DIGITALCOMMONS
@WAYNESTATE

Wayne State University

Wayne State University Dissertations

January 2020

# Tiling Optimization For Nested Loops On Gpus

Yuanzhe Li
*Wayne State University*

Follow this and additional works at: https://digitalcommons.wayne.edu/oa_dissertations

Part of the Computer Sciences Commons

## Recommended Citation

Li, Yuanzhe, "Tiling Optimization For Nested Loops On Gpus" (2020). *Wayne State University Dissertations*. 2362.
https://digitalcommons.wayne.edu/oa_dissertations/2362

This Open Access Dissertation is brought to you for free and open access by DigitalCommons@WayneState. It has been accepted for inclusion in Wayne State University Dissertations by an authorized administrator of DigitalCommons@WayneState.

**TILING OPTIMIZATION FOR NESTED LOOPS ON GPUS**

by

**YUANZHE LI**

**DISSERTATION**

Submitted to the Graduate School,

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

**DOCTOR OF PHILOSOPHY**

2020

MAJOR: Computer Science

Approved By:

______________________________

Advisor                                        Date

______________________________

______________________________

______________________________

# DEDICATION

For my wife Jingwen, child Avery, and parents Zhiyong and Tong, and all the things that bring us joy in life.

# ACKNOWLEDGEMENTS

This template is a combination of work done by Mike Catanzaro and Gabe Angelini-Knoll, both formerly of the WSU math department, with some additions and synthesizations by Clayton Hayes (clayton.hayes@wayne.edu) for broader use.

This template was last update updated on 2019-03-10 by Aaron Willcock (ez9213@wayne.edu) for the updated WSU dissertation and thesis formatting guidelines. The unedited, accepted latex upon which this thesis template was most recently adapted can be found at (https://github.com/aarontwillcock/wsu-ms-cs-tufc).

# TABLE OF CONTENTS

Dedication . . . . . . . . . . ii

Acknowledgements . . . . . . . . . . iii

List of Tables . . . . . . . . . . viii

List of Figures . . . . . . . . . . ix

Chapter 1 Introduction . . . . . . . . . . 1

1.1 Contributions . . . . . . . . . . 3

1.2 Dissertation Outline . . . . . . . . . . 5

Chapter 2 Overview of NVIDIA GPUs and CUDA . . . . . . . . . . 7

2.1 GPU Architecture . . . . . . . . . . 8

2.2 CUDA Terminology . . . . . . . . . . 10

2.3 GPU Properties . . . . . . . . . . 11

Chapter 3 OVERVIEW OF LOOP TILING . . . . . . . . . . 14

3.1 Loop Tiling on GPUs . . . . . . . . . . 14

3.2 Wavefront Parallelism . . . . . . . . . . 15

3.2.1 Square VS Non-Square . . . . . . . . . . 17

Chapter 4 Related Work . . . . . . . . . . 19

4.1 Higher-Dimensional Dynamic Programming . . . . . . . . . . 19

4.2 Wavefront Parallelism Optimization . . . . . . . . . . 20

4.3 Time-Space Tiling for Stencil Computation . . . . . . . . . . 23

Chapter 5 Optimizing Higher-dimensional DOACROSS Parallelism . . . . . . . . . . 28

5.1 Introduction . . . . . . . . . . 28

5.2 Tiling-Like Data-Partitioning Scheme . . . . . . . . . . 30

5.3 Dynamic Programming in the Parallel PTAS . . . . . . . . . . . . . . . . . . . . 33
5.4 GPU Implementation and Analysis . . . . . . . . . . . . . . . . . . . . . . . . . 37
5.4.1 Design and Challenges . . . . . . . . . . . . . . . . . . . . . . . . 39
5.4.2 Two-level Fine-grained Parallelism . . . . . . . . . . . . . . . . . . 43
5.4.3 Analysis of the Dynamic Programming . . . . . . . . . . . . . . . . . 45
5.5 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
5.5.1 Experimental Setup . . . . . . . . . . . . . . . . . . . . . . . . . . 47
5.5.2 Analysis of Results . . . . . . . . . . . . . . . . . . . . . . . . . . 48
5.6 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
Chapter 6 Optimizing Wavefront Parallelism with Non-Square Tiling . . . . . . . . 55
6.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
6.2 Problem Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
6.2.1 Low Cache Hit Rates . . . . . . . . . . . . . . . . . . . . . . . . . . 59
6.2.2 Advantages of Shared Memory . . . . . . . . . . . . . . . . . . . . . 60
6.2.3 Barriers to Shared Memory Use . . . . . . . . . . . . . . . . . . . . 60
6.3 Design and Challenges . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
6.3.1 Design Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
6.3.2 Tile Concurrency and Synchronization . . . . . . . . . . . . . . . . . 63
6.3.3 Concurrency VS Data Locality . . . . . . . . . . . . . . . . . . . . . 66
6.3.4 Shared Memory Efficiency . . . . . . . . . . . . . . . . . . . . . . . 67
6.3.5 Synchronization Counter . . . . . . . . . . . . . . . . . . . . . . . . 70
6.4 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
6.5 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73

6.5.1 Wavefront Applications . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
6.5.2 Test Cases and GPU Environment . . . . . . . . . . . . . . . . . . . . . . . 75
6.5.3 Memory Subsystem Efficiency . . . . . . . . . . . . . . . . . . . . . . . . 76
6.5.4 Performance: Multiple Tile Sizes . . . . . . . . . . . . . . . . . . . . . . . 79
6.5.5 Performance: Cache vs Shared Memory . . . . . . . . . . . . . . . . . . 80
6.6 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
Chapter 7 Time-Skewed Tiling Optimization for High Order 2D Stencil Computations on GPUs . . . . . . . . . . . . . . . . . . . . . . . . . . 84
7.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
7.2 Background and Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . 88
7.2.1 Disadvantages of Different Temporal Tiling . . . . . . . . . . . . . . . . 88
7.2.2 Difficulties of Using Existing Solutions . . . . . . . . . . . . . . . . . . 89
7.2.3 Concurrency Modeling on GPUs for Time-Skewed Tiling . . . . . . . 90
7.2.4 Our Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91
7.3 Design and Challenges . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
7.3.1 Two-level Parallelism . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
7.3.2 Stream Processing Scheme . . . . . . . . . . . . . . . . . . . . . . . . . . 95
7.3.3 Two-level Lock System . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
7.3.4 Data Access Pattern . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101
7.4 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102
7.4.1 Dependency Array Structure and Transfer . . . . . . . . . . . . . . . . . 103
7.4.2 Stream Indexing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104
7.4.3 Code for Lock Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . 106

7.4.4 Flow of Tile Processing . . . 106
7.5 Experimental Evaluation . . . 108
7.5.1 Experimental Background and Setup . . . 108
7.5.2 Experimental Results and Analysis . . . 109
7.6 Summary . . . 115
Chapter 8 Conclusion . . . 116
Chapter 9 List of Publications . . . 119
References . . . 120
Abstract . . . 133
Autobiographical Statement . . . 137

# LIST OF TABLES


Table 1 DP-table Size = 3456 . . . . . . . . . . 51

Table 2 DP-table Size = 8640 . . . . . . . . . . 51

Table 3 DP-table Size = 12960 . . . . . . . . . . 51

Table 4 DP-table Size = 20736 . . . . . . . . . . 51

Table 5 DP-table Size = 362880 . . . . . . . . . . 51

Table 6 DP-table Size = 403200 . . . . . . . . . . 51

Table 7 Profiling data for cache-based and shared memory-based mechanisms on GTX 1080 Ti. . . . . . . . . . . 77

Table 8 Execution time (ms): averaged for 100 repetitions. . . . . . . . . . . 79

Table 9 Performance data for the GTX 1080 Ti GPU. . . . . . . . . . . 82

Table 10 Performance data for the Tesla K40 GPU. . . . . . . . . . . 82

Table 11 Number of time steps can be processed for each pair of tile size and distance. . . . . . . . . . . 110

Table 12 Performance in GFLOPS for Moore Neighborhood Pattern. . . . . . . . . 111

Table 13 Performance in GFLOPS for Cross-Shaped Neighborhood Pattern. . . . 112

Table 14 Performance Comparison in GFLOPS for Cross-Shaped Neighborhood Pattern. . . . . . . . . . . 113

Table 15 Performance Comparison in GFLOPs per second for Moore Neighborhood Pattern. . . . . . . . . . . 114

# LIST OF FIGURES


Figure 1 The GPU denotes more processing cores and small cache and less sophisticated flow control. . . . . . . . . . . . . . . . . . . . . . . . . . 9

Figure 2 The wavefront parallelism. . . . . . . . . . . . . . . . . . . . . . . . . 16

Figure 3 Hyperplane Tiling vs Square Tiling. . . . . . . . . . . . . . . . . . . . . 17

Figure 4 Partitioning a $3$-D *DP-table* by a divisor $(3, 3, 3)$. . . . . . . . . . . . . . 32

Figure 5 Dependency graph for $OPT(2, 3)$ . . . . . . . . . . . . . . . . . . . . . 34

Figure 6 The number of non-zero dimensions influence the performance. . . . 50

Figure 7 Average running time vs. the size of *DP-table*. . . . . . . . . . . . . . . 53

Figure 8 Design of Host and GPU Device: solid arrows depict the flow of events and dashed arrows show the data communication. . . . . . . . . . . 61

Figure 9 The comparison of inter-tile concurrency for two different tile sizes. . 64

Figure 10 Data layout in global memory and shared memory. . . . . . . . . . . . 69

Figure 11 Shared memory-based mechanism achieves much higher efficiency rate on multiple metrics except L$2$ cache hit rate. . . . . . . . . . . . . 77

Figure 12 Average difference for four applications. . . . . . . . . . . . . . . . . . 83

Figure 13 Different tiling strategies [1] illustrate the tradeoff between concurrency, computation overhead, and memory latency. . . . . . . . . . . . 86

Figure 14 Design of Host and GPU Device: solid arrows depict the flow of events and dashed arrows show the data communication. . . . . . . . . . . . 93

Figure 15 Intra-tile parallelism and inter-tile parallelism. . . . . . . . . . . . . . 94

Figure 16 Tiles that are in same color are processed simultaneously. . . . . . . . 95

Figure 17 Shared memory organization for dependent elements and tile elements.101

Figure 18 Two patterns lead to different memory efficiency. . . . . . . . . . . . . 109

# CHAPTER 1 INTRODUCTION

A nested loop is a loop that contains another loop or another nested loop. Generally, executing a nested loop requires memory access to one or multiple data entries, which are addressed by the indexes of one or multiple loops. Thus, the memory layout of a nested loop can be depicted in a multi-dimensional block where the dimensions of the block match the depth of the nested loop. In many problems, a nested loop can lead to a tremendous number of operations even if the length of each loop is not unacceptable long. A typical solution for accelerating this process is to split the total workload into small pieces and executing these pieces concurrently. In parallel programming, a nested loop can be split into multiple blocks along either one or more loops. Thus, each block contains a smaller nested loop, which is processed by one OS thread.

The idea of accelerating nested loops on GPUs is similar, but a much greater concurrency can be achieved because of the massive threads capacity. In GPU programming, it is recommended to split a nested loop completely into iterations, instead of a small block of a nested loops and each thread processes one iteration only. The ability of achieving huge concurrency also makes GPUs efficient in solving nested loop problems.

Nested loops are used in a variety of problems and this dissertation discusses the work for optimizing nested loops, in Dynamic Programming (**DP**) problems [2, 3] and stencil problems [4, 5] on GPUs. In dynamic programming problems, each data entry is updated in place, so data dependence exists between the data entries, which prevents the loop from being fully parallelized and leads to loop-level parallelism, often called DOACROSS parallelism [6]. Thus, maximizing concurrency according to the synchronization primitives,

in the statement, is important to optimizing dynamic programming problems. In stencil problems, data entries are updated out of place at each time stamp, so there is no data dependence to restrict the concurrency and the statements within a loop can be executed independently. Thus, nested loops in stencil problems can be fully parallelized as DOALL parallelism [7]. In both cases, poor data locality and irregular workload can hurt the performance even though the code has obtained concurrency that can fully utilize compute capability.

Parallel implementation of some dynamic programming problems follows a specific data access pattern, which leads to *wavefront parallelism* [8]. All previously mentioned performance issues exist in the wavefront parallelism execution process. In most cases, obtaining large concurrency and balanced workload can notably increase the performance. This strategy is applied [9] and achieves a significant speedup on GPUs, which is $25$ times faster than the multicore parallel implementation. The details of the work is presented in chapter 5.

Moreover, having good data locality can further improve the performance when the data entries, required by the computation, cannot completely fit into the $L1$ cache. GPU offers a user-managed cache, called shared memory, to help with this optimization. Under certain circumstances, manually organizing the data entries in shared memory can significantly increase data reuse and reduce the number of memory transactions. In chapter 6, a data locality optimization work is presented, which obtains a sixfold speedup.

Similarly, no-dependence nested loops can be optimized if the data locality is improved without introducing extra high latency. In stencil problems, good data locality is obtained by reusing the cached data entries to calculate as many time stamps as possible. A popular

method, which is called overlapped tiling and studied in many existing works, like [10, 11, 12, 13], obtains both significant concurrency and good data locality. However, it also leads to issues like an imbalanced workload and repeated calculations.

In this dissertation, the tiling methods for optimizing the loop-level parallelism for nested loops on GPUs are presented and discussed. DOACROSS and DOALL parallelism are explored in Dynamic Programming problems and stencil problems, respectively. The proposed tiling approaches resolve data locality and workload balancing issues when dealing with DOALL parallelism and also consider the concurrency for DOACROSS parallelism.

## 1.1 Contributions

In this dissertation, we develop non-overlapped tiling optimization strategies for both DOALL and DOACROSS parallelisms for nested loops on GPUs. We propose three approaches and make the following contributions to the nested loop optimization on GPUs.

- The first contribution is the development of a parallel approximation algorithm for $P \mid\mid C_{max}$ based on the Polynomial Time Approximation Scheme PTAS, which is specially designed for GPUs. The parallel algorithm [14] requires solving a higher-dimensional dynamic programming problem and is based on parallelizing the PTAS [15]. The major challenge of making approximation algorithms, such as the one for $P \mid\mid C_{max}$, efficient on GPUs is to improve the execution time of the higher-dimensional dynamic programming procedure. We take into account the huge computing power offered by modern GPUs and exploit the potential DOACROSS parallelism. The proposed solution resolves the memory issues and improves the thread-level workload balance. To the best of our knowledge this is the first GPU approximation algorithm

for solving the problem on shared-memory systems, proposed in the literature. Our evaluation on the GPU considers as many as nine dimensions in order to assess the optimal decomposition of the various problem instances. We compare the performance of our GPU implementation with that obtained by the OpenMP implementation on a multicore CPU. The results show that our techniques yield an efficient GPU approximation algorithm for $P \mid\mid C_{max}$, with improved performance on large problem instances.

- In the second research component, we exploit the wavefront parallelism for 2D dynamic programming problems and 2D stencil problems, which are updated in place. We design a shared memory-based tiling mechanism to achieve balanced and optimized workloads with minimal overhead compared to existing state-of-the-art approaches. In addition, we provide a methodology for deriving the optimized thread blocks and tiles from the GPU architecture. We design the kernel configuration to significantly reduce the minimum number of synchronizations required and also introduce an inter-block lock to minimize the overhead of each synchronization. Moreover, GPU shared memory is used to replace the L1 cache for improving both spatial and temporal locality, showing that a shared memory-based approach achieves better data locality and coalesced global memory access than a cache memory approach [8] does. To the best of our knowledge, there is no existing work that addresses both the concurrency issue and the memory efficiency issue.

- Our third contribution is the development of a non-overlapped tiling approach for optimizing 2D stencil problems by tiling the time dimension. Unlike the overlapped

tiling approach, non-overlapped tiling does not have the required data entries to perform repeat calculations to reproduce the data lost along the temporal dimension, which makes non-overlapped tiling not feasible for optimizing the time dimension. In our development, we make time dimension tiling possible in the non-overlapped tiling optimization by using GPU shared memory and developing a data pattern for accessing the tiles in shared memory. In addition, we explore the tile-level parallelism in a stream-processing pattern and design a two-level lock system to coordinate the processing sequence. The streaming system tiles the time dimension and does not generate extra calculation and imbalanced workload. Moreover, the proposed non-overlapped tiling approach is developed as a general purpose implementation for solving the most frequently used 2D stencil computations, which have different number of stencil points.

The optimization methods presented in this dissertation use CUDA terminology and use NVIDIA GPUs as experimental hardware. Because the research is using shared memory for improving memory efficiency, our contributions can be obtained when the users can access and manipulate GPU shared memory directly.

## 1.2 Dissertation Outline

The dissertation is organized as follows. Chapter 2 gives a brief overview of NVIDIA GPUs and the related CUDA API. Chapter 3 describes the tiling strategy to optimize DOALL and DOACROSS parallelisms by comparing square tiling to non-square tiling and overlapped tiling to non-overlapped tiling. Related works of my three dissertation projects are presented in chapter 4. Chapter 5 describes the tiling style data-partitioning method

that adjusts the over-sized data matrix into GPU memory and fully utilizes the compute resources for DOACROSS parallelism on high-dimensional dynamic programming. Chapter 6 describes a highly optimized hyperplane tiling approach, which achieves a balanced workload and maximum resource utilization with an extremely low synchronization overhead, for improving wavefront parallelism on GPUs. Chapter 7 extends some of the ideas from Chapter 4 to develop a non-overlapped tiling approach that optimizes the 2D stencil problems by tiling the time dimension on GPUs. The non-overlapped tiling approach can be adapted for different stencil types. At the end, we make a conclusion for our dissertation work and discuss some of the future research directions in Chapter 7.

# CHAPTER 2 OVERVIEW OF NVIDIA GPUS AND CUDA

GPU computing is a popular approach to simulating complex models and performing massive calculations. Compared to the typical multicore CPUs, the developers can obtain much higher throughput from the latest Nvidia GPUs without consuming extra energy. The high throughput implies a greater potential for accelerating the intensive arithmetic calculations on the GPUs. Thus, the extensive research on developing optimal GPU applicable parallel algorithm is required in many fields. Efficient GPGPU, known as General-Purpose computing on Graphics Processing Units, requires good parallelism, memory coalescing, regular memory access, small overhead on data exchange between the CPU and the GPU, and few explicit global synchronizations, which are usually gained from optimizing the algorithms. Besides these advantages, the proper use of some novel properties provided on NVIDIA GPUs can offer further improvement.

On GPU devices, software developers are able to develop general purpose processing applications using the CUDA platform. CUDA (Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) model created by NVIDIA [16]. The CUDA platform provides direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels.

In this chapter, we briefly introduce two different NVIDIA GPU architectures that we use for performance evaluations as well as the major GPU features and relevant CUDA concepts.

## 2.1 GPU Architecture

A NVIDIA GPU is designed with multiple streaming multiprocessors and a global memory system that includes a high bandwidth unified memory [17, 18, 19] and an L2 cache shared by all the multiprocessors. In each multiprocessor, there are hundreds of processing cores, which must execute the same operation concurrently like a vector processing unit. In addition, a multiprocessor is also equipped with an L1 cache and a human-managed cache, called shared memory. In this dissertation, we evaluate the experimental performance and architecture of Kepler [17] and Pascal [18] GPU architectures.

**Streaming Multiprocessors** (SMs) are the part of the GPU that runs CUDA kernels. Each multiprocessor has its own processing cores, cache blocks, registers, and control units. A Kepler GK110 GPU includes 15 streaming multiprocessors and each multiprocessor is equipped with 192 cores and a 64 KB cache block, which can be partitioned between L1 cache and shared memory. A Pascal GP104 GPU includes around 20 streaming multiprocessors and each multiprocessor is equipped with 128 cores. Unlike the Kepler GK110 architecture, a Pascal GPU has a 96 KB shared memory block, which is separate from the individual 48 KB L1 cache, in each multiprocessor.

The 1080 Ti has a total of 3584 CUDA cores, 12 GB global memory, and each CUDA core has a clock rate of 1.63 GHz for a peak performance of around 11.6 TFLOPS. Whereas, the K40 is equipped with 12 GB memory and has 2880 cores at a clock rate of 745 MHz, which sum to a peak performance of 4.29 TFLOPS. In addition, the memory bandwidth of the GTX 1080 Ti and Tesla K40 are 484 GB/s and 288 GB/s, respectively. We divide the throughput by the memory bandwidth to calculate the arithmetic intensity as 24 FLOPs/byte for 1080 Ti

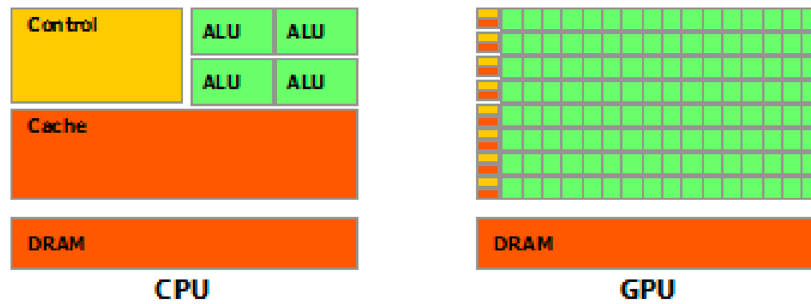



Figure 1: The GPU denotes more processing cores and small cache and less sophisticated flow control.

and 15 FLOPs/byte for K40.

**GPU Cores** The GPU is designed into a highly parallel architecture that has tremendous computational horsepower and very high memory bandwidth [16]. As it is shown in Fig. 1, unlike the traditional multicore CPUs, the GPUs are equipped with more than a thousand cores where each core has a lower clock rate and a much smaller and simpler cache. Therefore, GPUs are designed for intensive data processing rather than data caching and flow control.

**Shared Memory** can be used as a managed cache, which is especially useful when operations are performed on a certain block of data entries that are not stored consecutively. Significant benefit is obtained when using shared memory instead of cache in the study of matrix multiplication [20, 21]. Especially when the cache behavior is unpredictable, storing these data entries in shared memory guarantees data reuse. Also, it is more beneficial to use shared memory on Pascal GPUs because using shared memory frees L1 cache from storing non-contiguous data, which may lead to more efficient L1 cache usage.

## 2.2 CUDA Terminology

CUDA provides programmers with APIs for accessing a GPU's virtual instruction set and computing resources.

**Kernel:** A CUDA kernel consists of the operations, which are executed $N$ times in parallel by $N$ different CUDA threads on one or multiple multiprocessors.

**Thread:** In CUDA programming, threads are the basic processing unit that process the kernel's operations. CUDA threads do not have individual task schedulers, so a group of threads that share the same scheduler perform the same operations. To manage that many threads launched in a kernel, each thread is indexed with a unique ID.

**Thread Block:** In CUDA programming, a kernel function is launched with one or multiple thread blocks where a thread block is a programming abstraction which manages the thread and memory resources. First, a thread block is a cluster of threads, which contains up to $2048$ or $1024$ threads for different architectures. Second, the thread block manages memory access. Threads within the same block can access the same shared memory and L$1$ cache; however, data access between multiple blocks can only be completed in global memory. Third, a thread block resides only on one multiprocessor and the threads within a block share the in-processor memory and registers.

**Warp:** The multiprocessor creates, manages, schedules, and executes threads in groups of $32$ parallel threads called warps [16]. Thread blocks are partitioned into warps and each warp contains threads of consecutive, increasing thread IDs. A warp is like a SIMD (Single Instruction Multiple Data) machine, which can only execute one instruction with multiple data at one time. On Kepler and Pascal GPUs, full efficiency is realized when all $32$ threads

of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp executes each branch path taken, disabling threads that are not on that path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths.

**Stream:** A stream is a sequence of operations that execute in issue-order on the GPU. Thus, kernels scheduled to the same stream are executed in serial. To perform multiple CUDA kernels simultaneously, each kernel has to be launched in a unique stream. In some circumstances, launching kernels in multiple streams helps with obtaining maximum performance by leveraging concurrency and hiding data communication costs.

## 2.3 GPU Properties

The GPU is designed for massive parallelism, which requires many cores to fulfill the concurrency requirement. Thus, the delicate cache design is sacrificed on GPUs to tradeoff the performance and power cost. In order to fully utilize the GPU computing power, the programmers have to pay attention to a couple of GPU properties.

**Memory Hierarchy:** GPUs have a multi-layered memory hierarchy designed for different execution scope. In each thread block, registers are evenly assigned to each thread as its private memory. Then, each thread block has shared memory, which is on-chip and visible to all threads of the block. Shared memory can be used as managed cache with much higher bandwidth and much lower latency than global memory. Thus, data communication between the threads of a block can be performed in shared memory. In addition, each processor is also equipped with an on-chip L1 cache, used for fetching and storing data from off-chip memory. Also, a L2 cache is off-chip and accessible by all the processors.

The largest memory unit on a GPU is global memory, which is off-chip and all threads have access to. On a GPU, memory transactions are initiated by warps, where each warp consists of $32$ threads and a warp can process only one instruction at a time. Global memory is accessed via $32$, $64$, or $128$-byte memory transactions with the first address of each write or read memory transaction memory aligned to a multiple of the transaction size. When a warp performs a global memory access, it coalesces the memory accesses of the threads within the warp into one or more of these memory transactions depending on the size of the word accessed by each thread and the distribution of the memory addresses across the threads. Therefore, highest throughput can be achieved if $32$ threads in a warp request contiguous data entries that are aligned (memory coalescing). If $32$ threads read data entries located at non-contiguous addresses, it is possible that this warp performs up to $32$ memory transactions to complete the memory request.

**Bank Conflict:** Shared memory is organized into banks, which are equally-sized memory modules and should be accessed simultaneously to obtain high bandwidth. However, if multiple addresses of a memory request fall in the same memory bank, there is a bank conflict and the access must be serialized. On the other hand, $n$ addresses of a memory read or write request that fall in $n$ distinct memory banks can be satisfied simultaneously [16].

**Stream Processing:** On GPUs, each stream has its own operating context so that it may execute its commands out of order with respect to the other streams. Each stream can include multiple thread blocks and may utilize one or more GPU processors. The processor assignment of these thread blocks is not guaranteed and different streams could be distributed to the same processor during execution. Stream processing on GPUs is the technique to process multiple kernel functions concurrently, where each kernel launched

in a different stream. Thus, multiple kernel functions can be processed concurrently when each kernel is associated with an individual stream. However, multiple kernel functions that are assigned to the same stream are processed in sequence. Therefore, the number of concurrently processed kernel functions is limited by the number of streams as long as it is within the GPU capability. When we assign only one thread block to each kernel function, a mapping between the blocks and the streams is created. If the number of streams is the same as the number of processors, we will have one thread block processed in each processor because a GPU distributes the thread blocks evenly to all available processors.

# CHAPTER 3 OVERVIEW OF LOOP TILING

The loop tiling technique is used to improve data locality in a large-scale computation to ensure that the data entries in a tile fit perfectly into cache memory so that data can be reused. In addition, loop tiling also helps with obtaining better workload balance when data dependence exists within the nested loop.

## 3.1 Loop Tiling on GPUs

Tiling loops on a GPU can be completed in two steps. First, the loops are partitioned into small chunks in the host code. Thus, the parallelism is split into two level parallelism: inter-tile parallelism and intra-tile parallelism. Second, launching the tiles in one or multiple kernels is used to obtain the inter-tiles parallelism, and intra-tile parallelism is achieved by processing the inner iterations of each tile on the threads.

There are two approaches for obtaining inter-tile parallelism. One is that all the tiles are launched within the same kernel and processed by different thread blocks, so it generates a tile-block mapping. Streaming processing provides the alternative, which launches multiple kernels simultaneously. In this approach, one or multiple tiles are assigned to the thread blocks of a kernel, and multiple kernels are assigned to the streams. Therefore, the tiles that reside in the different stream context can be processed simultaneously.

Because the multiprocessor is equipped with many cores and limited memory capability, the tiling performance on GPUs is determined by the size of each tile. A large tile brings in enough concurrency but its memory requirement may exceed the GPU's memory capacity. On the other hand, a small tile can fit into the in-processor memory block but the concurrent workload may be insufficient for maximizing the use of the compute power.

## 3.2 Wavefront Parallelism

In a two-dimensional data matrix, wavefront parallelism can be visualized as the computation proceeding along diagonal waves, because each data entry is updated based on the already updated neighboring entries occurring in the same row, column, and diagonal if the dependence is uniform. A problem has uniform dependences if each data entry depends on a constant number of dependent subproblems as well as having a constant distance between itself and any one of its dependent subproblems, such as the local sequence alignment problem [2, 22]. Problems like matrix parenthesization [22] and the $P \mid\mid C_{max}$ [9] have non-uniform dependence. Fig. 2 illustrates the wavefront parallelism on a data matrix where each data entry has uniform dependence on its top and left cells. Data entries that reside in the same anti-diagonal, illustrated as dotted lines, could be executed concurrently since they depend on two adjacent entries at one preceding anti-diagonal level, where the dependencies are depicted as arrows. To obtain good concurrency, wavefront parallelism is often applied to applications that have nested loops and uniform dependences across iterations [23, 24, 25, 26]. Wavefront parallelism exploits the potential parallelism but it does not guarantee the efficiency of the parallel computation. The bi/tri-directional dependencies restrict the parallelism in the diagonal order, which can result in an imbalanced workload and poor data locality. To solve large problems, efficient approaches are required to eliminate these performance issues.

### Tiling Technique

The tiling technique is used to improve data locality in a large-scale computation to ensure that the data entries in a tile fit perfectly into cache memory so that data can be

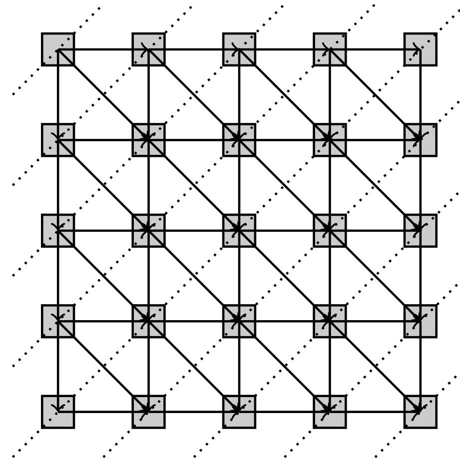


Figure 2: The wavefront parallelism.

reused. Fig. 3 shows two data matrices for the same DOACROSS parallelism, separated by the middle vertical line. The small squares in this picture represent the data entries and each data entry depends on the neighbors to its left and top. Thus, each data entry has a dependence in both outer loop and inner loop, which results in the fact that concurrency can be achieved along the diagonal direction. The dotted lines represent the concurrency that the data entries on the same dotted line can be executed simultaneously. In Fig. 3, the square tiling method generates two-level parallelism, which is shown on the righthand side of the vertical line. Inter-tile parallelism, represented by the tile indices, executes the concurrent tiles on multiple streaming multiprocessors and intra-tile parallelism, represented by the dotted lines, proceeds along diagonals, allowing concurrent data entries to be calculated by CUDA cores.

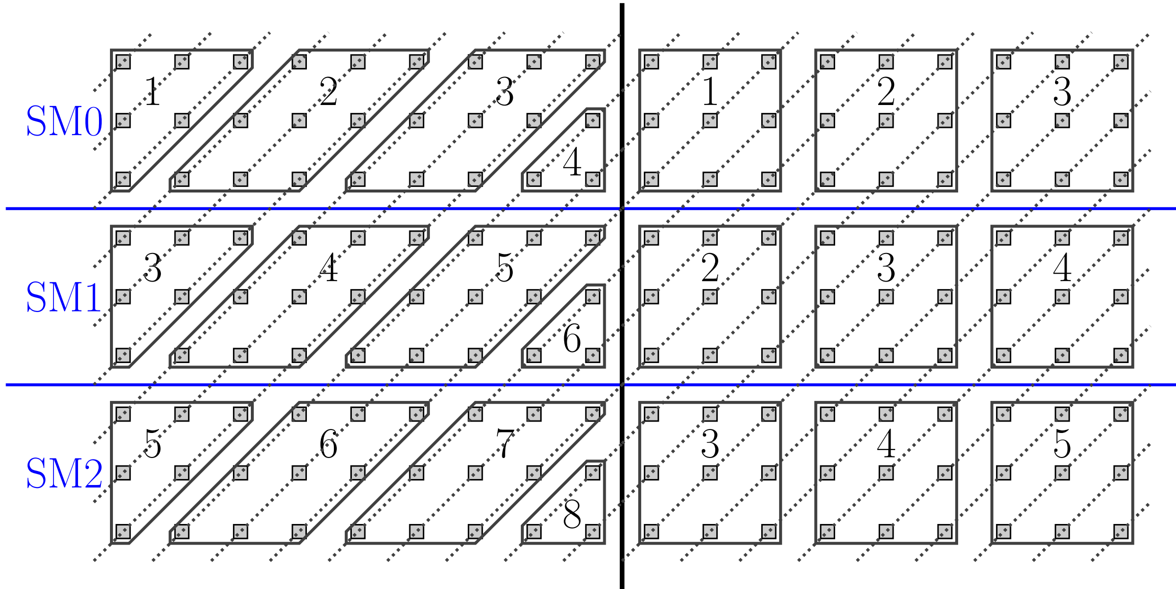



Figure 3: Hyperplane Tiling vs Square Tiling.

#### 3.2.1 Square VS Non-Square

Naturally, loop tiling partitions the loops evenly into multiple square-shaped tiles, as shown on the righthand side of Fig. 3. Square tiling improves the performance of DOALL parallelism because it contributes to efficient cache utilization. However, square tiling may not contribute optimal performance to DOACROSS parallelisms because an imbalanced workload still exists in both inter-tile parallelism and intra-tile parallelism and the average concurrency is relatively low.

Hyperplane tiling splits the data entries along the diagonals, which changes the data layout accessed by the memory operations and eliminates the intra-tile imbalanced workload. As shown in Fig. 3, hyperplane tiling, on the lefthand side of the vertical line, obtains balanced intra-tile workload in the parallelogram tiles. An imbalanced intra-tile workload appears only at the first and last tiles of each row; this overhead is negligible when the matrix size is large. The hyperplane tiling technique achieves a balanced intra-tile workload;

however, it is not necessarily efficient due to the imbalanced inter-tile workload caused by synchronization latency – the hyperplane tiles have to be synchronized with a global barrier to ensure the correctness of inter-tile data communication. Also, the global barrier forces all concurrent tiles to wait for the completion of the longest running task and idles some streaming multiprocessors. In this case, hyperplane tiling is a better option, which changes the data layout accessed by the memory operations and eliminates the intra-tile imbalanced workload.

# CHAPTER 4 RELATED WORK

## 4.1 Higher-Dimensional Dynamic Programming

To address “the curse of dimensionality", some researchers proposed approximations for higher-dimensional dynamic programming [27, 28, 29]. Despite huge advances in parallel computing, the parallel implementation of exact higher-dimensional dynamic programming problems, especially on the GPU, is not as well-studied as two-dimensional dynamic programming. Berger and Galea [30] implemented a multi-dimensional knapsack algorithm on the GPU by introducing the idea of combining coarse-grained parallelism and fine-grained parallelism, and improving the memory coalescing by fixing the number of dimensions. However, their technique works only for small problem sizes, as the size of a higher-dimensional table can grow quickly with the number of dimensions and is likely to exceed the GPU global memory.

Previous work investigated parallel dynamic programming, considering both coarse-grained (multiprocessor clusters) and fine-grained architectures (multicore CPUs, and many-core GPUs). A coarse-grained architecture, such as a multiprocessor cluster, usually achieves efficient local computations because of its powerful computational capabilities and large memory on each processor. But the inefficient inter-cluster communication and unbalanced workload are detrimental to the parallel performance. Some prior research has focused on reducing the inter-cluster communication, such as, partitioning the dynamic programming table into multiple rectangular segments [31, 32, 33]. In addition, work distribution schemes, like block-cyclic [32], have also been employed to balance the workload across processors. Implementing parallel dynamic programming on multi-

core CPUs and many-core GPUs requires a more sophisticated method of work distribution among threads, as a fine-grained architecture has many more computing resources to run the same anti-diagonal levels of sub-problems concurrently, especially for the GPU. A strategy for computing successive anti-diagonals of the dynamic programming table was applied to the multicore CPU to maximize parallel execution [14, 34]. Several researchers [35, 36, 37, 38] extended this strategy to achieve more fine-grained parallelism on the GPU, mostly for two-dimensional dynamic programming problems. Very few researchers investigated accelerating higher-dimensional dynamic programming problems on the GPU.

To address “the curse of dimensionality", some researchers proposed approximations for higher-dimensional dynamic programming [27, 28, 29]. Despite substantial advances in parallel computing, the parallel implementation of exact higher-dimensional dynamic programming problems, especially on the GPU, is not as well-studied as two-dimensional dynamic programming. Berger and Galea [30] implemented a higher-dimensional knapsack algorithm on the GPU by introducing the idea of combining coarse-grained parallelism and fine-grained parallelism, and improving the memory coalescing by fixing the number of dimensions.

## 4.2 Wavefront Parallelism Optimization

### Cache-Oblivious Wavefront

A cache-oblivious technique has been applied to wavefront parallelism recently [39]. As for other matrix-based applications, the data matrix is recursively split into smaller chunks until it can fit into cache so that data locality is improved. However, implementing

this cache-oblivious approach on GPUs is challenging because of the SIMT architecture. The recursive functions would consume limited GPU on-chip memory and it is difficult to distribute recursions to massive threads.

**Compensation-based Parallelism**

The compensation-based method is another approach introduced in [40, 41, 42], which breaks the multi-directional dependencies by ignoring the row-order data dependency. The data entries in the same row are executed concurrently and a correction is then applied to the intermediate results. Without the row-order data dependency, the execution is completed with a balanced workload. However, this method is not generic, because it changes the original data dependency as well as the sequence of computation operators on data entries [42]. Moreover, domain knowledge is required for correcting the final results, which makes this method difficult for users who have no related background.

**Tiling Technique**

On the other hand, the tiling technique is a general solution that can be applied to all problems of this kind. The idea of using the wavefront technique and tiling technique to maximize the innermost loop parallelism was first presented by Wolf and Lam [43]. The wavefront technique transforms the nested loop and makes the innermost loop a *DOALL*, so that maximized parallelism is obtained in the innermost loop. Then, the tiling technique reduces the synchronization cost and improves data locality. The square tiles used in the proposed algorithm cause an imbalanced workload issue on GPUs because the in-tile workload cannot be evenly distributed to the many threads. Also, this algorithm cannot be applied to the GPU directly because it does not map each data entry to the threads.

Di et al. [44, 26] adapt this tiling algorithm to GPUs to accelerate the successive over-

relaxation (SOR)-based applications. In these works, the square tiles are distributed to thread blocks so that each tile is processed by a GPU streaming multiprocessor and each thread processes one or more data entries within the tiles. However, only the tiles on the same diagonal can be processed concurrently because the outermost parallelism is achieved along the anti-diagonal. This leads to a processor-level concurrency issue at the beginning and the end of the execution, since the number of concurrent tiles is fewer than the number of processors. Moreover, due to the limited cache size on GPUs, the use of square tiles cannot improve the data locality in the L1 cache.

Malas et al. [45] accelerate stencil operations with hyperplane tiles on multicore CPUs and achieves about 2 to 3 times speedup comparing to the square tiling implementations. Bednárek et al. [3] apply the hyperplane tiling technique to their GPU implementation, which solves edit distance problem efficiently. The enhanced implementation not only minimizes the imbalanced workload but also optimizes the thread concurrency. Bednárek ignores the memory issue because the intermediate data entries are not needed in this work; therefore, most of the calculations can be performed using L1 cache. Di et al. [46] present a compiler framework that automatically parallelize nested loops on GPUs. In this work, the data matrix is split into hyperplane tiles and data entries of each tile are moved to shared memory before the execution, which enables coalesced memory access. However, this framework can only parallelize nested loops that have no dependence.

A GPU stream processing approach that reduces the synchronization cost and improves the processor-level concurrency is proposed by Belviranli [8]. In this implementation, a row of tiles are processed at the same processor, so local synchronization between each pair of these tiles is sufficient. Thus, the original global barriers can be eliminated and

unnecessary idle waiting is greatly reduced at each processor. But memory efficiency is not optimized. Because concurrent data entries are neither aligned nor consecutively stored in global memory, uncoalesced global memory accesses are inevitable and result in relatively inefficient memory access. Also, due to the limited cache size, it is difficult to reuse the cached data because each warp has to release its cache lines when accessing new data entries.

**Shared Memory**

On GPUs, shared memory can be used as a managed cache, which is especially useful when operations are performed on a certain block of data entries that are not stored consecutively. Significant benefit is obtained when using shared memory instead of cache in the study of matrix multiplication [20, 21]. The tiling technique is applied in matrix multiplication to reduce the number of data entries to be calculated concurrently, so these data entries can fit into shared memory. Due to the unpredictable cache behavior, storing these data entries in shared memory guarantees data reuse. Also, on the GPUs that have separate L1 cache and shared memory, using shared memory frees L1 cache from storing other non-contiguous data, which may lead to more efficient L1 cache usage.

## 4.3 Time-Space Tiling for Stencil Computation

The optimization of stencil computations has been studied for decades and they are still challenging for state-of-the-art multi-core and many-core architectures because of high memory bandwidth requirements. Since the array size is usually much larger than cache capacity, the computation forces subsequent sweeps through the array to reload data, which results in poor temporal locality.

The major solution is time-space tiling, which minimizes the space dimension's cache misses and reuses the cached data along time dimension. This strategy is widely used in most recent work [10, 47, 1, 12, 11, 48, 49, 50, 51, 52]. Various implementations are proposed to achieve the same objective but lead to different effectiveness according to the device architectures and the tradeoff between computation overhead and memory latency.

**Overlapped Tiling**

Overlapped tiling optimization performs redundant operations to address the dependence data when executing more than one time step. Krishnamoorthy [11] characterizes the situations in which tiling inhibits concurrent start and defines overlapped tiling approaches that enable concurrent start in the tiled space and resolve the load imbalance caused by tiling. Rawat et al. present a method for obtaining the optimal tile size and improve the overlapped tiling efficiency by managing GPU memory resources [12, 13]. Then, Rawat further improves the overlapped tiling with a fusion heuristic, which consumes more register resources to enable better temporal locality and reduce memory traffic [1]. Yount [49] specifically designed an overlapped tiling approach for the Intel Xeon Phi processor that utilizes the Phi processor's high-bandwidth memory, and optimizes the computation with SIMD instructions and vector-folding. Meng [52] points out the relationship between the performance and ghost zone of overlapped tiles and provides a method for finding the optimal ghost zone size automatically. Nguyen et al. [10] present a $3.5$D-blocking algorithm to address $3$D stencil problems with overlapped tiling. The $3.5$D tiling performs $2.5$D-spatial tiling and temporal tiling, which makes the $3$D stencil computation no longer memory bandwidth bounded. A flexible load-balancing scheme is also provided for distributing the array elements equally to the threads on both CPUs and GPUs. In ad-

dition, some code auto-generating compilers [53] have been built on top of overlapped tiling, like Halide [54], PATUS [55].

**Split Tiling**

Split tiling enables concurrent start without introducing computation overhead. However, it has trouble with intra-tile parallelism due to the tile shape. Krishnamoorthy [11] develop a split tiling approach by dividing a tile into sub-regions and scheduling the computation and communication to achieve concurrent start and load-balanced execution. Bondhugula [56] proposes a formalized diamond tiling, which is generalized to arbitrary stencil computations. This approach formalizes the conditions for the concurrent start for tiling hyperplanes and provides an approach to find such tiling hyperplane. Grosser et al. [48] split the tiles into a sequence of trapezoidal computation steps and develop an approach for generating split tiling code for GPUs in the PPCG [57] code generator. The proposed algorithm performs split tiling for stencil computations that have an arbitrary number of dimensions without the need for skewing or redundant computations. Malas et al. [45] combine the ideas of multicore wavefront temporal tiling with diamond spatial tiling to reduce memory bandwidth intensity in a 3D space grid, which shows performance advantages in bandwidth-starved computations and is optimized for multicore CPUs. Shrestha et al. [58] develop a jagged polygon tiling technique, which is a variant of diamond tiling. The jagged polygon tiling obtains the advantages of concurrent start and exploits inter-tile locality without compromising intra-tile parallelism.

**Cache Oblivious**

Cache Oblivious [59, 5] solutions are an alternative for resolving frequent memory access issues, which shares the same property of blocking time-space dimensions. In a

cache oblivious implementation, the blocking size is automatically determined to fit into the cache memory, which is achieved by continuously tiling the longer dimension in the recursive kernels. Bilardi [60] develops a cache oblivious algorithm for the problem of simulating large parallel machines on smaller machines in a time-space optimization manner. The algorithm applies to 1D and 2D spaces but does not generalizes to higher dimensions. Frigo [61] presents a more generalized cache oblivious algorithm for stencil computations, which solves arbitrary stencils computations in $n$-dimensional spaces. Tang et al. [62] develop a compiler and runtime system, called Pochoir, for implementing stencil computations on multicore CPUs. The Pochoir compiler is employed with a parallel cache-oblivious algorithm and it translates a domain-specific stencil language embedded in C++ into high-performing Cilk code for general $n$-dimensional stencil computations.

**Time Skewed Tiling**

Time skewed tiling tiles the temporal dimension according to the inter-tile dependence, which usually results in rectangular shapes in 1D computations, parallelogram tiles in 2D, and parallel-piped in 3D. Wonnacott [63] proposed the idea of a time skewing transformation to produce scalable locality for optimizing the computations, where data locality is the dominant issue. Later, this tiling optimization is used to eliminate the redundant computations [64, 65]. Andonov et al. [66, 67] discuss the method for selecting the optimal tile size for time skewing in a 2D grid.

**Hybrid Tiling**

Some attempts for optimizing stencil computations with hybrid tiling approaches are performed to obtain the advantages of concurrent start, load balancing, optimal data locality and good concurrency. Grosser et al. [68] develop a hybrid hexagonal/diamond tiling

approach, which can be used efficiently on GPUs. The hybrid tiling approach tiles the time dimension and outer spatial dimension into hexagonal tile shapes with diamond tiling along the other spatial dimensions. It involves no redundant computations and enables reuse along the time dimension while ensuring adequate parallelism.

**Other Optimizations**

In addition to the tiling approaches, some work also focus on processor-level optimization. Dursun et al. [69] emphasize processor-level optimization techniques in their work. They propose a hierarchical scalable parallelization scheme, which efficiently uses the hierarchical memory levels in Intel multicore CPUs. Similarly, optimization works like tuning memory resources and overlapping memory transaction latency with operations are also performed on GPUs [70, 71, 44]

# CHAPTER 5 OPTIMIZING HIGHER-DIMENSIONAL DOACROSS PARALLELISM

Optimizing DOACROSS parallelism on GPUs is challenging because of the inherently sequential relationship and irregular workload across sub-problems. This is especially the case for higher-dimensional problems, those with three or more dimensions, where dimensionality refers to the number of loops for the nested loop. In this chapter, we present techniques to optimize performance for higher-dimensional DOACROSS parallelism on GPUs. We obtain the DOACROSS parallelism from a higher-dimensional dynamic programming procedure, which has a structured data access pattern. The dynamic programming procedure is obtained from the best existing polynomial-time approximation scheme for the problem of scheduling jobs on identical parallel machines. We demonstrate that the proposed technique highly optimizes the higher-dimensional dynamic programming procedure on GPUs and experimental results show that the optimized GPU implementation outperforms an optimized OpenMP implementation.

## 5.1 Introduction

We study the optimization for the higher-dimensional DOACROSS parallelism for the problem of scheduling jobs on parallel identical machines to minimize makespan. The algorithm used to solve the problem is a Polynomial Time Approximation Scheme (PTAS) based on a higher-dimensional dynamic programming approach, where dimensionality refers to the number of variables in the dynamic programming equation characterizing the problem. Because the dynamic programming procedure accesses structured dependent data across multiple dimensions in each iteration, parallelizing the dynamic programming procedure leads to a regular DOACROSS parallelism and optimizing this procedure pro-

vides us with insight for optimizing higher-dimensional DOACROSS parallelism.

Although algorithms for several dynamic programming problems have already been ported to the GPU, challenges still remain, specially for higher-dimensional cases. In this study, the dimensionality refers to the number of variables in the dynamic programming equation characterizing the problems. Dynamic programming solutions are built from the solutions to sub-problems limiting the degree of parallelism that can be exploited. Furthermore, the workload imbalance among sub-problems increases with the number of dimensions. In addition, solving higher-dimensional dynamic programming problems can quickly exceed the GPU memory.

We consider higher-dimensional dynamic programming algorithms, those of three or more dimensions, and develop techniques to achieve an efficient implementation on the GPU. The proposed techniques resolve the memory issue and improve the thread-level workload balance. To illustrate the challenges and evaluate our techniques, we port to the GPU an approximation algorithm for scheduling jobs on identical parallel machines. This algorithm is a Polynomial Time Approximation Scheme (PTAS) based on an exact higher-dimensional dynamic programming approach. Our evaluation on the GPU considers as many as nine dimensions in order to assess the optimal decomposition of the various problem instances. We compare the performance of our GPU implementation with that obtained by an OpenMP implementation on a multicore CPU. The results show that our proposed tiling-like technique yields an efficient GPU algorithm with better performance on large problem instances, while also addressing the GPU memory limitations.

## 5.2 Tiling-Like Data-Partitioning Scheme

Even if the availability of the many cores on the GPU makes it possible to complete the massive calculations of a higher-dimensional dynamic programming (*DP*) problem in a relatively short time, the large storage requirement is still a challenge to GPU implementations. In the higher-dimensional *DP* problem, it is possible that each subproblem in the *DP-table* requires a big chunk of memory for temporarily holding the data of its dependent subproblems, so that even the execution of a relatively small size *DP* problem can also run out of memory. In this study, we resolve this issue by dividing the huge *DP-table* into many small blocks and performing executions on a number of blocks concurrently. Thus, we can save the memory usage by allocating memory only to the subproblems of these blocks. Here, we call the scheme of dividing the *DP* table, data-partitioning.

We now describe the idea of our proposed data-partitioning scheme for higher-dimensional *DP*. From a geometrical point of view, the partitioning evenly divides a multi-dimensional *DP-table* into multiple small blocks of the same size. The number of small blocks and the size of each block is determined by a vector, which we call $divisor$. A $divisor$ has the same number of dimensions as the multi-dimensional *DP-table*, and the value on each dimension represents the number of segments that this dimension is divided into. Since the subproblems of the higher-dimensional *DP-table* are stored in row-major order, the subproblems of each small block are stored dispersedly in the array of the *DP-table*. Thus, from the data storage point of view, the partitioning scheme reorganizes the storage order of the array to have the subproblems stored within the small blocks.

In the parallel *DP* problems considered in this chapter, the flow of computation moves

along the main diagonal, and the subproblems on each anti-diagonal are independent. Thus, the subproblems on the same anti-diagonal can be processed in parallel (as shown in Fig. 5). In a partitioned *DP-table*, we develop the same computation flow and parallelization on the blocks and the subproblems in each block separately. In other words, our implementation first processes the blocks on the same level in parallel and then parallelize the subproblems on the same in-block anti-diagonal level.

As an example, let us consider a $3$-dimensional *DP-table* $(M, N, L)$ which is evenly divided by a $divisor$, $(a, b, c)$, and each small block can be represented by a vector $(i, j, k)$, where $i \leq a$, $j \leq b$, and $k \leq c$. Thus, these blocks can be classified into different block-levels, and the vector indicates the block-level $l = i+j+k$ that a block belongs to. Here, the term "block-level" refers to the blocks that can be executed concurrently, which is similar to the term "anti-diagonal level" for concurrent subproblems. We can also index each small block with a unique value, which is calculated from $i \times b \times c + j \times c + k$, so that these small blocks can be stored in a sequence with the index. In addition, a subproblem, represented by $(x, y, z)$, belongs to the small block $(i, j, k)$ if $x \in \left[\frac{M \times i}{a}, \frac{M \times (i+1)}{a}\right]$, $y \in \left[\frac{N \times j}{b}, \frac{N \times (j+1)}{b}\right]$, and $z \in \left[\frac{L \times k}{c}, \frac{L \times (k+1)}{c}\right]$. Moreover, we can indicate the subproblem's in-block anti-diagonal level ($l = x + y + z$) and calculate the vector $(i, j, k)$ of the block it belongs to, where $i = \lfloor x / \frac{M}{a} \rfloor$, $j = \lfloor y / \frac{N}{b} \rfloor$, and $k = \lfloor z / \frac{L}{c} \rfloor$. All the subproblems $(x, y, z)$ that belong to a small block $(i, j, k)$ are stored consecutively in row-major order.

Fig. 4 shows an example of the data-partitioning scheme for a $3$-dimensional *DP-table*. The table consists of $6 \times 6 \times 6$ subproblems which are represented by the small cubes. Like the anti-diagonal parallelism in the $2$-dimensional *DP-table*, shown in Fig. 5, the addition of the vector values indicates the anti-diagonal level the subproblems belongs to. Thus, the

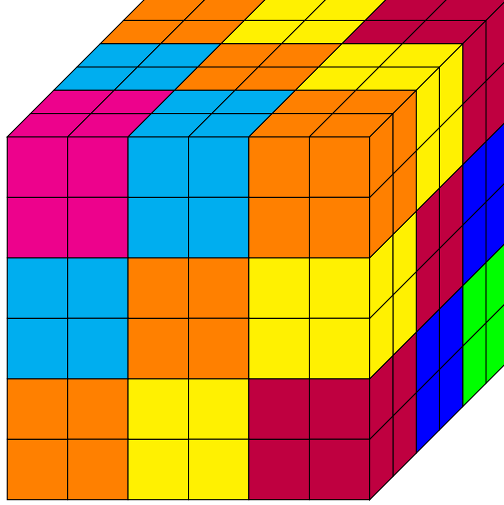


Figure 4: Partitioning a 3-D *DP-table* by a divisor $(3, 3, 3)$.

smallest subproblem $(0, 0, 0)$ is in the first anti-diagonal level, and the largest subproblem $(5, 5, 5)$ is in the $15$th level. After partitioning the *DP-table* with the $divior$, all the subproblems are classified into multiple $3$-dimensional blocks with a block size of $2 \times 2 \times 2$. Then, these blocks are also grouped into $7$ different block-levels which are represented by $7$ different colors, and the blocks with the same color are independent and can be executed concurrently. In addition, $8$ subproblems, in each block, are also classified into $4$ anti-diagonal levels, so that the in-block execution concurrency can also be obtained.

More details of the data-partitioning scheme are given in Section 5.4 where we employ the proposed techniques to a case study consisting of the PTAS algorithm for scheduling parallel identical machines. We will present the implementation of the dynamic programming procedure of the parallel version of PTAS on the GPU and analyze the advantages of using the data-partitioning scheme for resolving the problem efficiently. The block-level is indicated by $l = i + j + k$. Also, the index of the block can be calculated from $i \times N \times L + j \times L + k$. In addition, a configuration $(x, y, z)$ belongs to the block $(i, j, k)$ if

**Algorithm 1** PTAS for $P||C_{max}$ by Hochbaum and Shmoys [15]

1: **Input:** $n, m, \mathcal{T} = \{t_1, ..., t_n\}, \epsilon$
2: $LB \leftarrow \max\left\{\left\lceil \frac{1}{m}\sum_{j=1}^{n} t_j \right\rceil,\ \max_{j=1,\ldots,n} t_j\right\}$
3: $UB \leftarrow \left\lceil \frac{1}{m}\sum_{j=1}^{n} t_j \right\rceil + \max_{j=1,\ldots,n} t_j$
4: $k = \lceil 1/\epsilon \rceil$
5: **while** $LB < UB$ **do**
6: $\quad T = \lfloor (UB + LB)/2 \rfloor$
7: $\quad$ Partition jobs into short and long jobs
8: $\quad$ Round down long jobs to their nearest multiples of $\lfloor T/k^2 \rfloor$
9: $\quad OPT = DP(N, T)$
10: $\quad$ Obtain the schedule for rounded down long job sizes
11: $\quad$ **if** $OPT \leq m$ **then**
12: $\quad\quad UB = T$
13: $\quad$ **else**
14: $\quad\quad LB = T + 1$
15: Return the schedule

$x \in \left[\frac{M\times i}{3}, \frac{M\times(i+1)}{3}\right]$, $y \in \left[\frac{N\times j}{3}, \frac{N\times(j+1)}{3}\right]$, and $z \in \left[\frac{L\times k}{3}, \frac{L\times(k+1)}{3}\right]$.

## 5.3 Dynamic Programming in the Parallel PTAS

We illustrate our proposed techniques on a case study consisting of a parallel approximation algorithm for the problem of scheduling jobs on parallel identical machines to minimize makespan (denoted by $P \mid\mid C_{max}$) proposed by Ghalami and Grosu [14]. Their parallel algorithm requires solving a higher-dimensional dynamic programming problem and is based on parallelizing the PTAS by Hochbaum and Shmoys [15]. In what follows, we call the algorithm presented in [14], the *parallel PTAS*. The basic idea of the PTAS is to partition the set of jobs into two sets, long and short jobs, round down the processing times of the long jobs, and find an optimal schedule for the rounded long jobs using the dynamic programming procedure. The parallelization of the dynamic programming procedure is the core of the *parallel PTAS*.

We now briefly describe the PTAS, presented in Algorithm 1. The algorithm requires as

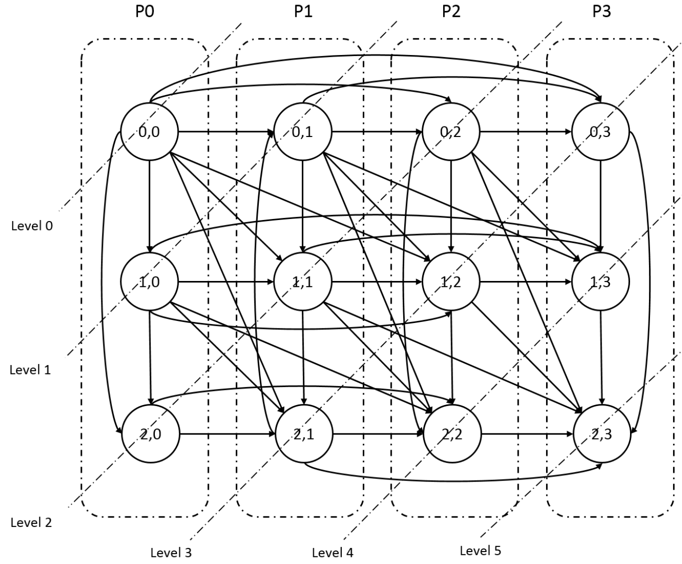



Figure 5: Dependency graph for $OPT(2, 3)$

input, the number of machines, $m$; the number of jobs, $n$; the processing times of the jobs $t_i$, $i = 1, \ldots, n$; and the relative error $\epsilon > 0$. We denote by $\mathcal{T}$ the multiset of jobs' processing times, i.e., $\mathcal{T} = \{t_1, ..., t_n\}$, and assume that all jobs' processing times are positive integers. The algorithm starts by computing the lower and upper bounds (denoted by $LB$ and $UB$) on the optimal makespan of the set of $n$ jobs on $m$ identical machines (Lines 2-3).

The algorithm performs a bisection search procedure for a target makespan value $T$ on the interval $[LB, UB]$ and determines a schedule for the long jobs that fits within $T$. Next, it rounds down the processing times of the long jobs to their nearest multiples of $\lfloor T/k^2 \rfloor$, so that long jobs are classified into $k^2$ dimensions, where $k = \lceil 1/\epsilon \rceil$. Then, the algorithm determines the number of jobs of each of the rounded sizes and creates a $k^2$-dimensional vector $N = (n_1, \ldots, n_{k^2})$, where $n_i$ is the number of rounded long jobs. After creating the vector $N$, the algorithm finds a schedule for the long rounded jobs with a makespan within time $T$. This is done by employing the *DP* algorithm which determines the suitable

number of machines to achieve a makespan within $T$. The *DP* algorithm generates the set $\mathcal{C}$ of all possible machine configurations. A *machine configuration* is a $k^2$-dimensional vector $(s_1, \ldots, s_{k^2})$ specifying an assignment of tasks to one machine and satisfying that the total rounded time is within $T$. The recurrence equation of the *DP* is given by

$$\begin{aligned} &OPT(n_1, \ldots, n_{k^2}) = \\ &1 + \min_{(s_1, \ldots, s_{k^2}) \in \mathcal{C}} OPT(n_1 - s_1, \ldots, n_{k^2} - s_{k^2}). \end{aligned} \tag{5.1}$$

where $OPT(n_1, \ldots, n_{k^2})$ is the minimum number of machines sufficient to schedule the set of jobs given by the vector $N$ and leading to a makespan within $T$. The idea behind this recurrence is that a schedule assigns some jobs to one machine and then assigns the rest of the jobs to as few machines as possible. It is important to observe that each entry requires at most $\lfloor 1/\epsilon \rfloor^{k^2}$ time to compute, and that the total number of entries is $n^{k^2}$ Hence, the values of $OPT(n_1, \ldots, n_{k^2})$ are the components of a dynamic programming table. Since $k^2$ is greater than 3, the *DP* procedure falls within the higher-dimensional dynamic programming. The dynamic programming formulation (Equation 5.1) also implies that the subproblems at each level depend on subproblems at more than one previous level. These subproblems correspond to the components of a table which we call the *DP-table*. Figure 5 shows the assignment of the subproblems for the example of a two-dimensional *DP-table* to a parallel system composed of four cores.

The dynamic programming formulation (Equation 5.1) of the PTAS is *non-serial monadic*, which means that there is a single recursive term in the dynamic programming formulation and the subproblems at each level depend on subproblems at more than one previous level. These subproblems correspond to the components of a table which we call the *DP-table*.

**Algorithm 2** *Parallel* $DP(N,T)$ by Ghalami and Grosu [14]

```
1:  Input: N = (n_1, ..., n_{k^2}), T
2:  σ ← (n_1 + 1)(n_2 + 1)...(n_{k^2} + 1)
3:  Let v^i = (v^i_1, ..., v^i_{k^2}) and OPT(v^i_1, ..., v^i_{k^2}) be the i-th entry of DP-table in row-major order,
    where i = 0, ..., σ − 1
4:  parallel for i = 0, ..., σ − 1 do
5:      d_i = 0
6:      for j = 0, ..., k^2 − 1 do
7:          d_i = d_i + v^i_j
8:  end parallel for
9:  n' = n_1 + ... + n_{k^2}
10: for l = 0, ..., n' do
11:     parallel for i = 0, ..., σ − 1 do
12:         if d_i = l then
13:             if i = 0 then
14:                 OPT(0, ..., 0) ← 0
15:                 break
16:             O_{v^i} ← ∅
17:             C_{v^i} ← all machine configurations of vector v^i
18:             for all (s_1, ..., s_{k^2}) ∈ C_{v^i} do
19:                 O_{v^i} ← O_{v^i} ∪ {OPT(v^i_1 − s_1, ..., v^i_{k^2} − s_{k^2})}
20:             min ← ∞
21:             for all o ∈ O_{v^i} do
22:                 if min > o then
23:                     min = o
24:             OPT(v^i_1, ..., v^i_{k^2}) ← min + 1
25:     end parallel for
26: return OPT(n_1, ..., n_{k^2})
```

The major contribution of the parallel PTAS algorithm [14] is related to the *DP*, which is based on two important characteristics of the computation of subproblems. First, the flow of computation moves along the main diagonal, and second, the subproblems on each anti-diagonal (denoted by Level x, in Figure 5) are independent. Thus, the subproblems on an anti-diagonal can be processed in parallel. Figure 5 shows the assignment of the subproblems for the example of a two-dimensional *DP-table* to a parallel system composed of four cores.

The parallelization of the higher-dimensional *DP* is presented in Algorithm 2. The

goal of the algorithm is to fill out the entire higher-dimensional *DP-table*. and find the optimal value of $OPT(N)$. First, the algorithm determines the size of the $DP$-table, $\sigma = \prod_{i=1}^{k^2}(n_i + 1)$ (Line 2). Next, the $P$ processors compute the sums of the distances of the vectors $v^i$, $i = 1, \ldots, \sigma$ in parallel (Lines 4-8). Because of the dependencies between the anti-diagonals, the parallel *DP* algorithm consists of $n' + 1$ sequential iterations, where $n'$ is the number of long jobs. The subproblems on each level $l$ (corresponding to anti-diagonal $l$) can be identified by the same $d_i$ value (Line 12) and executed by $P$ processors in parallel (Lines 11-25). For computing the optimal value of a subproblem, we need to know its dependencies on the preceding subproblems and use them in Equation (5.1). Therefore, the algorithm generates the set $\mathcal{C}_{v^i}$ of all possible machine configurations, $(s_1, \ldots, s_{k^2})$, for vector $v^i$ (Line 17). Next, the algorithm finds the location of all subproblems by searching the entire *DP-table* and reads their optimal values $OPT(v_1^i - s_1, \ldots, v_{k^2}^i - s_{k^2})$. Then, it places the values into multiset $\mathcal{O}_{v^i}$ (Lines 18-19) and determines the minimum among all values of the subproblems currently in $\mathcal{O}_{v^i}$, adds 1 to the minimum and assigns the value to subproblem $OPT(v_1^i, \ldots, v_{k^2}^i)$ (Lines 20-25). The ordering of iterations guarantees that at each level the algorithm already computed all the needed preceding subproblems.

## 5.4 GPU Implementation and Analysis

Since the *DP* procedure is the most expensive component of the PTAS in terms of running time, the parallelization of the *DP* algorithm becomes the major component of our GPU implementation. A straightforward port of the OpenMP implementation of the PTAS [14] to the GPU is inefficient, being about a hundred times slower than the OpenMP implementation. Thus, sophisticated designs using customized techniques are necessary

**Algorithm 3** GPU implementation of the PTAS

1: **Input:** $n, m, \mathcal{T} = \{t_1, ..., t_n\}, \epsilon, proc = 4, dim \in \{3, ..., 9\}$
2: $LB_p \leftarrow \frac{p}{proc} \max\left\{\left\lceil \frac{1}{m}\sum_{j=1}^{n} t_j \right\rceil,\ \max_{j=1,...,n} t_j\right\}, p = 0, ..., 3$
3: $UB_p \leftarrow LB_{p+1}, p = 0, ..., 2$
4: $UB_{proc-1} \leftarrow \left\lceil \frac{1}{m}\sum_{j=1}^{n} t_j \right\rceil + \max_{j=1,...,n} t_j$
5: $k = \lceil 1/\epsilon \rceil,\ LB = LB_0,\ UB = UB_{proc-1},\ count = 0$
6: **while** $LB < UB$ **do**
7:     **for** $p = 0, ..., proc - 1$ **do**
8:         $T_p = \lfloor (UB_p + LB_p)/2 \rfloor$
9:         Partition jobs into short and long jobs
10:         Round down long jobs to nearest multiples of $\lfloor T/k^2 \rfloor$
11:         Create a $k^2$-dimensional vector $N = (n_1, ..., n_{k^2})$
12:         $OPT_p = Partition(N, T_p, dim, p)$
13:     **for** $i = 0, ..., proc - 1$ **do**
14:         **if** $OPT_0 \le m$ **then**
15:             $UB = T_0$
16:             $LB = LB$
17:             $OPT = OPT_0$
18:         **else if** $OPT_{proc-1} > m$ **then**
19:             $UB = UB$
20:             $LB = T_{proc-1}$
21:             $OPT = OPT_{proc-1}$
22:         **else if** $OPT_i > m\ and\ OPT_{i+1} \le m$ **then**
23:             $UB = T_{i+1}$
24:             $LB = T_i$
25:             $OPT = OPT_{i+1}$
26:     $OPT_Array[count] = OPT$
27:     $count \leftarrow count + 1$

for achieving good performance on the GPU. Our GPU implementation of the PTAS and the higher-dimensional *DP* procedure are illustrated in Algorithms 3, 5, and 4.

The GPU PTAS is designed similarly as in Algorithm 1 and differentiated by the distinct execution scopes. In the GPU implementation of PTAS, presented in Algorithm 3, the $[LB, UB]$ interval is equally divided into four independent segments. The bisection search and the *DP* procedure is executed concurrently on each of these segments. Algorithm 5 presents the procedure for partitioning the higher-dimensional *DP-table* and the memory restructuring. Algorithm 4 shows the implementation of the higher-dimensional *DP* pro-

cedure and the work distribution which is designed for achieving the maximum execution concurrency.

In the rest of the chapter, we will use the term 'configuration' to refer to a subproblem of the higher-dimensional dynamic programming.

### 5.4.1 Design and Challenges

The high-dimensionality of dynamic programming poses significant challenges to GPU implementations and restricts the GPU performance due to two major issues. First, the dependencies among configurations are more complicated than those in the case of two-dimensional dynamic programming algorithms. In a two-dimensional dynamic programming table, a configuration only depends on the sub-configurations corresponding to three directions, horizontal, vertical, and diagonal. When this is applied to $n$-dimensional tables, a configuration can be updated from the sub-configurations that correspond to $n(n+1)/2$ directions. Thus, a configuration has many more potential sub-configurations, which are stored dispersedly in the higher-dimensional memory structure. The scattered memory access, called strided access, leads to low effective bandwidth (bus) utilization. The worst case happens when only one thread in the warp gets the requested data at each cache line. Thus, the warp reads data from the memory in a sequential manner, which can lead to significant overhead when the warp fetches data from the global memory, which is more likely to happen with large size instances. Second, the configurations in the same anti-diagonal level may have different numbers of sub-configurations because of the various dimensional structures among them. Consider the following example. The 3-dimensional configurations $(1, 2, 1)$ and $(0, 0, 4)$ are in the same anti-diagonal level because the sums of their dimensional sizes are the same, but configuration $(1, 2, 1)$ has $11$ sub-configurations,

and $(0, 0, 4)$ has only $4$. Since we use as many threads as possible to achieve the maximum concurrency when scheduling the configurations, in the same anti-diagonal level, to separate threads, the unequal workloads among the threads result in thread-level workload balancing issues. Furthermore, if the index of a sub-configuration, which represents its position in the memory of the dynamic programming table, is unknown during run time, it is necessary to iterate through the *DP-table* and search for the sub-configurations, shown in Algorithm 2 (Line $18$). Since the maximum size of the iteration is the same as the size of the *DP-table*, the search function can be time-consuming and becomes an additional major bottleneck.

**Algorithm 4** $GPU_DP((b_1, ..., b_{k^2}),\ block_offset)$

1: **Input:** $b^i = \{b_1, ..., b_{k^2}\}, block_offset$
2: Let $v^i = (v^i_1, \ldots, v^i_{k^2})$ and $OPT(v^i)$ be the $i$-th entry of *DP-table*
3: $b^i_offset \leftarrow block_offset$
4: $\#(AntiDiag_lvl) \leftarrow (block_size[1] + ... + block_size[k^2] + 1)$
5: **for** $lvl = 1, ..., \#(AntiDiag_lvl)$ **do**
6: $sizeof(lvl) \leftarrow$ number of configurations at each $lvl$
7: $FindOPT\langle\langle\langle gridSize, \frac{sizeof(lvl)}{gridSize}\rangle\rangle\rangle(b^i_offset)$
8: $b^i_offset \leftarrow b^i_offset + sizeof(lvl)$
9: Kernel synchronization and memory updates
10:
11: $FindOPT(b^i_offset)$ :
12: $tid = \text{blockDim.x} \times \text{blockIdx.x} + \text{threadIdx.x}$
13: $\#(v^{tid}_subconfig) = 1$
14: **for** $j = 0, ..., k^2 - 1$ **do**
15: ${v^{tid}}_j \leftarrow$ the value at address $b^j_offset + tid \times k^2 + j$
16: $\#(v^{tid}_subconfig) \leftarrow \#(v^{tid}_subconfig) \times ({v^{tid}}_j + 1)$
17: $\mathcal{C}_{v^{tid}} \leftarrow$ all sub-configurations of $v^{tid} = ({v^{tid}}_1, ..., {v^{tid}}_{k^2})$
18: //Get valid multisets $\mathcal{O}_{v^{tid}}$ from kernel *FindValidSub*
19: *FindValidSub*$\langle\langle\langle 1, \#(v^{tid}_subconfig)\rangle\rangle\rangle(v^{tid}, \mathcal{C}_{v^{tid}})$
20: //update *OPT* of the configuration $v^{tid}$ from its subsets' *OPT*
21: *SetOPT*$\langle\langle\langle 1, sizeof(\mathcal{O}_{v^{tid}})\rangle\rangle\rangle(\mathcal{O}_{v^{tid}}, ({v^{tid}}_1, ..., {v^{tid}}_{k^2}))$
22:
23: *SetOPT*$(\mathcal{O}_v, (v_1, ..., v_{k^2}))$ :
24: $tid = \text{blockDim.x} \times \text{blockIdx.x} + \text{threadIdx.x}$
25: Locate the block $b^i$ of vector $\mathcal{O}_v[tid]$
26: **for all** $(c_1, ..., c_{k^2})$ in $b^i$ **do**
27: **if** $(\mathcal{O}_v[tid]_1, ..., \mathcal{O}_v[tid]_{k^2}) == (c_1, ..., c_{k^2})$ **then**
28: $OPT[\mathcal{O}_v[tid]_1, ..., \mathcal{O}_v[tid]_{k^2}] = OPT[(c_1, ..., c_{k^2})]$
29: $min \leftarrow \infty$
30: **for all** $(ns_1, ..., ns_{k^2}) \in \mathcal{O}_v$ **do**
31: **if** $min > OPT(ns_1, ..., ns_{k^2})$ **then**
32: $min \leftarrow OPT(ns_1, ..., ns_{k^2})$
33: $OPT(v_1, ..., v_{k^2}) = min$

To obtain efficient performance on the GPU, we address all the issues discussed above by developing a data-partitioning scheme and applying the scheme to the dynamic programming component of PTAS. The experimental results show that our proposed technique achieves significant performance improvements for the large table size instances.

The implementation of the proposed data-partitioning scheme is illustrated in Algo-

rithm 5. The high level description idea of the data-partitioning was already presented in Section 5.2. The algorithm divides the *DP-table* into multiple higher-dimensional blocks along a number of specific dimensions, and the number of dimensions is defined by the parameter $dim$, which is in the range of $(3, ..., 9)$ in our experiments. In Algorithm 5, the *DP-table* is divided along the largest $dim$ dimensions (Line $10$), and the number of segments that each dimension is divided into are determined by $divisor$. The entries of the $divisor$ array are computed in lines $4$-$9$, based on the largest configuration, $N$, of the *DP-table*. Lines $6$-$8$ present the calculation of the size of each $divisor$'s dimension. The largest $dim$ dimensions of the $divisor$ are set to the largest divisors of $N$ that are smaller than the square roots of the dimensional sizes of $N$. Having the $divisor$, we are able to create an array of blocks that have different scope on each dimension. Then, the algorithm maps all the configurations into these blocks by matching the configurations' dimensional sizes to the scopes of these blocks. To obtain a group of fully functional blocks, the algorithm reorganizes the memory layout of the *DP-table* (Lines $20$-$27$) because the data-partitioning scheme requires the configurations of each block to be stored consecutively. With the newly organized memory layout, the algorithm is able to access the configurations of a block efficiently, which makes the block-level parallelism efficient. The code for classifying the blocks into different block-levels is given in lines $13$ and $15$. Then, the size of the blocks and the number of configurations of each block are calculated for the purpose of memory access (Lines 18-19). At the end, every four blocks of the same block-level are scheduled into four streams separately and executed concurrently.

### 5.4.2 Two-level Fine-grained Parallelism

Due to the limited execution concurrency, the straightforward port of the OpenMP implementation cannot utilize the many-core computing resources efficiently on GPUs. In Algorithm 2, only the subproblems on the anti-diagonal can be processed in parallel. All other iterations for finding the dependent subproblems of a designated subproblem are executed sequentially. A subproblem, in the dynamic programming table of PTAS, consists of a higher-dimensional vector representing a machine configuration.

In the dynamic programming procedure of PTAS, each configuration has a group of sub-configurations from which the job scheduling can be obtained. Thus, all the configurations of the same anti-diagonal level, and all the sub-configurations of the same configuration can be organized into a parent-child structure. Both “parents" and “children" can be distributed across multiple GPU blocks, which leads to more concurrent execution and results in more fine-grained parallelism. With the benefits of using the GPU feature, dynamic parallelism [72], the parent-child structure can be realized as a nested two-level fine-grained parallelism. The two-level nested parallelism is presented in Algorithm 4. Line 5 is the iteration that loops through all anti-diagonal levels of the higher-dimensional block. The kernel function *FindOPT* in line 7 is the “parent" at the first fine-grained level, which is called at every anti-diagonal level and maps all the configurations in the same anti-diagonal level to the GPU threads separately. In the kernel function *FindOPT*, each thread launches two other kernel functions, *FindValidSub* and *SetOPT* (Lines 19, 21).

**Algorithm 5** $Partition(N, T, dim, p)$

1: **Input:** $N = (n_1, ..., n_{k^2}), T, dim, p$
2: $optimal \leftarrow \infty$
3: $f_1 = 1$, $f_2 = 1$, $block_offset = 0$, $offset = 0$
4: $divisor \leftarrow \emptyset$
5: **for** $i = 1, ..., k^2$ **do**
6: $\quad div = \lfloor \sqrt{n_i + 1} \rfloor$
7: $\quad$ **while** $(n_i + 1) \bmod div \neq 0$ **do**
8: $\quad\quad div \leftarrow div - 1$
9: $\quad divisor \leftarrow divisor \cup div$
10: Keep the largest $dim$ dimensions of *divisor*, and set others to $1$
11: Generate the set of configurations $\mathcal{C}$ (*DP-table*) for $N$
12: Generate the set $\mathcal{B}$ of all blocks for the set $\mathcal{C}$
13: **for all** $(b_1, ..., b_{k^2}) \in \mathcal{B}$ **do**
14: $\quad lvl = b_1 + \cdots + b_{k^2}$
15: $\quad \mathcal{B}_{lvl} \leftarrow \mathcal{B}_{lvl} \cup \{b_1, ..., b_{k^2}\}$
16: $\#block_level \leftarrow$ the number of total block-levels
17: **for** $i = 1, ..., k^2$ **do**
18: $\quad block_size[i] = \frac{n_i+1}{divisor[i]}$
19: $\quad jobsPerBlock \leftarrow jobsPerBlock \times (block_size[i] + 1)$
20: **for all** $(c_1, ..., c_{k^2}) \in \mathcal{C}$ **do**
21: $\quad$ **for** $i = k^2, ..., 1$ **do**
22: $\quad\quad block[i] = \lfloor \frac{c_i}{block_size[i]} \rfloor$
23: $\quad\quad block_offset \leftarrow block_offset + block[i] \times f_1$
24: $\quad\quad f_1 \leftarrow f_1 \times divisor[i]$
25: $\quad\quad offset \leftarrow offset + (c_i - block_size[i]) \times f_2$
26: $\quad\quad f_2 \leftarrow f_2 \times block_size[i]$
27: $\quad \mathrm{M_offset}_{(c_1,...,c_{k^2})} \leftarrow block_offset \times jobsPerBlock + offset$
28: Reorganize $\mathcal{C}$'s memory layout with $\mathrm{M_offset}_{(c_1,...,c_{k^2})}$
29: **for all** $lvl < \#block_level$ **do**
30: $\quad$ **for all** $(b_1, ..., b_{k^2}) \in \mathcal{B}_{lvl}$ **do**
31: $\quad\quad GPU_DP\langle\langle\langle 1, 1, 0, streams \rangle\rangle\rangle((b_1, ..., b_{k^2}), block_size)$
32: $cudaMemcpy(h_opt, d_opt, size, DeviceToHost)$
33: **if** $optimal > h_opt$ **then**
34: $\quad optimal = h_opt$
35: **return** $optimal + 1$

These two kernel functions are the “children" at the second fine-grained level. *FindValidSub* helps finding the valid sub-configurations of the configuration, distributed to the “parent" thread, from all possible options (Line 19). Then the *OPT* of every valid sub-configuration is discovered from the dynamic programming table in function *SetOPT* (Lines

26-28), and the sub-configuration with the minimum *OPT* is used to update the *OPT* of the configuration (Lines 30-32).

The fine-grained two-level nested parallelism further increases the execution concurrency to near maximum and achieves some speedup; however, it is still inefficient compared to the existing OpenMP implementation, especially for large instances. Thus, additional changes are needed to underlying dynamic programming components of PTAS, the most time-consuming part of the application, to make it efficient on the GPU.

### 5.4.3 Analysis of the Dynamic Programming

In a fine-grained parallel dynamic programming implementation, especially when running on the GPU, many cores may stay idle for a considerable amount of time, as many anti-diagonal levels do not have enough work to fully occupy all computing resources. This is inevitable in fine-grained parallelism, but our data-partitioning scheme can alleviate the concurrency loss by improving the bus utilization for each warp when there are free computing resources available. For example, when no data-partitioning scheme is used, an anti-diagonal level of 32 configurations is scheduled to a warp, and thus, each thread in a warp executes on one configuration. With our data-partitioning implementation, the anti-diagonal level is divided into $b$ blocks, and each block needs a warp to work on the partial configurations. Thus, each warp has $32/b$ active threads on average. If $q$ cache lines are required when no data-partitioning scheme is employed, the cache line requirements of each warp of the data-partitioning implementation can be reduced to $q/b$.

Our data-partitioning scheme also addresses the thread-level workload imbalance issue. In the same example, instead of synchronizing all 32 threads of the same warp at the end of the anti-diagonal level, a synchronization, shown in line 9 in Algorithm 4, of

only $32/b$ threads is required by each block. Because the blocks of the same block-level are independent when they are executed concurrently, the in-block synchronization has no effect on other blocks, which implies the warps that have less work finish earlier than running them together in one warp. In addition, a block can continue its execution to the next in-block anti-diagonal level, without creating a race condition with the other blocks. This may improve the block-level workload balance because the overhead of one dense anti-diagonal level can be amortized by its following light levels, and vice versa. Therefore, the overhead of the warp-level synchronizations can be reduced. If all anti-diagonal levels that have heavy workload are in the same block, other blocks have to wait the completion of this block. In this case, the overhead of synchronizing these blocks is the same as the overhead of synchronizing the corresponding anti-diagonal levels when no data-partitioning scheme is used. However, using more warps also reduces the maximum computation capability because many threads are forcibly scheduled but have no work. Thus, the improvement on the effective bus utilization and the workload balance can only be obtained when there are idle cores available.

However, the use of a data-partitioning scheme also causes side-effects. First, it reduces the maximum parallelism which makes it not very efficient on small problem instances. Second, to access the correct memory address of a block, the data-partitioning scheme requires some calculations for the memory offset before the block execution is launched. Therefore, the blocks of the same block-level cannot be scheduled concurrently in one kernel call. Instead, these blocks are launched separately in different kernels (Lines 29-31 in Algorithm 5), and these kernels are scheduled sequentially in the default stream. To obtain block-level concurrency, we distribute the blocks of the same block-level, displayed

in Fig. 4 as blocks of the same color, into $4$ streams in a cyclic distribution manner (Line $31$). Because a CUDA stream has its own computing context, these blocks, in the different streams, can be executed concurrently. In our experiments, applying four streams to each data set provides the best performance for the majority of problem instances.

## 5.5 Evaluation

We investigate the performance of the proposed techniques by performing extensive experiments with the PTAS algorithm and running it on both the multicore CPU and the many-core GPU. We compare the performance of our proposed GPU algorithm, in terms of execution time, against the performance of the OpenMP algorithm. The comparisons are performed on the instances, classified into multiple groups based on their *DP-table* sizes.

### 5.5.1 Experimental Setup

The experiments with the OpenMP implementations are performed on a dual processor system equipped with two Intel Xeon $E5-2697v3$. The GPU implementation is evaluated on an Nvidia K40. The performance of the OpenMP implementation is presented for two configurations, $16$ cores and $28$ cores. The performance data of the GPU implementation is organized by the number of dimensions that the data-partitioning scheme is applied to. We run the experiments, partitioning between $3$ and $9$ dimensions separately, on the same instances. We use GPU-DIM$3$ to GPU-DIM$9$, to denote the cases corresponding to these different partitions. The problem instances are generated using the uniform distribution and considering different numbers of jobs and machines.

According to the approximation algorithm, the maximum number of the *DP-table*'s dimensions is determined by the error rate. In our experiments, we set $\epsilon$ to $0.3$ resulting in

a table with at most $16$ dimensions; however, the number of non-zero dimensions is unknown before the execution because it is determined not only by the jobs' processing times, but also by the target makespan value $T$. Since each interval $[LB, UB]$ has its unique $T$ in one instance, we can get multiple *DP-tables* of different sizes from each instance during the execution, and the running time of the instance is the addition of the running time of each *DP* execution. As the sizes of many *DP-tables* are close, we present the typical sizes to shrink the data set for the purpose of making readable figures and tables.

### 5.5.2 Analysis of Results

We first analyze the running time and the speedup of the GPU implementation, accelerated by the proposed techniques, by comparing it to the OpenMP implementation. The average running time for each considered size of the DP-table is shown in Fig. 7. We select $36$ dynamic programming tables of differing sizes from our much larger data set. These dynamic programming tables are specifically selected for displaying the efficiency of the GPU implementation across the range of sizes in the plots of Fig. 7. The $36$ table sizes are divided into three groups, and the ranges are $(100, ..., 10000)$, $(20000, ..., 100000)$, and $(110000, ..., 500000)$. To improve accuracy, we run the same experiment five times and collect all the performance of the selected table sizes, showing the averages of these five runs in the plots.

In the Fig. 7(a), the OpenMP code (denoted by OMP16 and OMP28) performs much better than the GPU code, because the small instances have much less concurrency and get few benefits from the reduction of the search function. Besides, the execution time of the GPU code is dependent on the number of non-zero dimensions in the *DP-table*. When one instance has a small number of non-zero dimensions, dividing along a large number of

dimensions cannot obtain further speedup. Consider the instance of table size of $3840$ as an example. This instance has $6$ non-zero dimensions, which leads to similar performance to the GPU executions that divide along $6$ to $9$ dimensions. Conversely, in most cases, partitioning along a small number of dimensions cannot obtain good efficiency on the instances that have a large number of non-zero dimensions. Thus, it is not a surprise that the worst performance is obtained in the case of GPU-DIM$3$ because of the less execution concurrency achieved by this implementation. A similar phenomenon occurs in all the instances, and we can conclude that the trade-off between the block complexity and the in-block workload determines the performance of the GPU implementation.

The GPU implementations are more efficient than the OpenMP implementations when the size of the instance's DP-table is larger than $30000$. As illustrated in Fig. 7(b) and 7(c), the best GPU performance is obtained by the implementations GPU-DIM$6$ and GPU-DIM$9$. Compared to the plots in Fig. 7(a) and (b), the lines of the plot in Fig. 7(c) are more regular and stable because the size of the instances in the third group are large enough to occupy all the GPU computing resources through the entire execution.

It is also possible that multiple instances share the same *DP-table* size but have a different number of non-zero dimensions. In this case, the same partitioning settings may perform differently on the instances with the same table size. Since the size of the *DP-table* as well as the number of non-zero dimensions of an instance are unknown before the execution, selecting the appropriate instances that can result in an expected table size and different number of non-zero dimensions is impossible. Therefore, due to space limitations, we filter the instances carefully from our data set and select two table sizes from each groups, used in Fig. 7.

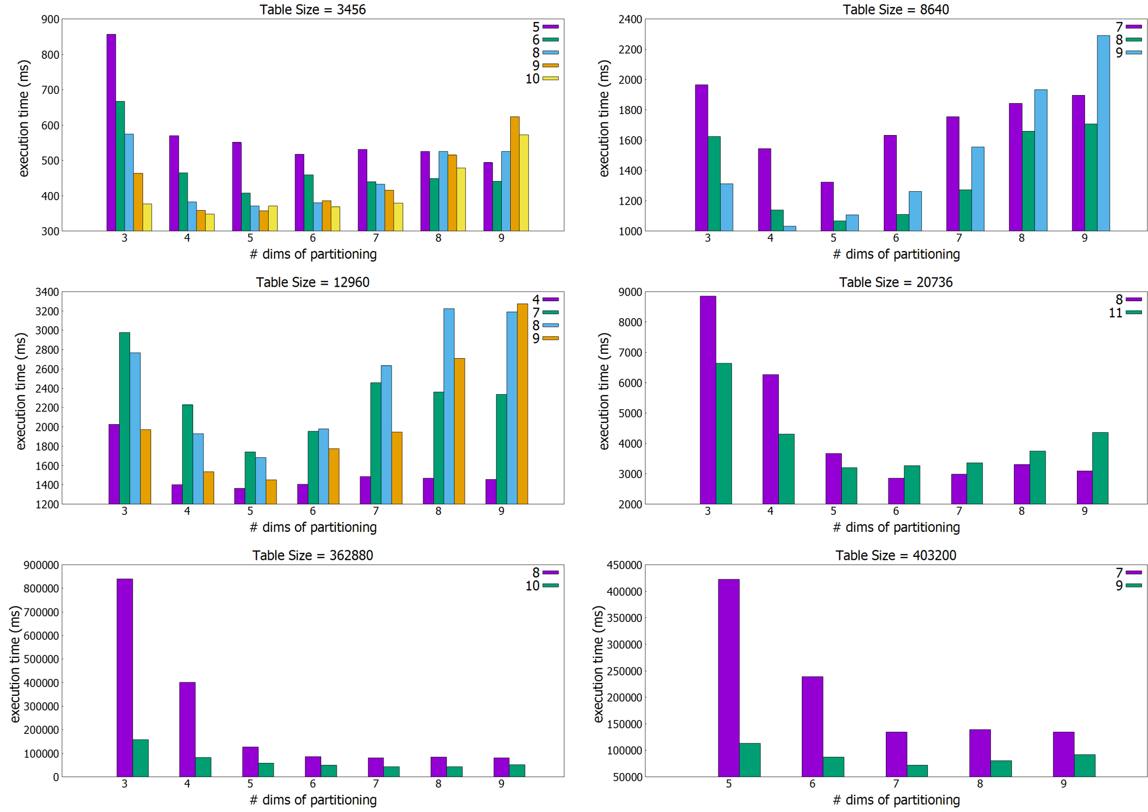



Figure 6: The number of non-zero dimensions influence the performance.

Fig. 6 illustrates the effects of different numbers of non-zero dimensions on the GPU performance. We separate the *DP-tables* of the same size according to the number of dimensions *DP-table* has, represented by the values given in the legend. The values along the horizontal axis are the number of dimensions that the *DP-table* is partitioned into. The performance data show that the best GPU performance of these $6$ selected table sizes is obtained separately when partitioning the table along $5$, $6$, and $7$ dimensions, which is similar to the results presented in Fig. 7. The execution of the table size of $403200$ is too slow when it is partitioned into $3$ or $4$ dimensions, and the running times exceed the wall clock time, which is set to $10,800,000$ milliseconds. Moreover, the *DP-tables* that have fewer dimensions are usually less efficient than the other *DP-tables* of the same table size

having more dimensions. However, exceptions still exist. Thus, we proceed with an in-depth analysis of the size of the in-block dimensions, which appears to be the major factor that can significantly affect the performance.

Table 1: DP-table Size $= 3456$

| #dim | dimension size | GPU-DIM3 | GPU-DIM5 |
|---|---|---|---|
| 5 | $(6,4,6,6,4)$ | $(3,4,3,3,4)$ | $(3,2,3,3,2)$ |
| 6 | $(2,6,3,4,6,4)$ | $(2,3,3,2,3,4)$ | $(2,3,1,2,3,2)$ |
| 8 | $(2,2,4,3,2,6,3,2)$ | $(2,2,2,1,2,3,3,2)$ | $(1,2,2,1,1,3,1,1)$ |
| 9 | $(3,2,3,2,2,2,2,3,4)$ | $(1,2,1,2,2,2,2,3,2)$ | $(1,1,1,2,2,2,2,1,2)$ |
| 10 | $(2,3,2,2,3,3,2,2,2,2)$ | $(2,1,2,2,1,1,2,2,2,2)$ | $(2,1,1,1,1,1,2,2,2,2)$ |

Table 2: DP-table Size $= 8640$

| #dim | dimension size | GPU-DIM3 | GPU-DIM5 |
|---|---|---|---|
| 7 | $(5,3,6,3,4,4,2)$ | $(1,3,3,3,2,4,2)$ | $(1,1,3,3,2,2,2)$ |
| 8 | $(5,6,2,3,2,2,4,3)$ | $(1,3,2,3,2,2,2,3)$ | $(1,3,2,1,2,2,2,1)$ |
| 9 | $(3,3,4,3,2,2,5,2,2)$ | $(1,3,2,3,2,2,1,2,2)$ | $(1,1,2,1,2,2,1,2,2)$ |

Table 3: DP-table Size $= 12960$

| #dim | dimension size | GPU-DIM3 | GPU-DIM5 |
|---|---|---|---|
| 4 | $(3,16,15,18)$ | $(3,4,5,6)$ | $(1,4,5,6)$ |
| 7 | $(4,5,3,6,4,3,3)$ | $(2,1,3,3,4,3,3)$ | $(2,1,1,3,2,3,3)$ |
| 8 | $(3,4,3,4,3,5,3,2)$ | $(3,2,3,2,3,1,3,2)$ | $(1,2,1,2,3,1,3,2)$ |
| 9 | $(3,3,3,2,3,4,2,5,2)$ | $(1,3,3,2,3,2,2,1,2)$ | $(1,1,1,2,3,2,2,1,2)$ |

Table 4: DP-table Size $= 20736$

| #dim | dimension size | GPU-DIM3 | GPU-DIM6 |
|---|---|---|---|
| 8 | $(4,4,6,6,2,3,3,2)$ | $(2,4,3,3,2,3,3,1)$ | $(2,1,2,2,1,1,1,1)$ |
| 11 | $(2,4,2,3,3,3,3,2,2,2,2)$ | $(2,2,2,1,1,3,3,2,2,2,2)$ | $(1,2,2,1,1,1,1,2,2,2,2)$ |

Table 5: DP-table Size $= 362880$

| #dim | dimension size | GPU-DIM3 | GPU-DIM7 |
|---|---|---|---|
| 8 | $(5,6,3,7,6,4,8,3)$ | $(5,3,3,1,5,4,4,3)$ | $(1,3,1,1,3,2,4,3)$ |
| 10 | $(3,3,3,4,5,7,2,3,4,4)$ | $(3,3,3,2,1,1,2,3,4,4)$ | $(3,3,1,2,1,1,2,1,2,2)$ |

Table 6: DP-table Size $= 403200$

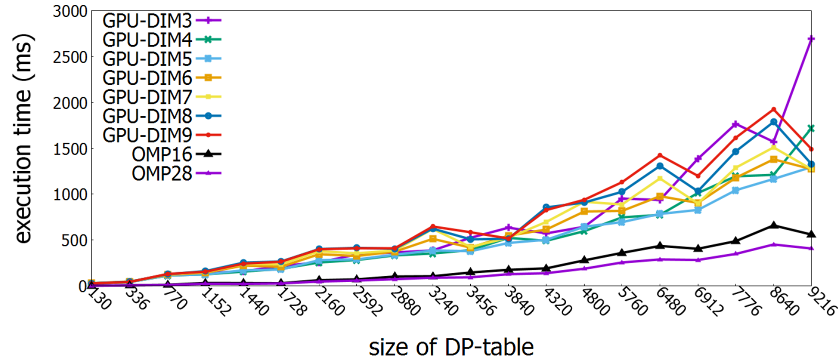
| #dim | dimension size | GPU-DIM3 | GPU-DIM7 |
|---|---|---|---|
| 7 | $(3,10,7,6,4,8,10)$ | $(3,5,7,6,4,4,5)$ | $(1,5,1,3,2,4,5)$ |
| 9 | $(4,5,4,2,3,5,7,3,8)$ | $(4,1,4,2,3,5,1,3,4)$ | $(2,1,2,2,1,1,1,3,4)$ |

To better understand the performance results, we need to investigate the effect of the dimensional sizes of the blocks. Again, we use the $6$ selected table sizes from the example data set and compare the block dimensional sizes of different partition settings for each *DP-table*. In Table 1, we compare the block's dimensional sizes of GPU-DIM3 to the

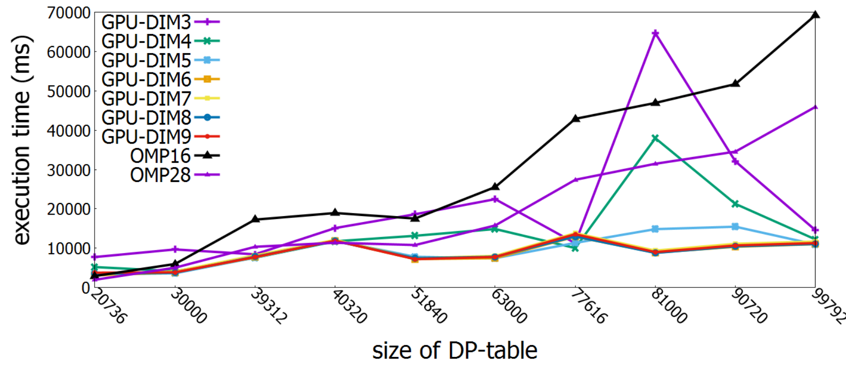
block's dimensional sizes of GPU-DIM5 for each of the *DP-tables* with different non-zero dimensions. Relating the differences to the performance, we can discover how a block's dimensional sizes influence the performance. The size of each block dimension is calculated according to the division, presented in Algorithm 5 (Lines $4$-$9$). The related data is presented in the Tables 1-6. The four columns in each table represent the number of non-zero dimensions, the dimensional sizes of the *DP-table*, the dimensional sizes of the block after partitioning the *DP-table* along three dimensions, and the dimension sizes of the block after partitioning the *DP-table* along a specific number of dimensions, from which the best performance is obtained.

We can conclude from Tables 1-6 and Fig. 6 that the best performance is usually obtained by the execution that has the most regular shaped blocks and the smallest in-block workload. Generally, a large number of non-zero dimensions is helpful to the block's regularity because the high-density dimensions can be scattered by the extra dimensions. In Table 1, even if the *DP-tables* of $5$ or $6$ non-zero dimensions have the same table size, and the executions of these two *DP-tables* have the same number of launched GPU blocks, we can still observe from Fig. 6 that the execution of $6$ non-zero dimensions is more efficient because the one additional non-zero dimension further improves the block regularity, as it is shown in column GPU-DIM$3$. In addition, the block's dimensional sizes of GPU-DIM5 has more regular shapes and less workload than GPU-DIM3 for all the *DP-tables*, and we can see from the Fig. 6 that the performance of GPU-DIM5 is better on all the *DP-tables* of different non-zero dimensions. This conclusion also applies to other *DP-table* sizes. The exceptions appear when there is a big difference between the number of the non-zero dimensions. In Table 3, we can see from column GPU-DIM$3$ and column GPU-DIM$5$ that

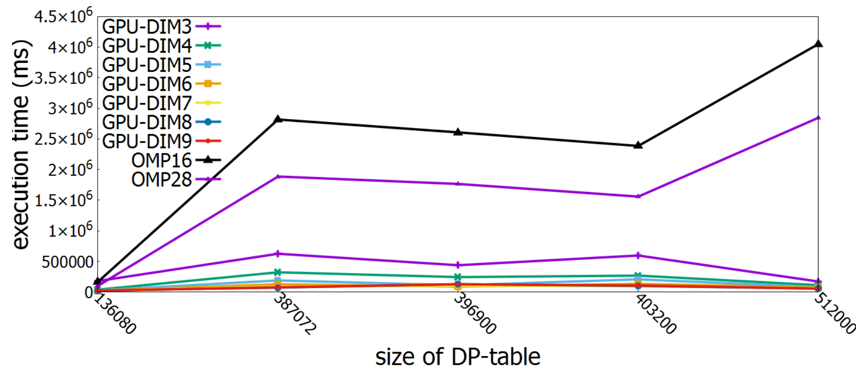
the irregular blocks, $(3, 4, 5, 6)$ and $(1, 4, 5, 6)$, perform much better than the other regular blocks. This is because the irregular blocks have much fewer non-zero dimensions which leads to much smaller in-block workload




(a) Instances with *DP-table* size 100 to 10000.




(b) Instances with *DP-table* size 20000 to 100000.




(c) Instances with *DP-table* size 100000 to 500000.

Figure 7: Average running time vs. the size of *DP-table*.

## 5.6 Summary

The proposed data-partitioning approach is an extension of tiling technique, which improves the GPU performance significantly and makes the GPU implementation perform better than the OpenMP implementation on large-scale higher-dimensional dynamic programming problems. To our knowledge, this is the first data-partitioning scheme specifically designed for addressing the performance and memory issue of higher-dimensional dynamic programming on the GPU. With the techniques, directly applied to the dynamic programming procedure, our study explores the potential of optimizing higher-dimensional DOACROSS parallelism on GPUs.

# CHAPTER 6 OPTIMIZING WAVEFRONT PARALLELISM WITH NON-SQUARE TILING

Wavefront parallelism is a well-known technique for exploiting the concurrency of applications that execute nested loops with uniform data dependencies. Recent research of such applications, which range from sequence alignment tools to partial differential equation solvers, has used GPUs to benefit from the massively parallel computing resources. To achieve optimal performance, tiling has been introduced as a popular solution to achieve a balanced workload. Because matrix-based dynamic programming algorithms usually have serial or non-serial dependences across the table, the massively parallel implementations are less efficient due to the imbalanced workload and cache contention caused by the heavy irregular memory access, which is also true for square tiling optimization. Thus, the non-square tiling technique is widely deployed for solving various scientific problems, like matrix-based dynamic programming problems and specific formulation of some stencil problems. Recent research, conducted on GPUs, achieves massive parallelism by expanding the wavefront loops and overcomes the imbalanced workload issue by splitting the data matrix into multiple hyperplane tiles. However, the use of hyperplane tiles increases the cost of synchronization and leads to poor data locality.

In this chapter, we present a highly optimized implementation of the wavefront parallelism technique that harnesses the GPU architecture. A balanced workload and maximum resource utilization are achieved with an extremely low synchronization overhead. We design the kernel configuration to significantly reduce the minimum number of synchronizations required and also introduce an inter-block lock to minimize the overhead of each synchronization. In addition, shared memory is used in place of the L1 cache. The

well-tailored mapping of the operations to the shared memory improves both spatial and temporal locality. We evaluate the performance of our proposed technique for four different applications: Sequence Alignment, Edit Distance, Summed-Area Table, and 2D-SOR. The performance results demonstrate that our method achieves speedups of up to six times compared to the previous best-known hyperplane tiling-based GPU implementation.

## 6.1 Introduction

Because of the equipped massive cores, achieving high memory efficiency is especially important to achieving full processor occupancy on GPUs, which can be improved by coalesced memory access patterns and data reuse of on-chip memory. However, optimizing memory accesses for applications that have unaligned or nonconsecutive data access patterns, as wavefront parallelism does, are challenging.

Wavefront parallelism is a technique for exploiting parallelism in nested loops. In a two-dimensional matrix, the computations proceed along diagonal waves, because each data entry is updated based on the already updated neighboring entries.

A problem has uniform dependences if each data entry depends has a constant number of dependent subproblems as well as a constant distance between itself and any one of its dependent subproblems, such as the local sequence alignment problem [2, 22]. Conversely, a problem has non-uniform dependences, such as the DP implementations of the matrix parenthesization problem [22] and the $P \mid\mid C_{max}$ problem [9]. During the computation, the execution of wave iterations are serialized to ensure the correctness for updating the data entries; the data entries of each wave iteration can be executed concurrently. Therefore, data dependencies prevent consecutively stored data from being processed in

parallel. Parallel processing the data entries in each diagonal wave requires access to non-contiguous memory addresses, so efficiency is degraded because of the multiple memory accesses required for updating each data entry.

The tiling technique is a general solution that changes the memory access pattern but not the operations, so that the original data dependency is never changed. In some studies [73, 44, 74, 26], the tiling technique is combined with wavefront parallelism to improve the memory efficiency. This approach splits the data matrix into multiple square tiles and assigns the data entries of each tile to a GPU streaming multiprocessor for exploiting data locality. Square tiling improves data locality but causes a serious imbalanced workload issue. The hyperplane tiling approach [75, 76, 77, 3, 45, 46, 26], as a non-square tiling technique, obtains a balanced workload by adjusting the memory access pattern. Hyperplane tiling splits data entries along the diagonals perpendicular to the data dependency, so each diagonal has an equal number of data entries which results in balanced workload and maximizes the core utilization. In addition, a stream processing-based implementation [8], further improves the efficiency of hyperplane tiling by reducing the synchronization overhead; however, intra-tile data locality and coalesced global memory accesses are not achieved.

In some cases, it is recommended to use on-chip shared memory [16] when it is difficult to improve data locality at the hardware-managed cache. Highly efficient solutions have been proposed in [13, 20, 21], which use shared memory instead of regular cache. Intuitively, it may be beneficial to construct the hyperplane tiles in shared memory. However, there is no obvious method to adapt the applications which require the wavefront parallelism technique and the hyperplane tiling technique to shared memory efficiently.

To the best of our knowledge, there is no existing work that addresses both the concurrency issue and the memory efficiency issue. In this chapter, we present a memory-optimized wavefront parallelism technique, which splits the data matrix into hyperplane tiles to achieve high concurrency. To address the memory efficiency issues, we adapt the tile processing to the GPU shared memory and directly manage the data accesses to obtain a coalesced memory access pattern and improve data locality. Because hyperplane tiling leads to an irregular data layout, we propose a general data transformation function that can be applied to wavefront applications to map the hyperplane shaped data block to the shared memory efficiently. Besides, in order to find the best tradeoff between core utilization and the limited shared memory capacity, we propose a general scheme for setting up the kernels on different NVIDIA GPU architectures. In addition, we improve the coarse parallelism workload balancing and minimize the synchronization overhead by processing the tiles using stream processing. The proposed work has the following major contributions:

- We show that a shared memory-based approach achieves better data locality and coalesced global memory access than a cache memory approach does.

- We design a shared memory-based tiling mechanism to achieve balanced and optimized workloads with minimal overhead compared to existing state-of-the-art approaches.

- We provide a methodology for deriving the optimized thread blocks and tiles from the GPU architecture.

- We develop a low-cost barrier lock to minimize the cross-kernel synchronization overhead.

- We evaluate the shared memory approach using four applications, running on NVIDIA GTX 1080 Ti and Tesla K40 GPUs and achieve up to six times speedup compared to the best existing approach that uses cache only.

## 6.2 Problem Statement

It is mentioned above that implementing the hyperplane tiling technique on hardware-managed cache cannot achieve good data locality. Here we explain the reasons that L1 cache reuse is low and global memory accesses are not coalesced. We also point out that replacing the global memory access with the shared memory access is not simple and an innovative implementation is needed for obtaining good shared memory efficiency.

### 6.2.1 Low Cache Hit Rates

GPU global memory is accessed via at most 128 byte memory transactions, which is also the size of a cache line. Because the concurrently executed data entries reside at different rows, as illustrated in Fig. 2, each cache line can hold only one desired data entry when the matrix row size is larger than a cache line. Therefore, the cache size required for storing all the concurrent data entries is 128 times of the size of data entries. As presented in [8], the best performance is achieved when 1024 data entries are executed concurrently. In other words, at least 128 KB cache memory is required to store all these data entries. Besides, extra space is also needed when the dependent data of the first data entry is not included in the same cache line. Considering that Kepler GPUs and Pascal GPUs are equipped with upto 48 KB L1 cache on each processor, which is increased to 64 KB on Turing GPUs and 96 KB on Volta GPUs, it is obvious that none of today's GPUs can keep 128 KB data in L1 cache so that even an optimized cache-based implementation cannot efficiently reuse L1 cache.

On the other hand, because the first data of each row in a hyperplane tile are not aligned to the address of multiple of 32, loading these data leads to repeated uncoalesced memory accesses.

### 6.2.2 Advantages of Shared Memory

Programmers retain control over when and which data entries should remain in shared memory during execution. With the data entries of the tile stored in shared memory, we can target the locations where these calculated results should be updated, so that memory operations no longer access global memory repeatedly and data locality is improved. Besides, more coalesced memory accesses are made. Because unaligned data entries appear only once per memory access request at each row, moving multiple consecutive tiles in one memory request reduces the uncoalesced memory access frequency. In addition, the access to the shared memory is almost perfectly coalesced because the data entries are aligned in the shared memory. Therefore, replacing the use of L1 cache with shared memory is a potential solution for improving the data locality.

Due to the limited shared memory capacity, transferring data between global memory and shared memory requires careful design to obtain high bandwidth utilization.

### 6.2.3 Barriers to Shared Memory Use

Unlike using square tiling in matrix multiplication, applying hyperplane tiling parallelism to shared memory is not that straightforward. Because the size of an optimal tile, which achieves the maximum concurrency, exceeds shared memory capacity, the tile applied to shared memory has to be truncated. However, reducing the tile size is not as simple as cutting the large cache tile evenly and different tile sizes lead to different concurrency and synchronization overhead. To obtain the best performance, it is important to have

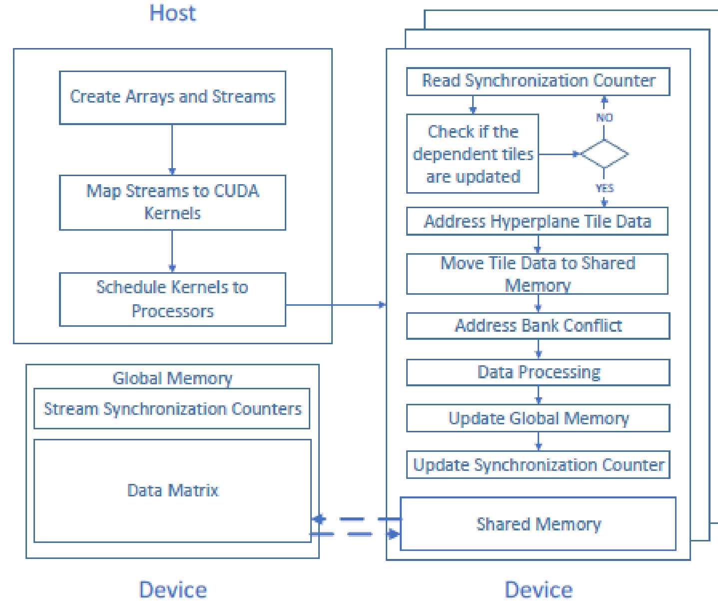



Figure 8: Design of Host and GPU Device: solid arrows depict the flow of events and dashed arrows show the data communication.

a method, which can derive the optimal tile size for the shared memory implementation from the problem size and the processor configuration.

Besides, transferring the hyperplane-aligned data entries between shared memory and global memory in the square pattern is not efficient, even if this transfer pattern is used as the major solution in many other works. We can see on the right hand side of Fig. 3 that including a hyperplane tile of data entries in a square data block leads to a waste of shared memory capacity which increases the data communication overhead. Therefore, a new data transfer pattern is also required to adapt to hyperplane tiling.

## 6.3 Design and Challenges

We present the design overview, which includes the shared memory-based stream processing, an innovative shared memory processing pattern, and an improved lock design, as well as the challenges of using shared memory for wavefront parallelism in this section.

### 6.3.1 Design Overview

Fig. 8 depicts the workflow of the approach. The complete GPU implementation consists of the host, which refers to the CPU and its memory and the device, which refers to the GPU and its memory. On the host, we first create a group of streams and allocate the data matrix to GPU global memory. Each GPU is equipped with multiple processors and we limit the maximum number of streams to the number of processors on the tested GPU. Then, these streams are mapped to all the kernels in a round robin manner, so a processor can process the next row of tiles as soon as it completes the current one. As illustrated in Fig. 9, a row of tiles are scheduled to one kernel, which has been mapped to a corresponding stream. In each kernel function, a row of tiles are fetched and processed in sequence.

On the device, all the processors receive the same instructions and apply these instructions to a different row of tiles concurrently. In each kernel function, thread $0$ first reads the stream index and uses that index to access the corresponding synchronization counter, meanwhile other threads are inactive. The counter value indicates whether the dependent data entries of the current tile are already updated. A tile can only be processed when the counter value is larger than a threshold which is calculated according to the tile index. Thus, thread $0$ reads the counter value repeatedly until it passes the threshold. Then, all threads are activated and use the thread indices as well as some offset values to access the corresponding data entries in global memory. In order to obtain coalesced memory access at shared memory, the data entries, which are misaligned in global memory, are re-positioned in shared memory so that they are stored as a square block instead of a hy-

perplane block. Before processing the data entries, padding has to be added to the data block in shared memory to avoid the bank conflict issue. After these data entries are executed, the results are copied back to global memory. At the end, thread $0$ updates the counter value of the next stream to notify the kernel in that stream that its dependent data entries are updated.

### 6.3.2 Tile Concurrency and Synchronization

Here we present the effect that tile concurrency and tile synchronization can have on performance. Also, we explain how the shared memory-based approach maximizes the tile concurrency and minimizes the synchronization costs. The different types of concurrency and synchronization are distinguished with prefix intra-tile and inter-tile. Intra-tile refers to the inside of a tile and inter-tile targets multiple tiles.

#### Intra-Tile Concurrency

Intra-tile concurrency refers to the number of data entries executed concurrently within the thread block. It is determined by the tile height in a hyperplane tile and contributes to the core utilization for the streaming multiprocessor. The latest NVIDIA GPUs have $128$ CUDA cores in each streaming multiprocessor. In cache-based approaches, the tile height is set to $1024$ to use all $1024$ threads in a thread block because the memory latency can be partially hidden by massive thread operations. However, it is different when the data entries are stored in shared memory. Because the memory access latency is already low in shared memory, we can obtain acceptable core utilization by assigning only $128$ threads or $256$ threads to each thread block.

#### Intra-Tile Synchronization

For wavefront parallelism, shown in Fig. 2, synchronization is required between the

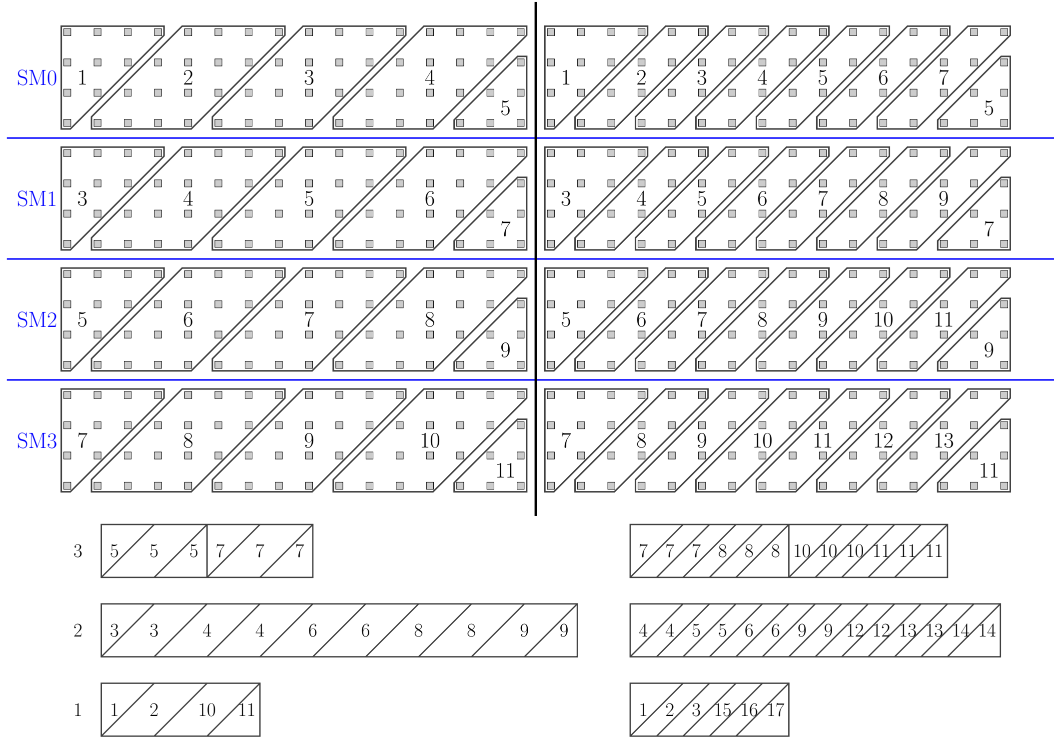



Figure 9: The comparison of inter-tile concurrency for two different tile sizes.

executions of data entries that are located on different adjacent anti-diagonals. Because the tiles of each stream are processed individually on their own processor, we perform local synchronization at each processor separately. Compared to synchronizing all the processors uniformly for each local update, separate local synchronization generates much lower synchronization cost.

**Inter-Tile Concurrency**

Inter-tile concurrency refers to the number of tiles executed concurrently across all the streaming multiprocessors. Since a tile can be executed only after the completion of its dependent tiles, some streaming multiprocessors are idle at the beginning and the end. Therefore, different tile widths effect the SM's concurrency. Fig. 9 illustrates the comparison between two different tile widths. The "wide" tiling, shown on the left, has a tile width

twice that of the “narrow" tiling, shown on the right. We extract all possible concurrency for the two different tiling methods and concatenate concurrent tiles for maximum concurrency as shown in the figure. The narrow tiling obtains higher peak concurrency but less second-level concurrency. Both the narrow tiling and the wide tiling have the same amount of work executed with the minimum concurrency. We consider the narrow tile as the unit workload and its width represents the unit time for executing the workload. Similarly, the triangular tile and the wide tile have twice the workload as well as twice the execution time. For a batch of tiles, we can calculate the approximate execution cost by following Equation 6.1, where $p$ represents all possible concurrency in the execution and $W_p$ is the total work that is executed with $p$ concurrency. In Fig. 9, the cost of the narrow tiling is $21.33$, which is slightly better than the wide tiling's efficiency of $22$.

$$E = \sum_{p} (W_p/p),\ p \in [\text{possible concurrency}] \tag{6.1}$$

**Inter-Tile Synchronization**

There is no overhead for synchronizing the tiles of the same row because these tiles are all executed by the same thread block and the execution of each tile is launched in sequential order. In this case, the inter-tile synchronization is already performed implicitly within each tile and the completion of the previous tile is guaranteed before the execution of the next tile starts. On the other hand, the tiles of different rows are executed in different streams, so the intra-tile synchronization performed within each thread block has no effect on the other thread blocks. Explicit barriers are required for synchronizing the tiles across multiple rows. In Fig. 9, the wide tiling has $5$ tiles in each row and $3$ of them

have a dependence on the previous row. Thus, the kernels of wide tiling have to wait till the valid counter values are updated on three tiles, which introduces less synchronization overhead because waiting for the valid counter value are required on five tiles for narrow tiling.

Processing the narrow tiles introduces much more overhead, due to more synchronization calls and less utilization of shared memory, even though it achieves slightly better inter-tile concurrency. We reach the conclusion that tiling the matrix with a wider tile size of $128 \times 96$ is more efficient than a narrower tile size of $256 \times 48$, where the tile size is represented by height $\times$ width.

### 6.3.3 Concurrency VS Data Locality

We already explained the effect that tile size could make on tile concurrency and synchronization. The tile size matters because tile size determines the number of threads allocated to each thread block. Here we provide further evidence to prove that the tile size also determines the tradeoff between concurrency and data locality. In addition, we propose a general method for determining the optimized tile size that achieves the best performance.

The height and width are limited due to the fixed shared memory size, which is usually at most $48$ KB for each thread block. Because each CUDA warp contains $32$ threads and each thread executes on one data entry at a time, it is optimal that the sizes of tile height and width be multiples of $32$. Thus, the tile sizes, shown as height $\times$ width, are limited to $64 \times 192$, $128 \times 96$, and $256 \times 48$ if each data entry is $4$ bytes long.

In this chapter, we assume that the height and width of each data matrix are powers of $2$. To ensure that the tile evenly subdivides the data matrix, we set the tile height ($T_h$)

and the tile width ($T_w$) to a power of $2$ as well, so the largest eligible tile requires $32$ KB shared memory, which is represented as $T_s$. We calculate $T_w$ by dividing the maximum effective shared memory by $T_h$, which is shown in Equation 6.2. $T_h$ is dominated by the number of cores in each streaming multiprocessor, which is represented by $c$. On modern GPUs, $c$ is either $192$ for the Kepler architecture or $128$ for the architectures released after Kepler. Thus, the optimized $T_h$ is constant at $128$ according to the equation. Even if setting $T_h$ to $128$ cannot fully utilize all $192$ cores on Kepler GPUs, the overall performance is still better because a small amount of computing capability is sacrificed to retain a higher cache (shared memory) per core rate.

$$\begin{aligned} T_h &= 2^{\lfloor \log_2 c \rfloor}, \\ T_w &= 2^{\lfloor \log_2 T_s \rfloor} / T_h. \end{aligned} \tag{6.2}$$

Therefore, the optimized tile size is $128 \times 64$ for most modern NVIDIA GPUs, which is also compatible to the optimized tile size that we concluded from the previous section. Thus, the optimal thread block size is set to $128$ threads. The actual tile size is also effected by the longest distance between a data element and its dependence, so the tile width might be adjusted to ensure that the tile height is no smaller than $128$.

### 6.3.4 Shared Memory Efficiency

Simply replacing L$1$ cache with shared memory is not efficient due to the uncoalesced memory access pattern and bank conflict. Moreover, it is difficult to obtain high bandwidth if data entries are moved to shared memory improperly.

#### Memory Coalescing

In order to have aligned data layout in shared memory, the unnecessary data entries that are in front of the first valid data entry at each row are ignored during the data transfer. We calculate the aligned new addresses with the thread index and row index and store the data entries at the re-positioned addresses to ensure that the intervals between each pair of concurrently executed data entries are constant.

Fig. 10 shows an example of the hyperplane data layout in global memory and square data layout in shared memory. In this example, we assume that each warp can fetch five data entries, which is also the size of a memory transaction. In global memory, two memory transactions are needed to acquire the required data entries, which are circled with dotted lines and full lines, at each row except the last one. After the required data entries are re-positioned in shared memory, only one memory transaction is needed per row.

The lefthand side depicts the data layout in global memory which leads to uncoalesced memory access. On the right side, all the concurrent data entries are aligned in the same column, so coalesced memory access can be obtained as long as the number of data entries at each row is multiple of $32$. The aligned shared memory enables coalesced memory access and with this mapping, all available threads access a column of elements, which is more efficient because of less index computation. After the calculation, the results have to be updated in global memory before proceeding to the next tile. The sequence of data transfer and execution for each tile is shown in Algorithm 6.

**Padding and Bank Conflict**

The actual number of data entries that are required for the execution is larger than the optimized tile size. The dependent data entries, which are enclosed in the dotted lines

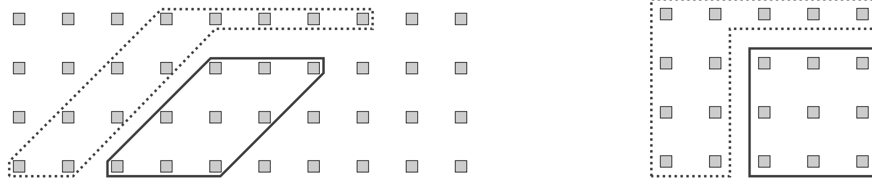


Figure 10: Data layout in global memory and shared memory.

should also be added to the size-optimal tile as padding entries and added to the front of each tile dimension, as shown in Fig. 10. Loading some extra elements as padding entries to each row of data can also avoid bank conflicts. Thus, the actual padding entries, which are added to each row, consists of the required dependent data entries and the possible extra elements used to avoid bank conflicts. This additional padding size should be relatively prime to the number of shared memory banks, which is $32$ on modern GPUs. Therefore, the actual tile size, moved to the shared memory, should be increased to $129 \times 67$.

**Data Transfer and Overhead**

Data transfer between global memory and shared memory is time consuming even if the tile size is small. In some applications, like sequence alignment, only the padding data needs to be moved to the shared memory before execution because each data entry depends only on its top and left neighbors. However, all the data entries in the data matrix have to be transferred to shared memory for the applications that process stencil-like computations. The data movement is completed in the function, shown in Algorithm 6 [line: 5, 8]. In the function, when we transfer the data between the shared memory and the global memory, we achieve the best bandwidth by moving a chunk of $64 \times 16$ data entries at a time with $1024$ threads. We cannot allocate more than $1024$ threads because we assign only one thread block to each streaming multiprocessor. As shown in Fig. 3,

the triangular tiles are inevitable at the beginning and end of each row and these have an imbalanced workload. We process the triangular tiles in the same way as processing the square tiles, because the intra-tile parallelism is the same as the original wavefront parallelism. Instead of moving the triangular tiles, we copy the minimum square tile that contains a triangular tile between the two memories. If the square tile is too large to fit into the shared memory, we split it into two rectangles and perform the operations on each of them sequentially.

### 6.3.5 Synchronization Counter

Synchronizing the blocks with a global barrier is not efficient because the dependence only exists between each pair of consecutive rows. The completion of one tile implicitly confirms completion of the execution of all preceding tiles in the same row. Therefore, to safely proceed with executing a tile, it is not necessary to synchronize all the tiles in the same anti-diagonal level. Instead, we just need to confirm completion of the dependent tile that resides in the prior row. However, keeping track of the status, which indicates the completion of processing the dependent tiles, for all the tiles is hard to implement and still not efficient. In fact, we just need to record the status of each row with a counter value, which indicates the number of already updated tiles. Therefore, in each row, the kernel increments the row counter as soon as it finishes the executions of a tile. In addition, the kernel reads the counter value of the previous row and checks if the dependent tiles are updated before processing each tile.

To avoid the false sharing issue, the lock array is declared with the “volatile" keyword, which ensures that cache copies stored in other L1 Caches will be informed when one of them is changed. We declared the lock array with the “volatile" keyword to avoid the false

sharing issue. The “volatile" keyword prevents the array from being cached, which means that all memory accesses revert to global memory, ensuring that we see the change when another streaming multiprocessor updates the value. No operations will be performed to these copies until the change is completely updated. Compared to PeerWave, the proposed lock design is simpler and more efficient. The lock for each row is not an array, but instead a counter of the proposed lock array, which is much smaller and requires fewer operations.

## 6.4 Implementation

**Data Transfer**

The data movement is completed in the function, shown in Algorithm 6 [line: 5, 8]. When we transfer the data between the shared memory and the global memory, we achieve the best bandwidth by moving a chunk of $64 \times 16$ data entries at a time with $1024$ threads. We cannot allocate more than $1024$ threads because we assign only one thread block to each streaming multiprocessor.

We process the triangular tiles in the same way as processing the square tiles, because the intra-tile parallelism is the same as the original wavefront parallelism. Instead of moving the triangular tiles, we copy the minimum square tile that contains a triangular tile between the two memories. If the square tile is too large to fit into the shared memory, we split it into two rectangles and perform the operations on each of them sequentially. Even if the overhead for moving the square tile is higher than the overhead for moving the triangular tiles, our implementation is more generic and requires less programming effort. Besides, the overhead for moving the triangular tiles becomes negligible when we are dealing with larger matrices.

**Algorithm 6** The operations executed at each tile.

1: **Input:** $dev_table[], dev_lock[], row_add, row_id, tile_id, len$
2: **Initial:** volatile __shared__ int $M[]$
3: $tile_add \leftarrow row_add + tile_id \times len$
4: LockRead($dev_lock, row_id, tile_id$)
5: moveToShare($M, dev_table, row_add$)
6: $__syncthreads()$
7: $M[i,j] \leftarrow$ Op$\{M[i-1,j], M[i,j-1], M[i-1,j-1]\}$
8: moveToGlobal($M, dev_table, row_add$)
9: LockWrite($dev_lock, row_id$)
10: $tile_id \leftarrow tile_id + 1$

**Spinlock Functions**

We use a spinlock to implement inter-tile synchronization. The spinlock consists of two function calls, *LockRead*() and *LockWrite*(), as shown in Algorithm 6 [line: 4, 9], and a synchronization counter array, as mentioned in Fig. 8. The counter array, $dev_lock[]$, has the same size as the number of rows in the matrix. A counter value indicates the number of tiles that have been executed in the corresponding row. In an intra-tile execution, the counter is incremented after the execution of the tile is completed. This operation is performed in the function *LockWrite*() where a *__syncthreads*() call is placed before the increment of the counter to ensure the completion by all threads in the block of the execution and the data transfer.

In function *LockRead*(), thread $0$ spins in an empty loop until it detects that the counter value is no smaller than $T$, while other threads are idle and wait for thread $0$. The counter value of each row can be accessed at $dev_lock[row_id]$, where $row_id$ is the index of the corresponding row. $T$ can be calculated according to equation 6.3 where $ID_{tile}$ refers to the index of a tile, $tileY$ and $tileX$ are the height and width of each tile, and $Xtiles$ represents the number of tiles that the data matrix splits X dimension into.

**Algorithm 7** Function design for LockRead() and LockWrite().

1: **Input** $thread, row_id, dev_lock[], T$
2: **LockRead():**
3: **if** $thread == 0$ **then**
4:     **while** $dev_lock[row_id] < T$ **do**
5:         {empty loop}
6: $__syncthreads()$
7:
8: **LockWrite():**
9: **if** $thread == 0$ **then**
10:     $dev_lock[row_id + 1] \leftarrow dev_lock[row_id + 1] + 1$
11: $__syncthreads()$

$$T = \min(ID_{tile} + tileY/tileX, Xtiles) \tag{6.3}$$

In function *LockWrite*(), thread $0$ updates the completion status of tiles by incrementing the counter, which is then accessed by the LockRead() function for the next row. A $__syncthreads()$ is used to force the other threads to wait for thread $0$ and also to ensure completion of the data transfer from shared memory to global memory.

## 6.5 Evaluation

Four wavefront applications, which have been used to evaluate the performance in many existing research works, are also used to evaluate the proposed shared memory-based mechanism. The performance data and memory efficiency metrics are used as the metrics to compare the overall efficiency of the shared memory-based mechanism and the cache-based mechanism. We implement the cache-based mechanism using PeerWave [8], which is the most optimized solution among the existing cache-based implementations, except use our proposed spinlock in place of their lock design.

### 6.5.1 Wavefront Applications

**Smith-Waterman (SW)**

This algorithm is used primarily to perform alignment of the input sequences in order to determine the matching patterns between the two sequences [2]. In the formula, $w(i,j)$ is the gap penalty function, which returns $3$ when $a_i$ equals $b_i$ and $-3$ otherwise.

$$M[i,j] = \max \begin{cases} M[i-1,j]-2, \\ M[i,j-1]-2, \\ M[i-1,j-1]+w(i,j), \\ 0 \end{cases}$$

**Wagner-Fischer (WF)**

This algorithm computes the edit distance between two sequences, which determines the difference by counting the minimum number of operations required to transform one sequence into the other [3].

$$M[i,j] = \begin{cases} M[i-1,j-1], & \text{if } a_i = b_i \\ \min \begin{cases} M[i-1,j]+1, \\ M[i,j-1]+1, \\ M[i-1,j-1]+1 \end{cases} & \text{otherwise} \end{cases}$$

**Successive Over-Relaxation (SOR)**

A specific formulation of 2D-SOR has been studied in [26, 44, 42], which performs

stencil computations on one data matrix in an iterative fashion for solving a linear system of equations. In this equation, the data entry $M[i, j]$ is calculated from its neighbors where $M[i-1, j]$ and $M[i, j-1]$ are newly updated and the other three entries are from the previous time step.

$$M[i,j] = (M[i-1,j] + M[i,j-1] + M[i,j] + M[i+1,j] + M[i,j+1])\,/\,5$$

**Summed-Area Table (SAT)**

Summed-area table (SAT) [78] is used in the image processing domain for generating the sum of values in a rectangular subset of a grid. The sum of all the data entries above and to the left of $(i, j)$ is efficiently computed in a single pass over the data matrix with the following equation.

$$M[i,j] = M[i,j] + M[i-1,j] + M[i,j-1] - M[i-1,j-1]$$

Since the arithmetic intensity for an equation can be approximately calculated using the number of operations divides the size of a data entry, we estimate the arithmetic intensity of SW, WF, SOR, and SAT, which are $1.5$ FLOPs/byte, $1.5$ FLOPs/byte, $1.25$ FLOPs/byte, and $1$ FLOPs/byte, respectively, according to the equations. Considering the high arithmetic intensity that a GPU can achieve, the performance of these four applications are bounded by the memory bandwidth.

### 6.5.2 Test Cases and GPU Environment

In the experiments, we test each application with $10$ different input matrix sizes, which are $2^{12}\times2^{12}$, $2^{13}\times2^{13}$, $2^{14}\times2^{14}$, $2^{15}\times2^{15}$, $2^{12}\times2^{15}$, $2^{13}\times2^{15}$, $2^{14}\times2^{15}$, $2^{15}\times2^{12}$, $2^{15}\times2^{13}$, $2^{15}\times2^{14}$.

The largest matrix size, $2^{15} \times 2^{15}$, is limited by the largest consecutive data block that we can allocate in GPU memory. In the implementation of the cache-based mechanism, 1024 threads are allocated to each thread block for fetching data and performing calculations. The proposed shared memory approach also uses 1024 threads for data transfers, but only 128 threads are active during the computation, which is restricted by the tile height.

The applications are evaluated on two NVIDIA GPUs: GTX 1080 Ti and Tesla K40. Both GPUs support up to 48 KB shared memory in each thread block. On each processor, the GTX 1080 Ti has an L1 cache of 48 KB and a shared memory of 96 KB, but the Tesla K40 has only a merged first level memory of 64 KB, which can provide either 48 KB shared memory or 48 KB L1 cache in one simulation. In the experiments using the Tesla K40, most of the first level memory is set to L1 cache and shared memory, respectively, for the two mechanisms. The 1080 Ti is equipped with 28 streaming multiprocessors with 128 CUDA cores for each streaming multiprocessor and the K40 has 15 streaming multiprocessors with 192 CUDA cores for each processor. In order to conduct the stream processing, the number of CUDA streams should be no more than the available streaming multiprocessors. Therefore, 28 and 15 CUDA streams are, respectively, used for the same applications, which are tested on both GPUs, to fully utilize the GPU computing capabilities.

### 6.5.3 Memory Subsystem Efficiency

We evaluate the memory efficiency on the GTX 1080 Ti GPU because it has larger L1 cache and shared memory. Profiling data for these applications are not exactly the same due to the different data dependence and memory access, but the shared memory approach requires much fewer DRAM transactions and achieves better shared memory (L1 cache) usage for all the applications compared to the cache approach. We evaluate the memory

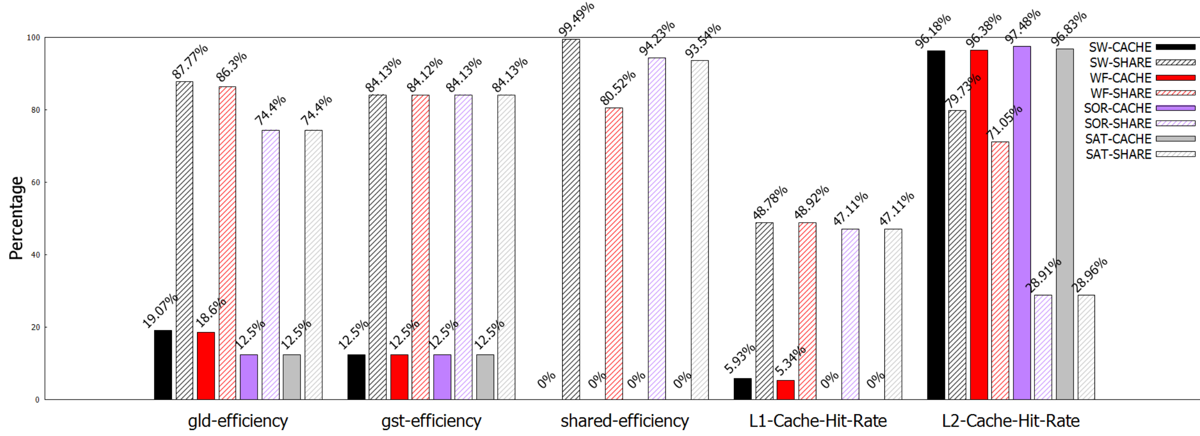



Figure 11: Shared memory-based mechanism achieves much higher efficiency rate on multiple metrics except L2 cache hit rate.

efficiency on the largest matrix, which is $2^{15} \times 2^{15}$. We use the NVIDIA profiling tool, *nvprof*, to collect the relevant information for cache, shared memory, and operations. The profiling data is collected as the average of all the kernels, so we evaluate the memory efficiency for the kernel function that performs the operations on a row of tiles. The optimized tile sizes are used for both the cache-based and the shared memory-based mechanisms, which are $1024 \times 256$ and $128 \times 64$, respectively. Because the optimized tile height of the cache-based mechanism is $8$ times larger than that of the shared memory-based mechanism, we perform data normalization by dividing the cache-based mechanism data metrics of “dram_read trans" and “dram_write trans" by $8$.

Table 7: Profiling data for cache-based and shared memory-based mechanisms on GTX $1080$ Ti.

| | Smith-Waterman | | Wagner-Fischer | |
|---|---|---|---|---|
| Metrics | CACHE | SHARE | CACHE | SHARE |
| dram_read_trans | $848,531$ | $566,242$ | $914,682$ | $570,701$ |
| dram_write_trans | $539,182$ | $489,500$ | $591,505$ | $489,107$ |

| | 2-D SOR | | Summed-Area Table | |
|---|---|---|---|---|
| Metrics | CACHE | SHARE | CACHE | SHARE |
| dram_read_trans | $995,162$ | $722,532$ | $921,199$ | $706,513$ |
| dram_write_trans | $597,278$ | $488,230$ | $904,226$ | $512,058$ |

To calculate a row of tiles, the number of device memory transactions required by the cache-based mechanism is more than the shared memory-based mechanism's, as shown in the first two rows of each table in Table 7. More device memory transactions leads to much larger overhead for accessing the required data entries. Fig. 11 depicts the percentage rate for multiple memory efficiency metrics. The shared memory-based mechanism achieves nearly a $50\%$ L1 cache hit rate on all four applications because only a small amount of data are directly read from or written to global memory during the execution, and most of these data access are consecutive and can be stored in L1 cache. However, the cache-based mechanism has limited L1 cache hits due to the imperfect data locality. Even though most of the required data is fetched by L2 cache in the cache-based mechanism, it is still less efficient considering the latency for accessing L2 cache. We do not use L1 cache hit rate as the metric for the shared memory-based mechanism because most calculations are performed in shared memory. However, we can control when data should be stored in or removed from the shared memory as well as the validity. So, we have a $100\%$ shared memory hit rate for accessing these data entries. The global load (gld) and global store (gst) efficiency indicate the ratio of requested global memory load/store throughput to required global memory load/store throughput. The higher ratio indicates that the shared memory-based mechanism uses fewer transactions, which is closer to optimal, to obtain the required data. This is because the shared memory-based mechanism achieves much better global memory coalescing. Moreover, near-optimal shared memory coalescing is obtained, shown as shared_efficiency. It is also true that the use of the shared memory demands many more integer instructions for locating the tiles and indexing the subproblems, which is necessary for copying the data between the shared memory and the global memory.

However, the optimized memory access contributes to a larger performance improvement.

Overall, the proposed mechanism achieves better memory efficiency because it requires fewer global memory transactions, has lower memory copy latency due to the coalesced memory access, and avoids the L1 cache misses by substituting shared memory for the cache.

### 6.5.4 Performance: Multiple Tile Sizes

In this subsection, we evaluate the shared memory-based mechanism and run experiments on the four applications, which are represented by the labels "SW", "WF", "SOR", and "SAT". Because of the relation, which is presented in Equation 6.2 and the size of shared memory, which is 48 KB, three tile sizes can be used in the shared memory-based mechanism, which are: $128 \times 32$, $128 \times 64$, and $256 \times 32$. The execution time of these three tile sizes are shown in Table 8 for the GTX 1080 Ti and K40 separately. The matrix size is set to $2^{15} \times 2^{15}$ for each application and the execution time is measured in milliseconds. To minimize the error, we repeat the same execution 100 times and use the average as the execution time of each experiment.

Table 8: Execution time (ms): averaged for 100 repetitions.

GTX 1080 Ti

| Tile Size (Height × Width) | SW | WF | 2D-SOR | SAT |
|---|---|---|---|---|
| $128 \times 64$ | 139 | 131 | 682 | 691 |
| $128 \times 32$ | 159 | 152 | 724 | 722 |
| $256 \times 32$ | 170 | 169 | 731 | 733 |

Tesla K40

| Tile Size (Height × Width) | SW | WF | 2D-SOR | SAT |
|---|---|---|---|---|
| $128 \times 64$ | 807 | 790 | 6,289 | 6,270 |
| $128 \times 32$ | 941 | 913 | 6,390 | 6,390 |
| $256 \times 32$ | 807 | 790 | 6,323 | 6,304 |

As shown in Table 8, the optimal tile size of $128 \times 64$ achieves the best performance in all cases. On the GTX $1080$ Ti GPU, because $128$ cores are available in each streaming multiprocessor, the shortest height that can still achieve an acceptable core utilization should be $128$. Tile $256 \times 32$ has the worst performance because the core utilization cannot be doubled but doubling the tile height halves the maximum inter-tile concurrency. Tile $128 \times 32$ has the best inter-tile concurrency and intra-tile concurrency; however, it requires more global memory accesses to complete the data transfer. On the Tesla K$40$ GPU, utilization of all cores can be achieved only by tile $256 \times 32$ because each multiprocessor has $192$ cores. However, tile $256 \times 32$ does not deliver much better performance compared to $128 \times 64$. Therefore, we should use tile size $128 \times 64$ for the similar applications and there is no necessity to adjust the tile size for these two NVIDIA GPU architectures.

### 6.5.5 Performance: Cache vs Shared Memory

We compare the execution time of our approach (SHARE) to the most optimized existing cache implementation (CACHE).

**Performance Data**

The execution time, in milliseconds, for the four applications, using the two GPUs, is depicted in Table 9 and Table 10. In general, the performance of each shared memory-based experiment is better than the corresponding cache-based experiment. In addition, the execution time on the GTX $1080$ Ti outperforms what is obtained on the K$40$ because of the faster clock rate, more streaming multiprocessors, and higher memory bandwidth. The shared memory-based approach achieves a $1.9$ to $6.7$ times speedup over the cache-based approach across all the applications on the $1080$ Ti, and a $1.4$ to $3.3$ speedup is obtained on the K$40$. To trade off the problem that each CUDA core of the K$40$ has less shared

memory on average, only $128$ threads are created to utilize the $192$ CUDA cores on each multiprocessor, which leads to a relatively lower efficiency.

**Performance Consistency**

We evaluate the performance consistency by comparing the performance data of two different matrices that have the same problem size. In our experiment, the comparable sets are $\{2^{15} \times 2^{12}, 2^{12} \times 2^{15}\}$, $\{2^{15} \times 2^{13}, 2^{13} \times 2^{15}\}$ and $\{2^{15} \times 2^{14}, 2^{14} \times 2^{15}\}$.

As shown in Table 9 and Table 10, the cache-based performance of the matrix $2^{15} \times 2^{14}$ is $10\% - 32\%$ slower than the performance of matrix $2^{14} \times 2^{15}$ on the GTX $1080$ Ti and $3\%-20\%$ slower on the K$40$. On the other hand, the shared memory-based implementation achieves extremely close performance for the two matrices of this comparable set on both GPUs. Similarly, we can observe that the performance difference between the two matrices is larger for the cache-based implementation, which is up to $27\%$ comparing to $22\%$ for the shared memory-based implementation for the comparable set of $\{2^{15} \times 2^{13}, 2^{13} \times 2^{15}\}$. For the comparable set $\{2^{15} \times 2^{12}, 2^{12} \times 2^{15}\}$, the shared memory-based mechanism still obtains quite consistent execution times on two different GPU architectures. However, the cache-based mechanism performs not as well on this comparable set where about $25\%$ performance difference exists in all four applications on GTX $1080$ Ti and $25\%$ to $35\%$ performance difference on K$40$.

Figure 12 depicts the data consistency of the shared memory-based mechanism and the cache-based mechanism separately executed on the two GPUs. The GTX $1080$ Ti is represented as index $1$ and the Tesla K$40$ is indexed with $2$. On K$40$, better data consistency is obtained from both mechanisms and the shared memory-based mechanism outperforms the cache-based mechanism significantly. While the performance difference ratio of the

Table 9: Performance data for the GTX 1080 Ti GPU.

| Time (ms) | SW | | WF | | 2D-SOR | | SAT | |
|---|---|---|---|---|---|---|---|---|
| Matrix | CACHE | SHARE | CACHE | SHARE | CACHE | SHARE | CACHE | SHARE |
| $2^{12} \times 2^{12}$ | 31 | 14 | 38 | 10 | 48 | 17 | 45 | 17 |
| $2^{13} \times 2^{13}$ | 68 | 21 | 81 | 21 | 123 | 49 | 112 | 49 |
| $2^{15} \times 2^{12}$ | 125 | 34 | 147 | 33 | 230 | 94 | 208 | 92 |
| $2^{14} \times 2^{14}$ | 146 | 44 | 171 | 43 | 329 | 174 | 309 | 188 |
| $2^{15} \times 2^{13}$ | 158 | 50 | 181 | 49 | 345 | 174 | 316 | 177 |
| $2^{12} \times 2^{15}$ | 169 | 53 | 208 | 53 | 299 | 108 | 273 | 108 |
| $2^{13} \times 2^{15}$ | 189 | 61 | 231 | 61 | 399 | 185 | 372 | 186 |
| $2^{14} \times 2^{15}$ | 243 | 73 | 282 | 72 | 605 | 343 | 569 | 353 |
| $2^{15} \times 2^{14}$ | 322 | 76 | 352 | 74 | 668 | 342 | 634 | 347 |
| $2^{15} \times 2^{15}$ | 862 | 139 | 877 | 131 | 1,651 | 682 | 1,510 | 691 |

Table 10: Performance data for the Tesla K40 GPU.

| Time (ms) | SW | | WF | | 2D-SOR | | SAT | |
|---|---|---|---|---|---|---|---|---|
| Matrix | CACHE | SHARE | CACHE | SHARE | CACHE | SHARE | CACHE | SHARE |
| $2^{12} \times 2^{12}$ | 121 | 26 | 138 | 25 | 284 | 108 | 248 | 107 |
| $2^{13} \times 2^{13}$ | 268 | 65 | 300 | 62 | 771 | 405 | 668 | 402 |
| $2^{15} \times 2^{12}$ | 491 | 134 | 538 | 127 | 1,411 | 800 | 1,290 | 798 |
| $2^{14} \times 2^{14}$ | 658 | 229 | 718 | 220 | 2,354 | 1,599 | 2,164 | 1,587 |
| $2^{15} \times 2^{13}$ | 669 | 238 | 750 | 229 | 2,348 | 1,607 | 2,184 | 1,593 |
| $2^{12} \times 2^{15}$ | 729 | 158 | 834 | 153 | 1,914 | 829 | 1,680 | 824 |
| $2^{13} \times 2^{15}$ | 835 | 230 | 944 | 220 | 2,732 | 1,598 | 2,460 | 1,587 |
| $2^{14} \times 2^{15}$ | 1,148 | 447 | 1,274 | 430 | 4,478 | 3,190 | 4,143 | 3,166 |
| $2^{15} \times 2^{14}$ | 1,377 | 447 | 1,495 | 429 | 4,627 | 3,190 | 4,278 | 3,165 |
| $2^{15} \times 2^{15}$ | 2,597 | 807 | 2,820 | 790 | 8,962 | 6,289 | 8,285 | 6,270 |

two mechanisms are very close on the GTX 1080 Ti. Considering that the execution time of comparable set $\{2^{15} \times 2^{14}, 2^{14} \times 2^{15}\}$ is much longer than set $\{2^{15} \times 2^{12}, 2^{12} \times 2^{15}\}$, the shared memory-based mechanism gets more benefits from data consistency. Overall, the proposed approach achieves better performance consistency compared to the cache-based mechanisms.

## 6.6 Summary

In this chapter we introduce a highly efficient hyperplane-tiling approach for exploiting wavefront parallelism on GPUs. Instead of relying on L1 cache, the mechanism used

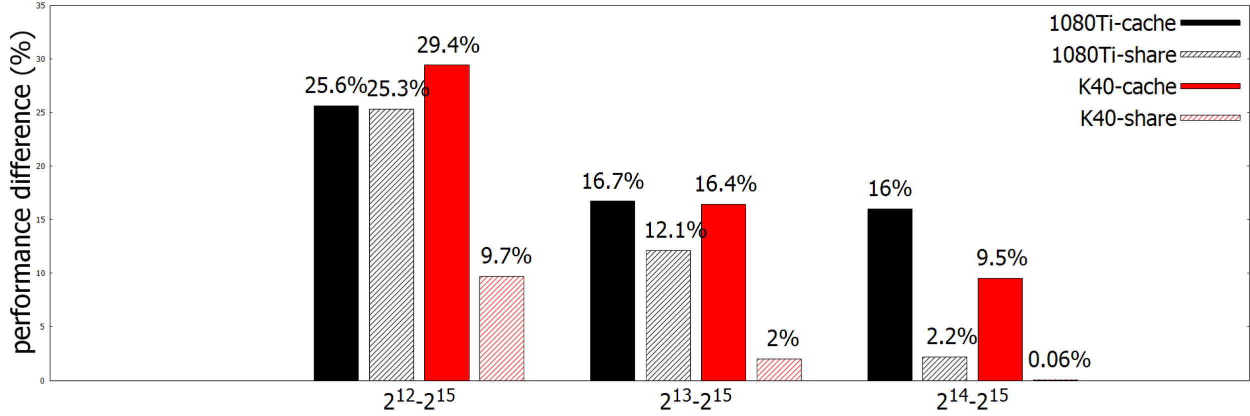



Figure 12: Average difference for four applications.

in prior research, we transfer the tile data to shared memory, which reduces the memory access latency and achieves better data locality. Our proposed shared memory implementation is a generalized solution, which can be applied to problems that execute nested loops with uniform data dependencies. We provide a formula to calculate the optimal tile size from the problem size and the GPU configuration, which determines the tradeoff between data locality and concurrency. Besides, we utilize the stream processing to minimize inter-tile synchronization overhead and provide a spin lock design which is easy to implement and has low overhead. We compare our approach to the best-existing solution and obtain up to six times speedup. The paper includes a detailed comparison and explanation of the performance difference between the two approaches.

# CHAPTER 7 TIME-SKEWED TILING OPTIMIZATION FOR HIGH ORDER 2D STENCIL COMPUTATIONS ON GPUS

Performance optimization of stencil computations has been widely studied in the literature, since they occur in many computationally intensive applications. In recent years, optimizing stencil computations on GPUs is especially popular due to the GPU's many-core architecture. Most of these optimization utilizes overlapped tiling together with GPU shared memory to address the inter-tile dependence. However, time-skewed tiling, which is also named hyperplane tiling, is rarely used due to the limited cache capacity on GPUs.

In this chapter, we present a time-skewed tiling approach for optimizing stencil processing on GPUs. The proposed approach utilizes GPU shared memory to reduce the global memory transactions and applies to arbitrary stencils in 2-dimensional spaces. To address the inter-tile dependences without introducing redundant computations, we develop a data access pattern for passing the dependent data to successor tiles efficiently in shared memory and obtaining cache reuse along the time dimension.

## 7.1 Introduction

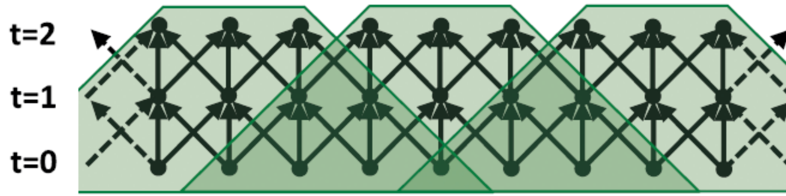
Stencil computation is widely used in partial differential equations [79] and image-processing applications [54] that require smoothing and filtering the array elements, as well as other applications, like music recognition [5]. The iterative computation involves a nested iteration, which updates array elements according to some fixed pattern, called a stencil. The nested iteration includes one or multiple loops for the spatial dimensions and the outer loop for the time dimension. A 2D stencil problem is resolved in a three-dimensional nested loop, which includes one time dimension and two spatial dimensions. A stencil computation traverses the space-time nested loop in an order that ensures the

computation of all array elements at time $t$ complete before computing any array elements at time $t + 1$. In addition, the computation results are updated out-of-place in the memory at each time step. Therefore, in a stencil computation, parallelism can be obtained within each time step because it has no dependence but is not independent across different time steps due to the dependence in time dimension.

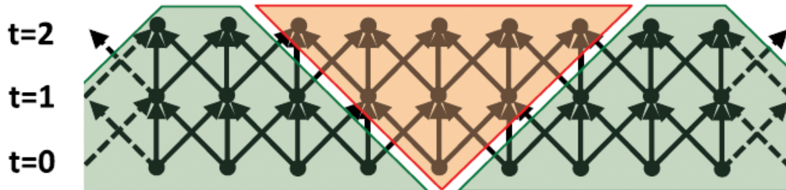
Even if the data entries can be processed simultaneously at each time step, the stencil computation is still bounded by memory bandwidth. The dependence in the time dimension forces the cache system to evict all data entries at the beginning of each time step. Moreover, if the size of the memory block required for computing the data entries of one time step exceeds the cache size, cache misses lead to more severe memory latency and further restrict the performance. This is even worse on GPUs due to the GPU's limited cache capacity.

One idea for optimizing stencil computation is reducing cache misses during the spatial dimension computation and tiling along the time dimension to reuse the cached data. The tiling technique is one popular solution for achieving the desired optimization and overlapped tiling is mostly studied in the high-performance community, especially for research conducted on GPUs, to address the inter-tile dependence by performing redundant computations. As shown in Fig. 13(a), overlapped tiling work around the inter-tile dependence by performing redundant operations. According to [10], $0.95$X extra operations are required if the ghost zone size is $10\%$ of the total grid elements and this number goes up to $3.62$X when ghost zone size increases to $20\%$. Besides, it also brings in intra-tile load imbalance.

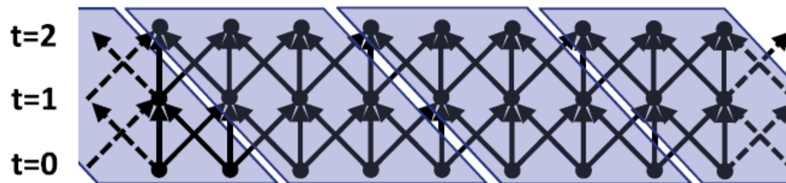
Unlike overlapped tiling, split tiling is developed to avoid redundant operations by




(a) Overlapped tiling has redundant operations and unbalanced workload.




(b) Split tiling has severe load imbalance issue and less inter-tile concurrency.




(c) Time-skewed tiling has lowest inter-tile concurrency among these three tiling strategy.

Figure 13: Different tiling strategies [1] illustrate the tradeoff between concurrency, computation overhead, and memory latency.

dividing the computation into multiple groups of tiles that can be processed simultaneously and each group produces the dependent data for the other groups. In Fig. 13(b), the tiles in green are processed simultaneously and the computation on orange tiles can only start when the green tile kernels are completed. In split tiling optimization, even if the redundant operations are avoided, the performance is still reduced due to less concurrency and severe load imbalance.

Time-skewed tiling, which is also called parallelogram tiling or hyperplane tiling, obtains the best spatial locality and avoids redundant computation without generating an unbalanced workload. In time-skewed tiling optimization, as shown in Fig. 13(c), dependent data are passed to successor tiles along each spatial dimension. Therefore, the

computation is serialized along one spatial dimension because of the preserved inter-tile dependence, which enforces a pipelined startup and provides limited concurrency. Because of the serialized computation and limited concurrency, this time-skewed tiling is rarely studied, especially on GPUs.

In our proposed time-skewed tiling approach, redundant operations are eliminated and intra-tile load balance is ensured. To minimize memory latency, an automatic time step variable determination method is developed to obtain optimal temporal locality. In addition, we develop a data access pattern to manage the memory address of tile data and dependent data, which helps with efficient utilization of GPU shared memory for storing tile data and passing dependent elements. Moreover, a circular pipeline scheduling mechanism is developed to increase the processor-level parallelism and ensure that there is no idle processor in the process. Our proposed optimization provides the following contributions:

- We present a time-skewed tiling approach for optimizing stencil problems on GPUs, which has not been developed before to the best of our knowledge.
- We develop a data access pattern to support the time-skewed tiling optimization, which manages shared memory efficiently and applies to arbitrary stencil problems in 2D space.
- We provide a different view towards the serialized dependence and prove that a limited concurrent solution may obtain good performance on GPUs when it has low computation overhead and memory latency.
- In the proposed time-skewed tiling approach, a circular pipeline technique is devel-

oped to minimize the unbalanced workload, caused by pipelined startup.

- We implement the pipelined computation with a stream processing scheme and design a two-level lock system for synchronizing the streams on GPUs.

## 7.2 Background and Motivation

In this section, we present the disadvantages of each tiling approach with respect to GPUs. Then, we discuss the difficulties for using some existing highly efficient solutions to optimize 2D stencil computations on GPUs. In addition, we propose an empirical analysis to the effect of scheduling limited inter-tile parallelism on GPUs when the tile processing is serialized. In the end, we explain our motivation of optimizing 2D stencil computations with time-skewed tiling on GPUs.

### 7.2.1 Disadvantages of Different Temporal Tiling

There is no perfect tiling strategy that can optimize data locality without bringing in penalty in other aspects. Load imbalance, occurs in overlapped tiling and split tiling, leads to a more serious performance overhead on GPUs because of CUDA massive-threads property. In addition, overlapped tiling and split tiling cannot obtain the optimal cache utilization due to the temporal blocking shapes. This is not desirable on GPUs because GPU has limited cache capacity and sacrificing cache resources may result in insufficient intra-tile parallelism. Time-skewed tiling has no above performance issue but it leads to pipelined start. More importantly, it enforces the serialized tile processing in each pipe and restricts the number of thread blocks that can be launched simultaneously in each kernel.

### 7.2.2 Difficulties of Using Existing Solutions

3.5D tiling is a highly efficient tiling mechanism for optimizing 3D stencil computations. The algorithm performs a 2.5D spatial tiling and an additional temporal tiling into on-chip memory. Because the 2.5D spatial tiling only blocks in two dimensions and streams through the third dimension, the amount of data entries, to be cached at each iteration, is significantly reduced so the bandwidth requirement is also reduced. This strategy changes the 3D stencil computation from memory bound to compute bound. However, this algorithm does not apply to 2D stencil computation because it cannot save bandwidth by pre-fetching a 2D layer when streaming along the third dimension. Therefore, 2D stencil computation is still memory bounded. Moreover, the temporal tiling, proposed in 3.5D algorithm, is not practical applying to GPUs. Because of GPU limited on-chip memory recourse, it is impossible to increase the inter-tile parallelism by caching more time steps in one tile.

Split tiling mechanisms like diamond tiling and trapezoidal tiling cannot retain the same efficiency on GPUs. Even if these mechanisms are naturally applicable to 2D grid, the load imbalance property is inevitable and leads to poor intra-tile parallelism on GPUs. This performance issue is more severe on diamond tiling that only a few steps, at the middle of the tile, may have enough operations for full core utilization.

As for code auto-generating compiler, it greatly save programmer's time from developing the tiling mechanisms; however, it usually fails to fully utilize hardware resources and has limitations in programming. For example, PPCG [57] has restrictions for using static allocated arrays, which limits the maximum problem size. Also, many compilers share a common problem that requiring the fixed tile size at compile time.

### 7.2.3 Concurrency Modeling on GPUs for Time-Skewed Tiling

Time-space tiling creates two-level parallelisms: inter-tile parallelism and intra-tile parallelism. Intra-tile parallelism is determined by the in-tile load balance and on-chip cache capacity, but inter-tile parallelism is restricted by the inter-tile dependence. Therefore, time-skewed tiling obtains the best intra-tile parallelism because the tile shape ensures the load balance. On the other hand, time-skewed tiling has limited inter-tile parallelism because the inter-tile dependence are not overlapped and have to be passed between adjacent tiles in a sequence.

The CUDA programming model for the time-skewed tiling on a 2D grid has the following properties. To have a better explanation, we use XY to represent a two-dimensional space where X and Y are the two dimensions and T represents the temporal dimension.

- Thread Mapping: Each tile is scheduled to one thread block to utilize on-chip cache and minimize the communication overhead.
- Block Mapping: The spatial tiles that reside on a XY plane and connected in a row along one spatial dimension, which is X dimension in this work, are processed in sequence in one or multiple kernels.
- Kernel Mapping: Adjacent rows of spatial tiles are started in a pipeline manner and processed in multiple kernels, which are launched simultaneously.

Above all, we can make the following conclusion. First, the processor utilization can be maximized by simultaneously processing multiple rows of tiles, so at least one kernel is launched on each processor. Second, the core utilization within each processor is deter-

mined by the spatial tile size and the ratio of operations per byte. Third, since 2D stencil computation is still memory bounded, creating multiple thread blocks and increasing the total amount of threads at each kernel may not bring in better core utilization due to the memory bandwidth congestion.

### 7.2.4 Our Motivation

It has been proven that both overlapped tiling and split tiling face the issue of large computation overhead. Because the stencil computation is still memory bounded, larger computation overhead would downgrade the overall performance. Therefore, we want to avoid this performance issue by developing a time-skewed tiling approach. Even if time-skewed tiling is not favored in the previous studies because of its limited concurrency issue, we believe that it is still possible to achieve an acceptable in-processor core utilization and obtain a good overall performance if cache resources are used efficiently.

The NVIDIA GPU architectures, Kepler and Pascal, share a common property that the maximum capacity of shared memory is no less than L1 cache. Thus, addressing the tiles in shared memory is more efficient at full utilization of GPU cache resources across all popular NVIDIA platforms. In addition, the non-coalesced memory access is inevitable in all existing tiling approaches because of the tile shape at temporal dimension. In this situation, the edge elements would be fetched into cache lines together with invalid data elements when they are first time read into L1 cache. The invalid data elements also reside in L1 cache through the entire computation, which wastes the limited cache space. In order to address the cache overhead and use shared memory efficiently, we develop a data access pattern for storing and accessing the valid data elements in shared memory efficiently. Moreover, most of the existing research projects focus on low-order stencil

computations that the distance is usually no larger than 2. Studying the efficiency of high-order stencil computation is another topic that we want to include in the development and analysis of our proposed tiling approach.

## 7.3 Design and Challenges

Fig. 14 illustrates the design overview of our tiling approach. This approach exploits the processor utilization in a streaming processing pattern, which is developed to address the processor-level load imbalance caused by pipelined start. A two-level lock system is designed to coordinate with stream processing, which ensures the kernel launch sequence and cache coherence for the updated dependence among adjacent tiles. In addition, the data access pattern is also presented.

### 7.3.1 Two-level Parallelism

We have explained that the full parallelism is obtained both inside the tile (intra-tile) and across multiple tiles (inter-tile). Therefore, we exploit both intra-tile and inter-tile parallelism in our proposed tiling optimization. In Fig. 15, the circle dots represent the data elements and the large squares represent tiles. On a XY plane, the tiles can be organized in 4 rows and the tiles that are in the same row are processed in sequence.

**Intra-Tile Parallelism**

As shown in Fig. 15, tile elements are processed by CUDA threads. Because each tile is processed in one thread block, the maximum parallelism is obtained when tile elements are processed by all in-block threads simultaneously. Each thread may process one or multiple data elements according to the tile size. To ensure efficient core utilization, we consider the latency hiding from two aspects: memory latency hiding and arithmetic latency hiding.

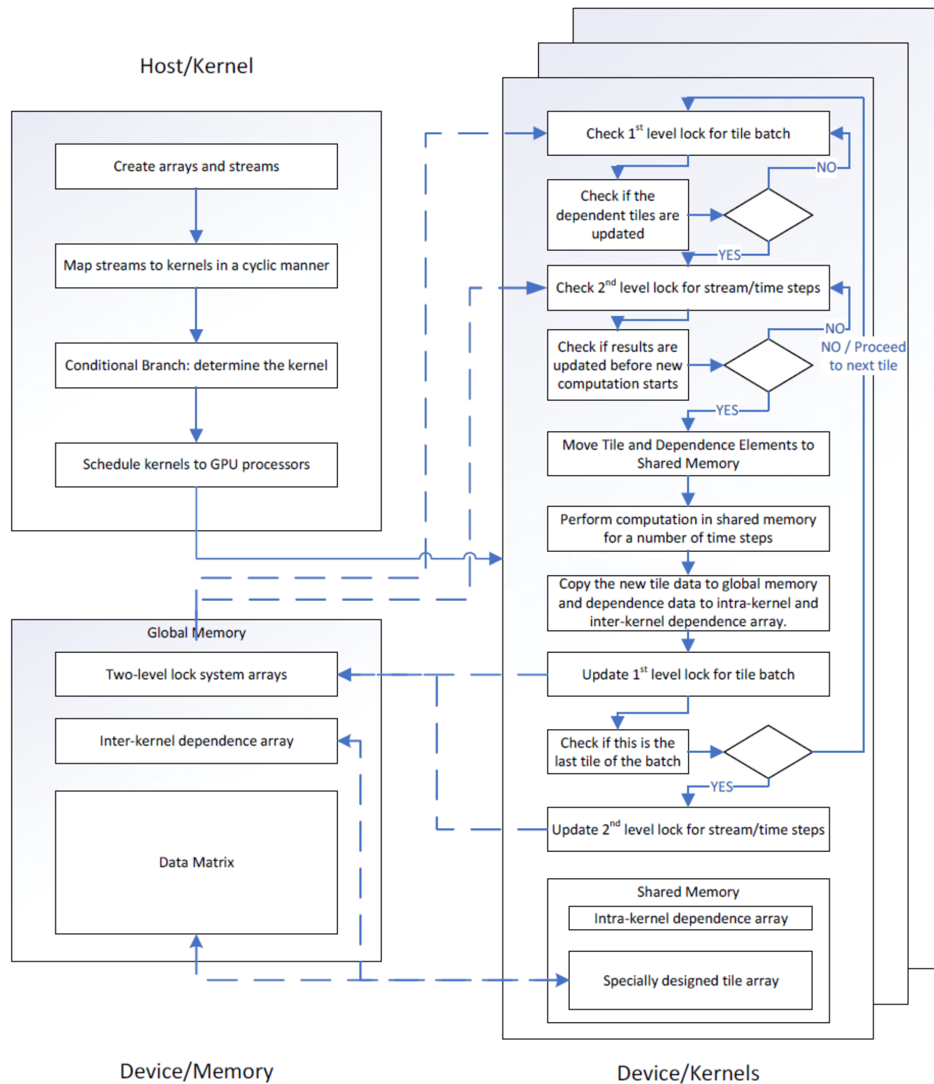



Figure 14: Design of Host and GPU Device: solid arrows depict the flow of events and dashed arrows show the data communication.

Since all required data elements are transferred to shared memory before the computation, no memory latency occurs during the computation. On the other hand, arithmetic latency can be completely overlapped if at least 24 warps are created on each multiprocessor [80]. In our tiling approach, we create 1024 threads for each block, which is sufficient to hide the latency and keep all cores busy during the computation.

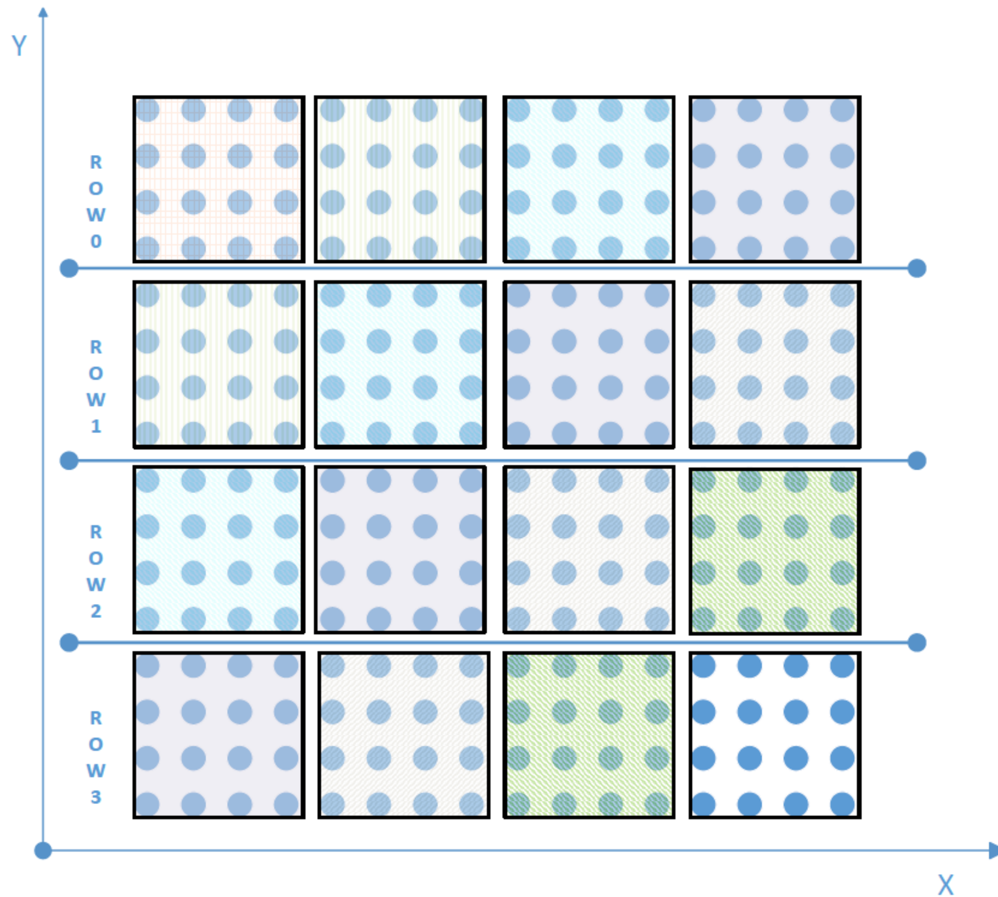



Figure 15: Intra-tile parallelism and inter-tile parallelism.

**Inter-Tile Parallelism**

In our tiling approach, inter-tile parallelism is obtained differently from the other tiling approaches. Because computation over a row of tiles along either X or Y dimension is serialized due to the dependences, inter-tile parallelism can only be obtained in a pipelined manner, which are presented by same color tiles in Fig. 15. We retain serial computation in the X dimension so the tiles, in a row along X dimension, are executed in sequence. In this chapter, “row" specifically represent a row of tiles along the X dimension. This serialized computation is implemented by creating only one block in each kernel and processing a row of tiles within this block in sequence. Then, we obtain inter-tile parallelism

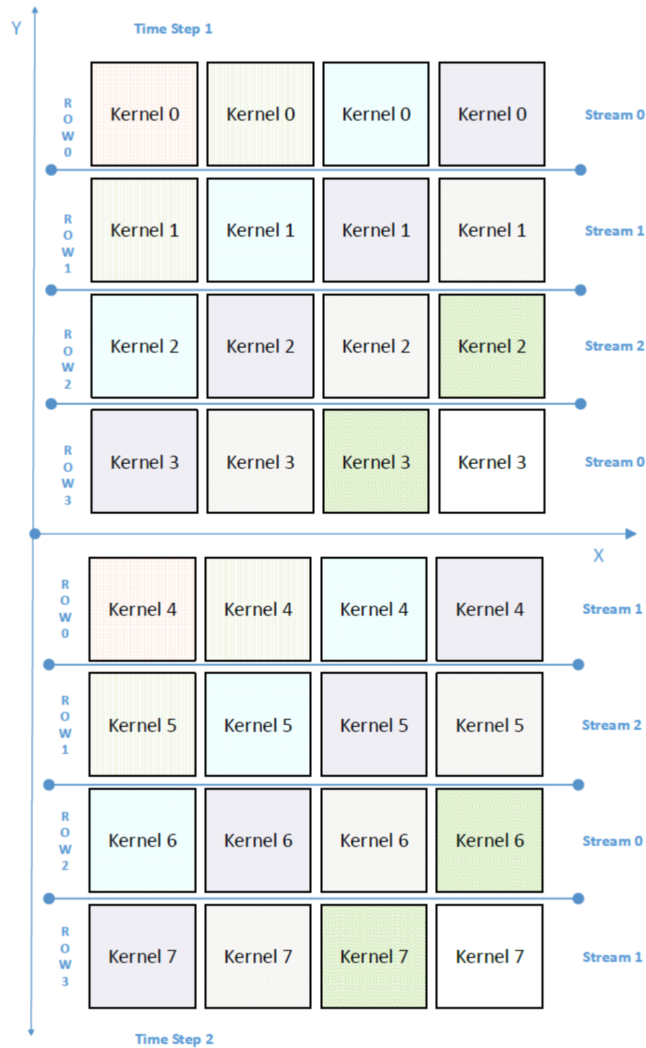



Figure 16: Tiles that are in same color are processed simultaneously.

by launching different rows of tiles, along the Y dimension, in different CUDA streams because kernels in different streams are launched simultaneously if the multiprocessors are available.

### 7.3.2 Stream Processing Scheme

Fig. 16 depicts the kernel-stream distribution as well as the stream parallelism. Because of the pipelined processing, the dependences in the Y dimension are staggered, so inter-tile parallelism is achieved at each anti-diagonal. Pipelined processing is realized by managing

the timing for processing the tiles in each kernel. Ideally, a kernel starts performing the computation of tile $n$ of row $k$ as soon as the dependent kernel completes its computation of tile $n$ of row $k-1$. This processing order is managed by a two-level lock system, which will be discussed in the next subsection.

**Intra-Kernel Dependent Data**

Processing each row of tiles in sequence in a single thread block provides an advantage for reducing memory latency. Passing dependent data between adjacent tiles not only leads to serialized computation but also enforces the communication between these tiles. If each tile is processed in a self-contained thread block, the communication has to be performed in global memory, which increases the memory traffic. In our time-skewed tiling approach, each row of tiles, along the X dimension, are streamed and processed in one thread block. Therefore, the dependent data, passing between every two tiles in each row, are stored in shared memory. Caching the inter-tile dependent data in shared memory significantly reduce the global memory transactions, which makes the data locality better than other tiling schemes.

**Inter-Kernel Dependent Data**

Except for intra-kernel dependences, dependent data are also passed among the tiles that reside in two adjacent rows and we call them inter-kernel dependent data. Because different rows of tiles are processed in different thread blocks, passing inter-kernel dependent data between every two adjacent rows must be done using global memory. This raises another memory issue because storing inter-kernel dependent data for all the rows at each time step requires a huge amount of memory space, which is not practical and generates

high overhead for allocating memory space.

Our stream processing scheme works around this issue by limiting the number of CUDA streams, created for processing the rows, and creating an inter-kernel dependent data array for these streams instead of rows. Thus, instead of the one-to-one row stream mapping, we create a limited number of streams, equal to the number of streaming multiprocessors, and allocate the kernels that process these rows into these streams. This strategy makes the total array size proportional to the length of the matrix in X dimension and significantly reduces the required memory space from $O(N^2 \times T)$ to $O(N)$ where $N$ represents the length of one spatial dimension of the input matrix and $T$ represents the total time steps.

**Intra-Stream Process**

Kernels that process the rows of all the time steps are distributed to a number of CUDA streams so multiple kernels are launched in each stream. Because the kernels that reside in the same stream are invoked sequentially, the memory safety for passing the dependent data is ensured implicitly. In addition, the serialized kernel launching property makes the reuse of the inter-kernel dependent data array possible because the kernels update the same dependency array in sequence as well.

**Inter-Stream Process and Task Scheduling**

We develop a task scheduling method to distribute the kernels that process the rows of all the time steps to the limited number of CUDA streams in a cyclic manner. Fig. 16 depicts the task scheduling for distributing kernels to $3$ streams in two adjacent time steps.

In the same time step, each stream can start launching a new kernel right after the

completion of the previous one as long as there is no dependence interval, which may put the kernel in idle waiting. The dependence interval occurs when the kernel, which is in anther stream and processing the predecessor row, has not completed the computation of certain tiles that have the dependent data required by the current kernel. This interval is relatively short because processing one row does not need to wait till the completion of processing the predecessor row. Instead, this inter-stream collaboration is synchronized on the tile basis so a new kernel, launched in a stream, can start processing its row as soon as the first few tiles of the predecessor row have been processed in the kernel distributed to another stream.

It is different when the inter-stream process reaches the end of one time step and is ready to proceed to the next time step. Theoretically speaking, the first row of each time step can be processed as soon as the tile elements are available because it requires no dependent data from the predecessor row. Thus, the inter-stream collaboration for the kernels that process the last row of one time step and the first row of the new time step, respectively, in separate streams has no dependency and allows simultaneous start. However, this is only guaranteed when mapping each row to an individual stream and the reuse of the dependency array brings in a new risk of overwritten data so a lock function is required to manage the starting of the next time step in our stream processing scheme.

### 7.3.3 Two-level Lock System

To manage the row processing either in the same time step or across multiple adjacent time steps, the stream processing scheme requires two lock functions to ensure the correctness and completeness of the data access between every two adjacent rows and between every two time steps. As shown in Fig. 16, we index the rows with increment ID for each

time step and the tiles for each row.

**In-Step Lock Function**

For every two adjacent rows, which are in the same time step, the kernel that processes the larger indexed row is always one tile behind the other kernel that processes the smaller indexed row because of the pipeline processing. The kernel that processes row $k$ needs to know the progress that the other kernel is making in row $k-1$. In other words, each kernel needs to update its status of tile completion and this status is accessed by another tile, which processes the next row of the same time step.

Thus, we create a counter variable for each row, used to record the number of tiles that have been processed at each point. The kernel increments this counter when it completes the computation and updates the data elements for one tile and each kernel reads the status counter of the kernel that processes the previous row before it starts processing a tile. If another kernel that processes the prior row has completed the computation to the tile that passes dependent data required by computing the current tile, it is safe for the target kernel to start the computation; otherwise, the kernel puts itself in a waiting status and keeps checking the status counter of the other kernel until the counter value is valid. An exception occurs in the first row because the kernel that processes the first row does not require dependent data, passed from the other kernel, so this kernel can start processing the tiles as soon as the tile elements are available.

**Across-Step Lock Function**

In our stream processing scheme, each CUDA stream is equipped with an inter-tile dependency array. To pass the dependent data to the kernel that processes the next row, each kernel updates its dependent data into the dependency array of the next stream. If

a kernel is launched at the last stream, it updates the dependent data to the dependency array of stream $0$.

The collaboration of the cyclic scheduling method and the fact that first kernel of each time step can start computation immediately generates a risk that the data elements, updated by the last kernel of each time step, would be overwritten by the next kernel, which is in the same stream and processes a row in the next time step. Fig. 16 depicts an example. In the first time step, four kernels that are in the last row are processed in sequence in stream $0$. At the same time, the kernels that are in the first row of second time step can be processed simultaneously because there is no dependence between these two rows. Therefore, the computation flow of time step $2$ is individual from processing the last row of time step $1$ and it is possible that the kernels of the second row in time step $2$ update dependent data to the inter-kernel dependency array of stream $0$. Because this dependence update could occur when one of the four kernels, which are $3$, $7$, $11$ and $15$, is still being processed, the dependence update, happened in time step $2$, would overwrite the data of stream $0$, which makes the calculation of these kernels incorrect.

We develop the second-level lock function and a lock array to hold the kernels waiting for a complete memory update. Similar to the status counter, each stream owns an array element, which is initialized to the total number of tiles in each row. Here we use row_size to represent the number of tiles in each row. Before starting the computation, each kernel reads the array element of the stream in which the next row is processed. An element value of row_size indicates that the data update is already completed in the next stream and it is now safe to pass the dependent data into that stream. If it is safe to start computation, a kernel clears the array element to $0$ and then sets it back to row_size when it completes the

Figure 17: Shared memory organization for dependent elements and tile elements.

computation for all the tiles of the row. Conversely, if the kernel reads value 0, it will be put into a empty loop to wait until the value is set back to row_size. Detailed implementation code is provided in Sec. 7.4.

### 7.3.4 Data Access Pattern

Unlike using L1 cache where the compiler manages the address implicitly, using shared memory requires direct human management. To accommodate the tile elements, intra-kernel dependent elements, and inter-kernel dependent elements in shared memory for each time step of a tile, we design a data access pattern that consists of three parts and is organized as shown in Fig. 17.

The small blocks in the given picture represent data elements and this table shows the organization of the array for storing all necessary data elements for processing a tile. The green region, which has $8 \times 8$ data elements, is an example of the tile in one time stamp. The grey region that has $10 \times 2$ data elements is reserved for storing inter-kernel dependent data. Then, the intra-kernel dependent data are stored in the blue region. The proposed two-level lock system ensures that the computation is performed only after all three regions are filled with correct data.

At the beginning of each step of a tile processing, tile elements of that step are copied to the green region while intra-kernel and inter-kernel dependent elements are stored into blue and grey regions, respectively. When the required elements are ready, the kernel is able to process the computation and get the updated tile elements for the next time step, which are located inside the red square. Then, the kernel moves the updated tile elements from the new address back to the green region and also copies the new intra-kernel and inter-kernel dependent elements to the corresponding addresses in the two dependency arrays. When a tile includes multiple time steps, the elements in all three regions are updated at each step and the tile elements, updated in the last time step, are written back to global memory. Therefore, the global memory transactions, performed in each tile processing, is significantly reduced because the tile elements are read/written from/to global memory only at the beginning and end of the tile processing and reused in shared memory during the computation. In addition, the global memory access latency, generated for updating inter-kernel dependent data, is also minimized in this data pattern for two reasons. First, the number of transactions required for fetching the inter-kernel dependent elements is low because these elements are stored consecutively and aligned in global memory. Second, the memory access latency can be overlapped with other operations because all the tile elements are executed simultaneously. Overall, we apply the data access pattern in tile processing and achieve near-optimal memory efficiency.

## 7.4 Implementation

In this section, we provide the details of our implementation in major functions, memory layout, and process flow.

### 7.4.1 Dependency Array Structure and Transfer

To better explain the structure of the dependency array structure, we define the following variables. dep_stride is the variable for quantifying the width of ghost zone [52]; tile_lenX is the length of the tile in the X dimension and tile_lenY is the length in the Y dimension; relatively, matrix_lenX and matrix_lenY are the length of the matrix in the X and Y dimensions; time_step is the number of time stamps, tiled in the temporal dimension; num_streams is the total number of available streams.

**Intra-Kernel Dependency Array.** In a kernel, the computation is performed across the tiles that concatenate in a row along the X dimension and the dependent data, passed between every two adjacent tiles, are located in the previous tile and near a joint edge. In Fig. 17, the blocks in blue are the intra-kernel dependent elements, required by the tile processing; the blocks, indexed with $1$ and $1/2$, are the dependent data, passed to the next tile. Thus, the structure of the intra-kernel dependency array is determined by dep_stride, tile_lenY, and time_step.

In tile processing, computation is performed in every time step of each tile so the access to the dependent data are required at each step. Thus, an array is created in shared memory to temporally store these dependent data for each tile. At the beginning of processing each time step, dependent data of that time step are moved from the dependency array to the computation array. After the computation, the new dependent data, used for processing the next time step of the next tile, are moved to the dependency array and overwrites the old values.

Because dep_stride usually does not equal to the warp size and these dependent data

are not stored consecutively, transferring these data requires more shared memory access. However, the overhead for the extra shared memory access is negligible due to the high memory bandwidth and low latency to access shared memory.

**Inter-Kernel Dependency Array.** In Fig. 17, the blocks in grey are the inter-kernel dependent elements and the blocks, indexed with $2$ and $1/2$, are the dependent data required by the next row. Thus, the structure of the inter-kernel dependency array is determined by matrix_lenX instead of tile_lenY. In tile processing, the kernel uses the tile index and the stream index to locate the address of the dependent elements in the large dependency array. The size of the inter-dependency array for all the streams is given in equation 7.1.

$$\text{array_size} = \text{num_streams} \times \text{time_steps} \times \text{dep_stride} \times \text{matrix_lenX} \tag{7.1}$$

Similar to the intra-kernel dependency operations, inter-dependent data are moved to the computation array at the beginning of each time step, then new dependent data, passed to the same indexed tile but in the next row, are moved back to the global dependency array. The number of elements to be passed is determined by tile_Xlen, which is a multiple of the warp size. Thus, transferring inter-dependent data is efficient because of full cache line utilization.

### 7.4.2 Stream Indexing

To distribute the kernels to the limited number of CUDA streams in a cyclic manner, it is important for each kernel to know the index of the stream where it is launched. Besides, stream indexes are also required for accessing the inter-kernel dependency array and counter arrays of the lock system. The method for obtaining the stream index directly

**Algorithm 8** Lock functions for coordinating the streams at each time stamp.

```
1: read_tile_lock_for_batch(volatile lock[], row_idx, tile_idx, YoverX, row_size, col_size,
   time_tile)
2: if threadIdx.x == 0 then
3:     limit = min(tile_idx + YoverX, row_size)
4:     while lock[time_tile × col_size + row_idx] < limit do
5:         { no operations }
6: __threadfence()
7: __syncthreads()

8: write_tile_lock_for_batch(volatile lock[], row_idx, col_idx, time_tile)
9: if threadIdx.x == 0 then
10:     lock[time_tile × col_size + row_idx + 1] += 1
11: __threadfence()
12: __syncthreads()
```

from the row index is shown in equation 7.2, which uses logic_stream as the variable of the stream index. However, this method is not applicable in most cases because the number of rows in each time step cannot be divided evenly, which makes the kernel-stream mapping between two adjacent time steps discontiguous. The discontiguous stream indexing between two adjacent time steps enforces a new pipelined start at each time step, which could significantly worsen the overall load imbalance. Thus, we provide our solution for deriving the stream index for each kernel and name the variable cur_stream. stream_offset is the remainder of number of rows in each time steps divided by num_streams; time_tile is the index of the tile in time dimension.

$$
\begin{aligned}
&\text{logic_stream} = \text{row_idx} \bmod \text{num_streams} \\
&\text{cur_stream} = (\text{logic_stream} + \text{stream_offset} \times \text{time_tile}) \bmod \text{num_streams}
\end{aligned}
\tag{7.2}
$$

**Algorithm 9** Lock functions for collaborating the streams across two adjacent time stamps.

```
1: read_time_lock_for_stream(volatile lock[], cur_stream, next_stream, row_size,
   row_idx)
2: if threadIdx.x == 0 then
3:     while lock[next_stream] < row_size do
4:         { no operations }
5: __threadfence()
6: __syncthreads()

7: write_time_lock_for_stream(volatile lock[], cur_stream, row_size)
8: if threadIdx.x == 0 then
9:     lock[cur_stream] = row_size
10: __threadfence()
11: __syncthreads()

12: clear_time_lock_for_stream(volatile lock[], cur_stream, row_size)
13: if threadIdx.x == 0 then
14:     lock[cur_stream] = 0
15: __threadfence()
16: __syncthreads()
```

### 7.4.3 Code for Lock Functions

The lock process that coordinates streams in each time step performs a comparison as the wait operation and an counter increment operation for signaling. In algorithm 8, variable YoverX is the quotient of tile_lenY divided by tile_lenX and tile_idx indexes the tile in each row. Algorithm 9 shows the design of the second lock that coordinates the streams across two adjacent time steps. In this lock, an extra function is performed to clear the counter variable to $0$ at the beginning of processing each row. next_stream is the index of the stream that launches the kernel, which processes the next row.

### 7.4.4 Flow of Tile Processing

Here we describe the flow of host process and device process separately in algorithm 10 and algorithm 11.

**Algorithm 10** Flow of host kernel for array initialization, indexing streams, and launching kernels.

1: Initialize arrays
2: Create streams
3: **while** $t <$ T_len **do**
4: **for** row_idx $= 0, \ldots,$ col_size **do**
5: Calculate stream index
6: GPU_Tile$\langle\langle\langle 1, 1024\rangle\rangle\rangle()$

The host kernel allocates array blocks in device global memory for tile elements, dependent elements, and lock counters. In addition, the host kernel creates a group of CUDA streams according to the number of multiprocessors. Then, the time dimension is tiled in a nested loop and the device kernels are launched in the inner iteration.

Flow of the CUDA kernel is shown in algorithm 11. Line $1$ performs a wait operation of the first-level lock to check if it is safe to start processing the new time step. When processing the new time step is permitted, a clear operation, shown in line $2$ is performed to set the counter variable to $0$ for the specific stream. Then, a loop is used to manage the processing sequence for a row of tiles, as shown in line $3$. In this first loop, the wait operation of the second-level lock is called to coordinate the streams and the required elements are moved into the computation array (line $4$-$5$). Next, the second iteration is applied to traverse all time steps of each tile, where tile_Tlen represents the number of time steps in each tile (line $6$). Between line $7$ and $16$, required elements are moved to shared memory and then the stencil computation is performed in function "Stencil$()$" (Line $10$). After the computation, dependent elements of the next time step are moved back to the dependency arrays and the updated tile elements are re-positioned for processing the next tile in function "Swap_Tile" (Line $14$). At the end, signal operations for the two locks are called accordingly to update the status of rows and streams (line $18$ and $19$).

**Algorithm 11** Flow of device kernel for processing tiles.

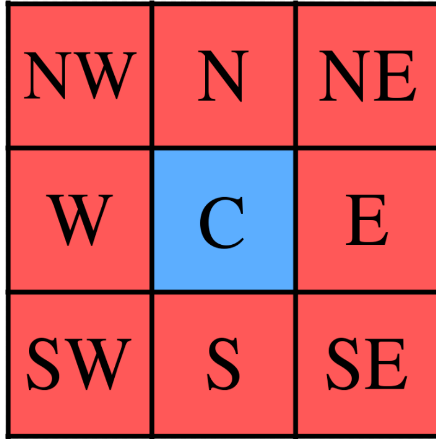
```
1: read_time_lock_for_stream(volatile lock[], cur_stream, next_stream, row_size,
   row_idx)
2: clear_time_lock_for_stream(volatile lock[], cur_stream, row_size)
3: for tile_idx = 0, . . . , row_size do
4:     read_tile_lock_for_batch(volatile lock[], row_idx, tile_idx, YoverX, row_size,
   col_size, time_tile)
5:     Move tile elements: global memory → shared memory
6:     for t = 0, . . . , tile_Tlen do
7:         Move dependent elements to shared memory
8:         __threadfence()
9:         __syncthreads()
10:        Stencil()
11:        __threadfence()
12:        __syncthreads()
13:        Move dependent elements to intra-kernel and inter-kernel dependency arrays
14:        Swap_Tile()
15:        __threadfence()
16:        __syncthreads()
17:    Move tile elements: shared memory → global memory
18:    write_tile_lock_for_batch(volatile lock[], row_idx, col_idx, time_tile)
19: write_time_lock_for_stream(volatile lock[], cur_stream, row_size)
```

## 7.5 Experimental Evaluation

The experiments are performed on a NVIDIA GTX 1080 Ti GPU. The GTX 1080 Ti is built with Pascal architecture, which has 96KB dedicated shared memory and 48KB L1 cache on each multiprocessor and a total of 3584 cores. However, a single block can use at most 48KB shared memory so at least two blocks are required for each kernel to obtain full shared memory utilization. To fully occupy the 28 multiprocessors, we create 28 CUDA streams so each multiprocessor always has one block to process during the computation.

### 7.5.1 Experimental Background and Setup

We build the experiments for the proposed time-skewed tiling approach on the 2D grids. The performance is evaluated from multiple aspects: distance, iterative method,



(a) Moore Neighborhood

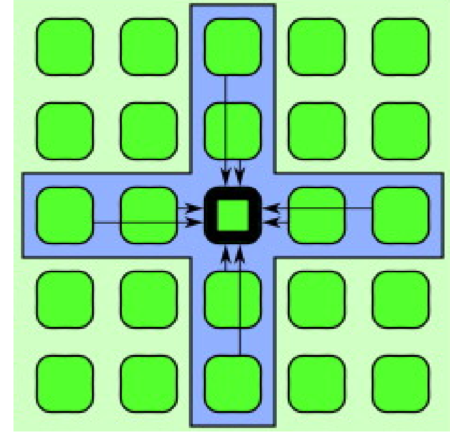

(b) Cross-Shaped Neighborhood

Figure 18: Two patterns lead to different memory efficiency.

and stencil pattern. Our experiments involves two stencil patterns including Moore neighborhood [81, 82], and cross-shaped neighborhood [83, 84]. Fig. 18a illustrates a Moore neighborhood pattern, which has a distance of $1$. The Moore neighborhood pattern provides the most efficient cache line utilization because data elements are stored consecutively in each row. A cross-shaped pattern that has $2$ units distance is shown in Fig. 18b. This stencil pattern has lower cache line efficiency.

Our evaluation is performed on Jacobi iteration methods, which are used in many computing applications, such as HEAT [85], Poisson solver [86], Gradient [87], and etc. We perform the Jacobi method on the Moore Neighborhood pattern and cross-shaped neighborhood pattern. The experiments are completed in six different spatial $2D$ matrix sizes, which are $2^8 \times 2^8$, $2^9 \times 2^9$, $2^{10} \times 2^{10}$, $2^{11} \times 2^{11}$, $2^{12} \times 2^{12}$, and $2^{13} \times 2^{13}$. Each computation has $512$ time steps and is involved in eight different distances, for each pattern.

### 7.5.2 Experimental Results and Analysis

In this section, we evaluate the performance of the proposed tiling approach and analyze the GPU resource management for obtaining optimal efficiency. Also, we compare the

Table 11: Number of time steps can be processed for each pair of tile size and distance.

| Tile_Size | dist= 1 | dist= 2 | dist= 3 | dist= 4 | dist= 5 | dist= 6 | dist= 7 | dist= 8 |
|---|---|---|---|---|---|---|---|---|
| $64 \times 64$ | 8 | 8 | 4 | N/A | N/A | N/A | N/A | N/A |
| $64 \times 32$ | 16 | 16 | 8 | 8 | 8 | 4 | 4 | 4 |
| $32 \times 32$ | 16 | 16 | 16 | 8 | 8 | 8 | 4 | 4 |

performance to a simple GPU implementation, a PPCG translated implementation [57], and an OpenACC implementation, in GFLOPs, where the simple GPU implementation is not optimized by any tiling.

**Time Steps**

There are three arrays that reside in shared memory, two computation arrays and one intra-kernel dependency array. According to the proposed data access pattern, each computation array consists of a tile elements region, intra-kernel dependent elements region, and inter-kernel dependent elements region. Because the intra-kernel dependency array would expand its size when the distance is increased, the number of time steps that can be processed in each tile has to be reduced to ensure the overall memory requirement is not exceeding the shared memory capacity. However, if the distance is so large that the memory requirement still exceeds the capacity, even if the number of time steps is already minimized, the number of tile elements to be calculated must be reduced.

Three different tile sizes are shown in Table 11. Only three distances are supported on the largest tile size, $64 \times 64$, because less memory can be allocated to intra-kernel dependency array. Having a smaller spatial tile leaves a larger memory block for the intra-kernel dependency array, which allows more time steps in a tile.

**Performance Analysis**

In all the experiments, each tile is processed by one kernel, which has only one thread

Table 12: Performance in GFLOPS for Moore Neighborhood Pattern.

| Problem_Size | Tile_Size | dist= 1 | dist= 2 | dist= 3 | dist= 4 | dist= 5 | dist= 6 | dist= 7 | dist= 8 |
|---|---|---|---|---|---|---|---|---|---|
| $2^8 \times 2^8$ | $64 \times 64$ | 15 | 21 | 31 | N/A | N/A | N/A | N/A | N/A |
| | $64 \times 32$ | 10 | 18 | 30 | 48 | 53 | 70 | 71 | 88 |
| | $32 \times 32$ | 14 | 28 | 40 | 60 | 70 | 94 | 96 | 119 |
| $2^9 \times 2^9$ | $64 \times 64$ | 28 | 59 | 84 | N/A | N/A | N/A | N/A | N/A |
| | $64 \times 32$ | 25 | 53 | 73 | 112 | 124 | 162 | 162 | 198 |
| | $32 \times 32$ | 31 | 62 | 114 | 172 | 199 | 260 | 258 | 322 |
| $2^{10} \times 2^{10}$ | $64 \times 64$ | 79 | 143 | 184 | N/A | N/A | N/A | N/A | N/A |
| | $64 \times 32$ | 63 | 116 | 159 | 244 | 270 | 354 | 348 | 426 |
| | $32 \times 32$ | 75 | 149 | 213 | 253 | 327 | 382 | 490 | 613 |
| $2^{11} \times 2^{11}$ | $64 \times 64$ | 147 | 259 | 336 | N/A | N/A | N/A | N/A | N/A |
| | $64 \times 32$ | 113 | 204 | 277 | 412 | 458 | 594 | 594 | 726 |
| | $32 \times 32$ | 62 | 143 | 207 | 281 | 352 | 458 | 444 | 564 |
| $2^{12} \times 2^{12}$ | $64 \times 64$ | 148 | 265 | 338 | N/A | N/A | N/A | N/A | N/A |
| | $64 \times 32$ | 116 | 208 | 284 | 422 | 467 | 608 | 608 | 741 |
| | $32 \times 32$ | 81 | 149 | 212 | 312 | 361 | 464 | 476 | 571 |
| $2^{13} \times 2^{13}$ | $64 \times 64$ | 148 | 271 | 342 | N/A | N/A | N/A | N/A | N/A |
| | $64 \times 32$ | 118 | 210 | 287 | 427 | 474 | 616 | 615 | 748 |
| | $32 \times 32$ | 79 | 148 | 212 | 313 | 361 | 462 | 480 | 577 |

block of $1024$ threads. Besides, each thread requires at least $40$ registers so each thread block needs $40$K registers, which is more than half of the total capacity and makes the block-level parallelism of each multiprocessor restricted by register resources. Therefore, only one thread block is processed on each multiprocessor at a time.

The experiments are performed on six different problem sets and the evaluation for each problem set also consists of three separate computations for different tile sizes. Each tile includes a certain number of time steps, which are shown in table 11. Table 12 provides the performance results of processing the Jacobi method on Moore neighborhood patterns. Good performance is obtained on large problem sets because more data updates are completed in the intra-kernel dependency array on shared memory. In addition, patterns that have long distance (high order) provide more operations, which are performed on the same cached tile block, so spatial locality is greatly improved and better performance can be achieved.

Table 13: Performance in GFLOPS for Cross-Shaped Neighborhood Pattern.

| Problem_Size | Tile_Size | dist= 1 | dist= 2 | dist= 3 | dist= 4 | dist= 5 | dist= 6 | dist= 7 | dist= 8 |
|---|---|---|---|---|---|---|---|---|---|
| $2^8 \times 2^8$ | $64 \times 64$ | 14 | 29 | 48 | N/A | N/A | N/A | N/A | N/A |
| | $64 \times 32$ | 11 | 23 | 39 | 58 | 78 | 114 | 142 | 184 |
| | $32 \times 32$ | 14 | 31 | 54 | 80 | 106 | 136 | 159 | 197 |
| $2^9 \times 2^9$ | $64 \times 64$ | 33 | 71 | 123 | N/A | N/A | N/A | N/A | N/A |
| | $64 \times 32$ | 29 | 61 | 118 | 165 | 220 | 289 | 317 | 411 |
| | $32 \times 32$ | 33 | 71 | 155 | 213 | 285 | 369 | 411 | 511 |
| $2^{10} \times 2^{10}$ | $64 \times 64$ | 95 | 193 | 316 | N/A | N/A | N/A | N/A | N/A |
| | $64 \times 32$ | 72 | 145 | 253 | 349 | 471 | 620 | 670 | 855 |
| | $32 \times 32$ | 80 | 171 | 291 | 413 | 546 | 681 | 779 | 951 |
| $2^{11} \times 2^{11}$ | $64 \times 64$ | 174 | 364 | 603 | N/A | N/A | N/A | N/A | N/A |
| | $64 \times 32$ | 130 | 267 | 445 | 647 | 842 | 1,080 | 1,207 | 1,541 |
| | $32 \times 32$ | 67 | 166 | 286 | 412 | 536 | 684 | 774 | 959 |
| $2^{12} \times 2^{12}$ | $64 \times 64$ | 180 | 375 | 605 | N/A | N/A | N/A | N/A | N/A |
| | $64 \times 32$ | 134 | 274 | 452 | 660 | 852 | 1,094 | 1,231 | 1,577 |
| | $32 \times 32$ | 79 | 177 | 292 | 422 | 547 | 698 | 788 | 978 |
| $2^{13} \times 2^{13}$ | $64 \times 64$ | 184 | 381 | 613 | N/A | N/A | N/A | N/A | N/A |
| | $64 \times 32$ | 135 | 275 | 455 | 660 | 864 | 1,111 | 1,250 | 1,601 |
| | $32 \times 32$ | 85 | 177 | 292 | 425 | 544 | 701 | 796 | 992 |

As shown in table 12, better performance is obtained on a tile size of $32 \times 32$ for small problem sets. Because the quotient of dividing a small problem set with large tile is smaller than the number of available CUDA streams, the inter-tile parallelism is not as good as it is obtained on a small tile. However, tiling with large tiles contributes better performance on large problem sets. Full inter-tile parallelism can be obtained by all tile sizes on large problem sets, so tiling with small tiles does not increase processor utilization. Oppositely, tiling with small tile sizes leads to higher memory access overhead because it requires more global memory transactions for updating the inter-kernel dependency array.

The same behavior also applies to the computation, which is performed on the cross-shaped neighborhood and shown in table 13. Because performing computation on the cross-shaped neighborhood requires much fewer operations than performed on Moore neighborhood, much better performance is achieved. However, executing fewer operations on the same amount of cached data elements also implies a worse spatial locality.

Table 14: Performance Comparison in GFLOPS for Cross-Shaped Neighborhood Pattern.

| Problem_Size | | dist= 1 | dist= 2 | dist= 3 | dist= 4 | dist= 5 | dist= 6 | dist= 7 | dist= 8 |
|---|---|---|---|---|---|---|---|---|---|
| $2^8 \times 2^8$ | PPCG | 2 | 6 | 11 | 20 | N/A | N/A | N/A | N/A |
| | OpenAcc | 46 | 107 | 191 | 292 | 406 | 537 | 654 | 805 |
| | plain_GPU | 51 | 126 | 221 | 348 | 451 | 584 | 722 | 900 |
| | Skew_Tiling | 14 | 31 | 54 | 80 | 106 | 136 | 159 | 197 |
| $2^9 \times 2^9$ | PPCG | 9 | 25 | 46 | 76 | N/A | N/A | N/A | N/A |
| | OpenAcc | 92 | 209 | 357 | 545 | 741 | 917 | 1,082 | 1,343 |
| | plain_GPU | 129 | 286 | 456 | 674 | 810 | 986 | 1,177 | 1,482 |
| | Skew_Tiling | 33 | 71 | 155 | 213 | 285 | 369 | 411 | 511 |
| $2^{10} \times 2^{10}$ | PPCG | 36 | 84 | 150 | 244 | N/A | N/A | N/A | N/A |
| | OpenAcc | 117 | 254 | 453 | 740 | 1,062 | 1,298 | 1,520 | 1,933 |
| | plain_GPU | 205 | 430 | 712 | 1,043 | 1,185 | 1,412 | 1,721 | 2,156 |
| | Skew_Tiling | 95 | 193 | 316 | 413 | 546 | 681 | 779 | 951 |
| $2^{11} \times 2^{11}$ | PPCG | 111 | 214 | 370 | 640 | N/A | N/A | N/A | N/A |
| | OpenAcc | 143 | 324 | 505 | 812 | 1,133 | 1,374 | 1,603 | 2,023 |
| | plain_GPU | 266 | 533 | 830 | 1,198 | 1,290 | 1,500 | 1,735 | 2,201 |
| | Skew_Tiling | 174 | 364 | 603 | 647 | 842 | 1,080 | 1,207 | 1,541 |
| $2^{12} \times 2^{12}$ | PPCG | 211 | 326 | 535 | 961 | N/A | N/A | N/A | N/A |
| | OpenAcc | 152 | 334 | 555 | 911 | 1,336 | 1,577 | 1,845 | 2,028 |
| | plain_GPU | 276 | 533 | 805 | 1,167 | 1,249 | 1,446 | 1,681 | 2,105 |
| | Skew_Tiling | 180 | 375 | 605 | 660 | 852 | 1,094 | 1,231 | 1,577 |
| $2^{13} \times 2^{13}$ | PPCG | 282 | 358 | 588 | 1,126 | N/A | N/A | N/A | N/A |
| | OpenAcc | 154 | 329 | 535 | 867 | 1,094 | 1,327 | 1,536 | 1,960 |
| | plain_GPU | 274 | 549 | 852 | 1,224 | 1,339 | 1,557 | 1,803 | 2,226 |
| | Skew_Tiling | 183 | 380 | 613 | 660 | 864 | 1,111 | 1,250 | 1,601 |

Therefore, better memory efficiency is obtained on Moore neighborhood than the cross-shaped neighborhood.

**Performance Comparison**

In the proposed time-skewed tiling approach, we added many extra operations for indexing the required data elements and moving data elements between shared memory and global memory. Thus, performing more operations on each tile can offset the overhead for executing these extra operations, so our proposed tiling method is more efficient if more data elements are involved in the computation. We compare our approach to three different implementations and illustrate the overall efficiency and performance improvement of our method in tables 15 and 14. Because PPCG requires a great amount of time for translating the code of higher-order computations, we can obtain PPCG performance data

Table 15: Performance Comparison in GFLOPs per second for Moore Neighborhood Pattern.

| Problem_Size | | dist= 1 | dist= 2 | dist= 3 | dist= 4 | dist= 5 | dist= 6 | dist= 7 | dist= 8 |
|---|---|---|---|---|---|---|---|---|---|
| $2^8 \times 2^8$ | PPCG | 2 | 6 | 9 | 16 | N/A | N/A | N/A | N/A |
| | OpenAcc | 39 | 76 | 101 | 121 | 133 | 135 | 144 | 167 |
| | plain_GPU | 46 | 87 | 110 | 127 | 135 | 142 | 150 | 149 |
| | Skew_Tiling | 15 | 27 | 40 | 60 | 70 | 94 | 96 | 119 |
| $2^9 \times 2^9$ | PPCG | 9 | 22 | 35 | 54 | N/A | N/A | N/A | N/A |
| | OpenAcc | 77 | 132 | 157 | 188 | 219 | 224 | 228 | 230 |
| | plain_GPU | 105 | 148 | 167 | 174 | 196 | 220 | 232 | 231 |
| | Skew_Tiling | 31 | 62 | 114 | 172 | 199 | 260 | 258 | 322 |
| $2^{10} \times 2^{10}$ | PPCG | 31 | 69 | 79 | 125 | N/A | N/A | N/A | N/A |
| | OpenAcc | 94 | 187 | 213 | 225 | 228 | 232 | 234 | 233 |
| | plain_GPU | 134 | 175 | 205 | 199 | 205 | 234 | 208 | 235 |
| | Skew_Tiling | 79 | 149 | 213 | 253 | 327 | 382 | 490 | 613 |
| $2^{11} \times 2^{11}$ | PPCG | 80 | 141 | 120 | 179 | N/A | N/A | N/A | N/A |
| | OpenAcc | 117 | 197 | 221 | 228 | 228 | 231 | 228 | 227 |
| | plain_GPU | 154 | 212 | 231 | 225 | 217 | 215 | 209 | 213 |
| | Skew_Tiling | 146 | 259 | 336 | 412 | 458 | 594 | 594 | 726 |
| $2^{12} \times 2^{12}$ | PPCG | 121 | 177 | 139 | 188 | N/A | N/A | N/A | N/A |
| | OpenAcc | 121 | 191 | 208 | 217 | 222 | 230 | 232 | 296 |
| | plain_GPU | 199 | 212 | 213 | 210 | 201 | 202 | 200 | 204 |
| | Skew_Tiling | 148 | 265 | 338 | 422 | 467 | 608 | 607 | 741 |
| $2^{13} \times 2^{13}$ | PPCG | 132 | 203 | 143 | 204 | N/A | N/A | N/A | N/A |
| | OpenAcc | 120 | 189 | 207 | 224 | 221 | 229 | 227 | 233 |
| | plain_GPU | 203 | 229 | 222 | 223 | 219 | 222 | 221 | 230 |
| | Skew_Tiling | 148 | 271 | 341 | 426 | 473 | 616 | 615 | 748 |

only for a distance of at most 4.

The computation of a cross-shaped neighborhood computation requires much fewer memory transactions to fetch the required data elements, so both plain GPU, PPCG and OpenAcc implementations achieve much higher GFLOPs on cross-shaped neighborhood patterns. However, our proposed method cannot utilize this benefit because a constant memory access pattern is used for transferring data between shared memory and global memory. The time spent on performing extra memory transactions degrades the overall performance, which is shown in table. 14. But table 15 illustrates a different story. To collect all required data elements when performing computation on moore neighborhood pattern, both plain GPU, PPCG, and OpenAcc implementations require more memory transactions, which makes the total transaction amount close to our method's. Thus, our

method achieves much better performance because of the better temporal locality.

## 7.6 Summary

We propose a time-skewed tiling method for optimizing stencil computations for 2D grids on GPUs. To implement the tiling method on GPUs, we design a data access pattern for storing the required data elements in shared memory. In addition, we propose a stream processing system to achieve inter-tile parallelism and develop a two-level lock system to manage the pipelined start of kernel processes. Also, we design the structures for dependency arrays, which enables efficient memory access for transferring the data elements. The proposed method achieves up to $3.5\times$ performance improvements when the stencil computation is performed on a Moore neighborhood pattern; however, it does not perform well on cross-shaped pattern. We conclude that time-skewed tiling optimization can provide considerable performance improvement on dense patterns and serialized in-row computation as well as pipelined start do not degrade the performance, as has been claimed in much earlier research. We plan to extend this method to 3D grids in our future work.

# CHAPTER 8 CONCLUSION

In my dissertation research, we focus on optimizing the different iterative problems that all have the computations performed in nested loops on GPUs. To address the different kinds of latency generated by various computation properties, respectively, we design the tiling optimization methods, which target the computation overhead and memory latency accordingly. Each tiling method can be used as a general solver to optimize the different problems of the same kind. Using the higher-dimensional data-partition approach, we improve a parallel algorithm that minimizes the total execution time for scheduling tasks to multiple identical machines. The multiple dimensions are automatically tiled with a priority setting that the largest dimension is always divided first. In order to reduce the memory latency and load imbalance of the applications that use wavefront parallelism, we propose a hyperplane-tiling approach based on a stream processing scheme. The wavefront parallelism is widely used in dynamic programming procedures and specialized stencil computations. The idea of hyperplane tiling is extended to optimize high order 2D stencil computations. We prove that the time-skewed tiling (hyperplane tiling) contributes good performance improvement on high-order Jacobi methods, which is different from what is claimed in many other research papers and still not comprehensively studied.

The proposed data-partitioning approach is an extension of the tiling technique, which automatically partitions some large dimensions to fit the tiled block into global memory and deliver optimal performance. Exhaustive experiments for different partition settings have shown that our improved algorithm improves the GPU performance significantly and makes the GPU implementation perform better than the OpenMP implementation on large-

scale higher-dimensional dynamic programming problems. To our knowledge, this is the first data-partitioning scheme specifically designed for addressing the performance and memory issue of higher-dimensional dynamic programming on the GPU. With these techniques, directly applied to the dynamic programming procedure, our study explores the potential of optimizing higher-dimensional DOACROSS parallelism on GPUs.

In a 2-dimensional grid, we focus on wavefront parallelism and introduce a highly efficient hyperplane-tiling optimization for exploiting parallelism and good data locality on GPUs. In GPU programming, optimizing wavefront parallelism usually faces performance issues of insufficient parallelism, load imbalance, and poor data locality. We improve the hyperplane tiling with a stream processing scheme and our scheme outperforms the most efficient existing hyperplane tiling optimization. Utilizing stream processing, which is coordinated with a spin lock design, minimizes the inter-tile synchronization overhead. Instead of relying on L1 cache, the mechanism used in prior research, we transfer the tile data to shared memory, which reduces the memory access latency and achieves better data locality. Our proposed shared memory implementation is a general solution, which can be applied to problems that execute nested loops with uniform data dependencies. We provide a formula to calculate the optimal tile size from the problem size and the GPU configuration, which determines the tradeoff between data locality and concurrency.

In 2D stencil computations, tiling is also an important strategy used to optimize the data locality and relief the performance latency, caused by the memory bandwidth bound property. Instead of studying overlapped tiling and split tiling, which are already widely discussed, we propose a time-skewed tiling method, which is designed for the GPU architecture. In order to improve data locality, we design a data access pattern for storing the

required data elements in shared memory. In addition, we propose a stream processing system to achieve inter-tile parallelism and develop a two-level lock system to manage the pipelined kernel processes. Also, we design the structures for dependency arrays, which enables efficient memory access for transferring the data elements. The proposed method achieves up to $3.5\times$ performance improvements when the stencil computation is performed on a Moore neighborhood pattern; however, it does not perform well on the cross-shaped pattern. We reach the conclusion that time-skewed tiling optimization can provide considerable performance improvement on dense patterns and serialized in-row computation. And pipelined start do not necessarily degrade performance as it is claimed in much prior research.

Our overall contribution of this thesis is to give efficient and general tiling optimization solutions for applications that perform computations in nested loops. We believe that our proposed solutions can be applied to many popular applications, like PTAS-based task scheduling, local sequence alignment, 2D-HEAT, etc. To the best of our knowledge, we are the first to use the proposed tiling methods to accelerate the execution of the three kinds of problems, mentioned above. Because we design the proposed tiling methods with the formulas that determine the optimal GPU resource utilization, these methods are extendable for different GPU platforms and have the potential for further improvement. Our high-dimensional partition method indicates that GPU parallel programming has a great potential for accelerating approximation algorithms. Since our time-skewed tiling method utilizes $50\%$ of the shared memory resources on each multiprocessor, it is possible that the method can be further improved if the register utilization can be reduced without degrading the parallelism.

# CHAPTER 9 LIST OF PUBLICATIONS

- **Li, Yuanzhe**, Laleh Ghalami, Loren Schwiebert, and Daniel Grosu. "A GPU Parallel Approximation Algorithm for Scheduling Parallel Identical Machines to Minimize Makespan." In 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 619-628. IEEE, 2018.

- **Li, Yuanzhe**, Loren Schwiebert, Eyad Hailat, Jason Mick, and Jeffrey Potoff. "Improving performance of GPU code using novel features of the NVIDIA kepler architecture." Concurrency and Computation: Practice and Experience 28, no. 13 (2016): 3586-3605.

- **Li, Yuanzhe**, and Loren Schwiebert. "Boosting Python performance on Intel Processors: A case study of optimizing music recognition." In 2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC), pp. 52-58. IEEE, 2016.

- Nejahi, Younes, Mohammad Soroush Barhaghi, Jason Mick, Brock Jackman, Kamel Rushaidat, **Yuanzhe Li**, Loren Schwiebert, and Jeffrey Potoff. "GOMC: GPU Optimized Monte Carlo for the simulation of phase equilibria and physical properties of complex fluids." SoftwareX 9 (2019): 20-27.

# REFERENCES

[1] P. S. Rawat, M. Vaidya, A. Sukumaran-Rajam, M. Ravishankar, V. Grover, A. Rountev, L.-N. Pouchet, and P. Sadayappan, "Domain-specific optimization and generation of high-performance gpu code for stencil computations," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1902–1920, 2018.

[2] T. F. Smith and M. S. Waterman, "Comparison of biosequences," *Advances in applied mathematics*, vol. 2, no. 4, pp. 482–489, 1981.

[3] D. Bednárek, M. Brabec, and M. Kruliš, "Improving matrix-based dynamic programming on massively parallel accelerators," *Information Systems*, vol. 64, pp. 175–193, 2017.

[4] K.-H. Yang, *Basic finite element method as applied to injury biomechanics*. Academic Press, 2017.

[5] Y. Li and L. Schwiebert, "Boosting python performance on intel processors: A case study of optimizing music recognition," in *Python for High-Performance and Scientific Computing (PyHPC), Workshop on*, pp. 52–58, IEEE, 2016.

[6] P. Unnikrishnan, J. Shirako, K. Barton, S. Chatterjee, R. Silvera, and V. Sarkar, "A practical approach to doacross parallelization," in *European Conference on Parallel Processing*, pp. 219–231, Springer, 2012.

[7] A. R. Hurson, J. T. Lim, K. M. Kavi, and B. Lee, "Parallelization of doall and doacross loops—a survey," in *Advances in computers*, vol. 45, pp. 53–103, Elsevier, 1997.

[8] M. E. Belviranli, P. Deng, L. N. Bhuyan, R. Gupta, and Q. Zhu, "Peerwave: Exploiting wavefront parallelism on gpus with peer-sm synchronization," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, pp. 25–35, ACM, 2015.

[9] Y. Li, L. Ghalami, L. Schwiebert, and D. Grosu, "A gpu parallel approximation algorithm for scheduling parallel identical machines to minimize makespan," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 619–628, IEEE, 2018.

[10] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey, "3.5-d blocking optimization for stencil computations on modern cpus and gpus," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–13, IEEE Computer Society, 2010.

[11] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Effective automatic parallelization of stencil computations," in *ACM sigplan notices*, vol. 42, pp. 235–244, ACM, 2007.

[12] P. S. Rawat, C. Hong, M. Ravishankar, V. Grover, L.-N. Pouchet, and P. Sadayappan, "Effective resource management for enhancing performance of 2d and 3d stencils on gpus," in *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit*, pp. 92–102, ACM, 2016.

[13] P. S. Rawat, C. Hong, M. Ravishankar, V. Grover, L.-N. Pouchet, A. Rountev, and P. Sadayappan, "Resource conscious reuse-driven tiling for gpus," in *Proceedings of*

*the 2016 International Conference on Parallel Architectures and Compilation*, pp. 99–111, ACM, 2016.

[14] L. Ghalami and D. Grosu, "A parallel approximation algorithm for scheduling parallel identical machines," in *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*, pp. 442–451, IEEE, 2017.

[15] D. S. Hochbaum and D. B. Shmoys, "Using dual approximation algorithms for scheduling problems theoretical and practical results," *Journal of the ACM*, vol. 34, no. 1, pp. 144–162, 1987.

[16] Nvidia Corporation, "CUDA Programming Guide," 2018.

[17] *NVIDIA's Next Generation Compute Architecture: Kepler GK110*.

[18] *NVIDIA GeForce GTX 1080*.

[19] *NVIDIA TURING GPU ARCHITECTURE*.

[20] X. Li, Y. Liang, S. Yan, L. Jia, and Y. Li, "A coordinated tiling and batching framework for efficient gemm on gpus," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pp. 229–241, ACM, 2019.

[21] T. Remmelg, T. Lutz, M. Steuwer, and C. Dubach, "Performance portable gpu code generation for matrix multiplication," in *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit*, pp. 22–31, ACM, 2016.

[22] A. Grama, V. Kumar, A. Gupta, and G. Karypis, *Introduction to parallel computing*. Pearson Education, 2003.

[23] G. M. Striemer and A. Akoglu, “Sequence alignment with gpu: Performance and design challenges,” 2009.

[24] Z. Li, A. Goyal, and H. Kimm, “Parallel longest common sequence algorithm on multicore systems using openacc, openmp and openmpi,” in *Embedded Multicore/Many-core Systems-on-Chip (MCSoC), 2017 IEEE 11th International Symposium on*, pp. 158–165, IEEE, 2017.

[25] K. Balhaf, M. A. Alsmirat, M. Al-Ayyoub, Y. Jararweh, and M. A. Shehab, “Accelerating levenshtein and damerau edit distance algorithms using gpu with unified memory,” in *Information and Communication Systems (ICICS), 2017 8th International Conference on*, pp. 7–11, IEEE, 2017.

[26] P. Di and J. Xue, “Model-driven tile size selection for doacross loops on gpus,” in *European Conference on Parallel Processing*, pp. 401–412, Springer, 2011.

[27] D. A. Castanon, “Approximate dynamic programming for sensor management,” in *Proc. 36th IEEE Conf. on Decision and Control*, vol. 2, pp. 1202–1207, 1997.

[28] D. Bertsimas and R. Demir, “An approximate dynamic programming approach to multidimensional knapsack problems,” *Management Science*, vol. 48, no. 4, pp. 550–565, 2002.

[29] V. Boyer, D. El Baz, and M. Elkihel, “Solution of multidimensional knapsack problems via cooperation of dynamic programming and branch and bound,” *European Journal of Industrial Engineering*, vol. 4, no. 4, pp. 434–449, 2010.

[30] K.-E. Berger and F. Galea, "An efficient parallelization strategy for dynamic programming on gpu," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pp. 1797–1806, IEEE, 2013.

[31] C. E. Alves, E. N. Cáceres, F. Dehne, and S. W. Song, "A parallel wavefront algorithm for efficient biological sequence comparison," in *International Conference on Computational Science and Its Applications*, pp. 249–258, Springer, 2003.

[32] M. Low, W. Liu, and B. Schmidt, "A parallel bsp algorithm for irregular dynamic programming," *Advanced Parallel Processing Technologies*, pp. 151–160, 2007.

[33] C. E. Alves, E. N. Cáceres, and F. Dehne, "Parallel dynamic programming for solving the string editing problem on a cgm/bsp," in *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pp. 275–281, ACM, 2002.

[34] B. Schmidt, H. Schröder, and M. Schimmler, "Massively parallel solutions for molecular sequence analysis," in *ipdps*, 2002.

[35] K. Nishida, Y. Ito, and K. Nakano, "Accelerating the dynamic programming for the matrix chain product on the gpu," in *Networking and Computing (ICNC), 2011 Second International Conference on*, pp. 320–326, IEEE, 2011.

[36] C.-C. Wu, J.-Y. Ke, H. Lin, and W.-c. Feng, "Optimizing dynamic programming on graphics processing units via adaptive thread-level parallelism," in *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pp. 96–103, IEEE, 2011.

[37] K. Nishida, K. Nakano, and Y. Ito, "Accelerating the dynamic programming for the optimal polygon triangulation on the gpu," in *International Conference on Algorithms and Architectures for Parallel Processing*, pp. 1–15, Springer, 2012.

[38] V. Boyer, D. El Baz, and M. Elkihel, "Solving knapsack problems on gpu," *Computers & Operations Research*, vol. 39, no. 1, pp. 42–47, 2012.

[39] Y. Tang, R. You, H. Kan, J. J. Tithi, P. Ganapathi, and R. A. Chowdhury, "Cache-oblivious wavefront: improving parallelism of recursive dynamic programming algorithms without losing cache-efficiency," in *ACM SIGPLAN Notices*, vol. 50, pp. 205–214, ACM, 2015.

[40] A. Khajeh-Saeed, S. Poole, and J. B. Perot, "Acceleration of the smith–waterman algorithm using single and multiple graphics processors," *Journal of Computational Physics*, vol. 229, no. 11, pp. 4247–4258, 2010.

[41] Y. Liu, A. Wirawan, and B. Schmidt, "Cudasw++ 3.0: accelerating smith-waterman protein database search by coupling cpu and gpu simd instructions," *BMC bioinformatics*, vol. 14, no. 1, p. 117, 2013.

[42] K. Hou, H. Wang, W.-c. Feng, J. S. Vetter, and S. Lee, "Highly efficient compensation-based parallelism for wavefront loops on gpus," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 276–285, IEEE, 2018.

[43] M. E. Wolf and M. S. Lam, "A loop transformation theory and an algorithm to maximize parallelism," *IEEE transactions on parallel and distributed systems*, vol. 2, no. 4, pp. 452–471, 1991.

[44] P. Di, H. Wu, J. Xue, F. Wang, and C. Yang, "Parallelizing sor for gpgpus using alternate loop tiling," *Parallel Computing*, vol. 38, no. 6-7, pp. 310–328, 2012.

[45] T. Malas, G. Hager, H. Ltaief, H. Stengel, G. Wellein, and D. Keyes, "Multicore-optimized wavefront diamond blocking for optimizing stencil updates," *SIAM Journal on Scientific Computing*, vol. 37, no. 4, pp. C439–C464, 2015.

[46] P. Di, D. Ye, Y. Su, Y. Sui, and J. Xue, "Automatic parallelization of tiled loop nests with enhanced fine-grained parallelism on gpus," in *2012 41st International Conference on Parallel Processing*, pp. 350–359, IEEE, 2012.

[47] T. Fukaya and T. Iwashita, "Time-space tiling with tile-level parallelism for the 3d fdtd method," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, pp. 116–126, ACM, 2018.

[48] T. Grosser, A. Cohen, P. H. Kelly, J. Ramanujam, P. Sadayappan, and S. Verdoolaege, "Split tiling for gpus: automatic parallelization using trapezoidal tiles," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pp. 24–31, ACM, 2013.

[49] C. Yount and A. Duran, "Effective use of large high-bandwidth memory caches in hpc stencil computation via temporal wave-front tiling," in *2016 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pp. 65–75, IEEE, 2016.

[50] H. Dursun, K.-I. Nomura, L. Peng, R. Seymour, W. Wang, R. K. Kalia, A. Nakano, and

P. Vashishta, “A multilevel parallelization framework for high-order stencil computations,” in *European Conference on Parallel Processing*, pp. 642–653, Springer, 2009.

[51] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick, “Impact of modern memory subsystems on cache optimizations for stencil computations,” in *Proceedings of the 2005 workshop on Memory system performance*, pp. 36–43, ACM, 2005.

[52] 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*, pp. 256–265, ACM, 2009.

[53] Y. Zhang and F. Mueller, “Auto-generation and auto-tuning of 3d stencil codes on gpu clusters,” in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pp. 155–164, ACM, 2012.

[54] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, “Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines,” in *Acm Sigplan Notices*, vol. 48, pp. 519–530, ACM, 2013.

[55] M. Christen, O. Schenk, and H. Burkhart, “Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures,” in *2011 IEEE International Parallel & Distributed Processing Symposium*, pp. 676–687, IEEE, 2011.

[56] U. Bondhugula, V. Bandishti, and I. Pananilath, “Diamond tiling: Tiling techniques

to maximize parallelism for stencil computations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 5, pp. 1285–1298, 2016.

[57] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gomez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for cuda," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, p. 54, 2013.

[58] S. Shrestha, G. R. Gao, J. Manzano, A. Marquez, and J. Feo, "Locality aware concurrent start for stencil applications," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 157–166, IEEE, 2015.

[59] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*, pp. 285–297, IEEE, 1999.

[60] G. Bilardi and F. P. Preparata, "Lower bounds to processor-time tradeoffs under bounded-speed message propagation," in *Workshop on Algorithms and Data Structures*, pp. 1–12, Springer, 1995.

[61] M. Frigo and V. Strumpen, "Cache oblivious stencil computations," in *ICS*, vol. 5, pp. 361–366, Citeseer, 2005.

[62] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The pochoir stencil compiler," in *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pp. 117–128, ACM, 2011.

[63] D. Wonnacott, "Achieving scalable locality with time skewing," *International Journal of Parallel Programming*, vol. 30, no. 3, pp. 181–221, 2002.

[64] Y. Song and Z. Li, “New tiling techniques to improve cache temporal locality,” *ACM SIGPLAN Notices*, vol. 34, no. 5, pp. 215–228, 1999.

[65] G. Jin, J. Mellor-Crummey, and R. Fowler, “Increasing temporal locality with skewing and recursive blocking,” in *SC’01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, pp. 57–57, IEEE, 2001.

[66] 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*, pp. 153–162, ACM, 2001.

[67] R. Andonov, S. Balev, S. Rajopadhye, and N. Yanev, “Optimal semi-oblique tiling,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 9, pp. 944–960, 2003.

[68] T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege, “Hybrid hexagonal/classical tiling for gpus,” in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, p. 66, ACM, 2014.

[69] H. Dursun, M. Kunaseth, K.-i. Nomura, J. Chame, R. F. Lucas, C. Chen, M. Hall, R. K. Kalia, A. Nakano, and P. Vashishta, “Hierarchical parallelization and optimization of high-order stencil computations on multicore clusters,” *The Journal of Supercomputing*, vol. 62, no. 2, pp. 946–966, 2012.

[70] A. Schäfer and D. Fey, “High performance stencil code algorithms for gpgpus,” *Procedia Computer Science*, vol. 4, pp. 2027–2036, 2011.

[71] H. Su, N. Wu, M. Wen, C. Zhang, and X. Cai, “On the gpu performance of 3d stencil computations implemented in opencl,” in *International Supercomputing Conference*, pp. 125–135, Springer, 2013.

[72] Y. Li, L. Schwiebert, E. Hailat, J. Mick, and J. Potoff, “Improving performance of gpu code using novel features of the nvidia kepler architecture,” *Concurrency and Computation: Practice and Experience*, vol. 28, no. 13, pp. 3586–3605, 2016.

[73] F. Zheng, X. Xu, Y. Yang, S. He, and Y. Zhang, “Accelerating biological sequence alignment algorithm on gpu with cuda,” in *Computational and Information Sciences (ICCIS), 2011 International Conference on*, pp. 18–21, IEEE, 2011.

[74] J. Kloetzli, B. Strege, J. Decker, and M. Olano, “Parallel longest common subsequence using graphics hardware.,” in *EGPGV*, pp. 57–64, 2008.

[75] M. A. Zidan, T. Bonny, and K. N. Salama, “High performance technique for database applications using a hybrid gpu/cpu platform,” in *Proceedings of the 21st edition of the great lakes symposium on Great lakes symposium on VLSI*, pp. 85–90, ACM, 2011.

[76] A. Tomiyama and R. Suda, “Automatic parameter optimization for edit distance algorithm on gpu,” in *International Conference on High Performance Computing for Computational Science*, pp. 420–434, Springer, 2012.

[77] S. Deorowicz, “Solving longest common subsequence and related problems on graphical processing units,” *Software: Practice and Experience*, vol. 40, no. 8, pp. 673–700, 2010.

[78] F. C. Crow, "Summed-area tables for texture mapping," in *ACM SIGGRAPH computer graphics*, vol. 18, pp. 207–212, ACM, 1984.

[79] P. Sobolevsky, S. Bakhanovich, and A. Gorbach, "Tiling optimization in numerically solving a multidimensional heat equation on a ring of processors," *Cybernetics and Systems Analysis*, vol. 46, no. 1, pp. 145–152, 2010.

[80] V. Volkov, *Understanding latency hiding on gpus*. PhD thesis, UC Berkeley, 2016.

[81] L. Maignan and J.-B. Yunes, "Moore and von neumann neighborhood n-dimensional generalized firing squad solutions using fields," in *2013 First International Symposium on Computing and Networking*, pp. 552–558, IEEE, 2013.

[82] D. A. Zaitsev, "A generalized neighborhood for cellular automata," *Theoretical Computer Science*, vol. 666, pp. 21–35, 2017.

[83] P. Micikevicius, "3d finite difference computation on gpus using cuda," in *Proceedings of 2nd workshop on general purpose processing on graphics processing units*, pp. 79–84, ACM, 2009.

[84] Y. Yang, H.-M. Cui, X.-B. Feng, and J.-L. Xue, "A hybrid circular queue method for iterative stencil computations on gpus," *Journal of Computer Science and Technology*, vol. 27, no. 1, pp. 57–74, 2012.

[85] V. Bandishti, I. Pananilath, and U. Bondhugula, "Tiling stencil computations to maximize parallelism," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, IEEE, 2012.

[86] A. McAdams, E. Sifakis, and J. Teran, "A parallel multigrid poisson solver for fluids simulation on large grids," in *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pp. 65–74, Eurographics Association, 2010.

[87] E. Sozer, C. Brehm, and C. C. Kiris, "Gradient calculation methods on arbitrary polyhedral unstructured meshes for cell-centered cfd solvers," in *52nd Aerospace Sciences Meeting*, p. 1440, 2014.

# ABSTRACT

## TILING OPTIMIZATION FOR NESTED LOOPS ON GPUS

by

**YUANZHE LI**

**May 2020**

**Advisor:** Dr. Loren Schwiebert

**Major:** Computer Science

**Degree:** Doctor of Philosophy

Optimizing nested loops has been considered as an important topic and widely studied in parallel programming. With the development of GPU architectures, the performance of these computations can be significantly boosted with the massively parallel hardware. General matrix-matrix multiplication is a typical example where executing such an algorithm on GPUs outperforms the performance obtained on other multicore CPUs. However, achieving ideal performance on GPUs usually requires a lot of human effort to manage the massively parallel computation resources. Therefore, the efficient implementation for optimizing nested loops on GPUs became a popular topic in recent years. We present our work based on the tiling strategy in this dissertation to address three kinds of popular problems. Different kinds of computations bring in different latency issues where dependencies in the computation may result in insufficient parallelism and the performance of computations without dependencies may be degraded due to intensive memory accesses. In this thesis, we tackle the challenges for each kind of problems and believe that other computations performed in nested loops can also benefit from the presented techniques.

We improve a parallel approximation algorithm for the problem of scheduling jobs on parallel identical machines to minimize makespan with a high-dimensional tiling method. The algorithm is designed and optimized for solving this kind of problem efficiently on GPUs. Because the algorithm is based on a higher-dimensional dynamic programming approach, where dimensionality refers to the number of variables in the dynamic programming equation characterizing the problem, the existing implementation suffers from the pain of dimensionality and cannot fully utilize GPU resources. We design a novel data-partitioning technique to accelerate the higher-dimensional dynamic programming component of the algorithm. Both the load imbalance and exceeding memory capacity issue are addressed in our GPU solution. We present performance results to demonstrate how our proposed design improves the GPU utilization and makes it possible to solve large higher-dimensional dynamic programming problems within the limited GPU memory. Experimental results show that the GPU implementation achieves up to $25\times$ speed up compared to the best existing OpenMP implementation.

In addition, we focus on optimizing wavefront parallelism on GPUs. Wavefront parallelism is a well-known technique for exploiting the concurrency of applications that execute nested loops with uniform data dependencies. Recent research on such applications, which range from sequence alignment tools to partial differential equation solvers, has used GPUs to benefit from the massively parallel computing resources. Wavefront parallelism faces the load imbalance issue because the parallelism is passing along the diagonal. The tiling method has been introduced as a popular solution to address this issue. However, the use of hyperplane tiles increases the cost of synchronization and leads to poor data locality. In this paper, we present a highly optimized implementation of the wave-

front parallelism technique that harnesses the GPU architecture. A balanced workload and maximum resource utilization are achieved with an extremely low synchronization overhead. We design the kernel configuration to significantly reduce the minimum number of synchronizations required and also introduce an inter-block lock to minimize the overhead of each synchronization. We evaluate the performance of our proposed technique for four different applications: Sequence Alignment, Edit Distance, Summed-Area Table, and 2D-SOR. The performance results demonstrate that our method achieves speedups of up to six times compared to the previous best-known hyperplane tiling-based GPU implementation.

Finally, we extend the hyperplane tiling to high order $2$D stencil computations. Unlike wavefront parallelism that has dependence in spatial dimension, dependence remains only across two adjacent time steps along the temporal dimension in stencil computations. Even if the no-dependence property significantly increases the parallelism obtained in the spatial dimensions, full parallelism may not be efficient on GPUs. Due to the limited cache capacity owned by each streaming multiprocessor, full parallelism can be obtained on global memory only, which has high latency to access. Therefore, the tiling technique can be applied to improve the memory efficiency by caching the small tiled blocks. Because the widely studied tiling methods, like overlapped tiling and split tiling, have considerable computation overhead caused by load imbalance or extra operations, we propose a time-skewed tiling method, which is designed upon the GPU architecture. We work around the serialized computation issue and coordinate the intra-tile parallelism and inter-tile parallelism to minimize the load imbalance caused by pipelined processing. Moreover, we address the high-order stencil computations in our development, which has not been comprehensively studied. The proposed method achieves up to $3.5\times$ performance improve-

ment when the stencil computation is performed on a Moore neighborhood pattern.

## AUTOBIOGRAPHICAL STATEMENT

Yuanzhe Li received his Bachelor's degree in Computer Science in 2012 from Xidian University, Xi'an, Shaanxi, China. He received a Master's degree in Computer Science in 2014 from Wayne State University, Detroit, Michigan, USA. His research interests include parallel computing, GPU computing, performance optimization, GPU-CPU heterogeneous system, etc.