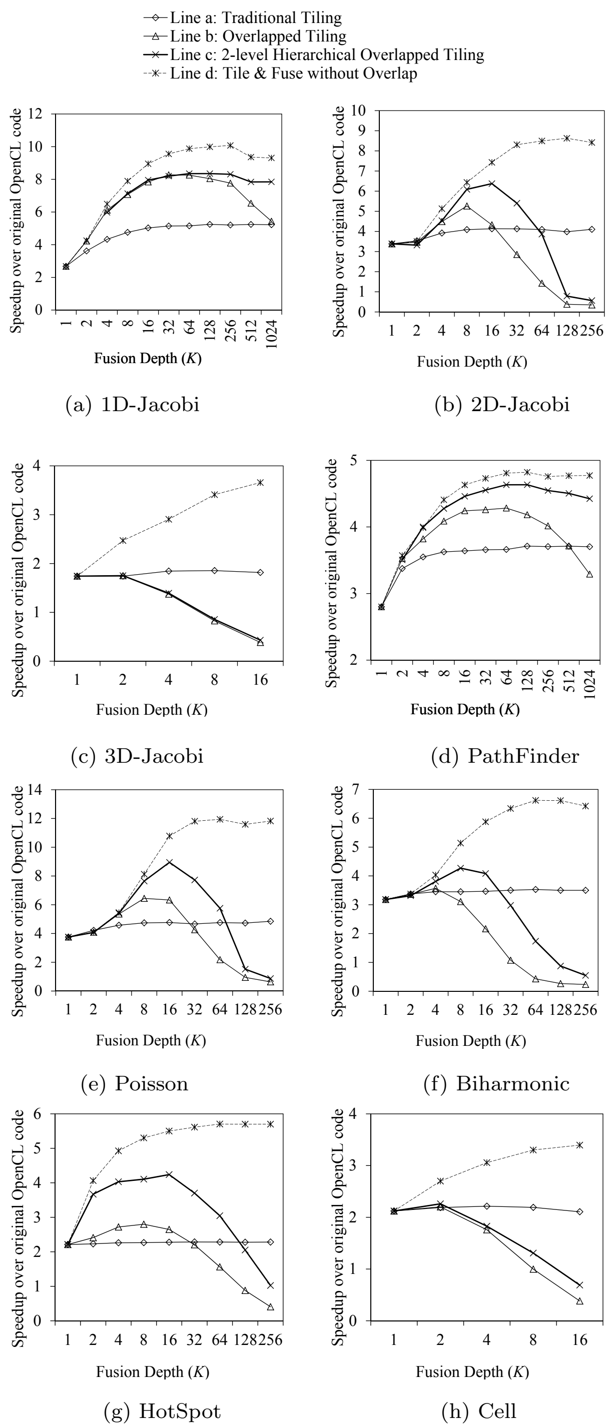


Figure 3.8: Evaluating the performance overlapped tiling and hierarchical overlapped tiling by scaling fusion depth K.

that implemented with low level primitives. However, overlapped tiling and 2-level hierarchical overlapped tiling only require synchronization within the work item groups, which is supported by the OpenCL standard. The discrepancy between overlapped tiling and Line d shows the overhead of the redundant computation introduced by overlapped tiling; the difference between traditional tiling and Line d shows the synchronization overhead saved by fusing the kernels. In Figure 3.8 it can be seen that Line d typically grows as the loop fusion depth K increases. This is because the number of synchronization operations removed grows with the depth. If do not count the cost of the redundant computation introduced, the performance benefit is always positive (if $RC = 0$, $\Delta Overhead$ in Equation 3.4 always increases when K increases). In fact, Line d defines the upper bound of performance for overlapped and hierarchical overlapped tiling.

For the two 1D benchmarks (1D-Jacobi and PathFinder), both plain overlapped tiling and hierarchical overlapped tiling can achieve significant speedup over traditional tiling, because the growth rate of redundant computation for overlapped tiling is low. For the 2D benchmarks (2D-Jacobi, Poisson, Biharmonic and HotSpot) the growth rate of the redundant computation is higher than for the 1D benchmarks. Thus, the interval of benefit (the values of fusion depth K for which lines b or c are above line a) of the 2D benchmarks is smaller than that of the 1D benchmarks for both, overlapped and hierarchical overlapped tiling. The figure also shows that the interval of benefit of hierarchical overlapped tiling is always bigger than that of plain overlapped tiling. Within the 2D benchmarks, Biharmonic shows the least speedup with overlapped tiling over traditional tiling. This is because the stencil of Biharmonic depends on 13 neighboring points versus 9 for 2D-Jacobi, 5 for Poisson, and 5 for Hotspot (see Table 4.2). As a result, the redundant computation of Biharmonic grows faster than that of the other 2D benchmarks. However, hierarchical overlapped tiling still achieves speedup over traditional tiling. Since the input data of Cell and 3D-Jacobi are 3-dimensional, the amount of redundant computation increases so fast that there is no opportunity for overlapped tiling.

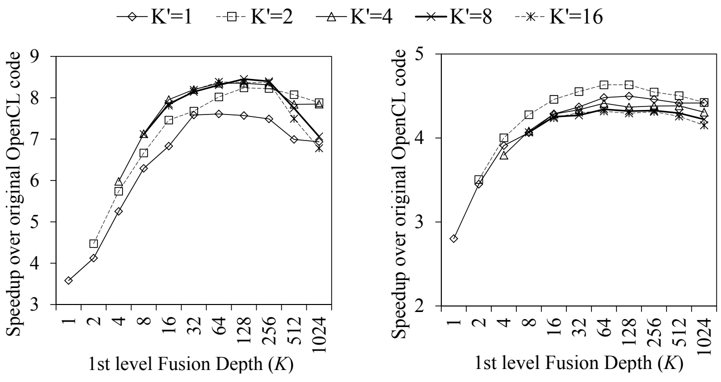
When comparing hierarchical overlapped tiling versus overlapped tiling the plots in Figure 3.8 show that hierarchical overlapped tiling performs better as the value of loop fusion depth K increases (the only exception occurs with the 3D benchmarks where all 3 tiling mechanisms behave the same), because the growth rate of redundant computation is lower for hierarchical overlapped tiling than for plain overlapped tiling. Hierarchical overlapped tiling is less sensitive to the value of K than overlapped tiling because, as mentioned above, the beneficial region of hierarchical tiling is larger than that of overlapped tiling.

The optimal value of fusion depth K value are determined by the value of t_s/t_c of the target platform. The values of K used in experiments are shown in Table 3.3.

	Overlapped Tiling	**Hierarchical Overlapped Tiling**
1D-Jacobi	$K = 32$	$K = 64$
2D-Jacobi	$K = 8$	$K = 16$
3D-Jacobi	$K = 2$	$K = 2$
PathFinder	$K = 32$	$K = 64$
Poisson	$K = 8$	$K = 16$
Biharmonic	$K = 4$	$K = 8$
HotSpot	$K = 4$	$K = 16$
Cell	$K = 1$	$K = 2$

Table 3.3: Loop Fusion depth K used in experiments.

Loop Fusion Depth for the Second Level of Tiling. Compared to the loop fusion depth K for the first level tiling of hierarchical overlapped tiling, the tuning of loop fusion depth K' for the second level tiling is more involved. This is partially because the performance of the second level tiling depends on the first level of tiling. Figure 3.9 shows the performance of hierarchical overlapped tiling for 1D-Jacobi and PathFinder with different pairs of K and K'. We can see that $K' = 2$ is the best choice for PathFinder; but there is no obvious optimal value for 1D-Jacobi. Thus, manual tuning may be required to find the optimal fusion depth K' for the second level of hierarchical overlapped tiling. However, the performance impact of the second level tiling is significantly smaller than that of the first level tiling. For instance, when K' is set to 2, the average performance loss is less than 5% compared to the best K'.



(a) 1D-Jacobi (b) PathFinder

Figure 3.9: Fusion depth K' for the second level of hierarchical overlapped tiling.

Input Size. Figure 3.10 shows the speedup of hierarchical overlapped tiling over plain overlapped tiling for the different benchmarks, as the input size increases. The speedups are computed using the best value of K. The figure shows that hierarchical overlapped tiling performs better than overlapped tiling for the 2D-benchmarks. It also shows, that the performance difference between hierarchical overlapped tiling and plain overlapped tiling becomes smaller as the input data size increases. This is because since the total number of worker threads remains constant, the tile size is determined by the input data size. With smaller tiles, the redundant computation has a higher impact; thus, overlapped tiling is less efficient, while hierarchical overlapped has more opportunities to reduce the overheads introduced by the redundant computation. With larger tiles, the redundant computation has less impact, and as a result, there is less difference between overlapped and hierarchical overlapped tiling. For the 3D benchmarks 3D-Jacobi and Cell, since the amount of redundant computation grows so fast, the optimal K for both plain overlapped tiling and hierarchical overlapped tiling is less than 2, so there is no obvious performance discrepancies between the two schemes.

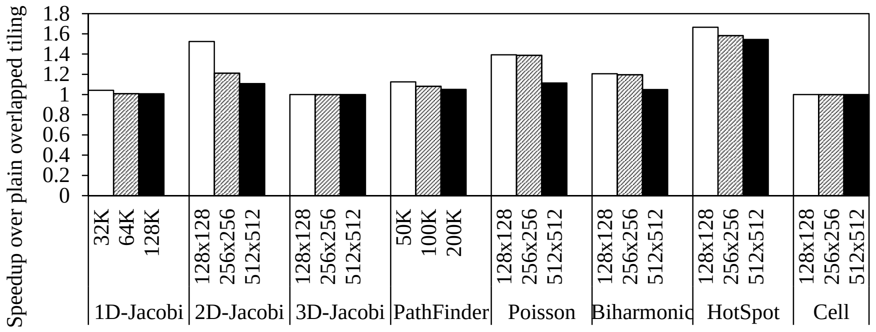


Figure 3.10: Speedup of hierarchical overlapped tiling over plain overlapped tiling with different input sizes. The horizontal axis is the input size for each benchmark.

3.5.5 Compilation Overhead

Since the experimental system transforms OpenCL kernel code at runtime, the overheads introduced need to be considered. The overhead consists of two parts: The OpenCL runtime that transforms the kernel code and the execution of the Omega Library. Caching the existing compilation result can help reduce both parts of the runtime overhead: if the sequence of kernels selected for optimization are the same as a previous sequence, no compilation needs to be done. For the benchmarks used in this paper, the OpenCL runtime compilation needs to be invoked at most twice, one for the steady state and another for the epilog. Figure 3.11 shows the compilation times of overlapped tiling and hierarchical overlapped tiling, normalized to the compilation time of the original program. On average, overlapped tiling requires 37% more compilation time, while hierarchical overlapped tiling costs about 60% more.

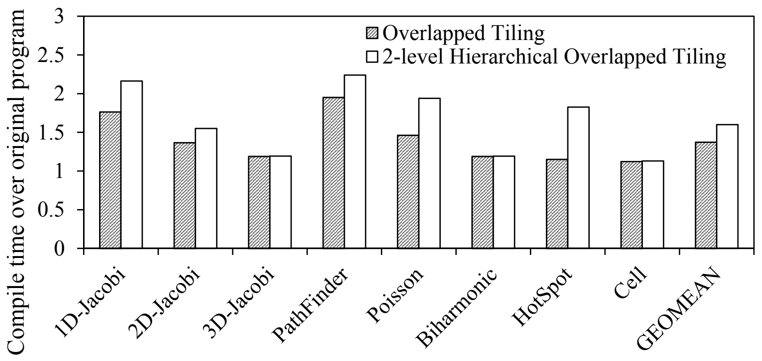


Figure 3.11: Compile time for overlapped tiling and hierarchical overlapped tiling, normalized to the compile time of the original OpenCL code.

3.6 Conclusion

In this chapter, a new transformation, hierarchical overlapped tiling, is proposed. By creating hierarchical overlapping tiles, the new transformation can reduce communication overhead among tiles while introducing smaller amount of redundant computation compared to plain overlapped tiling. An experimental system is implemented to apply the proposed transformations to OpenCL programs. Experimental results show that on average overlapped tiling and hierarchical overlapped tiling achieves 18% and 37% speedup over traditional tiling, respectively.

Chapter 4

Hiding Communication Latency with Conjugate-Trapezoid Tiling

4.1 Hiding Communication Latency

The Jacobi code in Figure 4.1-(a) is an example of a 1D ISL where for simplicity the boundary checking code has been ommitted. Figure 4.1-(b) shows the correponding standard tiled code for a distributed memory system with P nodes, where boundary elements must be explicitly sent to and received from neighboring tiles (Line 6-9) before the computation at each time step starts. Since the communication latency in a distributed memory systems is not negligible, if assuming that the computation time of Line 12 is t_{comp}, and the communication cost is t_{comm}, the total execution time of the code in Figure 4.1 is $T \times (t_{comp} \times N/P + t_{comm})$, where the communication overhead is $T \times t_{comm}$. Thus, depending on the value of t_{comm}, the performance penalty incurred due to the communication latency can be significant.

```
for(int t = 0 ; t < T; t++) {
    for(int i = 0; i < N; i++)
        Y[i]=(X[i-1]+X[i]+X[i+1])/3;
    swap(X, Y);
}
```

(a) Sequential 1-dimensional Jacobi code

```
rank = ... // the rank of the node
int w=N/P;
int lowerbound = rank*w;
int upperbound = (rank+1)*w;
for(int t = 0 ; t < T; t++) {
    MPI_Isend(X[lowerbound], ..., rank-1, t, ...);
    MPI_Isend(X[upperbound-1], ..., rank+1, t, ...);
    MPI_Recv(X[lowerbound-1], ..., rank-1, t, ...);
    MPI_Recv(X[upperbound], ..., rank+1, t, ...);

    for(int i = lowerbound; i < upperbound; i++)
        Y[i]=(X[i-1]+X[i]+X[i+1])/3;
    swap(X, Y);
}
```

(b) Standard tiled code for P nodes with no overlapping between communication and computation.

Figure 4.1: 1-dimensional Jacobi code with MPI.

4.2 Conjugate-Trapezoid Tiling

This section discusses the design of *Conjugate-Trapezoid Tiling.* First, Conjugate-Trapezoid Tiling is introduced with a 1-dimensional stencil example, and then the idea is generalized to stencils with higher dimensional stencils.

4.2.1 Definitions

The discussion in this chapter focuses on iterative stencil loops (ISL), which are iterative computations in which the elements of an array are updated using the values of neighboring elements, following a fixed pattern or stencil. An example of an ISL is shown in Figure 4.2, where in each time step element $i_0, i_1, ..., i_{d-1}$ of `Y` is computed in terms of a set of neighboring elements in `X`. The outermost t loop determines the number of timesteps.

```
for(int $t$ = 0 ; $t < T$; $t$++) {
  for(int $i_0$ = 0; $i_0 < N_0$; $i_0$++)
    for(int $i_1$ = 0; $i_1 < N_1$; $i_1$++)
      ...
        for(int $i_{d-1}$=0; $i_{d-1}$<$N_{d-1}$; $i_{d-1}$++) {
          $z_0$ = X[$\vec{f_0}(i_0, i_1, ..., i_{d-1})$];
          $z_1$ = X[$\vec{f_1}(i_0, i_1, ..., i_{d-1})$];
              ...
          $z_{m-1}$ = X[$\vec{f}_{s-1}(i_0, i_1, ..., i_{d-1})$];
          Y[$i_0, i_1, ..., i_{d-1}$] = $g(z_0, z_1, ..., z_{m-1})$;
        }
  swap(X, Y);
}
```

Figure 4.2: An ISL with a d-dimensional stencil. The total depth of the loop nest is $d + 1$.

The ISL forms an $(n + 1)$-dimensional iteration space I. The shape of I is a $(n + 1)$-dimensional hyper-cuboid:

$$I = span(E) = span(\begin{pmatrix} T, 0, 0, ..., 0 \\ 0, N_0, 0, ..., 0 \\ 0, 0, N_1, ..., 0 \\ ... \\ 0, 0, 0, ..., N_{n-1} \end{pmatrix}).$$

Each $\vec{f_k}$ is a $\mathbb{Z}^d \to \mathbb{Z}^d$ mapping. $\vec{f_0}$, $\vec{f_1}$, ..., $\vec{f}_{m-1}$ defines m dependence vectors: $\vec{d_k} = (1, d_{k,0}, d_{k,1}, ..., d_{k,n-1}), k = 0, 1, ..., m - 1$, where the first component of each dependence vector is 1. [1] These dependences are collected in a matrix:

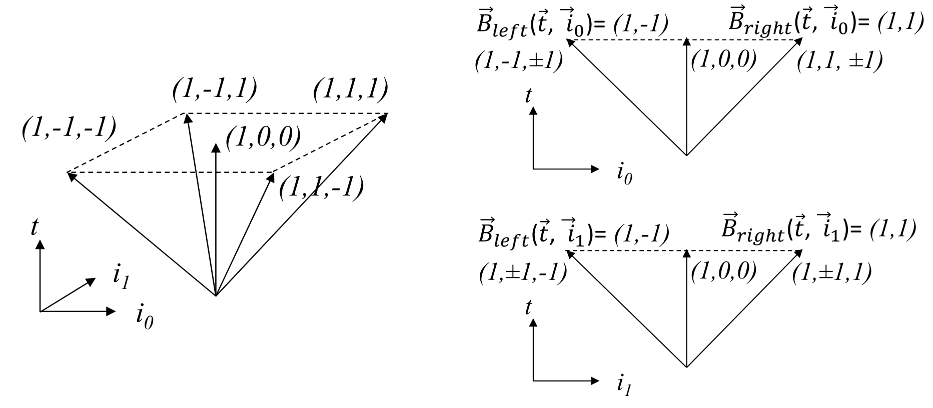
$$D = \begin{pmatrix} \vec{d_0} \\ \vec{d_1} \\ ..., \\ \vec{d}_{m-1} \end{pmatrix}.$$

[1]Because the values computed in iteration t_0 of the outermost loop are consumed in iteration $t_0 + 1$.

Here n is the dimensionality of the array involved in the computation and m is the number of point in the stencil.

The dependence vectors generate a cone in the $(n+1)$-dimensional space, as shown in Figure 4.3-(a) for a five-point stencil in a 3D space $< \vec{t}, \vec{i}_0, \vec{i}_1 >$. The projections of the cone on $< \vec{t}, \vec{i}_0 >$ and $< \vec{t}, \vec{i}_1 >$ planes are triangles, as shown in Figure 4.3-(b). The boundaries of the projected triangles are defined by the dependence vectors in the boundaries of the cone. Vectors $\vec{B}_{left}$ and $\vec{B}_{right}$ represent the boundarie edges of the projection of the dependence cone on the plane$< \vec{t}, \vec{i}_j >$ $(j = 0, 1, ..., d-1)$:

$$\vec{B}_{left}(\vec{t}, \vec{i}_j) = (1, min\{x_{k,j} | k = 0, 1, ..., m-1\})$$
$$\vec{B}_{right}(\vec{t}, \vec{i}_j) = (1, max\{x_{k,j} | k = 0, 1, ..., m-1\})$$



(a) Cone generated by the dependence vectors (b) Projections of the cone

Figure 4.3: Dependence vectors of a 5-point stencil

To facilitate later discussion, define $Proj_{<\vec{t},\vec{i}_j>}(\vec{u})$ as the projection of vector $\vec{u} = (u'$, u_0, u_1, ..., u_j, ..., $u_{n-1})$ on plane $< \vec{t}, \vec{i}_j >$:

$$Proj_{<\vec{t},\vec{i}_j>}(\vec{u}) = (u', u_j)$$

And define $Proj_{<\vec{i}_{j_0},\vec{i}_{j_1},...,\vec{i}_{j_{k-1}}>}(\vec{u})$ as the projection on the k-dimensional hyperplane $<\vec{i}_{j_0},\vec{i}_{j_1},...,\vec{i}_{j_{k-1}}>$:

$$Proj_{<\vec{i}_{j_0},\vec{i}_{j_1},...,\vec{i}_{j_{k-1}}>}(\vec{u}) = (u_{j_0}, u_{j_1}, ..., u_{j_{k-1}})$$

4.2.2 1-dimensional Stencil

Consider the 1-dimensional (2-dimensional iteration space) Jacobi example shown in Figure 4.1 , whose dependence matrix is:

$$D = \begin{pmatrix} \vec{d_0} \\ \vec{d_1} \\ \vec{d_2} \end{pmatrix} = \begin{pmatrix} 1, -1 \\ 1, 0 \\ 1, 1 \end{pmatrix}.$$

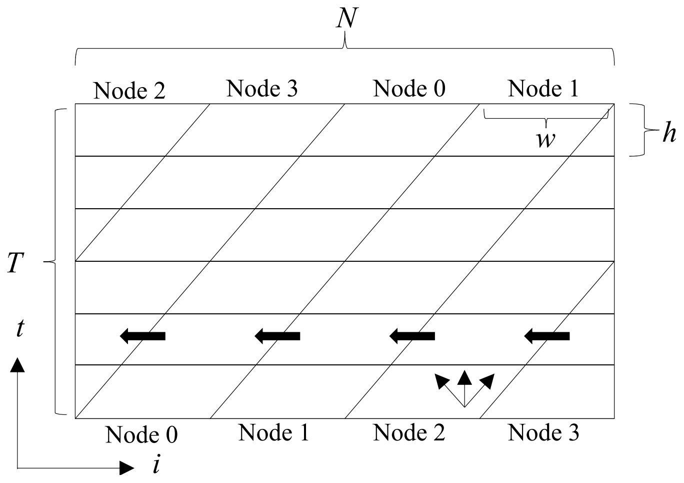


Figure 4.4: Tiling 1-dimensional Jacobi code. Thin arrows represent the dependence vectors. Thick arrows show the dependence between tiles. w and h are the parameters decided by users to determine the size of each tile.

The Conjugate-Trapezoid Tiling is composed of two levels of tiling. At the outer level, the iteration space is partitioned (tiled) into parallelogram tiles of width w and height h. For each tile, two edges are perpendicular to the t axis, and the other two edges are parallel to the dependence vector on the right boundary, $\vec{B}_{right}(\vec{t},\vec{i})$=$(1, 1)$. The tiling scheme is shown in Figure 4.4, in which w and h are the parameters decided by users to determine the size of each tile.

For a more precise description, we use the normal vector of each tiling hyper-plane of the tiling scheme, and make the reciprocal of the length of the normal vector be the distance between parallel tiling hyper-planes. Let $\vec{h}_0$ be the normal vector of the edge that is parallel to $\vec{B}_{right}(\vec{t},\vec{i})$, $\vec{h}_0 \perp \vec{B}_{right}(\vec{t},\vec{i})$, then $\vec{h}_0$ is:

$$\begin{aligned}\vec{h}_0 &= \frac{1}{w} \cdot \vec{B}_{right}(\vec{t},\vec{i}) \cdot \begin{array}{c} \begin{array}{cc} t & i \end{array} \\ \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \end{array} \\ &= \frac{1}{w} \cdot (-1, 1)\end{aligned}$$

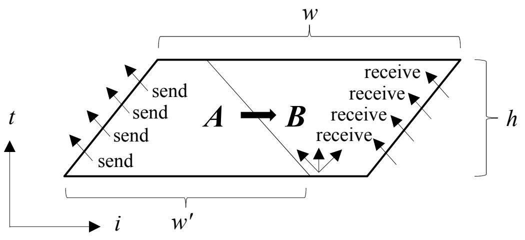
Note that the matrix $\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$ stands for a rotation transformation of 90 degrees in clockwise. And $\frac{1}{|\vec{h}_0|}$ is the distance between the tiling hyper-planes parallel to $\vec{B}_{right}(\vec{t},\vec{i})$. Similarly, let $\vec{h}_1$ be the normal vector of the edge that is perpendicular to $\vec{t}$. Intuitively $\vec{h}_1 = (1/h, 0)$ and $\frac{1}{|\vec{h}_1|} = h$. The paralleogram tiles in Figure 4.4 can be represented by the following matrix H, where each row of H is the normal vector to the edges of the parallelogram.

$$H = \begin{pmatrix} \vec{h}_0 \\ \vec{h}_1 \end{pmatrix} = \begin{array}{c} \begin{array}{cc} t & i \end{array} \\ \begin{pmatrix} -\frac{1}{w} & \frac{1}{w} \\ \frac{1}{h} & 0 \end{pmatrix} \end{array}$$

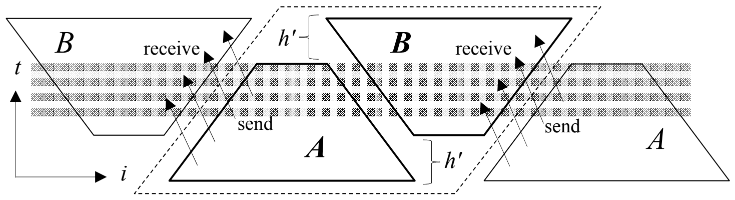
This tile shape has been chosen so that each tile only depends on its right neighbor tile. Thus, if the tiles in a row are distributed onto different nodes, a node only needs to receive data from its right neighbor node, as shown in Figure 4.4. This strategy also achieves data locality and load balance: data is reused if tiles in each (oblique) column are assigned to the same node; work load is balanced because all the tiles have the same area (tiles in the right and left boundaries can be mapped to the same node).

In order to execute the tiles within the same horizontal level in parallel, fine-grain inter-tile communication for every tile step must be done, as the example code in Figure 4.1-(b). However, with this strategy each time step a tile needs to send boundary data to its left neigbor, and receive data from its right neighbor, as shown in Figure 4.5-(a). To eliminate the communication overhead from the total execution time, a second level of tiling is applied. Each parallelogram is partitioned into two trapezoid subtiles (subtile A and B in Figure 4.5-(a)). where the edge that divides subtile A and B is parallel to the left-boundary dependence vector $\vec{B}_{left}(\vec{t},\vec{i})$=$(1,-1)$, so that there is only dependence from subtile A to subtile B. We call A and B conjugate trapezoid subtiles. Notice that these tiles can have exactly the same shape (if the stencil is symmetric), by appropriately choosing where to partition the outer tile. In the example in the Figure, this partition point is determined by w', which is computed as $h + w/2$.

As a result of this subtiles, *send* operations are in subtile A, and all *receive* operations are in subtile B. Subtile A can start execution right away, as it is independent of the B subtile, while tile B is delayed by h' time steps. Thus, each *receive* operation in subtile B is delayed by h' from its correponding *send* receive in subtile A. Thus, if the communication latency is smaller than the computation time of the h' time steps, the data sent can arrive to its destination before the *receive* operation. Thus, the communication latency can be totally hidden when using asynchronous *send* and *receive* operations. The dashed box in Figure 4.5-(b) stands for the scope for each tile after subtile B has been delayed; and the shaded stripe covers the amount of local computation between each pair of *send* and *receive.* The requirement of this scheme is that the bandwidth of the system must be large enough to allow h' messages on the fly during the computation time of h' time steps. The parameter



(a) Subtile the parallelogram tile into 2 conjugate trapezoid subtiles. There is a dependence from subtile A to subtile B.



(b) Delay subtile B by h' time steps to hide communication latency

Figure 4.5: Subtiling and delay

h' provides extra flexibility to our tiling scheme, as h' can be adjusted to fit target platforms with different configurations of bandwidth and communication latency.

4.2.3 2-dimensional Stencil

```
for(int t = 0 ; t < T; t++) {
  for(int i0 = 0; i0 < N0; i0++)
    for(int i1 = 0; i1 < N1; i1++)
      Y[i0][i1]=(X[i0-1][i1+1]+X[i0][i1+1]+X[i0+1][i1+1]
               +X[i0-1][i1]   +X[i0][i1]   +X[i0+1][i1]
               +X[i0-1][i1-1]+X[i0][i1-1]+X[i0+1][i1-1])/9;
    swap(X, Y);
}
```

Figure 4.6: Sequential 2-dimensional Jacobi code.

Consider next the Jacobi code in Figure 4.6 as an example of an ISL with a 2-dimensional stencil (with a 3-dimensional iteration space). The 9-point stencil is defined by the following dependence matrix:

$$D = \begin{pmatrix} \vec{d_0} \\ \vec{d_1} \\ ... \\ \vec{d_8} \end{pmatrix} = \begin{pmatrix} 1,0,0 \\ 1,1,0 \\ 1,-1,0 \\ 1,0,1 \\ 1,0,-1 \\ 1,1,1 \\ 1,1,-1 \\ 1,-1,1 \\ 1,-1,-1 \end{pmatrix}.$$

The projections of the dependence vectors on the $<\vec{t},\vec{i_0}>$ and $<\vec{t},\vec{i_1}>$ planes are:

$$\vec{B}_{left}(\vec{t},\vec{i_0}) = (1,-1) \quad \vec{B}_{right}(\vec{t},\vec{i_0}) = (1,1)$$
$$\vec{B}_{left}(\vec{t},\vec{i_1}) = (1,-1) \quad \vec{B}_{right}(\vec{t},\vec{i_1}) = (1,1)$$

As in the example in Figure 4.4, the outer tile can be computed using the right projection of the dependence vector on the planes $<\vec{t},\vec{i_0}>$ and $<\vec{t},\vec{i_1}>$. The projections of $\vec{h}_0$ and $\vec{h}_1$

are computed as follows:

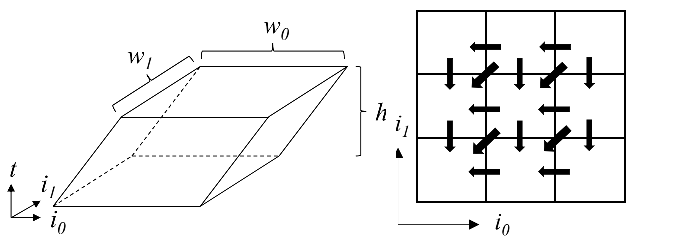
$$\begin{aligned}
Proj_{<\vec{t},\vec{i}_0>}(\vec{h}_0) &= \frac{1}{w_0} \cdot \vec{B}_{right}(\vec{t},\vec{i}_0) \cdot \begin{array}{cc} t & i_0 \\ \left(\begin{array}{cc} 0 & 1 \\ -1 & 0 \end{array}\right) \end{array} \\
&= \frac{1}{w_0} \cdot (-1, 1) \\
Proj_{<\vec{t},\vec{i}_1>}(\vec{h}_1) &= \frac{1}{w_1} \cdot \vec{B}_{right}(\vec{t},\vec{i}_1) \cdot \begin{array}{cc} t & i_1 \\ \left(\begin{array}{cc} 0 & 1 \\ -1 & 0 \end{array}\right) \end{array} \\
&= \frac{1}{w_1} \cdot (-1, 1)
\end{aligned}$$

The rest components in $\vec{h}_0$ and $\vec{h}_1$ are zeros. $\frac{1}{|\vec{h}_0|}$ and $\frac{1}{|\vec{h}_1|}$ stand for the distance between corresponding tiling hyper-planes. Similarly $\vec{h}_2 = (1/h, 0, 0)$. Thus, the resulting tile can be represented by the following matrix:

$$H = \left(\begin{array}{c} \vec{h}_0 \\ \vec{h}_1 \\ \vec{h}_2 \end{array}\right) = \begin{array}{ccc} t & i_0 & i_1 \\ \left(\begin{array}{ccc} -\frac{1}{w_0} & \frac{1}{w_0} & 0 \\ -\frac{1}{w_1} & 0 & \frac{1}{w_1} \\ 0 & 0 & \frac{1}{h} \end{array}\right) \end{array}$$

The resulting tiles are parallelepiped, as shown in Figure 4.7-(a). The top and bottom faces of the parallelepiped tiles are rectangles, while the other four faces are parallelograms. Figure 4.7-(b) shows the top-down view of tiles on the $< \vec{i}_0, \vec{i}_1 >$ plane. Since the slopes of side faces are determined by the right projections of the dependence vectors, each tile only depends on the neighbors above (this is actually the parallelepiped behind if we consider the

3D space) and on the right, as shown in Figure 4.7-(b).

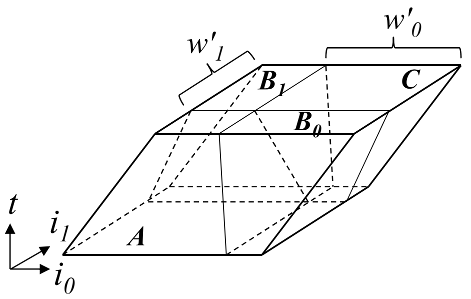


(a) Tile shape (b) Top-down view of tiles

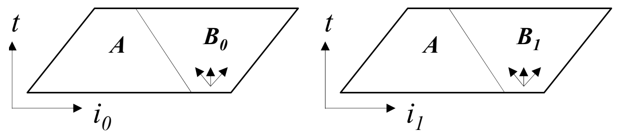
Figure 4.7: Tiling with 2-dimensional stencils. The thick arrows in (b) represent the dependence between tiles.

Next step is to do subtile the parallelepiped tiles, by applying the same subtiling strategy in Figure 4.5-(a), so that the slope of the boundary planes between tiles is determined by the left projection of the dependence vectors on the $< \vec{t}, \vec{i}_0 >$ and $< \vec{t}, \vec{i}_1 >$ planes. As a result of this tiling strategy, four subtiles are generated: A,B_0, B_1 and C, as shown in Figure 4.8-(a). The dependence relations among subtiles are: $A \rightarrow B_0$, $A \rightarrow B_1$, $B_0 \rightarrow C$, $B_1 \rightarrow C$ and $A \rightarrow C$. Subtile A and C are two pyramids of the same shape, but B_0 and B_1 are not pyramids. However, if $w'_0 = h + w_0/2$ and $w'_1 = h + w_1/2$, the projections of the subtiles on $< \vec{t}, \vec{i}_0 >$ and $< \vec{t}, \vec{i}_1 >$ planes are still conjugate trapezoids, as shown in Figure 4.8-(b).

Figure 4.9-(a) shows the slice of a tile for one time step, where all the tiles are rectangles. Because of the dependences among subtiles, the computations in subtile A must be done first; then B_0 and B_1 (the order of B_0 and B_1 is not important because there are no dependences between them); and finally, C. Based on the dependences among tiles shown in Figure 4.7-(b), each time step a tile only needs to send the data near the left and bottom boundaries (the shadowed area in the Figure) to the corresponding neighbor nodes. So only subtiles A, B_0 and B_1 send data, while subtile C, B_0 and B_1 receive data. There are five separate *send* operations: S_0 (from A to the lower-left neighbor), S_1 (from A to the left neighbor), S_2 (from A to the neighbor below), S_3 (from B_0 to the neighbor below), and S_4 (from B_1 to the



(a) Subtiling

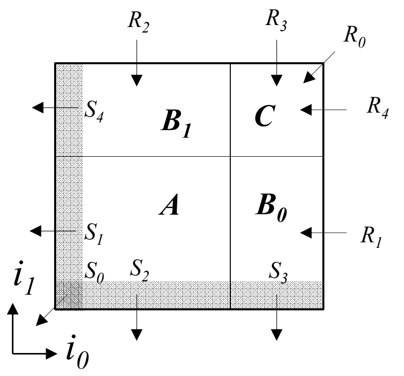


(b) Projections of subtiles

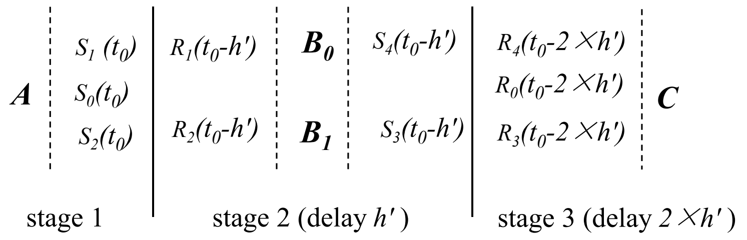
Figure 4.8: Subtiling with 2-dimensional stencils.

neighbor below). R_0, R_1, R_2, R_3 and R_4 are the corresponding *receive* operations. In total, each time step there are 14 computation and communication operations within a tile. Figure 4.9-(b) shows the order of these operations, as enforced by the dependences. Because each tile is assigned to a single node, we can schedule the above 14 operations in any sequential order that satisfies the dependencies in Figure 4.9-(b).

In order to hide communication latency, it is necessary to delay the computation associated with *receive* operations. The start of the computation of subtile B_0 and B_1 can be delayed by h' time steps from subtile A, so that the latency of transmitting the data from S_1 and S_2 to R_1 and R_2, respectively, is hidden. However, in subtile C, the R_3 and R_4 need to receive the data from S_3 and S_4 in tiles, B_0 and B_1, respectively. Hence execution of C should be delayed by h' time steps from subtile B_0 and B_1, or $2 \times h'$ time steps from subtile A. The projections of the subtiles after these delays are shown in Figure 4.10. The dashed trapezoids are the projections of the subtiles behind the others.



(a) Slice of a tile when $t = t_0$.



(b) Dependence among the operations within each time step.

Figure 4.9: Schedule of subtiles for each time step.$\forall j$, S_j and R_j are the corresponding *send* and *receive* pair.

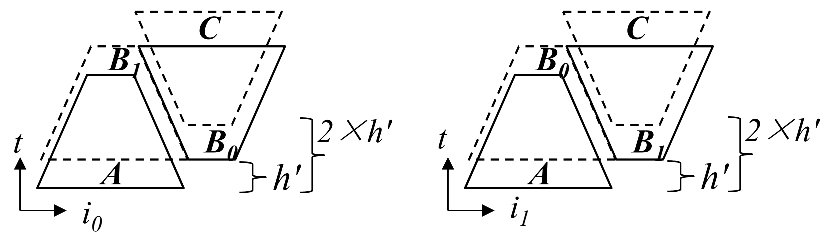


Figure 4.10: Projections of subtiles with 2-dimensional stencils.

4.2.4 Higher Dimensions

For ISLs with higher dimensional stencils, Conjugate-Trapezoid Tiling is more complicated. For example, the 3-dimensional Jacobi code is a 27-point stencil, where the iteration space is 4-dimensional and requires a 3×3 cube of elements to compute each output element.

Although we cannot draw the shape of 4-dimensional tiles, the approach discussed in Section 4.2.3 for a 3-dimensional iteration space can be used to apply tiling to the 4-dimensional iteration space. The 4-dimensional cone that contains all the dependence vectors is projected on to $< \vec{t}, \vec{i}_0 >$, $< \vec{t}, \vec{i}_1 >$ and $< \vec{t}, \vec{i}_2 >$ planes. Based on the projection of the dependence cone, we apply the same tiling strategy as in Figure 4.4 to produce parallelogram tiles on each 2-dimensional plane. The resulting tile is a 4-dimensional hyper-parallelepiped. Subtiles are computed as shown Figure 4.5-(a) on the projection of the hyper-parallelepiped tile on each plane, so that the projections of subtiles on each plane are conjugate trapezoids. The total number of subtiles within a tile is $2^3 = 8$. For a given time step, the slice of a hyper-parallelepiped tile is a cuboid, as shown in Figure 4.11, where subtile A *send* the boundary data to 7 neighbor nodes, and subtile D *receive* data from other 7 neighbor nodes. Thus, for each time step, the computation of subtile B_0, B_1 and B_2 depends on subtiles A, and subtiles C_0, C_1 and C_2 depend on subtiles B_0, B_1 and B_2, and subtile D depend subtiles C_0, C_1 and C_2. Hence, when delaying subtiles to hide communication latency, the subtiles are divided into four stages. The first stage includes subtile A, as well as the *send* and *receive* operations associated to A; the second stage, includes subtiles B_0, B_1 and B_2, which are delayed by h' time steps from subtile A; the third stage, includes subtiles C_0, C_1 and C_2, that are delayed by $2 \times h'$ time steps; and the fourth stage includes the subtile D, which is delayed $3 \times h'$ time steps.

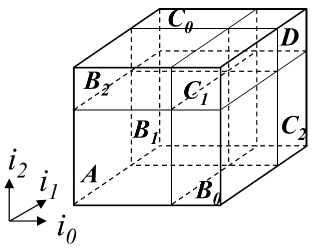


Figure 4.11: Slice of a 4-dimensional hyper-parallelepiped tile.

Based on the above discussion, the algorithm of Conjugate-Trapezoid Tiling for ISLs with

n-dimensional stencil (($n+1$)-dimensional iteration space) can be summarized as shown in Algorithm 1.

Algorithm 1 Conjugate-Trapezoid Tiling for ISL with d-dimensional stencil (($d+1$)-dimensional iteration space)

```
1: Compute $\vec{B}_{left}(\vec{t},\vec{i_j})$ and $\vec{B}_{right}(\vec{t},\vec{i_j})$ of the cone of dependence vectors for each plane $<\vec{t},\vec{i_j}>$ ($j$=0,1,...,$d-1$).
2: Compute $H = (\vec{h}_0^T, \vec{h}_1^T, ..., \vec{h}_{d-1}^T, \vec{h}_d^T)^T$. Each row of $H$ is the normal vector of a tiling hyperplane. The last row $\vec{h}_d = (1/h, 0, 0, ..., 0)$. Other rows $\vec{h}_j$ ($j = 0, 1, ..., d-1$) are computed as follows:
       $$Proj_{<\vec{t},\vec{i_j}>}(\vec{h}_j) = \frac{1}{w_j} \cdot \vec{B}_{right}(\vec{t},\vec{i_j}) \cdot \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$$
   The rest of the components in $\vec{h}_j$ are filled with zeros.
3: Tile the iteration space with the tiling hyperplanes defined by $H$. The height of each tile is $h$, and the width of each tile on $\vec{i_j}$ dimension is $w_j$. Distribute tiles on nodes organized in a $d$-dimensional mesh $<\vec{i}_0, \vec{i}_1, ..., \vec{i}_{d-1}>$.
4: Divide the projection of the tile on each $<\vec{t},\vec{i_j}>$ plane into two conjugate trapezoid subtiles. The boundary of between subtiles is parallel to $\vec{B}_{left}(\vec{t},\vec{i_j})$. This results in $2^d$ subtiles in each $(d+1)$-dimensional tile in total.
5: Partition the $2^d$ subtiles into $d+1$ stages according to the dependence among subtiles. Schedule the subtiles in order of stage. Delay the $k$-th stage by $k \times h'$ time steps ($k = 0, 1, .., d$).
6: In each subtile, receive necessary boundary data from neighbor nodes before computation, and send boundary data to neighbor nodes after computation.
```

4.3 Performance Model

This section introduces a performance model of Conjugate-Trapezoid Tiling which is used to the explain the experimental results.

4.3.1 Definitions

At each time step, the total number of stencil points computed by each node is:

$$N_{comp} = \prod_{j=0}^{d-1} w_j$$

N_{comp} stands for the volume of the d-dimensional tile slice at each time step. The order of N_{comp} is d. If the average computation time for each stencil point is t_{point}, the computation time at each time step would be:

$$t_{comp} = t_{point} \cdot N_{comp} = t_{point} \cdot \prod_{j=0}^{d-1} w_j$$

Only the data points at the boundaries of the tile need to be communicated (to be sent or received) with neighboring nodes. For 1-dimensional Jacobi, the total number of data points to be sent at each time step is 2. For 2-dimensional Jacobi, the grey area in Figure 4.9-(a) shows the data points to be sent. I assume that the messages sent to different destination nodes do not cause congestion with each other, so I only need to consider the largest message, which would cause the longest latency. The maximum number of data points to be sent at each time step is $2 \cdot max\{w_0, w_1\}$. In general, for a n-dimensional stencil, the maximum number of data points to be sent at each time step is:

$$V_{send} = c \cdot max\{\prod_{k=0}^{n-2} w_{j_k} | \forall \vec{j} = (j_0, j_1, ..., j_{n-2})\}$$

in which c is a constant determined by the stencil. For the Jacobi stencils of any dimension, $c = 2$, which means Jacobi stencil only requires each points to exchange data with its direct neighbor points. Intuitively, V_{send} is proportional to the size of one "surface" of the n-dimensional tile slice at each time step, so the order of V_{send} is $n - 1$. Because of symmetry, for any non-boundary node, the volume of data to be sent is equal to the volume of data to be received, so let $V_{send} = V_{recv} = V$.

For each *send-receive* pair, let $t_{latency}$ denote the latency. In practice $t_{latency}$ consists of two partss: the fixed startup overhead $t_{startup}$ and the transmission latency which is determined by the size of the data to be transmitted:

$$t_{latency} = t_{startup} + t_{trans} \cdot V$$

For small messages, the total latency is dominated by the startup overhead, so $t_{latency} \approx t_{startup}$. However, if the message is large, the total latency is proportional to the data size being transmitted:

$$t_{latency} \approx t_{trans} \cdot V \quad \sim \quad V$$

For each *send* or *receive* operation, even with asynchronous mode, there is some CPU time that cannot be overlapped with computations. Let t_{send} and t_{recv} denote the overhead of asynchronous *send* and *receive* operations, respectively. Similarly, because of symmetry, for any non-boundary node the total number of *send* operations must be the same to the total number *receive* operations at each time step. Let $N_{send} = N_{recv} = N_{msg}$. N_{msg} is determined by the number of neighbor points that needs to do data exchange, which is a constant for a given stencil. Because the total number of neighbor points for a n-dimensional stencil is $3^n - 1$, the upper bound of N_{msg} is $(3^n - 1)/2$. For the Jacobi stencil, the values of N_{msg} are 1, 3, 7 in the 1, 2, 3-dimensional cases, respectively.

4.3.2 Performance Model

If communication and computation are not overlapped, the total execution time at each time step is:

$$\begin{aligned}
t_{exec} &= t_{comp} + (t_{send} + t_{recv}) \cdot N_{msg} + t_{latency} \\
&= t_{point} \cdot N_{comp} + (t_{send} + t_{recv}) \cdot N_{msg} \\
&\quad + t_{startup} + t_{trans} \cdot V
\end{aligned}$$

Consider the Conjugate-Trapezoid Tiling proposed in this paper. Let the total number of points computed at each time step be N'_{comp}. At each time step in the steady state, $N'_{comp} \approx N_{comp}$. Because of subtiling, the size of the message sent by each subtile is smaller, so the maximum number of data points to be sent V' is smaller than V: $V' \leq V$. The cost is that the total number of *send* or *receive* operations, N'_{msg}, becomes larger: $N'_{msg} \geq N_{msg}$. For the Jacobi stencil, with subtiling the values of N'_{msg} are 1, 5, 19 in the 1, 2, 3-dimensional cases respectively. The number of messages to be sent for 2-dimensional Jacobi. is shown in Figure 4.9-(a). Figure 4.12 shows the number of messages to be sent for 3-dimensional Jacobi. For standard tiling, 7 messages need to be send to different neighboring tile (1 message for the data on the vertex, 3 messages for 3 edges and another 3 for the 3 faces), as shown in Figure 4.12-(a). After Conjugate-Trapezoid Tiling, some messages are divided into 2 (the original edge messages) or 4 messages (the original face messages) due to subtiling, as shown in Figure 4.12-(b). However, those message that are divided from the same original message are still to be sent to the same destination tile. The total number of messages becomes 19. The increase of the number of the messages may cause performance problems, which will be addressed in Section 4.3.3.

Because subtiles are delayed in Conjugate-Trapezoid Tiling, there are at least h' time steps between each *send* and *receive* pair. Let t'_{exec} denote the total execution time at each

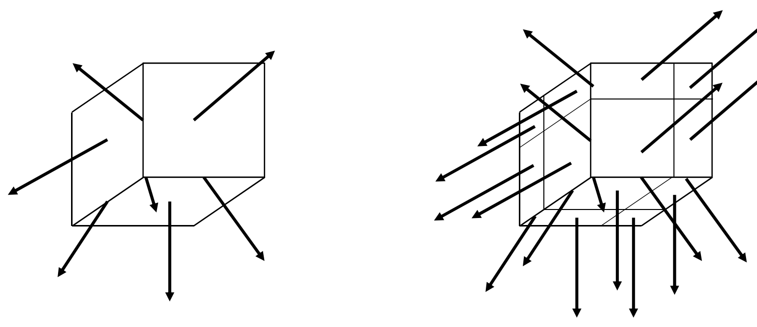


Figure 4.12: Number of messages to be sent for 3-dimensional Jacobi. Each arrow stands for a message to be sent to an neighboring tile. The arrows with the same direction means that the destination of the messages are the same.

time step with Conjugate-Trapezoid Tiling, and $t'_{latency} = t_{startup} + t_{trans} \cdot V'$ is the latency of the largest message. Then if $t'_{latency}$ is smaller than the total execution time of h' time steps,

$$h' \times t'_{exec} \geq t'_{latency} = t_{startup} + t_{trans} \cdot V'$$

the communication latency can be fully overlapped:

$$\begin{aligned} t'_{exec} &= t'_{comp} + (t_{send} + t_{recv}) \cdot N'_{msg} \\ &= t_{point} \cdot N'_{comp} + (t_{send} + t_{recv}) \cdot N'_{msg} \end{aligned} \tag{4.1}$$

According to above, the minimum number of delayed time step h' to fully hide communication latency is determined by:

$$h' \geq \frac{t'_{latency}}{t'_{exec}} = \frac{t_{startup} + t_{trans} \cdot V'}{t_{point} \cdot N'_{comp} + (t_{send} + t_{recv}) \cdot N'_{msg}} \tag{4.2}$$

The performance benefit achieved by Conjugate-Trapezoid Tiling is:

$$\begin{aligned}\Delta &= t_{exec} - t'_{exec} \\ &= t_{point} \cdot (N_{comp} - N'_{comp}) \\ &\quad +(t_{send} + t_{recv}) \cdot (N_{msg} - N'_{msg}) + t_{latency} \\ &\approx t_{latency} - (t_{send} + t_{recv}) \cdot (N'_{msg} - N_{msg}) \end{aligned} \tag{4.3}$$

4.3.3 Optimization

Communication-Coalesce Optimization

With Conjugate-Trapezoid Tiling, the number of *send* and *receive* operations at each time step, N'_{msg}, is greater than N_{msg}, which can cause performance degradation according to Equation 4.3. $N'_{msg} \geq N_{msg}$ is because the data to be exchanged with the same neighbor node are divided into two or more messages by the delayed subtiles. For example, in Figure 4.9, the destination of message S_1 and S_4 is the same node. However, after schedule of subtiles, within the same time step t_0, S_1 is provided by subtile A with the data stands for the original time step of t_0, but S_4 is provided by subtile B_0 with the data stands for the original time step of $t_0 - h'$. This leads to S_1 and S_4 becoming two separate sending operations, and so as S_2 and S_3. If postpone S_1 until the computation in subtile B_0 completes, the send operations of S_1 and S_4 can be merged into one because the destination node is the same. S_2 and S_3 can also be merged by postponing S_2 until subtile B_1 is complete. Similarly, for receive operations if bring R_4 forward before subtile B_0 starts, R_1 and R_4 can be merged because they have the same source node, and so as R_2 and R_3.

Figure 4.13 shows how communication operations are merged. Compared to Figure 4.9-(b), S_1 and S_2 are postponed while R_3 and R_4 are moved into an earlier stage. This optimization of merging communication operations is always possible, because send opera-

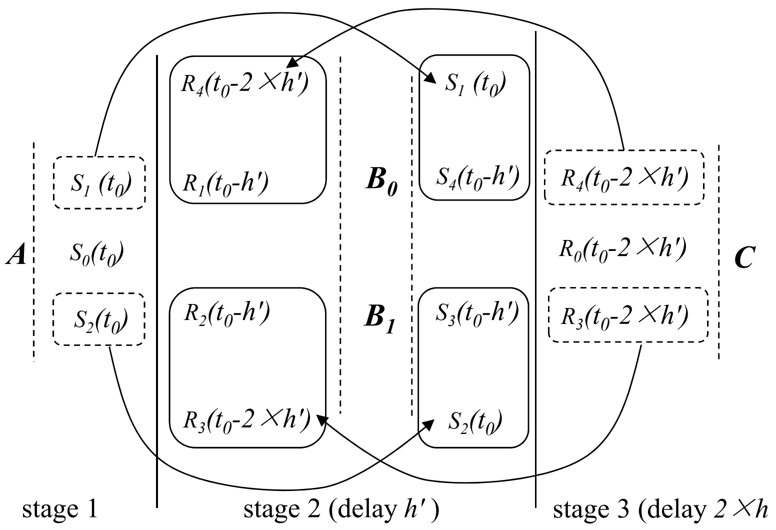


Figure 4.13: Merge communication operations. The *send* or *receive* operations with in the same dashed box can be merged. Compared to Figure 4.9-(b), the *send* or *receive* operations denoted by the dashed arrows are moved (delayed or brought forward) to the operations denoted by corresponding thick arrows.

tions only need to be postponed, and receive operations only need to be moved to an earlier stage. This optimization still preserve the correctness of communication. Although S_1 and R_4 are the corresponding send and receive operations, and S_1 and R_4 are moved into the same stage as shown in Figure 4.13, there are still h' time steps of execution between the corresponding send and receive operations in the pipeline (when S_1 sends out the data for time step t_0, R_4 is receiving the data for time step $t_0 - h'$). Hence this optimization does not affect the threshold of h' calculated in Equation 4.2.

Because the send or receive operations with the same destination or source node are merged into one operation, this optimization results in that $N'_{msg} = N_{msg}$ and $V' \approx V$ on the steady state of the pipeline.

Balanced Multi-Threading

If nodes have more than one processors, the computation can be evenly distributed to every processor because the stencil points at each time step are fully parallelizable. However, the overhead of asynchronous *send* or *receive* operations can not be reduced through parallelization. This optimization is to balance the overall load of computation and communication operations. As shown in Figure 4.14, the white bars stand for computation time, and the grey bars stand for communication operation overhead. When nodes work in sequential mode, the total execution time of each time step is $t'_{exec} = t'_{comp} + (t_{send} + t_{recv}) \cdot N'_{msg}$. If nodes work in parallel mode, assume each node has 4 processors, the computations can be divided into partitions and distributed on to each processor. However, the overhead of communication operation cannot be divided. As a result, instead of evenly dividing the computations in into 4 partitions, the partition(s) which produces data for sending or depends on the data to be received are designed to contain less computations. So the overall load distribution is balanced.

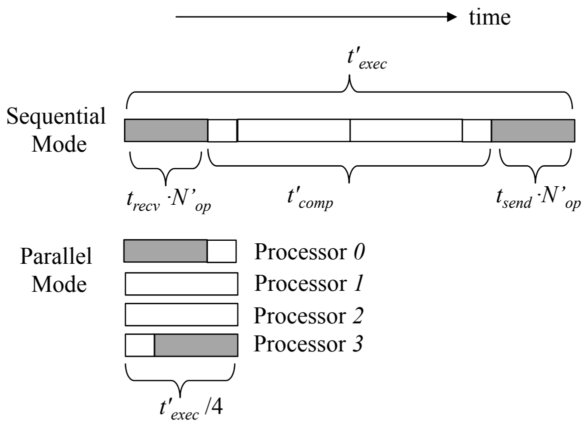


Figure 4.14: Balanced multi-threading with four processors. The white bars stand for computation time, and the grey bars stand for communication operation overhead.

In general, if each node has P processors, and $max\{t_{send}, t_{recv}\} \leq t'_{exec}/P$, ideally the execution time in parallel mode can be t'_{exec}/P if work load among threads are well balanced.

This is equivalent to that the amortized overhead of each communication operation is also reduced by parallelization, even though that the communication operation it self is not parallelized.

4.4 Unified Tiling Representation

This section uses the unified tiling representation framework defined in Section 2.1 to describe the Conjugate-Trapezoid Tiling discussed in this Chapter.

The application of Conjugate-Trapezoid Tiling focuses on regular ISLs with the format shown in Figure 4.2. For ISLs with n-dimensional stencils, the iteration space I is an $(n+1)$-dimensional hyper-cuboid, so the basis matrix of I is a diagonal matrix:

$$I = span(E) = span(\begin{pmatrix} T, 0, 0, ..., 0 \\ 0, N_0, 0, ..., 0 \\ 0, 0, N_1, ..., 0 \\ ... \\ 0, 0, 0, ..., N_{n-1} \end{pmatrix}).$$

The first dimension of I is the time dimension, consisting of the time steps of the ISL. The rest of the n dimensions are the space dimensions.

The stencil of an ISL may contain m points, which means that the computation of each data points depends on m neighboring data points from the previous time step. This results in m dependence vectors $\vec{d_0}$, $\vec{d_1}$, ..., $\vec{d}_{m-1}$ in the iteration space I. Because each time step only needs the results from the previous time step, the first component of each dependence vector must be 1. So the dependence matrix D must have the following form:

$$D = \begin{pmatrix} \vec{d_0} \\ \vec{d_1} \\ ... \\ \vec{d}_{m-1} \end{pmatrix} = \begin{pmatrix} 1, * \\ 1, * \\ ... \\ 1, * \end{pmatrix}.$$

Because the tile shape $\mathbb{T}$ is an $(n+1)$-dimensional parallelepiped, there must exist a tiling matrix T:

$$T = \begin{pmatrix} \vec{t_0} \\ \vec{t_1} \\ ... \\ \vec{t_n} \end{pmatrix},$$

such that $\mathbb{T} = span(T)$. Since the projection of $\mathbb{T}$ on $< \vec{i}_0, \vec{i}_1, ..., \vec{i}_{n-1} >$ is an n-dimensional hyper-rectangle, 1 to n rows of T must be:

$$\begin{aligned} \vec{t_1} &= (w_0, 0, 0, ..., 0), \\ \vec{t_2} &= (0, w_1, 0, ..., 0), \\ &... \\ \vec{t_n} &= (0, 0, 0, ..., w_{n-1}). \end{aligned}$$

For the first row of T, $\vec{t_0}$ is determined by the height of the parallelepiped tile h and the dependence vectors. More specifically, the projections on $\vec{t_0}$ on each $< \vec{t_0}, \vec{i_j} >$ is parallel to $\vec{B}_{right}(\vec{t}, \vec{i_j})$:

$$Proj_{<\vec{t}, \vec{i_j}>}(\vec{t_0}) = c_j \cdot \vec{B}_{right}(\vec{t}, \vec{i_j}).$$

And because the height of the parallelepiped tile is h, the first component of $Proj_{<\vec{t},\vec{i_j}>}(\vec{t_0})$ must be h:

$$Proj_{<\vec{t},\vec{i_j}>}(\vec{t_0}) = (h, x_j) = c_j \cdot \vec{B}_{right}(\vec{t}, \vec{i_j})$$

According to above equations:

$$\begin{aligned} x_j &= \frac{\vec{B}_{right}(\vec{t}, \vec{i_j}) \cdot (0,1)^T}{\vec{B}_{right}(\vec{t}, \vec{i_j}) \cdot (1,0)^T} \cdot h, \qquad j = 0, 1, ..., n-1 \\ \vec{t_0} &= (h, x_0, x_1, ..., x_j, ..., x_{n-1}). \end{aligned}$$

Since Conjugate-Trapezoid Tiling is tessellating tiling, the repetition matrix $R = T$. After tiling, there still exists "horizontal" dependences (dependence vectors that are perpendicular to the dimension of $\vec{t}$). The dependence matrix between tiles are as follows:

$$D' = \begin{pmatrix} 1, * \\ 1, * \\ ... \\ 1, * \\ 0, * \\ 0, * \\ ... \\ 0, * \end{pmatrix}.$$

The rows in D' with the first component 0 are the dependence vectors perpendicular to the $\vec{t}$ dimension. Because of these "horizontal" dependences, not enough parallelism can be exposed among tiles if each tile are executed atomically (as shown in Figure 4.4). However, after subtiling and delaying dependent subtiles, tiles within the same level can be executed in parallel. As a result, the following schedule of tiles become valid:

$$\begin{aligned} S &= \begin{pmatrix} 1 \\ 0 \\ ... \\ 0 \end{pmatrix}, \\ \mathbb{S}(\vec{i'}) &= \vec{i'} \cdot S = (i'_0, i'_1, ..., i'_n) \cdot S = i'_0. \end{aligned}$$

Similar to overlapped tiling, the purpose of the technique of subtiling and delaying dependent subtiles is to make the above schedule $\mathbb{S}$ valid and expose enough parallelism for coarse grain tiles.

To sum up, Table 4.1 shows the unified representation of Conjugate-Trapezoid Tiling.

4.5 Evaluation

In this section, Conjugate-Trapezoid Tiling is evaluated on a distributed memory machine and the experimental results are presented.

4.5.1 Target Platform

The measurement of the performance is done with a distributed memory cluster of 32 nodes. Each node has an Intel Xeon quad-core X3430 CPU at 2.4GHz. The nodes are interconnected with a gigabit Ethernet. The MPI environment is MPICH2. The compiler is GCC 4.7.2, with the option `-O3`.

Input	$I, D = \begin{pmatrix} 1,* \\ 1,* \\ ... \\ 1,* \end{pmatrix}$
D'	$D' = \begin{pmatrix} 1,* \\ 1,* \\ ... \\ 1,* \\ 0,* \\ 0,* \\ ... \\ 0,* \end{pmatrix}$, after subtiling: $D' = \begin{pmatrix} 1,* \\ 1,* \\ ... \\ 1,* \end{pmatrix}$.
$\mathbb{T}$	$T = \begin{pmatrix} \vec{t}_0 \\ \vec{t}_1 \\ ... \\ \vec{t}_n \end{pmatrix} = \begin{pmatrix} h, x_0, x_1, ..., x_j, ..., x_{n-1} \\ w_0, 0, 0, ..., 0 \\ 0, w_1, 0, ..., 0 \\ ... \\ 0, 0, 0, ..., w_{n-1} \end{pmatrix}$ $x_j = \frac{\vec{B}_{right}(\vec{t}, \vec{i}_j) \cdot (0,1)^T}{\vec{B}_{right}(\vec{t}, \vec{i}_j) \cdot (1,0)^T} \cdot h, \quad j = 0, 1, ..., n-1$
$\mathbb{R}$	$R = T.$
$\mathbb{S}$	$S = \begin{pmatrix} 1 \\ 0 \\ ... \\ 0 \end{pmatrix}.$

Table 4.1: Representation of Conjugate-Trapezoid Tiling

	Stencil Dimension	**Input Size**	**Points of Stencil**
1-D Jacobi	1	512K	3
2-D Jacobi	2	1024 x 512	9
3-D Jacobi	3	64 x 64 x 64	27
PathFinder	1	512K	3
Poisson	2	1024 x 512	5
Biharmonic	2	1024 x 512	13
HotSpot	2	1024 x 512	9
Cell	3	64 x 64 x 64	27

Table 4.2: Benchmarks

4.5.2 Implementation

The Conjugate-Trapezoid tiling discussed in this chapter is implemented in the Cetus compiler [21], which is a source-to-source C compiler. The inputs to the compiler are the sequential C codes of the ISLs. The dependence analysis result of Cetus provides the stencils definitions of the ISLs, which is the input to the tiling algorithm. The implementation use the Omega library [22] to perform tiling operations on the iteration space polyhedron. The code generation of Cetus is modified to generate parallel code with MPI APIs. In order to implement the optimization discussed in Section 4.3.3, the computation within each node is parallelized with Pthreads instead of OpenMP.

4.5.3 Benchmark

The same 8 ISL programs are evaluated as benchmarks as in Chapter 3: 1,2,3-dimensional Jacobi, PathFinder, Poisson, Biharmonic, HotSpot and Cell. These benchmarks contains 5 different stencils, from 1-dimensional to 3 dimensional. The input size for each benchmarks is listed in Table 4.2. Table 4.3 shows different stencils and the corresponding dependence vectors.

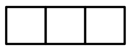
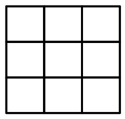
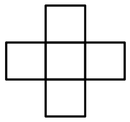
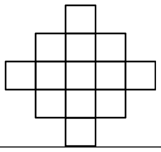
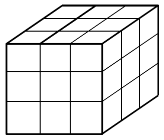
Stencil	Dependence Vectors	Benchmark
	$(1,-1),(1,0),(1,1)$	1-D Jacobi, PathFinder
	$(1,0,0)$, $(1,\pm1,\pm1)$, $(1,0,\pm1)$, $(1,\pm1,0)$	2-D Jacobi, HotSpot
	$(1,0,0)$, $(1,0,\pm1)$, $(1,\pm1,0)$	Poisson
	$(1,0,0)$, $(1,\pm1,\pm1)$, $(1,0,\pm1),(1,\pm1,0)$, $(1,0,\pm2),(1,\pm2,0)$	Biharmonic
	$\forall x=-1,0,1,\ \forall y=-1,0,1,$ $\forall z=-1,0,1, \quad (1,x,y,z)$	3-D Jacobi, CELL

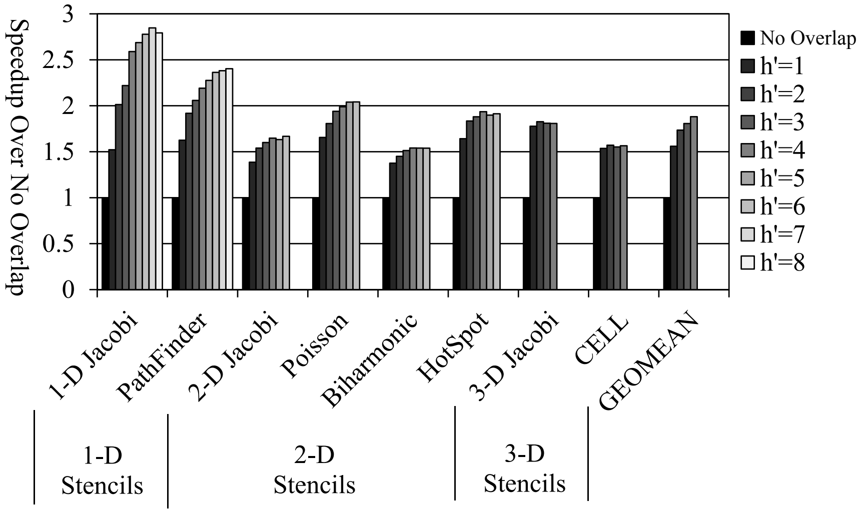
Table 4.3: Different stencils of the benchmarks

4.5.4 Experimental Result

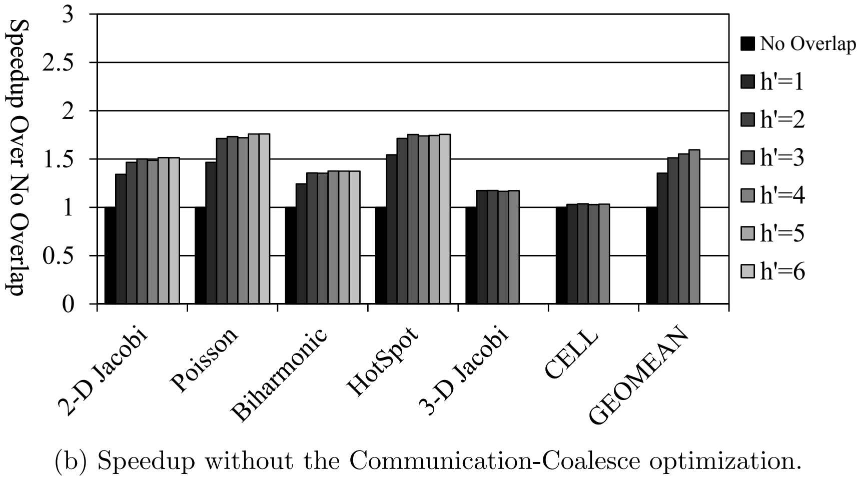
Performance Overview

Figure 4.15-(a) shows the speedup achieved by Conjugate-Trapezoid Tiling with different values of h'. The baseline is the performance achieved by tiling with no communication overlap. On average, Conjugate-Trapezoid Tiling achieves 1.9X speedup when $h' = 4$.

Among all the benchmarks, the common trend is that, as the value of h' becomes larger than 4, higher performance is achieved. If communication latency is longer than the computation time within a tile, increasing h' will delay the dependent subtiles, which are the receivers of messages, so that larger portion of communication latency can be hidden. This trend exactly shows the motivation of Conjugate-Trapezoid Tiling. However, It can also be seen that after exceeding a certain threshold, there is very little or no performance gain by keeping increase h'. This is because if h' is already large enough to hide the entire latency,



(a) Speedup with the Communication-Coalesce optimization.



(b) Speedup without the Communication-Coalesce optimization.

Figure 4.15: Speedup with different values of h'. The baseline is the performance achieved by tiling with no communication overlap. The figure shows the performance with h' up to 6 for 2-dimensional stencils and h' up to 4 for 3-dimensional stencils, because the trend shows that there would be no additional performance gains with a larger h'.

there will be no extra benefit with an even larger h'. Equation 4.2 in Section 4.3.2 discussed the threshold of h'. In Figure 4.15, for the benchmarks with 1-dimensional stencils (1-D Jacobi and PathFinder), the threshold is 6 or 7, while for the benchmarks with 2-dimensional stencils (1-D Jacobi, Poisson, Biharmonic and HotSpot) the threshold is 3 or 4. What is more, for 3-D Jacobi and Cell, which are the benchmarks with 3-dimensional stencils, peak performance is achieved when $h' = 1$. This can be explained by Equation 4.2 as follows. The input sizes for each benchmarks shown in Table 4.2 are chosen to make the number of stencil points allocated to each node be the same across all benchmarks. So N_{comp} is the same each benchmarks, and so is N'_{comp} because $N_{comp} \approx N'_{comp}$. However the amount of computation of higher dimensional stencils is usually larger than that of lower dimensional stencils, such as 1-D Jacobi (3 points) versus 2-D Jacobi (9 points). For higher dimensional stencils the amount of computation of each stencil point is larger, and more time consuming, too. This means that t_{point} is larger for higher dimensional stencils, which contributes one factor that leads to a smaller threshold of h' according to Equation 4.2. The other factor is that for higher dimensional stencils each tile has more neighboring tiles, so N_{msg} and N'_{msg} are also larger. The above two factors makes a larger denominator in Equation 4.2, and a smaller threshold of h' as a result.

Communication-Coalesce Optimization

Figure 4.15-(b) shows the speedup achieved without the Communication-Coalesce optimization introduced in Section 4.3.3. Since for 1-dimensional stencils $N_{msg} = N'_{msg} = 1$, there is no difference with or without the Communication-Coalesce optimization. So 1-D Jacobi and PathFinder are not included in Figure 4.15-(b). Compared to Figure 4.15-(a), it can be seen that 2-dimensional stencils benchmarks (2-D Jacobi, Poisson, Biharmonic and HotSpot) achieve a lower speedup without the Communication-Coalesce optimization. Another observation is that the peak performance is reached with a smaller h': without the optimization

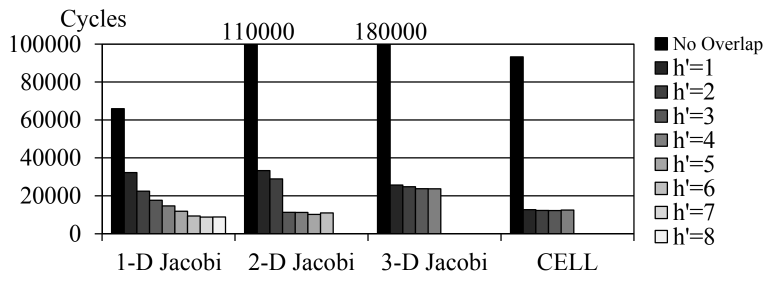


Figure 4.16: Communication overhead with different h'.

the peak performance is reached around $h' = 3$ while in Figure 4.15-(a) the peak performance is reached around $h' = 4$ for most 2-dimensional stencils benchmarks. Because without the optimization, the additional overhead introduced by $(N'_{msg} - N_{msg})$ more send and receive operations, the execution time at each time step becomes longer. So fewer time steps are needed to overlap with communication latency. This problem is more serious for benchmarks with 3-dimensional stencils, since for those benchmarks, without the optimization $N'_{msg} = 19$, which is much larger compared to $N_{msg} = 7$. As shown in Figure 4.15-(b), 3-D Jacobi and CELL can only achieve very small speedup with any h'. The overhead of additional communication operations almost kill the performance benefit gained by overlapping communication and computation.

Communication overhead

Figure 4.16 shows the communication overhead with different h'. The overhead is measured by the average number of cycles taken by each blocking receive call `MPI_Recv`. So this overhead shown in Figure 4.16 stands for the t_{recv} plus the part of $t_{latency}$ that is not overlapped with computation. The bar of "No Overlap" corresponds to $t_{recv} + t_{latency}$, which is the upper bound of the overhead shown in the figure. It can be seen that from 1-D Jacobi to 3-D Jacobi, as the dimension of the stencil increases, the value of $t_{recv} + t_{latency}$ also increases. This is because according to the performance model defined in Section 4.3.2, if assuming that t_{recv} and $t_{startup}$ are constant, the other part of $t_{latency}$ is proportional to the number of data points to be communicated (V, which is the volume of the data to be sent or received), and the order of V is $d-1$. This means that because higher dimensional stencils have larger communication volume, the communication overhead, if not hidden by overlapping with computations, is higher. In Figure 4.16 the "No Overlap" bar of CELL is lower than 3-D Jacobi. CELL and 3-D Jacobi share the same stencil, which means the number of data points to be communicated is the same. However, each data point of CELL only takes one byte storage, while each data point of 3-D Jacobi is a floating point value. So the actual communication volume V of CELL is smaller than that of 3-D Jacobi.

As the value of h' increases, the communication overhead decreases for every benchmark shown in Figure 4.16, because more portion of communication overhead is hidden by overlapping it with computations. This can explain the trend of performance shown in Figure 4.15. When h' exceeds some threshold, communication overhead does not reduce any more because $t_{latency}$ is fully hidden and the lower bound t_{recv} is reached. As in Figure 4.15, benchmarks with higher dimensional stencils reach the lower bound sooner with a smaller h' compared to benchmarks with lower dimensional stencils.

Input Size Sensitivity

Figure 4.17 shows the speedup with different input size for 1-D and 2-D Jacobi. For both benchmarks, h' reaches the threshold with a smaller value with larger input size. This is because larger input size results in larger N_{msg} and N'_{msg}. This makes the denominator in Equation 4.2 larger and thus a smaller threshold of h'. It also shows than for both benchmarks as the input size getting larger, the peak speedup achieved by Conjugate-Trapezoid Tiling is lower. This is because a larger input size leads to larger tile size if the total number of computation nodes remains the same. When the large tile size is large enough the program becomes CPU-bound instead of communication-bound. The optimization opportunity for

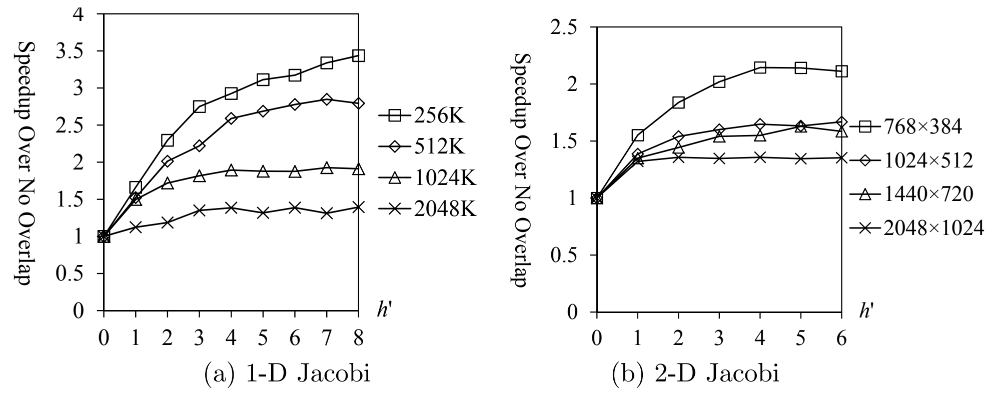


Figure 4.17: Input size sensitivity.

hiding communication latency for a cpu-bound program is small.

4.6 Conclusion

This chapter discusses the design and implementation of a new tiling scheme called *Conjugate-Trapezoid Tiling* for ISLs. The proposed tiling scheme divides each tile into subtiles, and schedules subtiles within each tile to overlap computation and communication. This chapter describes the tiling algorithm in polyhedron model, and compare Conjugate-Trapezoid Tiling and standard tiling analytically. The proposed tiling scheme is implemented in Cetus compiler to automatically generated code for d-dimensional stencils. Experimental results on a cluster show that Conjugate-Trapezoid Tiling is able to effectively reduce the overhead of communication latency, and achieves significant speedup over traditional tiling strategy.gy.

Chapter 5

Tile Shape Selection for Hierarchical Tiling

In this chapter, the tile shape selection for hierarchical tiling is studied. In hierarchical tiling, tile shapes at different levels interact with each other and together determine execution time. This means that, in order to minimize execution time, simply selecting tile shape for each level separately will not necessarily lead to the optimal choice; optimal tile shape selection can only be achieved by tackling hierarchical tiling in a global manner. Figure 5.1 shows an example of different tile shape choices. Arrows represent the dependences between tiles. Dashed lines show the tiles that can be executed in parallel in a wavefront schedule. The numbers stand for the order of the execution the wavefronts. Suppose the iteration space of the loop nest has two dependences: $\vec{d_0} = (1, 0)$ and $\vec{d_1} = (0, 1)$. There are different possible tile shapes for each level of tiling. The most common choice is the square. In Figure 5.1-(a), both levels use square tiles. The tiles along the same diagonal line can be executed in parallel. Assume that the computation time of each level 0 tile is one unit time, the total execution time of schedule of in Figure 5.1-(a) is $7 \times 7 = 49$ units of time. There are, however, better tile shapes. If the level 1 tile shape is changed to a parallelogram as shown Figure

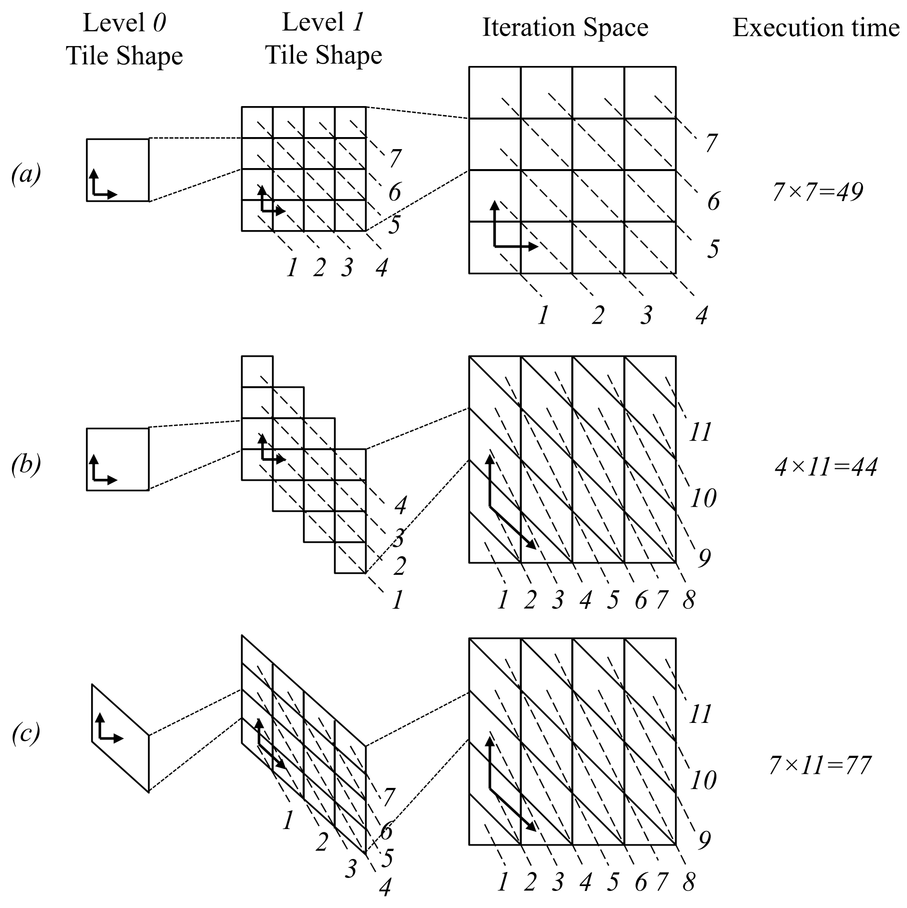


Figure 5.1: Different tile shape choices for hierarchical tiling. Arrows are the dependences between tiles. Dashed lines represent the tiles that can be executed in parallel in a wavefront schedule. The numbers stand for the order of the wavefronts

5.1-(b), the total number of time unit of execution time becomes $4 \times 11 = 44 < 49$. However, parallelogram shaped tile is not always the best choice. If we also choose parallelogram tile for the level 0 tiling, as shown in Figure 5.1-(c), the total execution time would increase to $7 \times 11 = 77$ units of time (here it is assumed that the parallelogram tile at level 0 takes the same time to execute as the square tile.). A quantitative model can be used to find the optimal tile shape choice combination for each level of tiling to achieve minimal execution time.

5.1 Problem Definition

Given the iteration space I with the shape of n-dimensional hyper-parallelepiped, assume $\vec{e_k} = (e_{k,0}, e_{k,1}, ..., e_{k,n-1})$, $k = 0, 1, ..., n-1$ are the n edge vectors of the hyper-parallelepiped, then the following matrix E is the basis matrix of I:

$$E = \begin{pmatrix} \vec{e_0} \\ \vec{e_1} \\ ... \\ \vec{e}_{n-1} \end{pmatrix}.$$

And $I = span(E)$.

Assume there are m dependences in I. The m dependence vectors $\vec{d_0}, \vec{d_1}, ..., \vec{d}_{m-1}$, forms an $m \times n$ dependence matrix D as follows:

$$D = \begin{pmatrix} \vec{d_0} \\ \vec{d_1} \\ ... \\ \vec{d}_{m-1} \end{pmatrix}.$$

Without loss of generality, I assume that $m \geq n$ and there are n dependence vectors that are linearly independent [1].

Then select a tile shape $\mathbb{T}$ with n-dimensional hyper-parallelepiped shape, and $\vec{t_k} = (t_{k,0}, t_{k,1}, ..., t_{k,n-1})$, $k = 0, 1, ..., n-1$ are the n edges of the hyper-parallelepiped tile, so the

[1] Otherwise, it must be possible to make at least one loop full permutable through a sequence of affine transformations, and then it can be simplified into a lower dimensional iteration space.

following matrix T is the tiling matrix (the basis matrix of space $\mathbb{T}$):

$$T = \begin{pmatrix} \vec{t_0} \\ \vec{t_1} \\ ... \\ \vec{t}_{n-1} \end{pmatrix}.$$

As discussed in previous chapters, if using tile shape $\mathbb{T}$(T) to tile the iteration space I, the tiled space I' is another n-dimensional iteration space. It is possible to apply tiling transformation for I' again. To describe an l-level hierarchical tiling, define a sequence of iteration spaces I_0, I_1,..., I_l:

$$I_0 = I, \quad I_1 = I'$$

and $\forall k = 1, 2, ..., l$, I_k is the tiled space of I_{k-1}. This follows the bottom-up approach of hierarchical tiling. Since each I_k must have the shape of n-dimensional parallelepiped, define E_k is the basis matrix for I_k:

$$\forall k = 0, 1, ..., l, \quad I_k = span(E_k).$$

At each level of tiling, the tile shape is $\mathbb{T}_k$. In order to simplify the problem, tile shapes are restricted to n-dimensional parallelepiped, too. Let T_k be the tiling matrix at each level,

$$\forall k = 0, 1, ..., l-1, \quad \mathbb{T}_k = span(T_k).$$

The tiles at different levels are scheduled independently. And each tile is executed atomically, which means there is no communication with other tiles at the same level before computation within it is completed.

With the definitions above, the problem is how to select tile shapes T_0, T_1, ..., T_{l-1} for

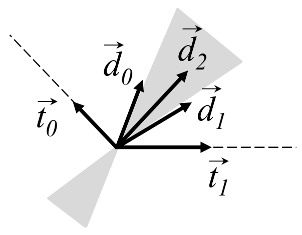


Figure 5.2: Constraints of tile shape imposed by dependences.

each level of tiling, so that with any valid scheduling of tiles on each level, the total execution time of the final loop nest is minimized.

5.2 Constraints on Tile Shape Selection

In the discussion of this chapter, tile shapes are restricted with hyper-parallelepiped. And the tiles on each tiling level are scheduled atomically. Besides, tiles are tessellating, which means there is no overlapped between tiles introduced to eliminate dependences. With the assumptions above, Figure 5.2 describes the constraints of tile shape selection imposed by dependences in general. In order to produce a valid tile shape, the infinite cone spanned by extending the tile edges $\vec{t}_0$, $\vec{t}_1$, ..., $\vec{t}_{n-1}$ must contain every dependence vector. For the example in Figure 5.2, $\vec{t}_0$ and $\vec{t}_1$ must not be in the shaded area. To describe the above observation formally, each dependence vector $\vec{d}_k$ must be covered by the cone spanned by the extension cords of $\vec{t}_0$, $\vec{t}_1$, ..., $\vec{t}_{n-1}$,

$$\begin{aligned} &\forall \vec{d}_k, \quad \exists \vec{a} = (a_0, a_1, ..., a_{n-1}) \geq \vec{0}_n = (0, 0, ..., 0) \\ &\vec{d}_k = a_0 \cdot \vec{t}_0 + a_1 \cdot \vec{t}_1 + ... + a_{n-1} \cdot \vec{t}_{n-1} = \vec{a} \cdot T \end{aligned} \tag{5.1}$$

Assume that every tile is always large enough so that inter-tile dependences only exist between adjacent tiles (including diagonal adjacent). This requirement can be expressed as follows:

$$\forall \vec{d_k}, \quad \exists \vec{a} = (a_0, a_1, ..., a_{n-1}) \leq \vec{1}_n = (1, 1, ..., 1) \tag{5.2}$$
$$\vec{d_k} = a_0 \cdot \vec{t_0} + a_1 \cdot \vec{t_1} + ... + a_{n-1} \cdot \vec{t}_{n-1} = \vec{a} \cdot T$$

The above two requirements can be merged into the following:

$$\forall \vec{d_k}, \quad \exists \vec{a}, \quad \vec{0} \leq \vec{a} \leq \vec{1}, \quad \vec{d_k} = \vec{a} \cdot T \tag{5.3}$$

which is the constraint imposed by dependences when choosing the tile shape.

When the original iteration space I is tiled by matrix T and produces the tiled space I', The relation among I, I' and T is studied as follows. First, assume the tiling transformation is an affine transformation, the shape of I' should still be an n-dimensional hyper-parallelepiped under any affine transformation. Suppose E' is the basis matrix of I' and $I' = span(E')$, which is the space of tiles. There must be some $n \times n$ matrix A such that:

$$E' = E \cdot A$$

Next, consider the effect of tiling transformation represented by A. As shown in Figure 2.4, after tiling, under the new coordinates system shown as axis labels i'_0 and i'_1, vector $\vec{t_0}$ and $\vec{t_1}$ become $(1, 0)$ and $(0, 1)$, respectively. More generally, we have the following equation ($\mathbb{1}_n$ denotes the $n \times n$ identity matrix):

$$T \cdot A = \begin{pmatrix} \vec{t_0} \\ \vec{t_1} \\ ... \\ \vec{t}_{n-1} \end{pmatrix} \cdot A = \begin{pmatrix} 1, 0, ..., 0 \\ 0, 1, ..., 0 \\ ... \\ 0, 0, ..., 1 \end{pmatrix} = \mathbb{1}_n$$

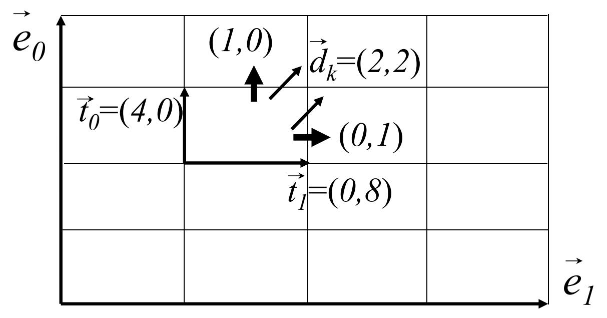


Figure 5.3: Inter-tile dependences (thicker arraows) generated by original dependence vector $\vec{d}_k$.

It can be concluded that $A = T^{-1}$. So the relation among I, I' and T is: $E' = E \cdot T^{-1}$. So T^{-1} represents the affine transformation of the tiling transformation with tile shape T.

When the original iteration space I is tiled by matrix T, each dependence vector $\vec{d}_k$ is also transformed to $\vec{d'}_k$ by the affine transformation represented by T^{-1}. T must satisfy the constraints of Equation (5.3),

$$\vec{d'}_k = \vec{d}_k \cdot T^{-1} = \vec{a} \cdot T \cdot T^{-1} = \vec{a}, \qquad \vec{0}_n \le \vec{a} \le \vec{1}_n \tag{5.4}$$

As shown in Figure 5.3, a dependence vector $(2,2)$ would result in three dependence vectors after tiling: $(1,0)$, $(0,1)$, and $(1,1)$. In general, a single dependence vector $\vec{d}_k$ in original iteration space I generates one or more dependences between tiles in the tiled iteration space I'. The rule is that, for $\vec{d'}_k = \vec{a} = (a_0, a_1, ..., a_{n-1})$, if $a_j > 0$, there is a dependence $(0, ..., 1, ..., 0)$ imposed on the j-th dimension in I'. In addition, if there are n dependence vectors $\vec{d}_{k_0}$, $\vec{d}_{k_1}$, ..., $\vec{d}_{k_{n-1}}$ which are linearly independent, the inter-tile dependences in I' generated by those n dependence vectors must include n orthonormal unit vectors.

Considering the above observation in the context of hierarchical tiling defined in last section, if at the k-th level of tiling, the dependence matrix in iteration space I_k is D_k, D_k

must have the following form:

$$
\begin{aligned}
&D_0 = D, \\
&D_k = \begin{pmatrix} \vec{d}_{k,0} \\ \vec{d}_{k,1} \\ \ldots \\ \vec{d}_{k,m-1} \end{pmatrix} = \begin{pmatrix} 1, 0, \ldots, 0 \\ 0, 1, \ldots, 0 \\ \ldots \\ 0, 0, \ldots, 1 \\ \ldots \end{pmatrix} = \begin{pmatrix} \mathbb{1}_n \\ * \end{pmatrix}, \\
&\forall \vec{d}_{k,j}, \quad k \geq 1, 0 \leq j < m, \qquad \vec{0} \leq \vec{d}_{k,j} \leq \vec{1}.
\end{aligned} \tag{5.5}
$$

5.3 Execution Model

Assume that the sequential execution time only depends on amount of computation, and that the execution time of each iteration is always the same. So, if the computations of iteration space $I = span(E)$ is not parallelized, the total execution time is proportional to the number of iterations within it:

$$Time_s(I) = |I| = |det(E)|. \tag{5.6}$$

The execution time of a parallelized iteration space depends on the schedule of iterations within the iteration space. In order not to be tied to any specific scheduling scheme, the concept of *ideal execution time* is used in analysis. Ideal execution time is the minimal execution time of an iteration space I that can be achieved by any valid schedule of iterations. If each iteration in I is viewed as an activity and D represents the dependences between activities, the ideal execution time is determined by the critical path in I. It is assumed that there is infinite hardware parallelism available at each level. In practice, the above assumption means that the amount of hardware parallelism should be always larger than the maximum number of iterations that can be executed in parallel. Under these assumptions, the critical path in I is the longest path of dependences between iterations.

Let $L(E, D)$ denote the length of the longest path of dependent iterations in the iteration space I with basis matrix E and dependence matrix D, the ideal execution time of I can be calculated as follows:

$$Time(I) = L(E, D) \tag{5.7}$$

Section 5.4 discusses how to calculate $L(E, D)$. In general, $L(E, D)$ depends on every $\vec{d_k}$ in D. However, if the dependence vectors are not linearly independent, we can ignore some rows in D when calculating $L(E, D)$. For example, the iteration space shown in Figure 5.4 has three dependence vectors $\vec{d_0}$, $\vec{d_1}$ and $\vec{d_2}$, with $\vec{d_2} = \vec{d_0} + \vec{d_1}$. P and P' are two dependence paths within the iteration space. If there is any dependence path containing a pair of neighboring iterations with the dependence of $\vec{d_2}$, such as P', it is always possible to replace $\vec{d_2}$ on the path with a combination of $\vec{d_0}$ and $\vec{d_1}$, as shown in Figure 5.4. This will resulting a longer dependence path. As a result, it can be concluded that the longest dependence path must only contain $\vec{d_0}$ and $\vec{d_1}$ (P), and other dependence vectors can be ignored [2] when calculating $L(E, D)$.

In particular, if there are n dependence vectors which are orthonormal unit vectors, it is safe to ignore other rows in D and only consider those n dependence vectors, because any other single dependence vector can be replaced by the combination of one or more orthonormal unit vectors, which will result in a longer dependence path. Based on above observation,

$$L(E, \begin{pmatrix} \mathbb{1}_n \\ * \end{pmatrix}) = L(E, \mathbb{1}_n) \tag{5.8}$$

[2] Assume that the iteration space is much larger compared to each dependence vector.

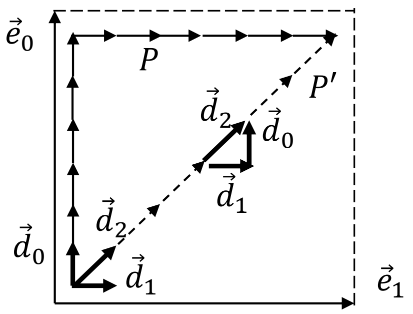


Figure 5.4: The dependent pathes (P and P') within an iteration space. The length of a dependent path is the number of iterations on the path. If any path contains $\vec{d_2}$ (P'), it is always possible to replace $\vec{d_2}$ with $\vec{d_1}$ and $\vec{d_0}$ on the same position, which resulting a longer path. This means the longest path (P) must only contain $\vec{d_0}$ and $\vec{d_1}$.

According to Equation 5.5, under the context of hierarchical tiling the dependence matrix each at each level $D_k = \begin{pmatrix} \mathbb{1}_n \\ * \end{pmatrix}$, $\forall k \geq 1$. As a result, Equation (5.8) can be used to simplify further analysis.

5.4 Calculate the Longest Dependent Path

When applying tiling transformations recursively to a given iteration space I, hierarchical tiling contains a tiling transformation at each level. Following a bottom-up approach for hierarchical tiling, first tiling matrix T_0 is used to tile the original iteration space $I_0 = I$, producing a new iteration space I_1. More generally, on the k-th level of tiling, tiling matrix T_k is applied to tile the iteration space I_k and generated the new iteration space I_{k+1}. Assume E_k is the basis matrix of I_k,

$$I_k = span(E_k), \qquad E_{k+1} = E_k \cdot T_k^{-1} = E_0 \cdot \prod_{j=0}^{k} T_j^{-1}$$

The iterations in the bottom level tiles (the finest grain) are executed in sequential mode, so the per-tile execution time can be calculated by Equation 5.6:

$$Time(T_0) = Time_s(T_0) = |det(T_0)|$$

For upper level tiles ($T_k, 1 \leq k < l$), each tile at the level immediately below is considered as a single iteration.Let D_k denote the dependences at the k-th level with $D_0 = D$, the per-tile execution time is calculated using Equation 5.7:

$$Time(T_k) = L(T_k, D_k) = Time(T_{k-1}) \cdot L(T_k, D_k)$$

Then the total execution time after tiling is:

$$\begin{aligned} Time(I) = Time(E_l) &= Time(T_{l-1}) \cdot L(E_l, D_l) \\ &= Time(T_{l-2}) \cdot L(T_{l-1}, D_{l-1}) \cdot L(E_l, D_l) \\ &= |det(T_0)| \cdot (\prod_{k=1}^{l-1} L(T_k, D_k)) \cdot L(E_l, D_l) \\ &= |det(T_0)| \cdot (\prod_{k=1}^{l-1} L(T_k, D_k)) \cdot L(E_0 \cdot \prod_{k=0}^{l-1} T_k^{-1}, D_l) \end{aligned}$$

According to Equation 5.5 and 5.8, the above equation can be simplified to the following form:

$$Time(I) = |det(T_0)| \cdot (\prod_{k=1}^{l-1} L(T_k, \mathbb{1}_n)) \cdot L(E_l, \mathbb{1}_n) \tag{5.9}$$

So, the problem of optimal tile shape selection for hierarchical tiling is equivalent to selecting a sequence of tiling matrices T_0, T_1, ..., T_{l-1} to minimize the above equation. Equation 5.9 does not include D; however, according to the discussion in Section 5.2 the shape of T_0 must conform the constraints (Equation 5.1 and 5.2) imposed by each dependence vector in D.

5.4.1 Calculate $L(T_k, \mathbb{1}_n)$

The meaning of $L(T_k, \mathbb{1}_n)$ is the length of the longest dependent path with dependence matrix $\mathbb{1}_n$ in tile T_k. The iteration space in tile T_k can be normalized to an n-dimensional hyper cube by applying the affine transformation [3] represented by T_k^{-1}. Thus,

$$L(T_k, \mathbb{1}_n) = L(T_k \cdot T_k^{-1}, \mathbb{1}_n \cdot T_k^{-1}) = L(\mathbb{1}_n, T_k^{-1}).$$

Consider a row in $\mathbb{1}_n$, $\vec{d_j}$. Since $D_k = \begin{pmatrix} \mathbb{1}_n \\ * \end{pmatrix}$, $\vec{d_j}$ is also a row in D_k. Because T_k represents the tile shape at the k-th level of tiling, T_k must conform to the constraints imposed by each dependence vector in D_k. According to Equation 5.1 and 5.2:

$$\exists \vec{a}_j, \quad \vec{0}_n \leq \vec{a}_j \leq \vec{1}_n, \qquad \vec{d_j} = \vec{a}_j \cdot T_k$$

Then, let D' denotes the dependences within the tile (normalized tile so that the tile is defined by tessellating edge vectors):

$$D' = \begin{pmatrix} \vec{d'}_0 \\ \vec{d'}_1 \\ \dots \\ \vec{d'}_{n-1} \end{pmatrix} = \mathbb{1}_n \cdot T_k^{-1} = \begin{pmatrix} \vec{d_0} \\ \vec{d_1} \\ \dots \\ \vec{d}_{n-1} \end{pmatrix} \cdot T_k^{-1} = \begin{pmatrix} \vec{a}_0 \\ \vec{a}_1 \\ \dots \\ \vec{a}_{n-1} \end{pmatrix}$$

$$\forall \vec{d_j}, \qquad \vec{0}_n \leq \vec{d'}_j = \vec{a}_j \leq \vec{1}_n$$

For the 2-dimensional cases, the intuitive explanation of the above observation is that the direction of each $\vec{d_j}$ must be in the upward, right or upper right direction, as shown in Figure

[3] In order to keep the problem in the integer domain, the affine transformation can be revised to $T_k^{-1} \cdot |det(T_k)|$, and the discussion in this section would be still valid.

5.5 (a) and (b). Since the normalized iteration space is a hyper-cube, the longest path must start from the bottom left corner, which is the base point $(0, 0, ..., 0)$.

The problem of computing $L(T_k, \mathbb{1}_n)$ is that finding the longest path P, which is a sequence of points $\vec{p}_0$, $\vec{p}_1$, ..., $\vec{p}_{L-1}$ (Figure 5.5-(c), each $\vec{p}_1$ stands for the coordinate of the point) such that:

$$\begin{aligned}
&\vec{p}_0 = \vec{0} = (0, 0, ..., 0), \\
&\forall j = 0, 1, ..., L-1, \\
&\qquad \vec{0} = (0, 0, ..., 0) \le \vec{p}_j \le (1, 1, ..., 1) = \vec{1}, \\
&\qquad \exists \vec{d'}_{r_j}, \quad \vec{p}_{j+1} = \vec{p}_j + \vec{d'}_{r_j}
\end{aligned} \tag{5.10}$$

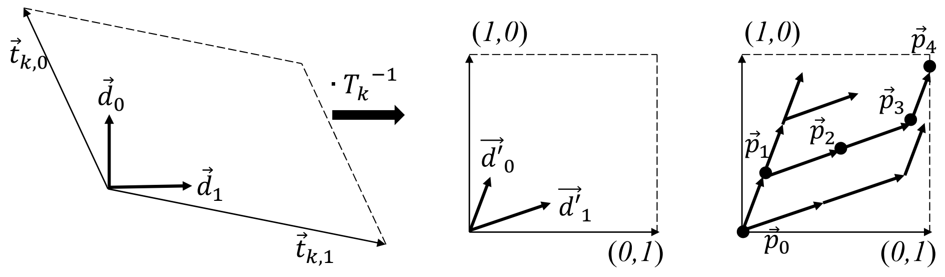
L is the length of the dependent path found. Then $L(T_k, \mathbb{1}_n) = max\{L\}$. We have that the last point $\vec{p}_{L-1}$ is defined as:

$$\begin{aligned}
\vec{p}_{L-1} = \vec{p}_{L-2} + \vec{d'}_{r_{L-2}} &= \sum_{j=0}^{L-2} \vec{d'}_{r_j} && 0 \le k_j < n \\
&= c_0 \cdot \vec{d'}_0 + c_1 \cdot \vec{d'}_1 + ... + c_{n-1} \cdot \vec{d'}_{n-1} \\
&= \vec{c} \cdot D' && \vec{c} = (c_0, c_1, ..., c_{n-1}) \ge \vec{0}, \quad \vec{c} \in \mathbb{Z}^n \\
&= \vec{c} \cdot T_k^{-1}
\end{aligned}$$

Because $\vec{0}_n \le \vec{d'}_j \le \vec{1}_n$, the condition $\vec{0} \le \vec{p}_j \le \vec{1}$ is satisfied if $\vec{0} \le \vec{p}_{L-1} \le \vec{1}$. And because $\sum_{j=0}^{n-1} c_j$ is the total number of steps of the path, so $max\{L\}$ can be solved through the following linear programming problem $LP(T_k^{-1}, \vec{0}, \vec{1})$:

$$\vec{0} \le \vec{c} \cdot T_k^{-1} \le \vec{1}, \quad \vec{c} \ge \vec{0}, \qquad max\{\sum_{j=0}^{n-1} c_j\}. \tag{5.11}$$

Therefore, we conclude that $L(T_k, \mathbb{1}_n) = LP(T_k^{-1}, \vec{0}, \vec{1})$.



(a) Original tile (b) Normalized tile (c) Dependent pathes

Figure 5.5: Determine the length of the longest dependent path within the iteration space of tile T_k. Each $\vec{p}_1$ stands for the coordinate of the point.

5.4.2 Calculate $L(E_l, \mathbb{1}_n)$

The meaning of $L(E_l, \mathbb{1}_n)$ is the length of the longest dependent path in iteration space of the highest level tiling. Similarly, it is possible to apply the affine transformation represented by E_l^{-1} to normalize the iteration space:

$$L(E_l, \mathbb{1}_n) = L(E_l \cdot E_l^{-1}, \mathbb{1}_n \cdot E_l^{-1}) = L(\mathbb{1}_n, E_l^{-1}).$$

Let D' denote the transformed dependence matrix:

$$D' = \begin{pmatrix} \vec{d'}_0 \\ \vec{d'}_1 \\ \dots \\ \vec{d'}_{m-1} \end{pmatrix} = \mathbb{1}_n \cdot E_l^{-1} = E_l^{-1}$$

However, unlike the case when calculating $L(T_k, \mathbb{1}_n)$, there is no guarantee that $\forall \vec{d'}_j,\ \vec{0} \leq \vec{d}_j \leq \vec{1}$. More intuitively, each dependence vector $\vec{d'}_j$ can point to any direction. So $L(E_l, \mathbb{1}_n)$ cannot be directly solved by the linear programming problem introduced in Equation (5.11).

Since dependence vectors $\vec{d'}_j$ can point to any direction, the longest dependent path does not necessarily start from base point $(0, 0, ..., 0)$ of the hyper-cube iteration space. Thus the dependent path P in the iteration space E_l is defined the same as in Equation 5.10 except that $\vec{p}_0$ does not necessarily equal to $\vec{0}$. To simplify the analysis, we define a related problem that is graphically depicted in Figure 5.6: find the longest path P', which is a sequence of points $\vec{p'}_0$, $\vec{p'}_1$, ..., $\vec{p'}_{L'-1}$ such that:

$$\begin{aligned}
&\vec{p'}_0 = (0, 0, ..., 0) = \vec{0}, \\
&\vec{-1} = (-1, -1, ..., -1) \leq \vec{p'}_{L'-1} \leq (1, 1, ..., 1) = \vec{1}, \\
&\forall j = 0, 1, ..., L-2, \quad \exists \vec{d'}_{r_j}, \quad \vec{p'}_{j+1} = \vec{p'}_j + \vec{d'}_{r_j} \qquad (5.12)
\end{aligned}$$

L' is the length of path P'. P' starts from that base point, and only requires that the coordinate of the end point $\vec{p'}_{L'-1}$ is within in the hype-cube surrounding the base point. $\vec{p'}_{L'-1}$ can be calculated as follows:

$$\begin{aligned}
\vec{p'}_{L'-1} &= \vec{p'}_{L'-2} + \vec{d'}_{r_{L'-2}} = \sum_{j=0}^{L-2} \vec{d'}_{r_j} && 0 \leq k_j < n \\
&= c_0 \cdot \vec{d'}_0 + c_1 \cdot \vec{d'}_1 + ... + c_{n-1} \cdot \vec{d'}_{n-1} \\
&= \vec{c} \cdot D' && \vec{c} = (c_0, c_1, ..., c_{n-1}) \geq \vec{0}, \quad \vec{c} \in \mathbb{Z}^n \\
&= \vec{c} \cdot E_l^{-1}
\end{aligned}$$

So $max\{L'\}$ can be solved through the following linear programming problem $LP(E_l^{-1}, \vec{-1}, \vec{1})$:

$$\vec{-1} \leq \vec{c} \cdot E_l^{-1} \leq \vec{1}, \quad \vec{c} \geq \vec{0}, \qquad max\{\sum_{j=0}^{n-1} c_j\}. \qquad (5.13)$$

Next step is to prove that $max\{L'\}$ is approximately equal to $max\{L\}$ so that $LP(E_l^{-1}, \vec{-1}, \vec{1})$ is a good approximation of $L(E_l, \mathbb{1}_n)$. First, given any path P, it is always possible to construct a path P' by letting $\vec{p'}_j = \vec{p}_j - \vec{p}_0$. So $max\{L\} \leq max\{L'\}$ is easily proven.

On the other hand, given a path P', the start point is $\vec{p'}_0 = \vec{0}$, and the end point is

$\vec{p'}_{L'-1} = (p'_0, p'_1, ..., p'_{n-1})$. In order to move path P' into the hyper-cubic space $span(\mathbb{1}_n)$, Construct $\vec{s} = (s_0, s_1, ..., s_{n-1})$ as follows:

$$\forall j = 0, 1, ..., n-1 \qquad s_j = \begin{cases} 1, & p_j < 0, \\ 0, & \text{else.} \end{cases}$$

By shifting the offset of $\vec{s}$, $\vec{p'}_0$ and $\vec{p'}_{L'-1}$ are moved into the hyper-cubic space $span(\mathbb{1}_n)$:

$$\begin{aligned} &\vec{p'}_s = \vec{p'}_0 + \vec{s}, && \vec{0} \le \vec{p'}_s \le \vec{1}, \quad \vec{p'}_s \in span(\mathbb{1}_n) \\ &\vec{p'}_e = \vec{p'}_{L'-1} + \vec{s}, && \vec{0} \le \vec{p'}_e \le \vec{1}, \quad \vec{p'}_e \in span(\mathbb{1}_n) \\ &\vec{p'}_e = \vec{p'}_s + \vec{c} \cdot D' \end{aligned} \tag{5.15}$$

Assume that the iteration space is much larger compared to the length of each dependence vector: $\forall \vec{d'}_j, |\vec{d'}_j| << 1$. Then it is always possible to find two point $\vec{p}_s$ and $\vec{p}_e$ in a small surrounding space around $\vec{p'}_s$ and $\vec{p'}_e$ respectively, such that:

$$\begin{aligned} &|\vec{p}_s - \vec{p'}_s| < \varepsilon, && \vec{p}_s \in span(\mathbb{1}_n), \\ &|\vec{p}_e - \vec{p'}_e| < \varepsilon, && \vec{p}_e \in span(\mathbb{1}_n), \\ &\vec{p}_e - \vec{p}_s = b \cdot \vec{c'} \cdot D', && b \in \mathbb{Z}, \vec{c'} \in \mathbb{Z}^n. \end{aligned} \tag{5.16}$$

Then construct new path P within the hyper-cubic space $span(\mathbb{1}_n)$ as follows:

$$\begin{aligned} &C = \sum_{j=0}^{n} c'_j, \qquad L = b \cdot C \\ &\vec{p}_0 = \vec{p}_s, \\ &\vec{p}_{j \cdot C + k} = j \cdot \vec{c'} \cdot D' + \vec{d'}_{j_k}, \qquad 0 \le j < b, \quad c'_{j_k} \ne 0 \\ &\vec{p}_{L-1} = \vec{p}_e. \end{aligned}$$

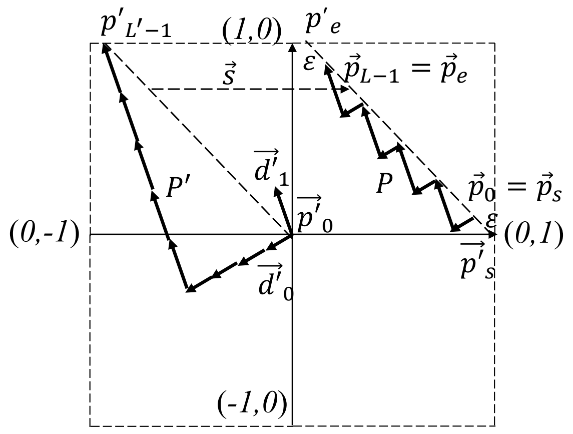


Figure 5.6: Construct path P according to a given path P'. The lengthes of P and P' are approximately equal.

The length of path P is $L = b \cdot C$. Considering Equation 5.15 and 5.16, there is:

$$L > L' - \lceil \frac{\varepsilon}{min\{|\vec{d'}_j|\}} \rceil = L' - \varepsilon',$$
$$max\{L\} > max\{L'\} - \varepsilon'.$$

The idea of above analysis is, we try to break path P' into small repeated pieces, and arrange the same number of repeated pieces in a line to construct P. Since the start and end points of P and P' are very close, the length of these two pathes should also be close. Figure 5.6 shows the intuition of above analysis.

Since $max\{L\} \leq max\{L'\}$ is already proved, it can be concluded that $max\{L\} \approx max\{L'\}$, with the error no larger than ε'. As a result, the solution of the linear programming problem $LP(E_l^{-1}, -\vec{1}, \vec{1})$ can be used to estimate $L(E_l, \mathbb{1}_n)$:

$$L(E_l, \mathbb{1}_n) = L(E_0 \cdot \prod_{k=0}^{l-1} T_k^{-1}, \mathbb{1}_n) \approx LP(E_l^{-1}, -\vec{1}, \vec{1})$$

5.4.3 Summary

According to the discussion in last two sections, the problem of computing execution time of hierarchical tiled loop nests can be transformed into the following form:

$$\begin{aligned}
Time(I) &= |det(T_0)| \cdot (\prod_{k=1}^{l-1} L(T_k, \mathbb{1}_n)) \cdot L(E_l^{-1}, \mathbb{1}_n) \\
&\approx |det(T_0)| \cdot (\prod_{k=1}^{l-1} LP(T_k^{-1}, \vec{0}, \vec{1})) \cdot LP(E_l^{-1}, \vec{-1}, \vec{1}) \\
&= f(E, T_0, T_1, ..., T_{l-1})
\end{aligned}$$

Given the basis matrix E of the original iteration space I, the optimal tile shape for hierarchical tiling is the sequence of tiling matrices $T_0, T_1, ..., T_{l-1}$ that minimize $f(E, T_0, T_1, ..., T_{l-1})$. If I is n-dimensional, each tiling matrix T_k contains n^2 elements. In total function f has $n^2 \cdot l$ variables. Besides, f is a non-linear function. Finally, because f contains terms of the solutions to linear programming problems, the problem of optimal tile shape selection is a multidimensional, nonlinear, bi-level programming problem.

5.5 Automatic Tile Shape Selection

According to the discussion in last section, optimal tile shape selection for hierarchical tiling is a multidimensional, nonlinear, bi-level programming problem, which is not easy to solve analytically. As a result, simulated annealing is adopted to select tile shape automatically according the analytical model introduced in last section. The sketch of simulated annealing algorithm to select tile shape is shown in Algorithm 2.

Line 8 in Algorithm 2 $valid(NewS)$ checks whether the randomly generated $NewS$ is a valid tiling solution, otherwise $NewS$ will be generated again. This includes checking each T_k in $NewS$ satisfy the constraints imposed by dependences (Equation 5.1 and 5.2). This

Algorithm 2 Sketch of simulated annealing algorithm to select tile shape for hierarchical tiling

```
$Solution = Solution_0 =< \mathring{T}_0, \mathring{T}_1, ..., \mathring{T}_{n-1} >$;
$Time = f(E, Solution_0) = f(E, \mathring{T}_0, \mathring{T}_1, ..., \mathring{T}_{n-1})$;
$Solution_{best} = Solution$;
$Time_{best} = Time$;
$Step = 0$;
while $Step < Step_{max}$ do
   $NewS = < T_0, T_1, ..., T_{n-1} > = Solution + random(\Delta)$;
   if $valid(NewS)$ then
      $NewT = f(E, NewS)$;
      $Temp = temperature(Step, Step_{max})$;
      if $accept(NewT, Time, Temp, random())$ then
         $Solution = NewS$;
         $Time = NewT$;
      end if
      if $Time < Time_{best}$ then
         $Solution_{best} = Solution$;
         $Time_{best} = Time$;
      end if
      $Step = Step + 1$;
   end if
end while
```

also includes checking that the maximum number of parallelizable tiles does not exceed the hardware parallelism at each level. In addition, it is necessary to keep each tile large enough to aromatize the overhead of tiling. So we add an additional requirement in $valid(NewS)$ such that the iterations within each tile T_k much be more than a certain threshold Th_k at each level of tiling:

$$\forall T_k, \quad |det(T_k)| \geq Th_k$$

Note that upper bound is not set for the size of each tile T_k. This is because the simulated annealing algorithm is self-adaptive and is able to converge to tiles with suitable size. The solution with very large tiles will lead to high execution time and hence the probability to be accepted in Line 11 of Algorithm 2 is low. For initial solution $Solution_0 {=} < \mathring{T}_0, \mathring{T}_1, ..., \mathring{T}_{n-1} >$, just choose the smallest possible tile that is valid for each level of tiling: $|det(\mathring{T}_k)| = Th_k$.

5.6 Unified Tiling Representation

The tiling schemes studied in this chapter are tessellating, atomic tiling, and the shape of iteration space and tile shapes are restricted to be n-dimensional hyper-parallelepipeds. As a result, there exists an basis matrix E such that $I = span(E)$, and tiling matrix T such that $\mathbb{T} = span(T)$. For tessellating tiling where the tile shapes are n-dimensional hyper-parallelepipeds, the repetition matrix $R = T$. This chapter studies performance of the best possible schedule of tiles in stead of any particular scheduling scheme. Table 5.1 shows the unified representation of a single level of tiling. Because hierarchical tiling is done in the bottom-up approach in this chapter, the input for the kth level of tiling is: $I_k = I_{k-1} \cdot T_k^{-1}$, $D_k = D'_{k-1}$.

Input	$I = span(E), D$
D'	$D' = \begin{pmatrix} 1,0,0,...,0 \\ 0,1,0,...,0 \\ 0,0,1,...,0 \\ ... \\ 0,0,0,...,1 \\ * \end{pmatrix}$
$\mathbb{T}$	the selected tile shape T
$\mathbb{R}$	$R = T$.
$\mathbb{S}$	Best possible $\mathbb{S}$ that minimizes the ideal execution time

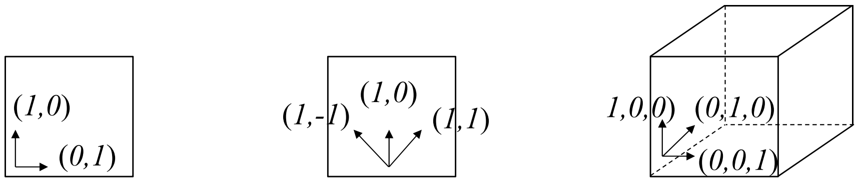
Table 5.1: The Unified Tiling Representation

5.7 Evaluation

5.7.1 Environment Setup

Two platforms with hierarchical parallelism are step up for evaluation. One is the GPU platform consisting of an NVIDIA GeForce GTX 480 graphic card with 15 MPs and 480 SPs in total. The other platform is a 32-node cluster; each node has 4 Quad-Core AMD Opteron processors, or 16 cores. Because both platforms only have a 2-levels hierarchy, only 2-level hierarchical tiling is evaluated. For the GPU, the hardware parallelism at level 1 is 15, which is the number of MPs. However, although the number of SPs within each MP is 32, we consider the hardware parallelism at level 0 as 512 for the experiments. This is because NVIDIA GPUs require a certain number of concurrent hardware threads for each MP to achieve high throughput. For the cluster, the hardware parallelisms at two levels are 16 and 32, which are the number of cores per node and the total number of nodes.

Two stencil computation programs are used as benchmarks: Gauss-Seidel and Jacobi. 1-D stencils use 1-dimensional input array, and result in 2-dimensional iteration spaces, while 2-D stencils result in 3-dimensional iteration spaces. The iteration space of 1-D Gauss-Seidel contains two dependence vectors: $(1, 0)$ and $(0, 1)$, and the iteration space of 1-D Jacobi contains three dependence vectors: $(1, -1)$, $(1, 0)$ and $(1, 1)$. There are three dependence



(a) 1-D Gauss-Seidel (b) 1-D Jacobi (c) 2-D Gauss-Seidel

Figure 5.7: The dependence vectors in the iteration space of stencil computation programs.

vectors for 2-D Gauss-Seidel: $(1, 0, 0)$, $(0, 1, 0)$ and $(0, 0, 1)$. Figure 5.7 shows the dependence vectors in the iteration space of stencil computation programs.

An automatic system which uses simulated annealing is implemented to select tile shape for hierarchical tiling. The lp_solve[8] is used to solve the linear programming problems in the model. The Omega Library[22] is used to do the tiling transformation and code generation. On the GPU platform, we generate OpenCL[23] code. On the cluster platform we generate hybrid MPI-OpenMP code: the higher level tiles are parallelized using MPI across nodes, and the lower level tiles are parallelized with OpenMP across processor cores within each node.

5.7.2 Performance

The performance of three common tiling schemes is compared with that of the code generated by our system: *Wavefront*, *Diamond* and *Skewing*, as shown in Table 5.2. The optimal sizes for the tiles of these three tiling schemes are chosen under the condition that the amount parallelism exposed does not exceed the hardware parallelism at the corresponding level.

Figure 5.8 shows the speedup of the difference hierarchical tiling schemes for 2-dimensional the iteration spaces for both GPU and cluster platforms, and Figure 5.9 shows the performance data for 3-dimensional iteration spaces. The horizontal axis is the size of iteration space. The vertical axis is the speedup over *Wavefront* or *Diamond*. On average, the tiling

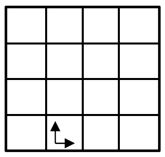
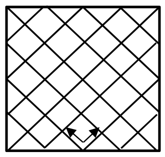
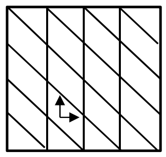
	Wavefront	Diamond	Skewing
Tile Shape			
Tiling Matrix	$\begin{pmatrix} x, 0 \\ 0, x \end{pmatrix}$	$\begin{pmatrix} x, -x \\ x, x \end{pmatrix}$	$\begin{pmatrix} x, 0 \\ -x, x \end{pmatrix}$

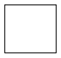
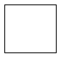
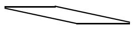
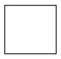
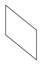
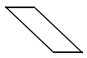
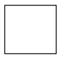
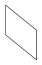
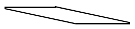
Table 5.2: Common tiling schemes: *Wavefront*, *Diamond* and *Skewing*.

scheme with automatically selected tile shape can achieve over 90% speedup over *Wavefront* for Gauss-Seidel stencil, and 20%-30% speedup over *Diamond* for Jacobi stencil. The speedup achieved is consistent over different iteration space sizes. This means our automatic system can always find better tile shapes compared to common tiling schemes.

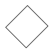
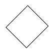
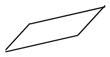
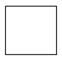
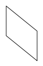
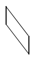
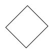
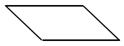
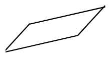
For 1-D Jacobi stencil shown in Figure 5.8-(c) and (d), it can be seen that on the GPU platform *Skewing* shows a 10% speedup over *Diamond*, but no speedup is observed on the cluster platform. It is shown that the ideal execution times of *Diamond* and *Skewing* are the same for the Jacobi stencil. However, the amount of parallelism within each higher level tile varies less with *Skewing* than *Diamond*. This reason is that, as mentioned before, the execution model of NVIDIA GPU requires a certain number of concurrent hardware threads to achieve high throughput, *Skewing* shows better performance than *Diamond* on GPU.

5.7.3 Tile Shape

Table 5.3 shows the tile shape selection for 1-D Gauss-Seidel and Jacobi on GPU platform of each tiling scheme corresponding to Figure 5.8. Table 5.4 shows the tile shape selection for 2-D Gauss-Seidel on GPU. T_0 and T_1 are the tiling matrix at each level, and $T_0 \cdot T_1$ represents the tile shape in the original iteration space. In Table 5.3, under each matrix the shape of the tile is shown graphically. It can be seen that the common tiling schemes of *Wavefront*, *Diamond* and *Skewing* are combinations of regular tile shapes, including squares, diamonds

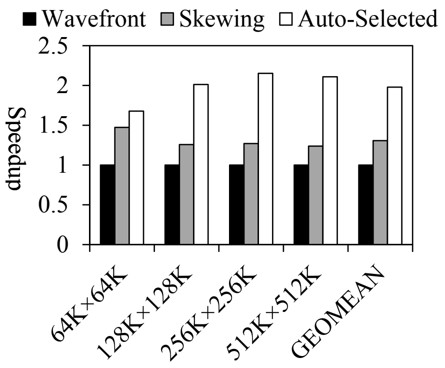
	Wavefront	Skewing	Auto-Selected
T_0	$\begin{pmatrix} 32, 0 \\ 0, 32 \end{pmatrix}$ 	$\begin{pmatrix} 32, 0 \\ 0, 32 \end{pmatrix}$ 	$\begin{pmatrix} 2, -30 \\ 0, 12 \end{pmatrix}$ 
T_1	$\begin{pmatrix} 256, 0 \\ 0, 256 \end{pmatrix}$ 	$\begin{pmatrix} 256, 0 \\ -256, 256 \end{pmatrix}$ 	$\begin{pmatrix} 256, -250 \\ 0, 56 \end{pmatrix}$ 
$T_1 \cdot T_0$	$\begin{pmatrix} 8192, 0 \\ 0, 8192 \end{pmatrix}$ 	$\begin{pmatrix} 8192, 0 \\ -8192, 8192 \end{pmatrix}$ 	$\begin{pmatrix} 512, -10680 \\ 0, 672 \end{pmatrix}$ 

(a) 1-D Gauss-Seidel on GPU (128K×128K)

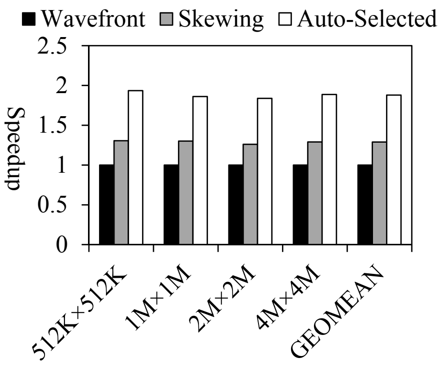
	Diamond	Skewing	Auto-Selected
T_0	$\begin{pmatrix} 18, -18 \\ 18, 18 \end{pmatrix}$ 	$\begin{pmatrix} 18, -18 \\ 18, 18 \end{pmatrix}$ 	$\begin{pmatrix} -2, -18 \\ 8, 8 \end{pmatrix}$ 
T_1	$\begin{pmatrix} 256, 0 \\ 0, 256 \end{pmatrix}$ 	$\begin{pmatrix} 256, 0 \\ -256, 256 \end{pmatrix}$ 	$\begin{pmatrix} 64, 0 \\ -512, 512 \end{pmatrix}$ 
$T_1 \cdot T_0$	$\begin{pmatrix} 4608, -4608 \\ 4608, 4608 \end{pmatrix}$ 	$\begin{pmatrix} 4608, -4608 \\ 0, 9216 \end{pmatrix}$ 	$\begin{pmatrix} -128, -1152 \\ 5120, 13312 \end{pmatrix}$ 

(b) 1-D Jacobi on GPU (128K×128K)

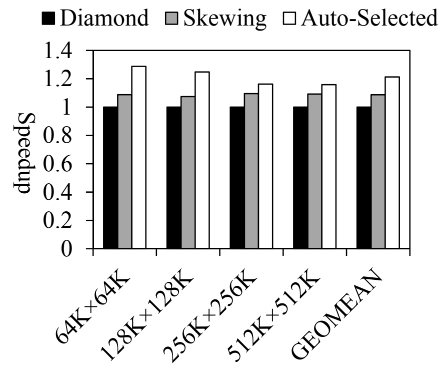
Table 5.3: Tile shape selection for 1-D Gauss-Seidel and Jacobi.



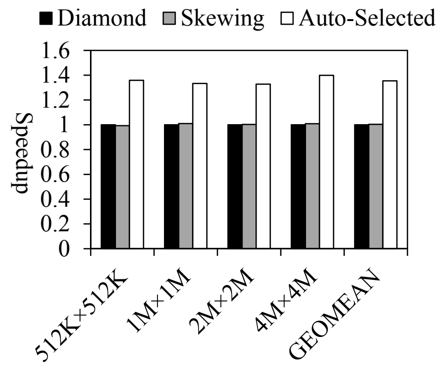
(a) 1-D Gauss-Seidel on GPU



(b) 1-D Gauss-Seidel on cluster



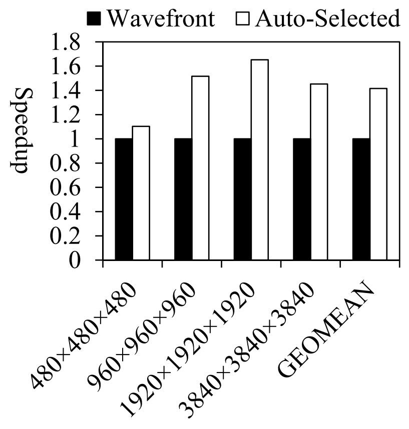
(c) 1-D Jacobi on GPU



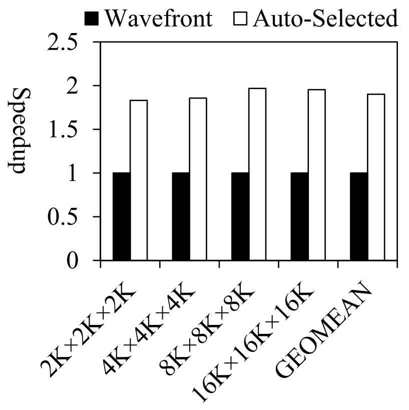
(d) 1-D Jacobi on cluster

Figure 5.8: Hierarchical tiling performance for 2-dimensional iteration space on GPU and cluster platforms. The horizontal axis is the size of the iteration space. The vertical axis is the speedup over *Wavefront* or *Diamond.*

and regular parallelograms. However, the automatically selected tile shapes are irregular parallelograms. Since it is nonintuitive to figure out the performance impact of a particular tiling scheme with irregular tile shapes (especially for higher dimensional iteration spaces), it is necessary to build an analytic model to evaluate tile shape selection quantitatively, and select tile shape automatically.



(a) 2-D Gauss-Seidel on GPU



(b) 2-D Gauss-Seidel on cluster

Figure 5.9: Hierarchical tiling performance for 3-dimensional iteration space on GPU and cluster platforms. The horizontal axis is the size of the iteration space. The vertical axis is the speedup over *Wavefront*.

	Wavefront	Auto-Selected
T_0	$\begin{pmatrix} 16,0,0 \\ 0,16,0 \\ 0,0,16 \end{pmatrix}$	$\begin{pmatrix} 16,-48,0 \\ 0,16,0 \\ 0,-48,16 \end{pmatrix}$
T_1	$\begin{pmatrix} 16,0,0 \\ 0,16,0 \\ 0,0,16 \end{pmatrix}$	$\begin{pmatrix} 8,-8,0 \\ 0,8,0 \\ 0,-16,16 \end{pmatrix}$
$T_1 \cdot T_0$	$\begin{pmatrix} 256,0,0 \\ 0,256,0 \\ 0,0,256 \end{pmatrix}$	$\begin{pmatrix} 128,-512,0 \\ 0,128,0 \\ 0,-1024,256 \end{pmatrix}$

Table 5.4: Tile shape selection for 2-D Gauss-Seidel on GPU (1920×1920×1920).

5.7.4 Model Accuracy

Figure 5.10 compares the real execution time and the ideal execution time. The data is for $128K \times 128K$ 1-D Gauss-Seidel running on GPU, with tile shape choices listed below:

$$T_0 = \begin{pmatrix} 32,0 \\ 0,32 \end{pmatrix}, \quad T_1 = \begin{pmatrix} 256,0 \\ x,256 \end{pmatrix}.$$

The value of x in T_0 is scaled from 0 to -320. The shape of resulting T_0 changes from a square to a parallelogram. It is shown that the trends of the real execution time and the

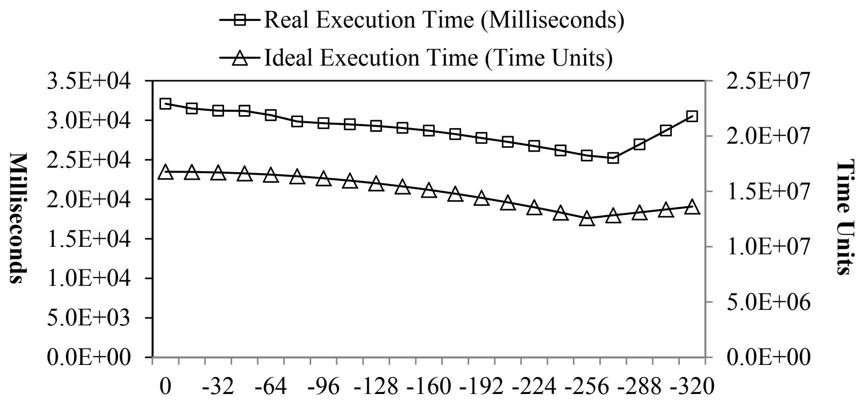


Figure 5.10: Comparison between the real execution time and the ideal execution time for 1-D Gauss-Seidel. The left axis is for real execution time and the axis on the right is for ideal execution time.

ideal execution time are exactly the same when tile shape varies. The result shows that ideal execution time commutated by the model in this paper can be used to direct tile shape selection.

5.8 Conclusion

In this chapter an analytical model is built to analyze the relation between the tile shape at each level of hierarchical tiling and the ideal execution time of the tiled loop nest. It is shown that the problem of optimal tile shape selection for hierarchical tiling is a multidimensional, nonlinear, bi-level programming problem. An automatic system which uses simulated annealing together with our analytical model is implemented to automatically select tile shape for hierarchical tiling. The tile shape selected by the automatic system can be quite different from the intuitive regular shapes. The experimental results show that irregular tile shapes may have the potential to achieve higher performance over regular tiling schemes.

Currently the model focuses on the interaction between tile shape and parallelism exposure, and uses ideal execution time as the optimizing goal. The model could be more accurate by considering other factors affecting locality and communication. Besides, given the complexity of the multi-dimensional nonlinear optimization problem in our model, at present simulated annealing is sued to find the near-optimal solution instead of the guaranteed optimal one. In future, it is possible to use the state-of-the-art optimization problem solving algorithms or libraries to solve the tile shape selection problem directly.

Chapter 6

Related Work

Loop tiling is a traditional but effective optimization for programs whose execution time is dominated by loops, such as programs with stencil computations. Numerous optimizations based on the tiling of iteration spaces have been proposed for improving data locality [39, 42, 1, 2, 36, 14, 33, 38, 41, 45, 48], or exploiting parallelism [40, 43, 6, 18, 44, 7, 47]. Most of these works use polyhedral model as the representation of the tiling transformation. Bondhugula et al. design and implement Pluto [11], which can automatically transform loops for parallelism and locality based on the polyhedral model. Their transformation integrates traditional tiling techniques. However, existing research on tiling mainly focuses on tessellating tiling and scheduling tiles atomically. There have been ideas of taking advantage of the hierarchy of the hardware [29, 27, 32, 24, 10, 16, 9]. Hierarchical tiling as an effective optimization to exploit hardware hierarchy, has been proposed for better usage of memory hierarchy and enhancing parallel schedule [12, 17, 13].

For existing work on non-tessellating tiling, the closest work to the Overlapped Tiling discussed in this thesis is that of Krishnamoorthy et al.[26]. Their approach allows overlap between neighboring tiles to achieve balanced schedule of parallel tiles. Other works such as Ripeanu et al. [35] and Meng et al. [30] describe performance models to predict the optimal amount of redundant computation for stencil computations in a grid environment with message passing or for GPUs, respectively. However, in those paper only single level of overlapped tiling is considered, so that as more communication/synchronization overhead is eliminated, the overhead introduced by redundant computations also increases in a higher order, which would kill the performance benefit of overlapped tiling. The most important difference between this thesis and existing work is the Hierarchical Overlapped Tiling transformation. Based on the observation that the overhead of redundant computation is the main drawback of the overlapped tiling approach, Hierarchical Overlapped Tiling applies overlapped tiling hierarchically. In this way, Hierarchical Overlapped Tiling is able to take advantage of the hierarchy of the hardware and provide more room of balance between the additional overhead introduced by redundant computation and the communication/synchronization overhead eliminated.

The purpose of tiling with non-atomic tiles is usually to achieve balanced schedule when parallelizing tiles. When parallelizing ISLs, a typical strategy is to do loop skewing and wavefronting. However, wavefronting strategy would lead to an imbalanced schedule [26]. In order to achieve balanced schedule when tiling ISLs, Krishnamoorthy et al. [26] propose split tiling. The idea of split tiling is similar to Conjugate-Trapezoid Tiling: each tile is divided into sub-regions, and the sub-region without dependence is processed first, followed by the sub-regions with dependences. Since inter-tile communication needs to be done between the computation of sub-regions within a tile, split tiling is a tiling scheme with non-atomic tiles. On the other hand, split tiling can be also viewed as a tiling scheme with more than one tile shapes as discussed in Section 2.1.3, and the set of neighboring tiles with different shapes forms a hyper-parallelepiped-shaped super tile. Tang et al. [37] implement the Pochoir compiler, which automates a trapezoidal decomposition for stencil code. Their decomposition algorithm is very similar to split tiling. So it can be viewed as another example of tiling with non-atomic tiles. However, their work is based on shared memory machines, and the inter-tile communication overhead is not studied.

For split tiling and the Pochoir compiler, inter-tile communication is still aggregated at the beginning or end of the execution of tiles or subtiles/sub-regions; inter-tile communication is not interleaved with computations. So although the tiles/super-tiles are non-atomic, subtiles and sub-regions are still execution atomically. Because communication latency cannot be overlapped with computation time, the overall performance could still suffer from the overhead caused by communication latency. This might be tolerable for shared memory machines, but could cause performance problems for distributed machines, which is expected to have higher inter-node communication latency. Demmel et al. [15] proposed techniques to reduce communications for sparse matrix computations under distributed memory environment. Their technique allows communication during the execution within a tile, and schedule computation and communications to reduce the penalty of latency. However, their results show that speedups are only achieved for machines with very high communication latency; no speedup is achieved by their parallel algorithm on machines with fast network. On contrast, the Conjugate-Trapezoid Tiling introduced in this thesis can change the number delayed steps h' to fit target platforms with different communication latency.

It is well known that tile shape, as well as tile size, can have a significant impact on locality, inter-tile communication and parallelism [31, 46, 28]. However, most of the existing research on tiling mainly focus on choosing tile size assuming those of regular shapes (such as rectangles, or regular parallelograms) that can be produced with the help of simple auxiliary transformations such as loop skewing. This is because the performance impact of irregular tiles shapes is usually nonintuitive and it is not easy to develop a strategy to find a good tile shape among irregular shapes. Besides, it is also difficult to write code with irregular tiles manually. Xue [46] presents an approach to find the parallelogram or hyper-parallelepiped tile shape that is able to minimize the amount of communication between tiles. This work does not consider the impact of tile shape on parallel scheduling. Högstedt et al. [18] introduce a model to select the tile shape that minimizes the execution time. Since their work only consider a single level of tiling, their model shows that the complexity to determine of execution time when the tiles are parallelograms is equivalent to the complexity of linear programming problem. None of these works discussed above studies the tile shape selection problem in the context of hierarchical tiling. Renganarayana and Rajopadhye [34] have the closest work of using the model of linear programming problem to optimize tiling schemes. Their work presents a model to estimate the overall execution time of loop nests. Their framework determines the optimal tile sizes for hierarchical tiling by solving a convex optimization problem. However, their work only considers hyper-rectangle tiles, as well as the shape of the iteration space. Although it is more difficult to analyze the performance impact of hierarchical tiling with the general tile shapes, experimental results given in this thesis show that irregular tile shapes have the potential to achieve higher performance over regular shapes.

Chapter 7

Conclusions

This thesis studies the application of tiling techniques for stencil compucations. previous studies of tiling optimizations mainly focused on a single level tiling, tessellating tiling, regular shape tile, and executing tiles atomically. This thesis discusses several novel tiling techniques, including hierarchical tiling, non-tessellating tiling, executing tiles non-atomically, irregular tile shapes, and combinations of these techniques.

Contributions of this thesis include:

- The introduction of a unified tiling representation framework to represent general tiling schemes, including non-tessellating tiling and executing tiles non-atomically. An automatic code generator is developed to facilitate this tiling representation framework, which was used for the evaluation of the proposed tiling schemes.

- Overlapped Tiling and Hierarchical Overlapped Tiling are introduced in this thesis. They belong to the category of non-tessellating tiling. Both of these tiling schemes aim at eliminating communication/synchronization overhead by introducing redundant computation. The second scheme also takes advantage of hardware hierarchy to reduce the amount of redundant computation.

- The introduction of the design and evaluation of Conjugate-Trapezoid Tiling. Conjugate-Trapezoid Tiling pipelines intra-tile computation and inter-tile communication, so that the communication latency can be hidden by overlapping with computation time.

- A novel approach to the tile shape selection problem for hierarchical tiling. It is concluded that optimal tile shape selection for hierarchical tiling is a multidimensional, nonlinear, bi-level programming problem. Experimental results show that the irregular tile shapes have the potential to outperform intuitive tiling shapes.

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] N. Ahmed, N. Mateev, and K. Pingali. Tiling imperfectly-nested loop nests. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society.

[3] M. Alpert. Not just Fun and Games. *Scientific American*, 4, 1999.

[4] W. F. Ames. *Numerical Methods for Partial Differential Equations.* Academic, San Diego, CA, sencond edition, 1977.

[5] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '91, pages 39–50, New York, NY, USA, 1991. ACM.

[6] R. Andonov, S. Balev, S. Rajopadhye, and N. Yanev. Optimal semi-oblique tiling. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '01, pages 153–162, New York, NY, USA, 2001. ACM.

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

[8] M. Berkelaar, K. Eikland, and P. Notebaert. Lpsolve. http://lpsolve.sourceforge.net/5.5/.

[9] G. Bikshandi. *Parallel Programming with Hierarchically Tiled Arrays*. PhD thesis, UIUC, 2007.

[10] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarán, D. Padua, and C. von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 48–57, New York, NY, USA, 2006. ACM.

[11] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[12] L. Carter, J. Ferrante, and S. F. Hummel. Hierarchical tiling for improved superscalar performance. In *Proceedings of the 9th International Symposium on Parallel Processing*, IPPS '95, pages 239–245, Washington, DC, USA, 1995. IEEE Computer Society.

[13] L. Carter, J. Ferrante, S. F. Hummel, B. Alpern, and K.-S. Gatlin. Hierarchical tiling: A methodology for high performance. Technical report, 1996.

[14] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 279–290, New York, NY, USA, 1995. ACM.

[15] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick. Avoiding communication in sparse matrix computations. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12, April.

[16] B. B. Fraguela, J. Guo, G. Bikshandi, M. J. Garzarán, G. Almási, J. Moreira, and D. Padua. The hierarchically tiled arrays programming approach. In *Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems*, LCR '04, pages 1–12, New York, NY, USA, 2004. ACM.

[17] A. Hartono, M. M. Baskaran, C. Bastoul, A. Cohen, S. Krishnamoorthy, B. Norris, J. Ramanujam, and P. Sadayappan. Parametric multi-level tiling of imperfectly nested loops. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 147–157, New York, NY, USA, 2009. ACM.

[18] K. Högstedt, L. Carter, and J. Ferrante. Selecting tile shape for minimal execution time. In *Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures*, SPAA '99, pages 201–211, New York, NY, USA, 1999. ACM.

[19] W. Huang, M. R. Stan, K. Skadron, K. Sankaranarayanan, S. Ghosh, and S. Velusam. Compact thermal modeling for temperature-aware design. In *Proceedings of the 41st annual Design Automation Conference*, DAC '04, pages 878–883, New York, NY, USA, 2004. ACM.

[20] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 319–329, New York, NY, USA, 1988. ACM.

[21] T. A. Johnson, S.-I. Lee, L. Fei, A. Basumallik, G. Upadhyaya, R. Eigenmann, and S. P. Midkiff. Experiences in using cetus for source-to-source transformations. In *Proceedings of the 17th international conference on Languages and Compilers for High Performance Computing*, LCPC'04, pages 1–14, Berlin, Heidelberg, 2005. Springer-Verlag.

[22] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega Library interface guide. Technical report, 1995.

[23] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.

[24] I. Kodukula, K. Pingali, R. Cox, and D. Maydan. An experimental evaluation of tiling and shackling for memory hierarchy management. In *Proceedings of the 13th international conference on Supercomputing*, ICS '99, pages 482–491, New York, NY, USA, 1999. ACM.

[25] G. Kreisel and J.-L. Krivine. *Elements of mathematical logic.* North-Holland Pub. Co., 1967.

[26] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 235–244, New York, NY, USA, 2007. ACM.

[27] A. Leung, N. Vasilache, B. Meister, M. Baskaran, D. Wohlford, C. Bastoul, and R. Lethin. A mapping path for multi-gpgpu accelerated computers from a portable high level programming abstraction. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 51–61, New York, NY, USA, 2010. ACM.

[28] A. W. Lim, G. I. Cheong, and M. S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *Proceedings of the 13th international conference on Supercomputing*, ICS '99, pages 228–237, New York, NY, USA, 1999. ACM.

[29] J. Liu, Y. Zhang, W. Ding, and M. Kandemir. On-chip cache hierarchy-aware tile scheduling for multicore machines. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 161–170, Washington, DC, USA, 2011. IEEE Computer Society.

[30] J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 256–265, New York, NY, USA, 2009. ACM.

[31] H. Ohta, Y. Saito, M. Kainaga, and H. Ono. Optimal tile size adjustment in compiling general DOACROSS loop nests. In *Proceedings of the 9th international conference on Supercomputing*, ICS '95, pages 270–279, New York, NY, USA, 1995. ACM.

[32] N. Park, B. Hong, and V. Prasanna. Tiling, block data layout, and memory hierarchy performance. *IEEE Transactions on Parallel and Distributed Systems*, 14(7):640–654, July.

[33] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for non-shared memory machines. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 111–120, New York, NY, USA, 1991. ACM.

[34] L. Renganarayana and S. Rajopadhye. A geometric programming framework for optimal multi-level tiling. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, SC '04, pages 18–, Washington, DC, USA, 2004. IEEE Computer Society.

[35] M. Ripeanu, A. Iamnitchi, and I. T. Foster. Cactus application: Performance predictions in grid environments. In *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, Euro-Par '01, pages 807–816, London, UK, UK, 2001. Springer-Verlag.

[36] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, pages 215–228, New York, NY, USA, 1999. ACM.

[37] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The pochoir stencil compiler. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM.

[38] M. Wolf, D. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. *International Journal of Parallel Programming*, 26:479–503, 1998.

[39] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 30–44, New York, NY, USA, 1991. ACM.

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

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

[42] M. Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, Philadelphia, PA, USA, 1989. Society for Industrial and Applied Mathematics.

[43] M. Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, Supercomputing '89, pages 655–664, New York, NY, USA, 1989. ACM.

[44] D. Wonnacott. Time skewing for parallel computers. In *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '99, pages 477–480, London, UK, UK, 2000. Springer-Verlag.

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

[46] J. Xue. Communication-minimal tiling of uniform dependence loops. In *Languages and Compilers for Parallel Computing*, volume 1239, pages 330–349. Springer Berlin / Heidelberg, 1997.

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

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

[49] X. Zhou, J.-P. Giacalone, M. J. Garzarán, R. H. Kuhn, Y. Ni, and D. Padua. Hierarchical overlapped tiling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 207–218, New York, NY, USA, 2012. ACM.