SCALABLE PARALLEL TRIDIAGONAL ALGORITHMS WITH DIAGONAL PIVOTING AND THEIR OPTIMIZATION FOR MANY-CORE ARCHITECTURES

BY

LI-WEN CHANG

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Adviser:

Professor Wen-mei W. Hwu

ABSTRACT

Tridiagonal solvers are important building blocks for a wide range of scientific applications that are commonly performance-sensitive. Recently, many-core architectures, such as GPUs, have become ubiquitous targets for these applications. Therefore, a high-performance general-purpose GPU tridiagonal solver becomes critical. However, no existing GPU tridiagonal solver provides comparable quality of solutions to most common, general-purpose CPU tridiagonal solvers, like Matlab or Intel MKL, due to no pivoting. Meanwhile, conventional pivoting algorithms are sequential and not applicable to GPUs.

In this thesis, we propose three scalable tridiagonal algorithms with diagonal pivoting for better quality of solutions than the state-of-the-art GPU tridiagonal solvers. A SPIKE-Diagonal Pivoting algorithm efficiently partitions the workloads of a tridiagonal solver and provides pivoting in each partition. A Parallel Diagonal Pivoting algorithm transforms the conventional diagonal pivoting algorithm into a parallelizable form which can be solved by high-performance parallel linear recurrence solvers. An Adaptive R-Cyclic Reduction algorithm introduces pivoting into the conventional R-Cyclic Reduction family, which commonly suffers limited quality of solutions due to no applicable pivoting. Our proposed algorithms can provide comparable quality of solutions to CPU tridiagonal solvers, like Matlab or Intel MKL, without compromising the high throughput GPUs provide.

To my parents and my family
for their unconditional love and support.

ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Wen-mei W. Hwu, for his tremendous mentorship and support. He has always motivated my work and been patient with me. His wisdom will keep inspiring me in my professional career and personal life.

I would like to thank all members of the IMPACT research group, past and present, for their help and camaraderie. They are, in no particular order, Chris Rodrigues, Sara Baghsorkhi, Alex Papakonstantinou, John Stratton, I-Jui Sung, Xiao-Long Wu, Nady Obeid, Victor Huang, Deepthi Nandakumar, Hee-Seok Kim, Nasser Anssari, Tim Wentz, Daniel Liu, Izzat El Hajj, Steven Wu, Abdul Dakkak, Wei-Sheng Huang, Carl Pearson, Tom Jablin, and Jon Larson.

Also, I want to thank Marie-Pierre Lassiva-Moulin, Andrew Schuh, and Xiaolin Liu for their help, and convey my gratitude to a lot of people I met during my time here, including Wooil Kim, Chia-Jen Chang, Chih-Sheng Lin, Judit Planas, Jose Cecilia, Juan Gómez Luna, Javier Cabezas, Isaac Gelado, Nacho Navarro, Hiroyuki Takizawa, Xuhao Chen, and those people I forget to mention or do not know the names of, especially the cleaner of my office.

Finally, I would like to thank my parents and my significant other for their unconditional love and support. Thank you everyone.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

A-R-CR	Adaptive R-Cyclic Reduction
CF	Continued Fraction
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
CR	Cyclic Reduction
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit
HSA	Heterogeneous System Architecture
MATLAB	Matrix Laboratory
MAGMA	Matrix Algebra on GPU and Multicore Architectures
MKL	Math Kernel Library
MPI	Message Passing Interface
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
PCR	Parallel Cyclic Reduction
PDE	Partial Differential Equation
RHS	Right Hand Side
SM	Streaming Multiprocessor
SPMD	Single Program, Multiple Data
TR	Tricyclic Reduction

CHAPTER 1

INTRODUCTION

Tridiagonal solvers are ubiquitous routines for performance-sensitive scientific computing. Partial differential equation (PDE) solvers are the most common applications applying tridiagonal solvers, since PDEs with neighboring references can be modeled as one or multiple tridiagonal systems [1, 2]. Moreover, tridiagonal solvers commonly serve as pre-conditioners in iterative numerical solvers [3]. Interpolation, such as cubic spline interpolation [4], is also a classic application of tridiagonal solvers for deriving coefficients or interpolated values. Parallelization is the most intuitive way to accelerate performance-sensitive computing. Recently, many-core architectures, like graphics processing units (GPUs), have dominated parallelized computational domains, not only because of the high-throughput computation and memory system GPUs deliver, but also because of general-purpose programming model, such as CUDA [5] and OpenCL [6], GPUs provide. However, developing an efficient parallel tridiagonal solver is challenging due to inherent dependence within a tridiagonal matrix, and implementing it on a massively parallel architecture like a GPU is even more challenging.

Most GPU tridiagonal algorithm research, especially the early studies [4, 7, 8, 9, 3, 10, 11, 12, 13], focused on applications solving a massive number of independent tridiagonal matrices. This strategy exploits embarrassing parallelism from independent matrices for efficiently utilizing GPUs. However, the performance might become limited when the number of independent matrices shrinks. Also, when applications solve multiple huge matrices, the limited size of GPU device memory may prefer to solve one entire large matrix at a time on a GPU instead of multiple portions from multiple independent matrices, to avoid redundant data transfer between host and device. In other words, when the size of single matrix increases, the number of independent matrices solved on a GPU decreases. Therefore, those studies for a massive number of matrices only provide limited, context-specific applications.

In order to build a more broadly applicable tridiagonal solver, it is necessary to develop a scalable tridiagonal solver for one or a few large matrices. However, only few of recent GPU tridiagonal solvers [14, 15, 16, 17] can efficiently solve a single large tridiagonal matrix. A scalable partitioning method is extremely critical for solving a single large matrix efficiently. Beyond partitioning, low memory requirement and high memory efficiency are also important for a tridiagonal solver, since most tridiagonal algorithms are memory-bound. Most previous studies, however, are still far from peak GPU memory bandwidth.

Most importantly, to the best of our knowledge, all existing GPU tridiagonal solvers exhibit a potential problem for providing unstable numerical solutions for general matrices. It is mainly because the algorithms, the *LU* algorithm or the Cyclic Reduction algorithm [1, 18], used by those solvers are known to be numerically unstable for certain distributions of matrices. Although, in practice, those tridiagonal solvers can be still applicable for specific kinds of matrices, such as strictly column diagonally dominant matrices, a stable tridiagonal solver is still extremely important for general applications on GPUs. Pivoting algorithms, such as partial pivoting or diagonal pivoting, are widely used to maintain numerical stability for CPU tridiagonal solvers but are less naturally suited to GPUs due to inherent dependence of those algorithms. A recent development [19] of linear algebra libraries for heterogeneous architectures (CPUs and GPUs) applied hybridization of partial pivoting on CPU and the rest parallel computation on GPUs to multiple classical routines. This strategy might not be efficient for tridiagonal solvers due to the relatively high ratio of data transfer overhead between CPU and GPU to kernel computation time on GPUs. In other words, in order to build an efficient, numerically stable tridiagonal solver, it is crucial to develop a scalable GPU pivoting algorithm.

In this thesis, we investigate the possibility of efficient pivoting algorithms on GPUs by using the diagonal pivoting method [20], which potentially has a more cache-friendly memory access pattern due to no row interchange. Three parallel tridiagonal algorithms related to the diagonal pivoting method are proposed. First, we propose a SPIKE-based tridiagonal solver on GPUs (Chapter 3). The SPIKE algorithm [21, 22] partitions a large matrix into multiple small disjoint sub-matrices, which can be solved in parallel, and provides a scalable GPU solution for a tridiagonal solver. Most importantly, diagonal pivoting can be further enabled in each disjoint sub-matrix for a more numerically stable solution. Second, we develop a scalable parallel diagonal pivoting algorithm for a single tridiagonal matrix (Chapter 4). This algorithm models the diagonal pivoting method as one or few linear recurrences, and then solves them using scalable parallel linear recurrence solvers. Third, we design an adaptive R-Cyclic Reductions algorithm with diagonal pivoting (Chapter 5). In this algorithm, the R value (1 or 2) of the R-Cyclic Reduction algorithm [23, 1, 18] changes row-by-row by the structure (1-by-1 or 2-by-2 pivots) of diagonal pivoting. This strategy can avoid numerical instability of the classic Cyclic Reduction or R-Cyclic Reduction algorithms.

This thesis makes the following contributions:

- We first introduce pivoting methods into GPU tridiagonal solvers. To the best of our knowledge, none of existing GPU tridiagonal solvers applies pivoting method. All three of our proposed tridiagonal algorithms provide comparable quality of solutions to the CPU tridiagonal solvers in Intel MKL [24] and Matlab [25].

- We first introduce the SPIKE algorithm into the field of GPU computing. The SPIKE algorithm is a powerful domain partitioning algorithm, which provides high scalability and has only a low overhead. Our SPIKE-based tridiagonal solver demonstrates comparable performance to the art-of-the-state GPU tridiagonal solvers.

- We first parallelize the full-range diagonal pivoting method. A parallel diagonal pivoting algorithm is proposed to provide the functionality of diagonal pivoting within the full range of matrix. Our parallel diagonal pivoting algorithm can benefit from the scalable parallel linear recurrence solvers, and demonstrates comparable performance to the art-of-the-state GPU tridiagonal solvers.

- We first enable a pivoting method in the R-Cyclic Reduction algorithms. The R-Cyclic Reduction algorithms are conventionally considered to have a potential numerically instability problem. An adaptive R-Cyclic Reduction algorithm is proposed to enable the functionality of pivoting.

- We introduce multiple optimization mechanisms to improve memory efficiency or reduce memory requirement. Multiple optimization mechanisms are applied to enable scalable pivoting on GPU tridiagonal solvers. Our optimization strategies are not only applicable to GPU tridiagonal solvers and/or GPU diagonal pivoting, but also able to support general computation on other architectures.

In the rest of thesis, we use the following terminology. Each function executed on a GPU device is called a “kernel” function, written in a single-program, multiple-data (SPMD) form in CUDA or OpenCL. Each instance of a kernel is executed by a “thread”, and a group of threads is called a “thread block”, guaranteed to perform concurrently on the same streaming multiprocessor (SM). Within a thread block, subgroups of threads called “warps” are guaranteed to perform in lockstep, executing one instruction for all threads in the warp at once. If threads in a warp have branch divergence, those branches are executed one-by-one for all of the threads in that warp.

Commonly, one GPU device contains multiple SMs. Each SM has a register file for data or instruction operands privately accessed by each thread and scratchpad memory for shared data among threads within a thread block. A modern GPU SM tends to have an L1 cache, shared by thread blocks executed in the same SM. Multithreading switches different warps (within a thread block or among thread blocks) to hide the latency of memory or computation. A shared L2 cache exists among all or some SMs on a GPU device, while global memory is shared among all SMs.

The remaining chapters in the thesis are organized as follows: Chapter 2 defines the terminology of tridiagonal solvers and reviews the selected algorithms related to our proposed algorithms. Chapters 3, 4 and 5 describe our proposed algorithms, the SPIKE-Diagonal Pivoting algorithm, the Parallel Diagonal Pivoting algorithm, and the Adaptive R-Cyclic Reduction algorithm, respectively, and cover their optimization strategies and limitations. Chapter 6 evaluates our proposed algorithms in terms of numerical stability and single GPU performance. Chapter 7 covers related work and Chapter 8 concludes.

CHAPTER 2

RELATED TRIDIAGONAL SOLVER ALGORITHMS

This chapter covers all of related tridiagonal algorithms to our work. Although, as mentioned in Chapter 1, modern parallel tridiagonal solvers can solve multiple independent matrices simultaneously, for a simpler explanation, a single matrix case is used in the following thesis without extra annotation. In the single matrix case, the tridiagonal solver solves a **nonsingular** tridiagonal system, $Tx = d$, where T is an n-by-n tridiagonal matrix, and both x and d are one-column vectors with n elements. Equation 2.1 defines the scalar entries in T, x and d. Note that both rows and columns of T, and entries of x and d begin at 0 and end at $n-1$.

$$Tx = \begin{bmatrix} b_0 & c_0 & & & \\ a_1 & b_1 & c_1 & & \\ & a_2 & \ddots & \ddots & \\ & & \ddots & \ddots & c_{n-2} \\ & & & a_{n-1} & b_{n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_{n-1} \end{bmatrix} = d \tag{2.1}$$

2.1 Parallel Tridiagonal Solver Algorithms

The selected parallel tridiagonal solver algorithms are categorized into three major types of algorithms, which are SPIKE-based, Linear recurrence-based, and R-Cyclic Reduction algorithms, for a systematic explanation. In the following section, only one representative algorithm per category is mainly discussed, and the corresponding modified or extended algorithms are further discussed when they are related.

2.1.1 SPIKE Algorithm

The SPIKE algorithm [21, 22], illustrated in Figure 2.1, is a domain decomposition algorithm that partitions a band matrix into multiple disjoint block diagonal sub-matrices with the rest off-diagonal sub-matrices. In this thesis, we only focus on the tridiagonal matrix, which is a special type of band matrices. The original tridiagonal matrix, T, is decomposed into multiple **invertible** block diagonal sub-matrices, T_i, and off-diagonal entries, a_{hi} and c_{ti}. Meanwhile, the corresponding vectors, x and d, have similar partitions, x_i and d_i. In the sub-matrix, the partial results y_i's can be defined for satisfying Equation 2.2.

$$T_i y_i = d_i \tag{2.2}$$

The matrix, T, can be further defined as the product of two matrices, the block-diagonal matrix, D, and the spike matrix, S, illustrated in Figure 2.1, where v_i and w_i can be solved by Equation 2.3.

$$T_i v_i = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ c_{ti} \end{bmatrix} \text{ and } T_i w_i = \begin{bmatrix} a_{hi} \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{2.3}$$

Note that, in practice, Equation 2.2 and 2.3 can be further merged and solved in one routine, since they share the same matrix T_i.

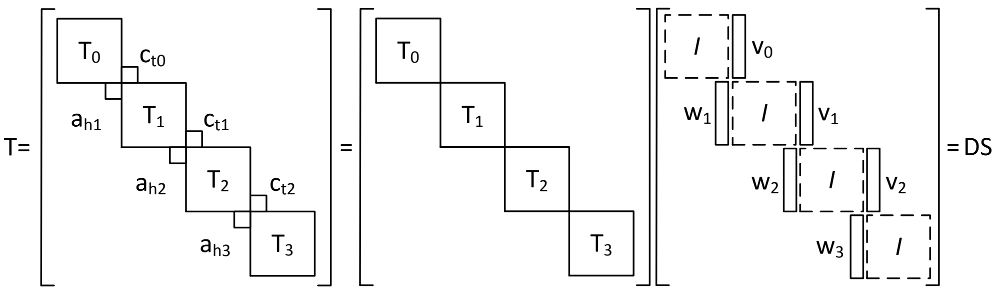

Figure 2.1: The SPIKE algorithm partitions a tridiagonal matrix T into 4 diagonal sub-matrices, and forms $T = DS$, where D is a block diagonal matrix and S is a spike matrix.

After forming the matrices D and S, the SPIKE algorithm solves $Dy = d$ for y, and then uses a special form of S to solve $Sx = y$ [22]. The spike matrix, S, can also be considered a block tridiagonal matrix and can be solved by a block tridiagonal solver algorithm, such as the block Cyclic Reduction (block CR) algorithm [1]. Since S has identity block diagonal sub-matrices and single-column off-diagonal sub-matrices, the computation for solving S can be further reduced to solving the dependencies among the first and last variables, called the glue variables (superscript with (h) and (t)), of each block, $\hat{S}$ in Equation 2.4. This strategy can dramatically reduce the computation cost from $O(n)$ to $O(P)$, where n is the size of a matrix and P is the number of partitions. The complete solution of x can be solved by propagating the glue variables.

$$\hat{S} = \begin{bmatrix} 1 & 0 & v_0^{(h)} & & & & & \\ 0 & 1 & v_0^{(t)} & & & & & \\ & w_1^{(h)} & 1 & 0 & v_1^{(h)} & & & \\ & w_1^{(t)} & 0 & 1 & v_1^{(t)} & & & \\ & & & w_2^{(h)} & 1 & 0 & v_2^{(h)} & \\ & & & w_2^{(t)} & 0 & 1 & v_2^{(t)} & \\ & & & & & w_3^{(h)} & 1 & 0 \\ & & & & & w_3^{(t)} & 0 & 1 \end{bmatrix} \tag{2.4}$$

It is worth mentioning that since the SPIKE algorithm is a domain decomposition algorithm, it does not limit applied algorithms to solve Equation 2.2 and 2.3, and $Sx = y$. Different applied solvers might give different numerical stability for the entire tridiagonal solver.

2.1.2 Linear Recurrence-based Tridiagonal Solver: Parallel Continued Fraction-*LU* Algorithm

The parallel Continued Fraction (CF)-*LU* algorithm [26] is a tridiagonal solver algorithm based on the parallel *LU* decomposition algorithm. In the *LU* decomposition, the original tridiagonal matrix, T, is decomposed into a product of two bidiagonal triangular matrices, L and U, Equation 2.5. The *LU* algorithm solves $Ly = d$ for y and then $Ux = y$.

$$L = \begin{bmatrix} 1 & & & & \\ l_1 & 1 & & & \\ & l_2 & 1 & & \\ & & \ddots & \ddots & \\ & & & l_{n-1} & 1 \end{bmatrix} \text{ and } U = \begin{bmatrix} f_0 & c_0 & & & \\ & f_1 & c_1 & & \\ & & \ddots & \ddots & \\ & & & f_{n-2} & c_{n-2} \\ & & & & f_{n-1} \end{bmatrix} \tag{2.5}$$

The coefficients l_i's and f_i's can be defined as follows:

$$l_i = a_i/f_{i-1},\ 1 \le i \le n-1 \text{ and } f_i = \begin{cases} b_0, i = 0 \\ b_i - l_i c_{i-1}, 1 \le i \le n-1 \end{cases} \tag{2.6}$$

The coefficients f_i's can be further formulated as continued fractions, Equation 2.7.

$$f_i = \begin{cases} b_0, i = 0 \\ b_i - \dfrac{a_i c_{i-1}}{f_{i-1}}, 1 \le i \le n-1 \end{cases} \tag{2.7}$$

Meanwhile, f_i's can be also defined as $f_i = \theta_i/\theta_{i-1}$, and θ_i can be solved as a linear recurrence, Equation 2.8.

$$\theta_i = \begin{cases} 1, i = -1 \\ b_0, i = 0 \\ b_i\theta_{i-1} - a_i c_{i-1}\theta_{i-2}, 1 \le i \le n-1 \end{cases} \tag{2.8}$$

The parallel CF-*LU* algorithm parallelizes the *LU* decomposition by computing coefficients l_i's and f_i's in parallel using parallel linear recurrence solvers [27, 28].

Also, the bidiagonal solver for L and U can be formulated to linear recurrences, as follows:

$$y_i = \begin{cases} d_0, i = 0 \\ -l_i y_{i-1} + d_i, 1 \le i \le n-1, \end{cases} \tag{2.9}$$

$$x_{n-i-1} = \begin{cases} y_{n-1}, i = 0 \\ -\dfrac{c_{n-i-1}}{f_{n-i-1}} x_{n-i} + y_{n-i}, 1 \le i \le n-1 \end{cases} \tag{2.10}$$

Therefore, the parallel CF-*LU* algorithm can solve a tridiagonal matrix in parallel.

The CF-*LU* algorithm is not the first linear recurrence-based tridiagonal solver. Before the CF-*LU* algorithm, the linear recurrence-based tridiagonal solvers [29, 30] tend to formulate x_i's as Equation 2.11.

$$x_i = \begin{cases} -\frac{b_0}{c_0}x_0 + \frac{d_0}{c_0}, i = 1 \\ -\frac{b_{i-1}}{c_{i-1}}x_{i-1} - \frac{a_{i-1}}{c_{i-1}}x_{i-2} + \frac{d_{i-1}}{c_{i-1}}, 1 < i \leq n-1 \end{cases} \tag{2.11}$$

All x_i's can be represented as a linear composition of x_0 by solving the linear recurrence (Equation 2.11), and x_0 can be solved using the last row, $a_{n-1}x_{n-2} + b_{n-1}x_{n-1} = d_{n-1}$, of T. After that, x_i's can be solved based on its own linear composition of x_0. Although these algorithms require only one linear recurrence, they suffer from severe numerical instability even in a strictly column diagonally dominant matrix due to using c_i's as denominators.

2.1.3 R-Cyclic Reduction Algorithm

The R-Cyclic Reduction (R-CR) algorithms [23, 1, 18] are a family of algorithms, including the classic Cyclic Reduction (CR) algorithm [23, 1, 18] ($R = 2$) and the Tricyclic Reduction (TR) algorithm [31] ($R = 3$). For a simpler explanation, we first explain the CR algorithm in detail, and then generalize the concept to the R-CR algorithms by comparing the TR algorithm with the CR algorithm.

The CR algorithm recursively eliminates the variables in radix-2 rows of a tridiagonal matrix using its adjacent two radix-2 rows; e.g., the variables of Row $k2^p$ can be eliminated using Row $k2^p + 2^{p-1}$ and Row $k2^p - 2^{p-1}$, where p is the level of recursion, k is a positive integer, and an undefined row index can be directly ignored. After a level of recursion, an effective tridiagonal matrix can be extracted in terms of a half unknown variables of the matrix before the recursion. Figure 2.2 illustrates the first level of CR on a 4-by-4 tridiagonal matrix, which is reduced to a 2-by-2 then a 1-by-1 effective matrix. These processes form a forward reduction phase.

When the effective matrix becomes 1-by-1 or small enough to be solved directly, the corresponding variables can be solved. After a effective matrix is solved, the resolved variables can be used to solve a larger effective matrix, which is the matrix before the recursive step. Using the same example in Figure 2.2, we can get x_0 and x_2 by solving the small effective matrix. After that, x_0 and x_2 can be used to solve x_1 and x_3 using Row 1 and 3 on the large

$$\begin{bmatrix} b_0 & c_0 & & \\ a_1 & b_1 & c_1 & \\ & a_2 & b_2 & c_2 \\ & & a_3 & b_3 \end{bmatrix} \rightarrow \begin{bmatrix} b'_0 & 0 & c'_0 & \\ a_1 & b_1 & c_1 & \\ a'_2 & 0 & b'_2 & 0 \\ & & a_3 & b_3 \end{bmatrix} \Rightarrow \begin{bmatrix} b'_0 & c'_0 \\ a'_2 & b'_2 \end{bmatrix}$$

Figure 2.2: One level of the CR forward reduction on a 4-by-4 matrix: a_2 and c_2 on Row 2 are eliminated by Row 1 and 3. Similarly, c_0 is eliminated by Row 1. After that, Row 0 and 2 can form an effective 2-by-2 matrix.

matrix, since the variables (x_0 and x_2) coupled with the coefficients a_1, a_3, and c_1 are known. Therefore, the effective matrices can be solved from small to large recursively. The solving processes form a backward substitution phase.

The TR algorithm is similar to the CR algorithm but recursively eliminates the variable in radix-3 rows instead of radix-2 rows. Row $k3^p + 1$ and $k3^p + 2$ locally eliminate the variables of each other, Figure 2.3. The results of this local elimination can be used to eliminate Row 0 and 3, and meanwhile are commonly not stored back for reducing memory requirement. Figure 2.4 shows the computed results in the memory system, and the last small matrix is the corresponding effective matrix of the TR algorithm.

$$\begin{bmatrix} b_0 & c_0 & & & & \\ a_1 & b_1 & c_1 & & & \\ & a_2 & b_2 & c_2 & & \\ & & a_3 & b_3 & c_3 & \\ & & & a_4 & b_4 & c_4 \\ & & & & a_5 & b_5 \end{bmatrix} \rightarrow \begin{bmatrix} b_0 & c_0 & & & & \\ a'_1 & 1 & 0 & c'_1 & & \\ a'_2 & 0 & 1 & c'_2 & & \\ & & a_3 & b_3 & c_3 & \\ & & & a'_4 & 1 & 0 \\ & & & a'_5 & 0 & 1 \end{bmatrix}$$

Figure 2.3: The local elimination of TR forward reduction on a 6-by-6 matrix One level of CR forward reduction on a 6-by-6 matrix: Row 1 and Row 2 can eliminate each other, and Row 4 and Row 5 can eliminate each other.

By increasing the number of rows for elimination, the performance might be improved due to the number of recursive levels decreased. Different rows or variables in the same level of recursion can be eliminated or solved in parallel, while ones in the different level might be in sequential. The *R*-Cyclic Reduction algorithms generalize the idea by recursively eliminating

$$\begin{bmatrix} b_0 & c_0 & & & & \\ a_1 & b_1 & c_1 & & & \\ & a_2 & b_2 & c_2 & & \\ & & a_3 & b_3 & c_3 & \\ & & & a_4 & b_4 & c_4 \\ & & & & a_5 & b_5 \end{bmatrix} \rightarrow \begin{bmatrix} b'_0 & 0 & 0 & c'_0 & & \\ a_1 & b_1 & c_1 & & & \\ & a_2 & b_2 & c_2 & & \\ a'_3 & 0 & 0 & b'_3 & 0 & 0 \\ & & & a_4 & b_4 & c_4 \\ & & & & a_5 & b_5 \end{bmatrix} \Rightarrow \begin{bmatrix} b'_0 & c'_0 \\ a'_3 & b'_3 \end{bmatrix}$$

Figure 2.4: One level of the TR forward reduction on a 6-by-6 matrix: a_3 and c_3 on Row 2 are eliminated by Row 1 and 4 from the left matrix in Figure 2.3. Similarly, c_0 is eliminated by Row 1 from Figure 2.3. After that, Row 0 and 3 can form an effective 2-by-2 matrix.

variables for the radix-R rows for a tridiagonal matrix.

Two modifications of the CR algorithm are worth mentioning. First, the Parallel Cyclic Reduction (PCR) algorithm [32] recursively eliminates variables in all rows instead of the radix-2 rows in the CR algorithm. Therefore, the PCR algorithm can avoid the backward substitution phase, but introduces more computation in the forward reduction phase. In the end, the PCR algorithm tends to have higher total complexity when the size of a matrix increases. Second, the block CR algorithm solves a block tridiagonal matrix, instead of a (scalar) tridiagonal matrix, by extending scalar operations into matrix (or block) operations. For a special form of block tridiagonal matrices, e.g. a spike matrix in the SPIKE algorithm, the block CR algorithm can be further simplified.

2.2 Diagonal Pivoting Algorithm

The diagonal pivoting algorithm [20] for tridiagonal solvers is proposed to improve numerical stability of the classic LU algorithm. Although the LU algorithm with partial pivoting provides a similar quality of solution and is widely used for tridiagonal solvers on CPUs, it is not efficiently applicable on GPUs due to inefficient memory operations of row interchange. The diagonal pivoting algorithm avoids row interchange by selecting a 1-by-1 or 2-by-2 diagonal pivot blocks. The original tridiagonal matrix can be factorized using Equation 2.12.

$$T = \begin{bmatrix} P_d & C \\ A & T_r \end{bmatrix} = \begin{bmatrix} I_d & 0 \\ AP_d^{-1} & I_{n-d} \end{bmatrix} \begin{bmatrix} P_d & 0 \\ 0 & T_s \end{bmatrix} \begin{bmatrix} I_d & P_d^{-1}C \\ 0 & I_{n-d} \end{bmatrix}, \tag{2.12}$$

where P_d is either a 1-by-1 ($\begin{bmatrix} b_0 \end{bmatrix}$) or a 2-by-2 ($\begin{bmatrix} b_0 & c_0 \\ a_1 & b_1 \end{bmatrix}$) pivot block, and the Schur complement T_s can be defined in Equation 2.13.

$$T_s = T_r - AP_d^{-1}C = \begin{cases} T_r - \dfrac{a_1 c_0}{b_0} e_1^{(n-1)} e_1^{(n-1)T}, \text{for 1-by-1 } P_d \\ T_r - \dfrac{a_2 b_0 c_1}{\Delta} e_1^{(n-2)} e_1^{(n-2)T}, \text{for 2-by-2 } P_d, \end{cases} \tag{2.13}$$

where $\Delta = b_0 b_1 - a_1 c_0$, $e_1^{(k)}$ is the first column vector of the k-by-k identity matrix, and the superscript T means a transpose matrix.

Since T_s is still tridiagonal (Equation 2.13) and can be further factorized using Equation 2.12, the original matrix T can be recursively factorized in LBM^T, where L and M are lower triangular matrices and B is a block-diagonal matrix with either 1-by-1 or 2-by-2 blocks. After the LBM^T factorization, the diagonal pivoting algorithm solves $Lz = d$ for z, then $By = z$ for y, and finally $M^T x = y$.

Multiple applicable pivoting criteria [20] to determine how to choose 1-by-1 or 2-by-2 P_d in Eq. 2.12 have been proposed. In this thesis, we implement the asymmetric Bunch-Kaufman pivoting. The asymmetric Bunch-Kaufman pivoting chooses a 1-by-1 pivot at Row 0 if the leading entry b_0 is sufficiently large relative to adjacent entries, i.e., $|b_0| \sigma \geq \kappa |a_1 c_0|$, where $\sigma = max \{|a_1|, |a_2|, |b_1|, |c_0|, |c_1|\}$ and $\kappa = \frac{\sqrt{5}-1}{2} \approx 0.62$. Otherwise, a 2-by-2 pivot is chosen.

CHAPTER 3

SPIKE-DIAGONAL PIVOTING

As mentioned in Chapter 1, scalable partitioning and supporting pivoting are extremely important for building a broadly applicable tridiagonal solver. However, partitioning is still challenging due to extra overheads introduced from sharing data, communication, and increased complexity. Only few previous studies can efficiently partition the workload of a tridiagonal solver. Adding a pivoting support for a tridiagonal solver is even more challenging. To the best of our knowledge, no existing GPU tridiagonal solvers support pivoting. The main reason is because most of the existing tridiagonal solvers use the R-CR algorithms, especially the CR or PCR algorithms. Since the R-CR algorithms have a regular access pattern with small data sharing in a level of recursion, the workloads can be partitioned in a near-trivial way. However, the same fact makes pivoting almost impossible. These motivate us to investigate another scalable partitioning method for a pivoting support.

We propose a SPIKE-Diagonal Pivoting tridiagonal solver for GPUs. The SPIKE algorithm (Section 2.1.1) can decompose a tridiagonal matrix into multiple disjoint block diagonal sub-matrices, which can be solved in parallel. The SPIKE algorithm guarantees high scalability due to its disjoint partitioning and its low overhead for solving the spike matrix S. Then, the diagonal pivoting algorithm (Section 2.2) is applied to solve each block diagonal sub-matrices (Equation 2.2 and 2.3 in Section 2.1.1) for guaranteeing numerical stability in each sub-matrix. Our solver is the first tridiagonal solver using the SPIKE algorithm and/or a pivoting method on GPUs.

Since the sub-matrices are solved independently in the SPIKE algorithm and each of them is solved by the sequential diagonal pivoting method, it is intuitive to let each thread solve a sub-matrix. In order to maximize performance, three optimization strategies are applied. First, since each

Parts of this chapter appeared in the International Conference on High Performance Computing, Networking, Storage and Analysis [33]. The material is used with permission.

sub-matrix in the SPIKE algorithm contains consecutive rows, which are stored consecutively in memory and solved by a thread, therefore, threads in a warp have a strided access pattern, which dramatically reduces GPU memory performance. A data layout transformation method (Section 3.2.1) is applied to ensure our kernels have a stride-one memory access pattern and gain the most bandwidth possible from the GPU memory. Second, the heavily data-dependent control flow in the thread-parallel diagonal pivoting method causes thread divergence and further widely scattered memory accesses, which makes GPU caches nearly useless. A dynamic tiling technique (Section 3.2.2) is introduced to keep divergence under control so the GPU caches can be effectively utilized and deliver competitive memory performance. Last, since our tridiagonal solver is memory-bound, a kernel fusion strategy is proposed to further reduce memory bandwidth requirement by merging the LBM^T decomposition and the following L and B solvers.

In the end of this chapter, we discuss the potential extensions and/or limitations for the SPIKE-Diagonal Pivoting algorithm. These not only demonstrate applicability the SPIKE-Diagonal Pivoting algorithm can provide, but also lead us to the next proposed algorithm, the Parallel Diagonal Pivoting algorithm.

3.1 SPIKE-Diagonal Pivoting Algorithm

The SPIKE-Diagonal Pivoting algorithm is a hybrid algorithm of the SPIKE algorithm and the diagonal pivoting method. The flowchart of the algorithm is shown in Figure 3.1. A **massive** number of sub-matrices decomposed by the SPIKE algorithm can fully utilize GPUs. The partitioning step determines the number of sub-matrices by the degree of thread-level parallelism of GPUs. Also, to ensure a stride-one memory access pattern for the major kernels, the partitioning step marshals the data using the number of sub-matrices.

Then, the thread-parallel sub-matrix solver for the collection of independent sub-matrices T_i's computes both the solutions y_i (Equation 2.2) to the sub-matrices as well as the w_i and v_i components (Equation 2.3) of the spike matrix S. It can be considered as a RHS matrix with 3 column vectors that are solved together. In our SPIKE-Diagonal Pivoting algorithm, the diago-

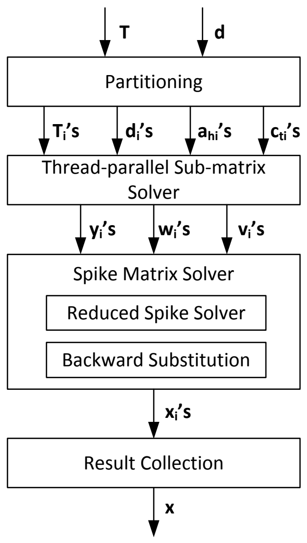

Figure 3.1: The flowchart of the SPIKE-Diagonal Pivoting algorithm.

nal pivoting method is applied. For w_i, since only the leading entry (a_{hi}) is nonzero, the computation for solving L can be further simplified. Similarly, for v_i, the last entry (c_{ti}) is nonzero, so the computation for solving L, B, and M^T can be reduced. Therefore, the overhead of solving 3 column vectors over the cost of solving 1 column vector is ignorable.

After all sub-matrices are solved, the spike solver for solving $Sx = y$ is invoked, first solving the reduced system (Equation 2.4), then followed by the computation of all remaining variables through backward substitution. This strategy dramatically reduces the computation cost for solving $Sx = y$. Therefore, the thread-parallel sub-matrix solver constitutes the majority of the computation in our solver. Finally, the result collection step marshals the computed results back to the original layout, since the data layout has been changed by the partitioning step.

3.2 Optimization for SPIKE-Diagonal Pivoting Algorithm

Three optimization mechanisms are proposed to boost the performance of our SPIKE-Diagonal Pivoting solver. Data layout transformation is introduced to improve memory efficiency for regular memory access patterns. Then, dynamic tiling is proposed to keep divergence under control and improve memory efficiency for irregular memory access patterns. Meanwhile, kernel fusion is applied to reduce memory bandwidth requirement.

3.2.1 Data Layout Transformation

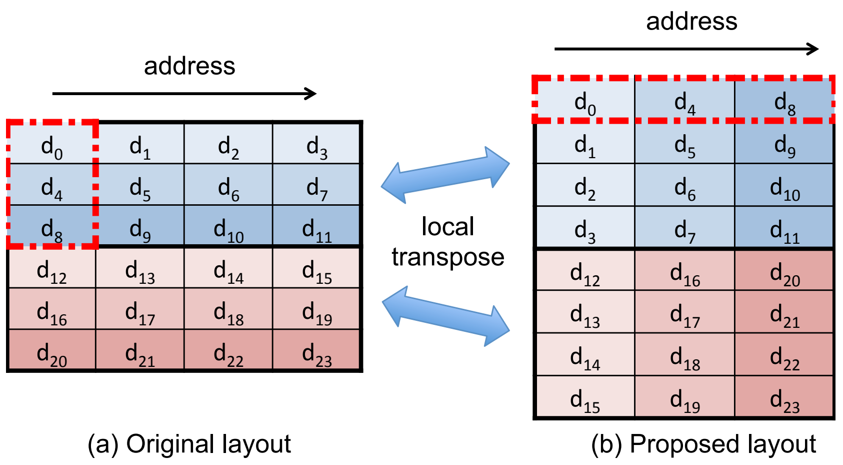

Figure 3.2: An illustration for data layout transformation for 2 warps. Each warp contains 3 threads, and each thread solves a 4-row sub-matrix.

For the GPU memory system to perform well, the memory address accessed by a warp should be either coalesced or very close together. However, in our thread-parallel solver, adjacent rows are accessed by a single thread sequentially, and widely spread rows are simultaneously accessed by a warp. Therefore, in the partitioning step of the proposed algorithm, these spread rows are marshaled into consecutive addresses by data layout transformation. Figure 3.2 illustrates the relationships between layouts before and after data marshaling with 2 warps, each containing 3 threads, each solving a 4-row sub-matrix. In the original layout, all coefficients (d_i) are placed into consecutive locations, such as d_0, d_1, d_2, and d_3. When the threads start accessing data from the left (dashed-boxed) column, a strided access pattern is created. In our proposed data layout, a local transpose on the data in each warp relocates a column to a row, which provides a coalesced memory access. This local transpose is similar to the Array of Structures of Tiled Arrays layout analyzed by Sung et al. [34].

Figure 3.3 demonstrates the performance improvement on an NVIDIA

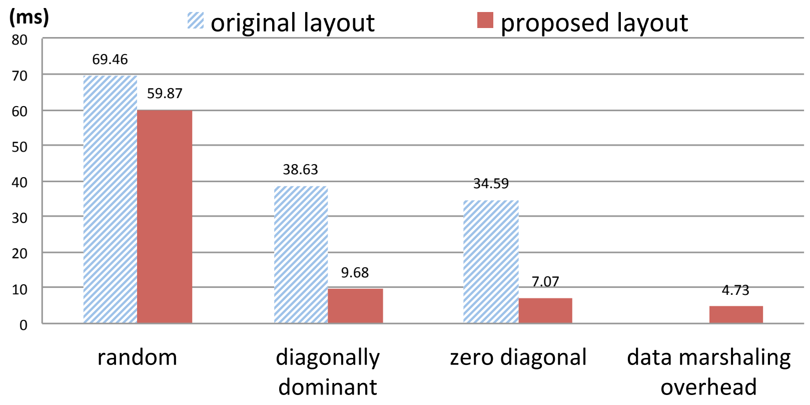

Figure 3.3: Performance improvement in the thread-parallel sub-matrix solver by applying data layout transformation on 3 types of 8M-size matrices.

GTX480 GPU for the proposed data layout transformation by comparing three kinds of matrices (8 million rows), which are either randomly generated, strictly column diagonally dominant, or having a zero diagonal with nonzero off-diagonal entries. The random matrix has strong branch divergence due to randomly chosen pivots, while the strictly column diagonally dominant matrix and the zero diagonal matrix result in no branch divergence with always 1-by-1 diagonal pivots or 2-by-2 pivots, respectively. Our proposed data layout outperforms the original one up to 4.89×, due to better memory efficiency. For the regular memory accesses without branch divergence, such as the strictly column diagonally dominant matrix and the zero diagonal matrix, our proposed data layout shows perfectly coalesced memory accesses and gives speedups over the original one by a factor of 3.99-4.89×. A small overhead of data marshaling is also observed, and it might be amortizable in iterative applications. On the random matrix, which has irregular memory accesses due to branch divergence, the performance improvement drops to only 1.16×. Since the memory access pattern is irregular and data-dependent, the (static) data layout transformation can only provide limited performance improvement. At the same time, since each thread on the random matrix chooses 1-by-1 or 2-by-2 pivots independent of other threads, the cache lines of data required by each thread on a particular iteration become fragmented and cause the low cache efficiency.

3.2.2 Dynamic Tiling

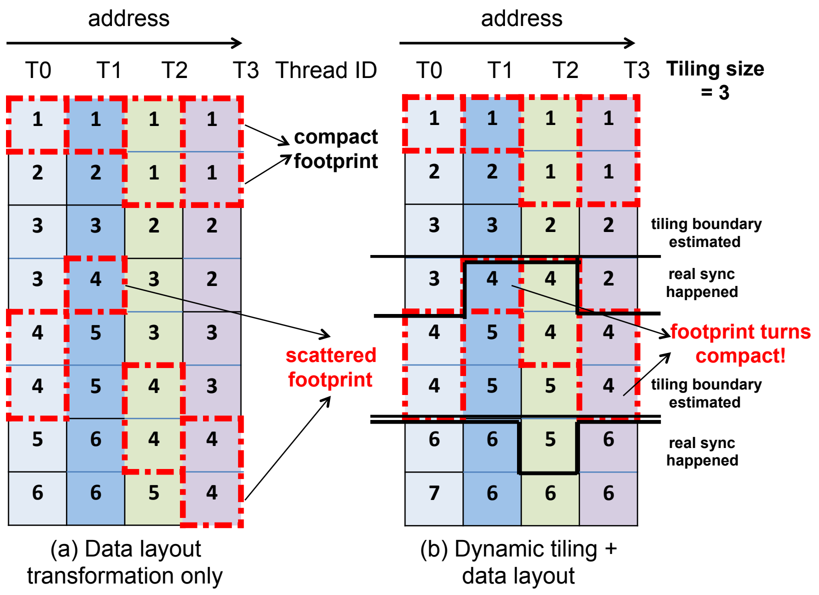

Figure 3.4: An illustration for dynamic tiling for 4 threads.

Figure 3.4(a) illustrates the fragmented memory footprint on the random matrix when 4 threads iterate through rows of their sub-matrices in the proposed layout of the previous section. The numbers in the boxes represent the number of iteration. In Iteration 1, the threads touch two lines of data. Although the access is not perfectly coalesced, the GPU L1 cache mechanism can still deliver reasonable bandwidth. Data accessed on Iteration 1 can be cached and might be used by Thread $T0$ and $T1$ on Iteration 2 if the cache line is not replaced. However, as threads consume different numbers of rows through each iteration, by the time they get to Iteration 4, the accessed data are scattered across 5 lines. If data touched by the threads are far enough apart, the number of cache lines to avoid evicting before all threads consume their data increases. At the same time, since the threads bring more cache lines into the L1 cache, it also increases the chances of evicting useful cache lines. Therefore, when the memory access footprint exceeds a certain number, which is related to the cache capacity and the number of scheduled threads, the memory system performance degrades significantly.

In order to counteract this problem, we propose a dynamic tiling mechanism, which bounds the size of the memory access footprint of the threads in a warp by inserting an intentional barrier synchronization for a warp. Listing 3.1 and 3.2 show the relationships between pseudo codes before and after dynamic tiling. Since a warp executes in lockstep, it is always synchronized if no branch divergence. Therefore, we change the loop structure of iterations to make the control flow converged every L^{th} ($L = 3$ in our example in Figure 3.4(b)) cache line the threads in a warp access. Here, k is the local row index in each sub-matrix, and is used to represent the memory footprint of each thread. Note k does not represent the index of iteration. Since the original `while` loop becomes two nested loops, a hidden barrier synchronization is automatically inserted before the `for` loop ends. This barrier synchronization for a warp between the smaller `while` loops bounds the number of cache line the threads access by forcing "fast" threads to wait for "slow" threads within a warp. At the first barrier synchronization in Figure 3.4(b), two fast threads (Thread T_2 and T_3) idle one iteration to wait for the other two slow threads (Thread T_0 and T_1). After dynamic tiling, the size of the footprint is bounded by a tuned tiling parameters and the footprint becomes more compact.

Listing 3.1: Baseline

```
k=0
while(k<n) {
   if(condition) {
      1-by-1 pivoting
      k+=1
   } else {
      2-by-2 pivoting
      k+=2
   }
}
```

Listing 3.2: Dynamic tiling

```
k=0
for(i=0; i<n/L; i++) {
   n_barrier=(i+1)*L
   while(k<n_barrier) {
      if(condition) {
         1-by-1 pivoting
         k+=1
      } else {
         2-by-2 pivoting
         k+=2
      }
   }
   //a hidden barrier
   //for a warp
}
```

Figure 3.5 demonstrates the performance improvement of dynamic tiling. Our dynamic tiling strategy provides an extra 3.56× speedup on top of data layout transformation when branch divergence is heavy, and shows a com-

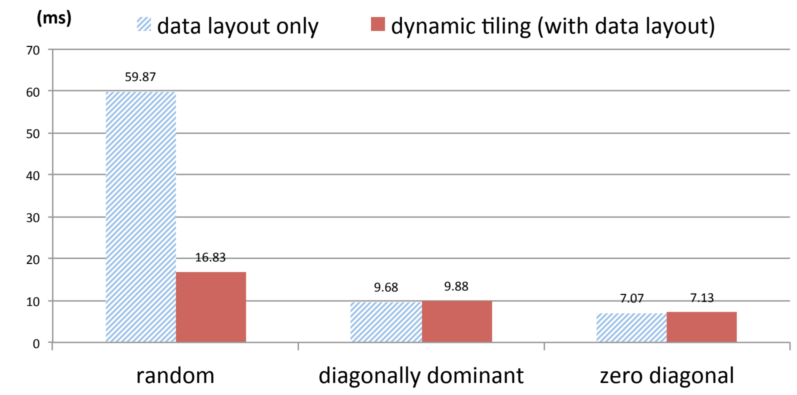

Figure 3.5: Performance improvement in the thread-parallel sub-matrix solver by applying dynamic tiling on 3 types of 8M-size matrices.

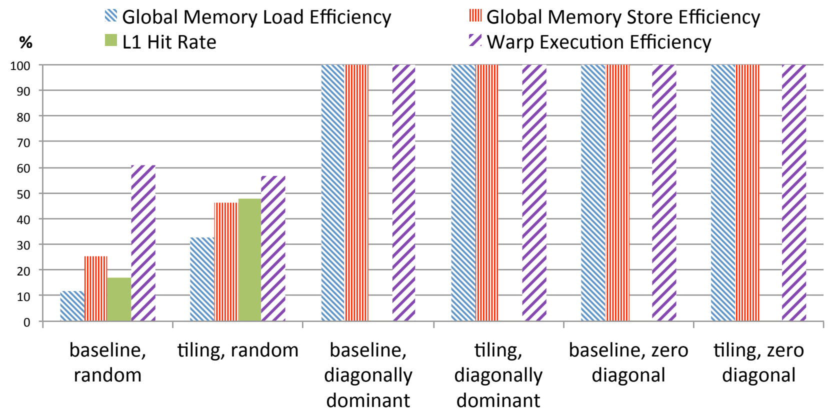

Figure 3.6: Performance counters for dynamic tiling.

pletely ignorable overhead when there is no divergence. Figure 3.6 further analyzes dynamic tiling using the hardware performance counters from the NVIDIA Visual Profiler [35]. The non-divergent matrices exhibited perfect memory utilization with or without dynamic tiling, as expected, and therefore they have the 0% L1 rates because of coalescing memory accesses. Without dynamic tiling, the random matrix can only have 11.7% and 25.3% in the global memory load efficiency and global memory store efficiency respectively. The low L1 cache hit rate of 17.2% implies that the memory footprint is too large that the cache could not contain all data, evicting cache lines before the data are consumed. By applying dynamic tiling, the global memory load efficiency and global memory store efficiency are improved from 11.7% to 32.5% and 25.3% to 46.1% respectively. At the same time, the L1 cache hit rate is significantly boosted from 17.2% to 48.0%, with only a minor decrease in warp execution efficiency (branch divergence). These metrics support our conclusion that dynamic tiling improves memory efficiency by more effectively utilizing the hardware cache, and show that the random matrix can have compatible performance to the non-divergent matrices by applying dynamic tiling.

An alternative dynamic scratchpad tiling might be applied to replace our proposed dynamic (cache) tiling by moving data into scratchpad memory instead of relying on L1 cache. Using scratchpad instead of L1 caches might avoid possible cache evicting for useful data, but might cause extra costs for scratchpad memory access requests and a small unused portion of scratchpad due to irregular accesses. On the other hand, dynamic scratchpad tiling might be extremely useful for a GPU without L1 caches or with L1 caches that do not support global accesses. One example for the latter case is the NVIDIA Kepler architecture. The L1 cache on Kepler is accessible only by local memory accesses, such as register spills and stack data, but not by global memory accesses. Therefore, different GPU architectures might benefit one or the other.

3.2.3 Kernel Fusion

Since the L and B solvers access the same data and have a memory access pattern similar to that of the LBM^T decomposition, these two parts can be merged to reduce memory bandwidth requirement. Kernel fusion might degrade kernel performance due to increased resource pressure. However, in our case, the LBM^T decomposition tends to have much higher resource pressure than the L and B solvers and can tolerate the increased resource the kernel fusion introduces. Therefore, kernel fusion is completely applicable in our case, and can significantly reduce memory bandwidth requirement. To be clear, after the kernel fusion, the matrices L and B can be consumed on the fly without storing back in memory. However, the matrix U and the pivoting structure have to be stored in memory and would not be used until the whole LBM^T decomposition completes.

3.3 Extensions of SPIKE-based Algorithm

Multiple extensions are further investigated. Our SPIKE-Diagonal Pivoting method can be extended to support multi-GPUs or heterogeneous clusters (CPUs + GPUs) by using OpenMP and MPI. Also, since the SPIKE algorithm does not limit applied solvers for sub-matrices, either the LU algorithm (Thomas algorithm) or the CR algorithms can be used for sub-matrix solvers. However, as mentioned in Chapter 1, in either the LU algorithm or the CR algorithm, stable solutions are only guaranteed for strictly column diagonally dominant matrices. Two extensions, SPIKE-Thomas [33] and SPIKE-CR [36] solvers, can provide higher throughputs over the existing tridiagonal solvers on diagonally dominant matrices. Last, for multiple right-hand-side (RHS) vectors, the LBM^T decomposition can be shared by multiple RHS vectors.

3.3.1 Multi-GPU and Cluster Extension

Our SPIKE-Diagonal Pivoting algorithm can support multi-GPUs or heterogeneous clusters (CPUs + GPUs). OpenMP [37] can be applied for multi-threads or multi-GPUs in a node, while MPI [38] can be used for communication among different nodes. Intel MKL `gtsv` is used for CPUs, while our thread-parallel diagonal pivoting solver is applied for GPUs.

3.3.2 SPIKE-Thomas Algorithm

Since the SPIKE algorithm does not restrict the algorithm applied for the sub-matrix solver, the Thomas algorithm (LU algorithm) can be applied to replace the diagonal pivoting algorithm if the input matrix is known as a strictly column diagonally dominant matrix. Under this situation, considering the branch of 1-by-1 pivot is always taken in the diagonal pivoting algorithm, the Thomas algorithm can provide a simpler control flow, fewer memory accesses, and less register pressure than the original diagonal pivoting algorithm. By hard-coding the Thomas algorithm instead of the diagonal pivoting algorithm as the sub-matrix solver, performance can be further improved.

3.3.3 SPIKE-CR Algorithm

Similar to the SPIKE-Thomas algorithm, the CR algorithm can be applied as the sub-matrix solver if the input matrix is known as a strictly column diagonally dominant matrix. In the SPIKE-CR algorithm, each sub-matrix is solved collaboratively by threads in a thread block instead of being solved by a single thread. The CR algorithm potentially can provide finer parallelism within a sub-matrix than the Thomas algorithm. Also, since threads in a thread block access a single sub-matrix, it has a perfectly coalesced access pattern, requiring no data layout transformation. Moreover, both a_{hi} and c_{ti} can be considered as parts of computation in the CR algorithm, so there is no overhead for solving w_i and v_i. However, a high-performance CR-based tridiagonal solver requires more optimizations than a Thomas-based solver. Common optimization techniques for the CR algorithm can be applied to the SPIKE-CR Algorithm. First, a register packing mechanism [7] is widely applied to reduce the required size of scratchpad memory the CR algorithm desires. Also, a hybrid of CR-PCR [13] by applying a warp-level PCR can be used to avoid the potential low utilization within a warp in the CR algorithm. Another level of the SPIKE algorithm can be further applied to minimize communication cost among warps within a thread block. By applying the above optimization techniques, the SPIKE-CR algorithm can outperform the SPIKE-Thomas by a factor of 1.23×.

3.3.4 Multiple Right-Hand-Side Vectors

Since multiple RHS vectors share the same input matrix, which has the same diagonal pivots, there is no need to re-execute the LBM^T decomposition on multiple RHS vectors. Therefore, the matrices L and B might be stored back in memory for sharing across multiple RHS vectors.

3.4 Limitation of SPIKE Algorithm

One major limitation of the SPIKE-Diagonal Pivoting algorithm lies in the assumption that each sub-matrix has to be invertible (non-singular). In a general matrix, a singular sub-matrix might happen during partitioning. At the same time, since most SPIKE-based algorithms, including ours, partition a matrix only based on the size of the matrix, it is extremely difficult to eliminate this corner case. Therefore, in this situation, the undefined solution returned by the SPIKE-Diagonal Pivoting algorithm does not match the result computed by the sequential diagonal pivoting algorithm. This limitation leads us to parallelizing the full diagonal pivoting algorithm.

CHAPTER 4

PARALLEL DIAGONAL PIVOTING ALGORITHM

A small range (within a sub-matrix) of diagonal pivoting provides stable numerical solutions of sub-matrices in the SPIKE-Diagonal Pivoting algorithm without compromising parallelism. However, due to this limited range of diagonal pivoting, the SPIKE-Diagonal Pivoting algorithm might not give a comparable quality of solution to the conventional diagonal pivoting algorithm in a certain type of matrix. This motivates us to investigate a Parallel Diagonal Pivoting algorithm for providing pivoting within the whole range of a tridiagonal matrix instead of a single sub-matrix.

A classic failure for a domain decomposition method like the SPIKE algorithm is mainly caused by singular or near-singular partitioning. Although the disjoint partitioning without communication used the SPIKE algorithm leads us to a highly scalable tridiagonal algorithm, this pattern causes the singular partitioning. Therefore, instead of relying on the partitioning without communication, our Parallel Diagonal Pivoting algorithm uses linear recurrences, which can be solved in parallel with a small amount of communication between two adjacent partitions.

First, we model the diagonal pivoting algorithm as a linear recurrence. Inspired by the CF-LU Algorithm (Section 2.1.2), the LBM^T decomposition can be modeled as a second-order linear recurrence. A major challenge of modeling the LBM^T decomposition is its branch divergence. Although by definition a linear recurrence can have a condition, it may potentially degrade the performance, especially on GPUs. We propose a linear recurrence-based LBM^T decomposition without branch divergence. Then, since our method takes a CF-LU-like strategy, the method also suffers from the drawback of the parallel CF-LU algorithm, which is a potential overflow problem. In Equation 2.8, θ_i's might overflow due to the large-value coefficients a_i's, b_i's, and c_i's. In order to counteract this problem, a normalization strategy is introduced. Last, similar to the CF-LU algorithm, the solvers for L and

M^T can be represented as linear recurrences and the solver for B only has local computation. Therefore, the proposed linear recurrence-based diagonal pivoting algorithm is completely parallelizable.

Since our Parallel Diagonal Pivoting algorithm is tightly coupled with linear recurrences, the throughput of parallel linear recurrence solvers completely dominates the efficiency of our method. Three optimization techniques are proposed to customize the parallel linear recurrence solvers for our Parallel Diagonal Pivoting algorithm. First, for a second-order linear recurrence solver, communication exists among threads and even among thread blocks. Since each thread block executes independently in the modern GPU programming models, communication among thread blocks is not well-defined during kernel execution and requires kernel termination, which creates massive redundant memory accesses and huge memory bandwidth requirement. Two methods are proposed to address inter-block communication and boost memory throughput.

Second, though our solutions for inter-block communication are efficient, they are not cost-free. Given a fixed-size matrix, the frequency of inter-block communication is inversely proportional to the amount of data processed in a thread block. By tiling more data in a thread block, the total cost of inter-block communication decreases. In order to tile maximal data in a thread block, a unified tiling mechanism is proposed to fully utilize SM resources, including both registers and scratchpad. Overusing SM resources might impact GPU multi-threading and then downgrade performance significantly. Our proposed unified tiling mechanism can balance each resource usage to avoid overusing one or both of them.

Last, the L and B solvers can be further merged with the LBM^T decomposition phase to reduce memory bandwidth requirement. A major possible challenge for this kernel fusion is branch divergence of the L and B solvers. This branch divergence is unavoidable and that is the reason that dynamic tiling is proposed for the SPIKE-Diagonal Pivoting algorithm to improve its cache efficiency instead of eliminating the branch divergence. By merging the L and B solvers into the LBM^T decomposition, the branch divergence is reintroduced to our non-divergent decomposition. Another possible challenge is increased resource pressure, which might degrade kernel performance. A kernel fusion optimization is proposed to reduce memory bandwidth requirement without introducing high overheads.

Similar to the previous chapter, in the end of this chapter, we also discuss the potential extensions and/or limitations for the Parallel Diagonal Pivoting algorithm. These demonstrate the applicability of the algorithm.

4.1 Linear Recurrence-based Parallel Diagonal Pivoting Algorithm

4.1.1 Linear Recurrence Formulation

In the diagonal pivoting method (Section 2.2), the leading entry $\hat{b}_i$ of the Schur complement T_s (Equation 2.13) can be further defined in Equation 4.1. Without loss of generality, we assume a pivot happens at Row 0.

$$\hat{b}_i = \begin{cases} \hat{b}_1 = b_1 - \dfrac{a_1 c_0}{b_0}, \text{for a 1-by-1 pivot } P_d \\ \hat{b}_2 = b_2 - \dfrac{a_2 b_0 c_1}{\Delta}, \text{for a 2-by-2 pivot } P_d, \end{cases} \tag{4.1}$$

where $\hat{b}_i$ can be also considered as the leading entries of the sub-blocks in B, and apparently $\hat{b}_0$ can be defined as b_0. Since Equation 2.13 is used recursively to define the LBM^T decomposition, the coefficients $\hat{b}_i$ can be defined recursively using equations similar to Equation 4.1. Particularly when a 1-by-1 pivot happens, $\hat{b}_1$ becomes f_1 in Section 2.1.2. If 1-by-1 pivots are always picked, the LBM^T decomposition can be simplified into the LU decomposition, where U is equal to BM^T, and $\hat{b}_i$'s and f_i's become equivalent. In this case, θ_i of Equation 2.8 in Section 2.1.2 is well-defined and satisfies $\hat{b}_i = f_i = \theta_i / \theta_{i-1}$.

Now, considering a 2-by-2 pivot happens at b_0, we have $\hat{b}_2 = b_2 - \frac{a_2 b_0 c_1}{\Delta}$. Since $\theta_{-1} = 1$ and $\theta_0 = b_0$ are defined (in Equation 2.8), Δ is equal to θ_1, Equation 4.2.

$$\theta_1 = b_1 \theta_0 - a_1 c_0 \theta_{-1} = b_1 b_0 - a_1 c_0 = \Delta \tag{4.2}$$

Then, since θ_1 is nonzero, by multiplying θ_1 on both sides, we have the following equation,

$$\hat{b}_2 \theta_1 = b_2 \theta_1 - a_2 b_0 c_1 = b_2 \theta_1 - a_2 c_1 \theta_0 = \theta_2 \tag{4.3}$$

Therefore, $\hat{b}_2 = f_2 = \theta_2/\theta_1$ is still satisfied. The above deduction shows a 2-by-2 pivot does not change the leading entries of $\hat{b}_2$. To be clear, the $\hat{b}_2$ from one 2-by-2 pivot is the same as the $\hat{b}_2$ from two 1-by-1 pivots. A 2-by-2 pivot happening at b_0 only changes the definition of $\hat{b}_1$ without changing $\hat{b}_2$ if two 1-by-1 pivots happen at b_0. Due to a 2-by-2 pivot at b_0, $\hat{b}_1$ is not a leading entry anymore. More importantly, Equation 2.8 is still satisfied in the LBM^T decomposition, and can be used to compute the well-defined leading entries $\hat{b}_i$'s of B.

The linear recurrence of the LBM^T decomposition can be written as follows:

$$\theta_i = \begin{cases} 1, i = -1 \\ b_0, i = 0 \\ b_i\theta_{i-1} - a_i c_{i-1}\theta_{i-2}, 1 \leq i \leq n-1, \end{cases} \tag{4.4}$$

which is the same as Equation 2.8, and then

$$\hat{b}_i = \frac{\theta_i}{\theta_{i-1}}, \text{ for a 1-by-1 pivot at } i-1 \text{ or a 2-by-2 pivot at } i-2 \tag{4.5}$$

The pivoting criterion (Section 2.2) can be rewritten as follows:

$$|\theta_i|\,\sigma \geq \kappa\,|a_{i+1}c_i\theta_{i-1}|, \tag{4.6}$$

where $\sigma = max\{|a_{i+1}|, |a_{i+2}|, |b_{i+1}|, |c_i|, |c_{i+1}|\}$.

4.1.2 Linear Recurrence-based LBM^T Decomposition

There are two major types of parallel linear recurrence solvers, which are a group-structure and a tree-structure linear recurrence solvers. The group-structure solvers take a divide-and-conquer approach for computing propagation coefficients of different groups in parallel, while the tree-structure solvers execute feasible non-dependent computation or communication within a whole linear recurrence in parallel. Intuitively, the group-structure solvers are similar to the SPIKE algorithm, while the tree-structure solvers are similar to the R-Cyclic Reduction algorithms.

A hierarchal, hybrid-structure parallel linear recurrence solver is used for our high-performance LBM^T decomposition. The hierarchy of our algorithm is shown as follows: within each thread, a sequential solver is applied;

within each warp, either a group-structure or a tree-structure (using the Kogge-Stone algorithm [28]) solver is applied; within each thread block, a tree-structure (also using the Kogge-Stone algorithm) solver is applied for different warps; among thread blocks, a group-structure solver is applied with (sequential) producer-consumer communication.

4.1.3 Normalization to Avoid Overflowing

The variables θ's might overflow, since Equation 4.4 (or Equation 2.8) is only related to coefficients a_i's, b_i's and c_i's but no d_i's. If Row i of a matrix is multiplied by a large value P, θ_i and all θ's after i also increase by P. Considering multiple rows of a matrix are multiplied by large values, after a certain row j, θ_j and all θ's after j may overflow and then generate wrong coefficients $\hat{b}_i$'s or f_i's. However, in the conventional (sequential) LBM^T or LU decomposition, this overflow never happens. This is mainly because the conventional decomposition directly computes the leading coefficients $\hat{b}_i$'s or f_i's instead of using a linear recurrence of θ_i's. To counteract this problem, a normalization scheme is necessary to avoid θ overflowing. Intuitively, each θ_i should be normalized by θ_{i-1}, and then θ_i becomes $\hat{b}_i$. However, this strategy cannot work since $\hat{b}_i$ might not be well-defined for each i in the diagonal pivoting algorithm. For example, $\hat{b}_1$ is not a leading entry when a 2-by-2 pivot happens at Row 0. Therefore, a division by zero or a small value might happen at $\hat{b}_1$. To avoid this situation, we can use Equation 4.6 to enable normalization. Therefore, normalization does not happen at a non-leading entry.

Since a normalization scheme indeed is a division operation, it is expensive and might degrade the performance of our solver. An alternative normalization scheme, optionally performing a division when the values of θ or the coefficients are larger than a threshold, is proposed to avoid expensive division operations.

4.2 Optimization for Parallel Diagonal Pivoting Algorithm

Three optimization mechanisms are proposed to boost the performance of our Parallel Diagonal Pivoting algorithm. First, efficient inter-block communication for our producer-consumer communication pattern is introduced to reduce memory bandwidth requirement. Then, unified tiling is proposed to minimize the frequency and the total cost of inter-block communication. Kernel fusion is further applied to reduce memory bandwidth requirement.

4.2.1 Producer-Consumer Inter-block Communication

As mentioned above, inter-block communication is critical for our parallel linear recurrence-based diagonal pivoting algorithm. In our algorithm, a producer-consumer inter-block communication pattern is applied to minimize inter-block communication for a fixed number of thread blocks. In order to support this inter-block communication pattern, we propose the following two methods.

Lock-free Message Passing is proposed to avoid the high overhead of kernel termination for global synchronization. Different from conventional GPU lock-free inter-block communication mechanisms [39], which rely on memory fences or atomic operations, our message passing strategy packs messages (data for communication) and a ready signal into an uncached memory request. This can avoid high-cost memory fences and atomic operations, and then dramatically boost performance. Figure 4.1 illustrates our proposed method. Through micro-benchmarking, we recognize that a memory request on a small range of (aligned) consecutive addresses is always completed at the same time. A memory request is commonly formed by a whole cache line or word that is read or written at the same time. Since it is a single memory request, no other threads can see a partial result of this cache line or word. This is similar to the atomicity property in a database system. Therefore, this property can be used as a ready signal to confirm whether a message is ready. By using micro-benchmarking, the range is determined as 128 Bytes (a cache line size) for both NVIDIA Fermi and Kepler GPUs. A producer writes a package, containing both a signal and a message of data,

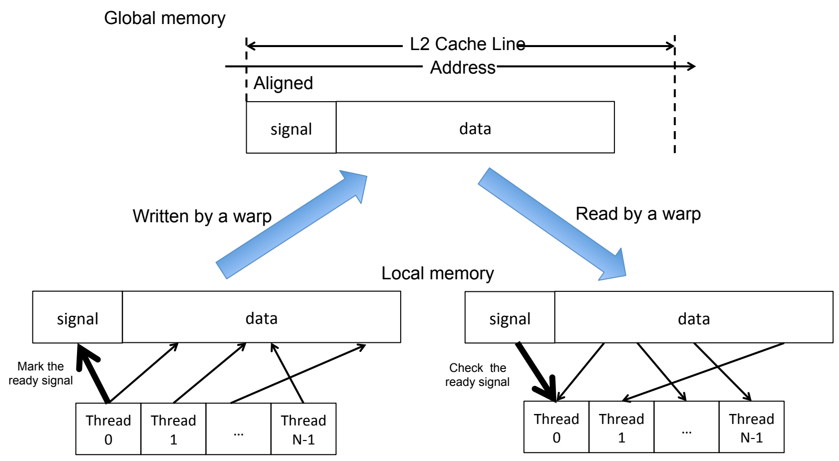

Figure 4.1: An illustration for lock-free message passing.

to a pre-determined address using a warp, which executes in lockstep, while a consumer keeps pooling the address to check the signal also using a warp. When the signal becomes ready, the consumer can confirm that it receives the correct message from the producer.

A similar method related to ours was proposed by Yan et al. [40]. Yan's method merges the single-variable message and the signal together, and a single address is used by both the message and signal at the same time. A specialized unused value of the message is chosen to represent the ready signal. Yan's method is not practical as our method, since not every message has an unused value for the ready signal, and Yan's method only supports single-variable messages. Our method splits addresses for the message and the signal, and then packs them into a single memory request. Our lock-free message passing method potentially can support different types communication patterns other than the producer-consumer pattern, but in this thesis it is applied only for the producer-consumer pattern.

Dynamic Parallelism is an alternative for implementing inter-block communication. Dynamic parallelism [41] is supported in NVIDIA Kepler GPUs to allow kernel launch within a kernel. This functionality can be applied for inter-block communication by launching a consumer kernel in a producer kernel. The message from the producer to the consumer can be passed through arguments of the consumer kernel. This strategy is extremely efficient due to eliminating pooling of the consumer. Different from our lock-free message passing method, allowing overlapping the consumer's execution with the producer's before the inter-block communication, dynamic parallelism serializes this overlapped part. Although this sequentialization might introduce a risk for degrading performance, massive multi-threading of GPUs might still fully utilize the memory bandwidth. Dynamic parallelism is only supported by a few most advanced GPUs. Therefore, for the GPUs without dynamic parallelism, our proposed lock-free message passing can be applicable.

High-throughput platform atomic operations supported by HSA [42] might be applied for either lock-free message passing or dynamic parallelism in the producer-consumer inter-block communication pattern. Also, a dedicated hardware support for GPU cache coherence from HSA might also simplify the implementation of this inter-block communication pattern.

4.2.2 Unified Tiling

Since inter-block communication is not completely cost-free, it is still important to control its usage rate, which is proportional to the number of thread blocks and is inversely proportional to the amount of data processed in a thread block. Therefore, intuitively, tiling more data processed in a thread block can decrease the frequency of inter-block communication, and then can further reduce the total communication cost. We propose a unified tiling mechanism that systematically tiles data to on-chip resources (registers and scratchpad) and fully utilizes these resources. In our unified tiling, registers and scratchpad are forming a unified tiling space (UTS) for data tiling. All private variables of a thread are stored in the UTS, and are assigned to either a register or scratchpad memory. This strategy eliminates the redundant storage if there are two copies.

The UTS strategy has the following benefits. First, since our method includes both registers and scratchpad memory, it can potentially generate a bigger tile in the UTS instead of two small tiles in the register file and scratchpad memory. Second, since the sizes of a register file and scratchpad memory vary from architecture to architecture, our method simplifies autotuning for different architectures. Third, our method can resolve avoidable register spilling or avoid low occupancies due to an imbalanced resource allocation, which means one resource is overused and the other resources are underutilized.

The UTS allocation can be implemented as a register allocation step of a compiler. Our method is implemented in an alternative way in this thesis as follows: All UTS variables are assigned into register file in the beginning, and the feedbacks of register pressure from the compiler are used to determine the UTS allocation. To address high register pressure, we intentionally spill private variables into scratchpad memory instead of global memory. The spilled variables are selected preferentially from the private variables used for data tiling due to their longer live ranges. Intuitively, our alternative UTS implementation can be considered as a simplified register allocator with register spilling on scratchpad memory. Also, the pressure of scratchpad memory is monitored to avoid low occupancies due to overusing scratchpad memory. To be precise, the scratchpad memory used for inter-block communication is pre-reserved and not considered as a part of the UTS. Our implementation may require multiple iterations for compiling since it uses the compiler feedback.

4.2.3 Kernel Fusion

As discussed in Section 3.2.3, kernel fusion is applicable for the LBM^T decomposition with the following L and B solvers. In our linear recurrence-based method, this merging might introduce the unavoidable branch divergence of the L and B solvers to the LBM^T decomposition. However, as we demonstrate in dynamic tiling, memory efficiency, instead of divergence, is the first-order effect for performance. By using the non-divergent LBM^T decomposition, the memory access pattern is already regularized. Therefore, the re-introduced branch divergence from the L and B solvers has a limited performance influence.

Since M^T is solved by a reverse-ordered linear recurrence solver, it cannot be merged with the other in-ordered linear recurrences. However, since a group-structure solver is applied among thread blocks and the LBM^T decomposition does access the same data the M^T solver uses, potentially the workload for computing propagation coefficients can be migrated into the LBM^T decomposition kernel. Due to the almost identical sizes of the propagation coefficients and the original matrix coefficients of M^T, there is no memory requirement reduction in this merging. On the other hand, this migration might change the ratio of the computation costs between the LBM^T decomposition kernel, which is already merged with the L and B solvers, and the M^T solver. Different sizes of matrices or different GPUs might prefer different kernel fusion.

4.3 Extensions of Linear Recurrence-based Tridiagonal Solvers

Similar to Section 3.3, multiple extensions are further investigated for the linear recurrence-based tridiagonal solvers. The parallel CF-LU algorithm is the most intuitive extension. Also, for RHS vectors, the LBM^T decomposition can be shared for multiple RHS vectors.

4.3.1 Parallel CF-LU Algorithm

Since the parallel CF-LU algorithm (Section 2.1.2) shares the same linear recurrence with the parallel diagonal pivoting algorithm, most of our optimization techniques discussed above are applicable for the parallel CF-LU algorithm. Since the CF-LU algorithm does not have divergence, multiple conditions and branches can be avoided. The parallel CF-LU algorithm can provide a stable solution if the input matrix is known as a strictly column diagonally dominant matrix.

4.3.2 Multiple Right-Hand-Side Vectors

Similar to the discussion in the SPIKE-Diagonal Pivoting algorithm (Section 3.3.4), there is no need to re-execute the LBM^T decomposition on multiple RHS vectors. The matrices L and B might be stored back in memory for sharing across multiple RHS vectors.

4.4 Limitation of Parallel Diagonal Pivoting Algorithm

One major limitation of the Parallel Diagonal Pivoting algorithm might be lack of high-performance extensions for multi-GPUs or heterogeneous clusters. In a GPU, results of adjacent groups in a group-structure linear recurrence solver are computed on adjacent thread blocks to avoid long latencies. However, in multi-GPUs, if consecutive groups are scheduled on the same GPU, that might cause long latencies across the groups among different GPUs; if interleaved groups are on the same GPU, that might require inter-GPU communication, which is not applicable without kernel termination for the current architectures. Kernel termination enables the extension for multi-GPUs but increases memory bandwidth requirement on each GPU. That becomes a trade-off. Similar to multi-GPUs, in multi-nodes, inter-node communication is required. Although MPI provides this functionality, kernel termination is still needed.

CHAPTER 5

ADAPTIVE R-CYCLIC REDUCTION ALGORITHM WITH DIAGONAL PIVOTING

Conventionally, the R-Cyclic Reduction algorithms (Section 2.1.3) are considered only working for strictly column diagonally dominant matrices. As mentioned in Chapter 3, using the R-CR algorithms, such as the CR or PCR algorithms, is the main reason causing no pivoting in most existing GPU tridiagonal solvers. In this chapter, we investigate the possibility of supporting pivoting in an R-CR-like algorithm.

We propose an Adaptive R-Cyclic Reduction (A-R-CR) algorithm with diagonal pivoting. Considering the R-Cyclic Reduction algorithms are a family of algorithms, each of which has a different R, we investigate a modified R-cyclic algorithm to dynamically support different R values on different rows. Inspired by the diagonal pivoting method, we recognize that different R values on different rows can be considered as different diagonal pivots.

Our A-R-CR algorithm is the first R-CR algorithm with pivoting. This chapter focuses on the design of our A-R-CR algorithm instead of its optimization techniques, since the optimization techniques for the R-CR algorithms are widely studied. Our A-R-CR algorithm currently requires the diagonal pivoting method to determine the structure of diagonal blocks, i.e. R values of rows. Although the diagonal pivoting method might be merged into the A-R-CR algorithm for reducing memory bandwidth requirement, it still introduces a large overhead. However, our A-R-CR algorithm is still applicable for those matrices with known or pre-computed pivoting structures. Also, the diagonal pivoting method might not be the most efficient pivoting strategy for the A-R-CR algorithm. We leave the investigation of other pivoting methods for the A-R-CR algorithm as future work.

5.1 Extended Support for Different R's in R-Cyclic Reduction Algorithm

In the following explanation, we use a terminology, m-(u, l), to define an R-CR operation eliminating dependence of an m-row block using u prior rows and l posterior rows. Apparently, 1-$(1, 1)$, 1-$(2, 2)$, 1-(R, R), and k-(k, k) operations can be defined for the conventional CR, TR, R-CR, and block CR with a block size k, respectively. Since only tridiagonal matrices are considered, each parameter of m, u, or l can be limited to either 1 or 2, resulting eight kinds of combination for different R-CR operations.

5.1.1 1-(u, l) R-Cyclic Reduction

The 1-(u, l) is considered operating on Row i by using information of Row $i - u$ to Row $i + l$ (Equation 5.1).

$$T = \begin{bmatrix} \ddots & \ddots & \ddots & & & & & \\ & a_{i-2} & b_{i-2} & c_{i-2} & & & & \\ & & a_{i-1} & b_{i-1} & c_{i-1} & & & \\ & & & a_i & b_i & c_i & & \\ & & & & a_{i+1} & b_{i+1} & c_{i+1} & \\ & & & & & a_{i+2} & b_{i+2} & c_{i+2} \\ & & & & & & \ddots & \ddots & \ddots \end{bmatrix} \tag{5.1}$$

Two diagonal blocks are chosen as follows:

$$D^- = \begin{cases} b_{i-1}, \text{for 1-by-1 } D^- \\ \begin{bmatrix} b_{i-2} & c_{i-2} \\ a_{i-1} & b_{i-1} \end{bmatrix}, \text{for 2-by-2 } D^- \end{cases} \tag{5.2}$$

and

$$D^+ = \begin{cases} b_{i+1}, \text{for 1-by-1 } D^+ \\ \begin{bmatrix} b_{i+1} & c_{i+1} \\ a_{i+2} & b_{i+2} \end{bmatrix}, \text{for 2-by-2 } D^+ \end{cases} \tag{5.3}$$

Here, we can assume both D^- and D^+ are invertible by choosing u and l wisely. If u is equal to 2, three linear operations for D^- can be further defined as follows:

$$\begin{aligned} D^- w^- = D^- \begin{bmatrix} w_h^- \\ w_t^- \end{bmatrix} &= \begin{bmatrix} a_{i-2} \\ 0 \end{bmatrix} \\ D^- v^- = D^- \begin{bmatrix} v_h^- \\ v_t^- \end{bmatrix} &= \begin{bmatrix} 0 \\ c_{i-1} \end{bmatrix} \\ D^- y^- = D^- \begin{bmatrix} y_h^- \\ y_t^- \end{bmatrix} &= \begin{bmatrix} d_{i-2} \\ d_{i-1} \end{bmatrix} \end{aligned} \tag{5.4}$$

Otherwise, $u = 1$, we have

$$\begin{aligned} D^- w^- = D^- w_h^- &= D^- w_t^- &= a_{i-1} \\ D^- v^- = D^- v_h^- &= D^- v_t^- &= c_{i-1} \\ D^- y^- = D^- y_h^- &= D^- y_t^- &= d_{i-1} \end{aligned} \tag{5.5}$$

Similarly, D^+ has the corresponding linear operations. In the end, the 1-(u,l) R-CR operation at Row i can be defined as follows:

$$\begin{aligned} b_i' &= b_i - a_i v_t^- - c_i w_h^+ \\ a_i' &= -a_i w_t^- \\ c_i' &= -c_i v_h^+ \\ d_i' &= d_i - a_i y_t^- - c_i y_h^+ \end{aligned} \tag{5.6}$$

5.1.2 2-(u,l) R-Cyclic Reduction

Then, we define the 2-(u,l) CR operates on Row i and Row $i+1$ using information from Row $i-u$ to Row $i+1+l$ (Equation 5.7).

$$T = \begin{bmatrix} \ddots & \ddots & \ddots & & & & & & & \\ & a_{i-2} & b_{i-2} & c_{i-2} & & & & & & \\ & & a_{i-1} & b_{i-1} & c_{i-1} & & & & & \\ & & & a_i & b_i & c_i & & & & \\ & & & & a_{i+1} & b_{i+1} & c_{i+1} & & & \\ & & & & & a_{i+2} & b_{i+2} & c_{i+2} & & \\ & & & & & & a_{i+3} & b_{i+3} & c_{i+3} & \\ & & & & & & & \ddots & \ddots & \ddots \end{bmatrix} \tag{5.7}$$

We can define similar diagonal blocks, $\overline{D}$'s, and vectors $\overline{w}$'s, $\overline{v}$'s, and $\overline{y}$'s. Then, $\overline{D}^-$, $\overline{w}^-$, $\overline{v}^-$ and $\overline{y}^-$ are identical as D^-, w^-, v^-, and y^-, respectively, while $\overline{D}^+$, $\overline{w}^+$, $\overline{v}^+$ and $\overline{y}^+$ are similar to D^+, w^+, v^+, and y^+, respectively, by increasing one row. To be clear, we define $\overline{D}^+$ as follows:

$$\overline{D}^+ = \begin{cases} b_{i+2}, \text{for 1-by-1 } \overline{D}^+ \\ \begin{bmatrix} b_{i+2} & c_{i+2} \\ a_{i+3} & b_{i+3} \end{bmatrix}, \text{for 2-by-2 } \overline{D}^+ \end{cases} \tag{5.8}$$

The 2-(u, l) R-CR at Row i and Row $i+1$ can be defined as follows:

$$\begin{aligned} b'_i &= b_i - a_i \overline{v}_t^- \\ a'_i &= -a_i \overline{w}_t^- \\ c'_i &= c_i \\ d'_i &= d_i - a_i \overline{y}_t^- \end{aligned} \tag{5.9}$$

and

$$\begin{aligned} b'_{i+1} &= b_{i+1} - c_{i+1} \overline{w}_h^+ \\ a'_{i+1} &= a_{i+1} \\ c'_{i+1} &= -c_{i+1} \overline{v}_h^+ \\ d'_{i+1} &= d_{i+1} - c_{i+1} \overline{y}_h^+ \end{aligned} \tag{5.10}$$

5.2 Structure of Diagonal Blocks

As mentioned above, the structure of diagonal blocks in our A-R-CR algorithm performs as diagonal pivots. In order to get a proper structure of diagonal blocks, the parallel diagonal pivoting algorithm or the conventional diagonal pivoting algorithm can be applied. However, similar to the conventional R-CR algorithm, the A-R-CR performs recursively. The structure of diagonal blocks might change in different levels of recursion. For example, a row in a 1-by-1 diagonal block in a level of recursion might form a 2-by-2 diagonal block with another row in another level of recursion. Therefore, after each level of recursion, the structure of diagonal blocks needs to be reorganized before another level of recursion is performed.

This restructuring diagonal blocks might limit the applicability of the A-

R-CR algorithm due to introducing a large pivoting overhead. On the other hand, the A-R-CR algorithm is still applicable to those matrices with known or pre-computed structures. One of the applications for the A-R-CR algorithm is solving a tridiagonal matrix with multiple RHS vectors, each of which shares the same structure of diagonal blocks.

5.3 Comparison with SPIKE Algorithm

The A-R-CR algorithm has multiple similarities with the SPIKE algorithm. Both of them decompose a matrix into multiple diagonal sub-matrices or diagonal blocks. Also, the SPIKE algorithm tends to use the block CR algorithm to solve the reduced spike matrix. Similarly, the A-R-CR algorithm solves a matrix recursively using adaptive R-CR operations.

However, the A-R-CR algorithm has few differences from the SPIKE algorithm and provides following benefits. First, the A-R-CR algorithm has a fine-grained and adaptive decomposition, only including either 1-by-1 or 2-by-2 diagonal blocks, while the SPIKE algorithm tends to use a coarse-grained and regular decomposition. Although the SPIKE algorithm does allow a fine-grained and adaptive decomposition, that might dramatically increase the complexity of solving the reduced spike matrix $\hat{S}$. Also, as mentioned in Section 3.4, the regular decomposition of the SPIKE algorithm might introduce singular sub-matrices, while the adaptive decomposition can avoid this situation in the A-R-CR algorithm. Second, the block CR algorithm used for solving the reduced spike matrix S in the SPIKE algorithm is not adaptive, while adaptive R-CR operations are used in each level of recursion of the A-R-CR algorithm. The non-adaptive block CR algorithm might also introduce an unstable numerical solution.

Due to the benefits of the adaptive decomposition, the A-R-CR algorithm can be applied to refine the SPIKE algorithm. One potential application is using the A-R-CR algorithm to replace the block CR algorithm for solving the reduced spike matrix $\hat{S}$. Although the A-R-CR algorithm has a higher overhead than block CR algorithm, the overhead is ignorable due to the small size of $\hat{S}$. Another potential application is applying the adaptive structure of diagonal pivots, which in fact is a part of diagonal pivoting, to avoid the singular partitioning in the SPIKE algorithm.

5.4 Optimization for Adaptive R-Cyclic Reduction Algorithm

Most of the optimization techniques applied to the conventional CR algorithm potentially can be applied to the A-R-CR algorithm. However, register packing, as one critical optimization technique for the CR algorithm, might not be applicable due to its requirement for a regular access pattern. Conventional scratchpad tiling is more suitable for the A-R-CR algorithm due to its support for an irregular access pattern. Depending on the size of a matrix, partitioning methods, such as [14, 13], might be applicable.

5.5 Limitation of Adaptive R-Cyclic Reduction Algorithm

As mentioned in the above sections, the A-R-CR algorithm restructures diagonal blocks in each level of recursion and introduces the overhead for adaptive decomposition. Also, irregularity of adaptive decomposition can cause heavy unavoidable branch divergence and might impact the performance on current GPU architectures. These two factors limit the A-R-CR algorithm only applicable for small tridiagonal matrices or those matrices with known structures. In order to counteract these limitations, an efficient pivoting strategy for the A-R-CR algorithm needs to be further investigated.

CHAPTER 6

EVALUATION

In this chapter, we evaluate both the numerical stability and performance of our proposed algorithms, the SPIKE-Diagonal pivoting algorithm, the parallel diagonal pivoting algorithm, and the A-R-CR algorithm. All of our solvers are double-precision and running on an Intel Xeon X5680 CPU and an NVIDIA C2050 GPU with CUDA 4.1.

In the numerical stability evaluation, we compare the proposed algorithms against the tridiagonal solvers in CUSPARSE, Matlab, and Intel MKL. We use a non-pivoting GPU tridiagonal solver, `gtsv` (renamed as `gtsv_nopivot` in CUSPARSE 5.5), in CUSPARSE 4.1, and pivoting CPU tridiagonal solvers, `backslash` and `gtsv`, in Matlab 2013b and MKL 11.1, respectively. Then, in the single GPU performance evaluation, we use CUSPARSE 4.1 as the baseline.

6.1 Numerical Stability Evaluation

In this evaluation of numerical stability, we test the quality of a solution using the $l2$-norm backward residuals (Equation 6.1) instead of providing a mathematical proof of the backward stability.

$$\text{backward residual} = \frac{\|Tx - d\|_2}{\|d\|_2} \tag{6.1}$$

We test 18 types of nonsingular tridiagonal matrices of size 512, including 16 matrices chosen to challenge the robustness and numerical stability of the tridiagonal algorithm in recent literature [33, 43, 20, 44] and 2 matrices carefully chosen with singular or near-singular decomposition to challenge the parallel algorithms. The description and the condition number of each tridiagonal matrix are listed in Table 6.1, while the corresponding backward

Table 6.1: Matrix types used in numerical stability evaluation

Matrix Type	Condition Number	Description
1	4.41e+04	Matrix entries randomly generated from a uniform distribution on [-1,1] (denoted as `U(-1,1)`)
2	1.00e+00	A Toeplitz matrix, main diagonal entries are 1e8, off-diagonal entries are from `U(-1,1)`
3	3.52e+02	gallery('lesp',512) in Matlab: real eigenvalues smoothly distributed in the interval approximately [-2*512-3.5, -4.5]
4	2.75e+03	Matrix entries from `U(-1,1)`, the 256th lower diagonal entry is multiplied by 1e-50
5	1.24e+04	Main diagonal entries from `U(-1,1)`, off-diagonal entries chosen with 50% probability either 0 or from `U(-1,1)`
6	1.03e+00	A Toeplitz matrix, main diagonal entries are 64 and off-diagonal entries are from `U(-1,1)`
7	9.00e+00	inv(gallery('kms',512,0.5)) in Matlab: Inverse of a Kac-Murdock-Szego Toeplitz matrix
8	9.87e+14	gallery('randsvd',512,1e15,2,1,1) in Matlab: a randomly generated matrix, condition number is 1e15, 1 small singular value
9	9.97e+14	gallery('randsvd',512,1e15,3,1,1) in Matlab: a randomly generated matrix, condition number is 1e15, geometrically distributed singular values
10	1.29e+15	gallery('randsvd',512,1e15,1,1,1) in Matlab: a randomly generated matrix, condition number is 1e15, 1 large singular value
11	1.01e+15	gallery('randsvd',512,1e15,4,1,1) in Matlab: a randomly generated matrix, condition number is 1e15, arithmetically distributed singular values
12	2.20e+14	Matrix entries from `U(-1,1)`, then the lower diagonal entries are multiplied by 1e-50
13	3.21e+16	gallery('dorr',512,1e-4) in Matlab: an ill-conditioned, diagonally dominant matrix
14	1.14e+67	A Toeplitz matrix, main diagonal entries are 1e-8, off-diagonal entries are from `U(-1,1)`
15	6.02e+24	gallery('clement',512,0) in Matlab: main diagonal entries are 0; eigenvalues include plus and minus odd integers small than 512
16	7.1e+191	A Toeplitz matrix, main diagonal entries are 0, off-diagonal entries are from `U(-1,1)`
17	3.27e+02	Main diagonal entries are 1 and off-diagonal entries are 0 on both Row 0 and 511, the rest diagonal entries are 0 and off-diagonal entries are 1
18	3.78e+18	Main diagonal entries are 0 and off-diagonal entries are 1 on both Row 255 and 256, the 254th main diagonal entry is 1e15, the rest entries are from `U(-1,1)`

Table 6.2: Backward residuals among algorithms

Matrix Type	SPIKE-Diagonal Pivoting	Parallel Diagonal Pivoting	A-R-CR	CUSPARSE	MKL	Matlab
1	1.82e-14	3.72e-14	1.23e-13	**7.14e-12**	1.88e-14	1.21e-14
2	1.27e-16	1.28e-16	1.18e-16	1.69e-16	1.03e-16	1.03e-16
3	1.55e-16	1.46e-16	1.45e-16	2.57e-16	1.35e-16	1.32e-16
4	1.37e-14	5.09e-15	3.76e-14	**1.39e-12**	3.10e-15	3.24e-15
5	1.07e-14	7.84e-15	1.29e-14	1.82e-14	1.56e-14	1.07e-14
6	1.05e-16	1.05e-16	9.97e-17	1.57e-16	9.34e-17	9.34e-17
7	2.42e-16	2.41e-16	2.37e-16	5.13e-16	2.52e-16	2.20e-16
8	2.14e-04	1.40e-03	2.14e-04	**~~1.50e+10~~**	3.76e-04	2.14e-04
9	2.32e-05	3.82e-05	9.80e-05	**~~1.93e+08~~**	3.15e-05	1.35e-05
10	4.27e-05	4.27e-05	3.38e-05	**~~2.74e+05~~**	3.21e-05	2.68e-05
11	7.52e-04	6.79e-04	2.91e-02	**~~4.54e+11~~**	2.99e-04	3.03e-04
12	5.58e-05	4.88e-05	5.04e-05	5.55e-04	2.24e-05	3.06e-05
13	5.51e-01	3.42e+01	3.37e-01	**~~1.12e+16~~**	3.34e-01	3.47e-01
14	2.86e+49	3.19e+49	**~~2.97e+55~~**	2.92e+51	1.77e+48	2.21e+47
15	2.09e+60	1.53e+60	9.79e+59	**~~Nan~~**	1.47e+59	3.69e+58
16	**~~Inf~~**	**~~Inf~~**	**~~Inf~~**	**~~Nan~~**	**~~Inf~~**	4.7e+171
17	**~~Nan~~**	8.14e-16	7.38e-16	**~~Nan~~**	4.87e-16	4.87e-16
18	**~~3.26e-03~~**	3.13e-14	3.31e-14	**~~Nan~~**	9.93e-15	9.10e-15

residuals of each solver are shown in Table 6.2. Since the size of matrices is only 512, the SPIKE-Diagonal Pivoting and Parallel Diagonal Pivoting algorithms perform with 64 partitions, instead of the maximal number of partitions for a GPU. Both CUSPARSE and the A-R-CR algorithm use the fine-grained parallelism for each row and block respectively, while both of the solvers in Intel MKL and Matlab are sequential.

In Table 6.2, bold numbers indicate solutions with backward residuals 100× larger than the baseline results, which are from the default Matlab tridiagonal solver, while struck-through ones highlight solutions with significantly worse backward residuals (1 million times) than the baseline results. Among 18 types of test matrices, the SPIKE-Diagonal Pivoting algorithm fails Type 17 and 18 due to singular or near-singular decomposition discussed in Section 3.4, and fails Type 16 due to an extreme ill-conditioned matrix; the Parallel Diagonal Pivoting algorithm, the A-R-CR algorithm, and MKL only fail Type 16 for the same reason; the A-R-CR algorithm fails one more matrix, Type 14, probably also due to an extreme ill-conditioned matrix; CUSPARSE fails 9 matrices due to no pivoting applied and Matlab produces results with finite backward residuals to all matrices.

Pivoting is extremely important for robust quality of solutions. Non-pivoting tridiagonal solvers, such as `gtsv` in CUSPARSE 4.1, only provides tolerable solutions for context-specific applications. In general applications, non-pivoting tridiagonal solvers are not reliable.

In our proposed algorithms, the SPIKE-Diagonal Pivoting algorithm tends to provide solutions of quality comparable to those of Intel MKL and Matlab, when good partitioning happens in the SPIKE-Diagonal Pivoting algorithm. However, singular or near-singular partitioning might dramatically degrade the quality of solutions in the SPIKE-Diagonal Pivoting algorithm and limit its applicability. Adaptive partitioning such as our A-R-CR algorithm can avoid this singular or near-singular partitioning and restore the robust quality of solutions. Similarly, communication among partitions, such as our Parallel Diagonal Pivoting, allowing more accurate diagonal pivoting also provides a robust quality of solutions.

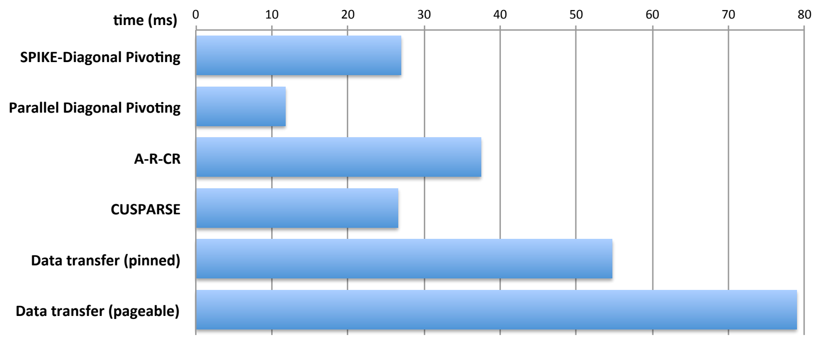

Figure 6.1: Single GPU performance comparison among GPU tridiagonal solvers.

6.2 Single GPU Performance Evaluation

We evaluate the performance of our proposed GPU tridiagonal solvers, in which we compare against the results of `gtsv` in CUSPARSE 4.1 on an NVIDIA C2050 GPU.

Figure 6.1 shows the performance for an 8-million row random (Type 1) matrix. In our evaluation, both the SPIKE-Diagonal Pivoting algorithm and the A-R-CR algorithm have comparable performance (1.5% and 41.1% slowdowns, respectively) to CUSPARSE, while the Parallel Diagonal Pivoting algorithm outperforms CUSPARSE by a factor of 2.25×.

Although the SPIKE-Diagonal Pivoting algorithm is extremely scalable for partitioning the workloads, the CR algorithm used in CUSPARSE is also able to divide the workloads in multiple SMs with a small overhead due to a regular access pattern with small data sharing. On the other hand, both data layout transformation and dynamic tiling introduce small overheads for handling the irregularity in the SPIKE-Diagonal Pivoting algorithm, while the CR algorithm does not have this overhead due to its regular pattern. In the end, they have similar performance.

Different from the SPIKE-Diagonal Pivoting algorithm, the A-R-CR algorithm has all of the overheads the CR algorithm has. On top of those, the A-R-CR algorithm needs to restructure diagonal blocks in each level of recursion and has an irregular access pattern. Therefore, the A-R-CR algorithm tends to have more execution time than CUSPARSE, while in terms of quality of solutions, the A-R-CR algorithm is much better.

Compared to the CR algorithm or the SPIKE algorithm, the Parallel Diagonal Pivoting algorithm has a negligible overhead for both partitioning and irregular computation. Meanwhile, both inter-block communication and kernel fusion significantly reduce the memory bandwidth requirement. All of these factors make the Parallel Diagonal Pivoting algorithm extremely suitable for GPUs.

CHAPTER 7

RELATED WORK

Most early GPU tridiagonal solvers are application-driven and focus on solving a massive number of small-size tridiagonal matrices. Sengupta et al. [12], Goddeke et al. [3], and Davidson et al. [7] implemented the CR algorithm on GPUs, while Egloff [8, 9] applied the PCR algorithm. Davidson's register packing technique can efficiently reduce the required size of scratchpad memory the CR algorithm desires. Sakharnykh first proposed a thread-parallel tridiagonal solver using the Thomas algorithm (LU algorithm) and then extended it to a PCR-Thomas algorithm using the PCR algorithm as a partitioning method. Zhang et al. [13] first systematically introduced a hybrid strategy for GPU tridiagonal solvers, by combining the Thomas, CR, PCR, and Recursive Doubling (RD) algorithms. The RD algorithm is a linear recurrence-based algorithm using c_i's as denominators (Section 2.1.2).

Only a few GPU solvers support a small number of large-size matrices. Kim et al. [17] and Davidson el al. [16] both extended Sakharnykh's PCR-Thomas algorithm to further support a large-size matrix. CUSPARSE [15] implemented the CR algorithm for supporting a large-size matrix by dividing workloads into multiple SMs. Argüello et al. [14] proposed a scalable partitioning method, called split-and-merge, for the CR algorithm to further reduce its memory bandwidth requirement. Murphy [45] applied the LU decomposition on a CPU and then the CR algorithm for both L and U solvers on a GPU. Murphy's strategy potentially pipelines the workloads between CPUs and GPUs.

No existing GPU tridiagonal solver had pivoting. For a general matrix solver, MAGMA [19] applied hybridization of partial pivoting on CPU and the rest parallel computation on GPUs. Also, as mentioned is Section 2.1.1, the SPIKE algorithm can support a band matrix in general. Li et al. [46] implemented a GPU SPIKE solver without pivoting for a band matrix. Our SPIKE-Diagonal Pivoting algorithm was published earlier than Li's work.

CHAPTER 8

CONCLUSION

In this thesis, we study scalable parallel tridiagonal algorithms with pivoting for providing robust quality of solutions for general-purpose applications on many-core architectures, such as GPUs. Previous GPU tridiagonal libraries were relatively context-specific, only applicable for certain kinds of matrices, such as a massive number of independent matrices, limited sizes of matrices, or matrices with specific properties, to efficiently compute a valid solution with a certain quality. We propose three scalable tridiagonal algorithms that are much more broadly applicable. Our proposed algorithms can provide solutions of comparable quality to the most common, general-purpose CPU tridiagonal solvers in existing packages like Matlab or Intel MKL, and demonstrate comparable performance on the state-of-the-art high-performance GPU tridiagonal solvers in existing packages like CUSPARSE.

The proposed algorithms can be extended to other parallel architectures, such as multi-core CPUs, FPGAs, or clusters, and still maintain reasonable quality of solution and high performance because of scalable diagonal pivoting. Meanwhile, the proposed optimization techniques can also be generalized to other applications or architectures. Our SPIKE-Diagonal Pivoting algorithm is published in [33] and included as `gtsv` in NVIDIA CUSPARSE 5.5 or later versions. The extended SPIKE-Thomas algorithm is published in [33], while the SPIKE-CR algorithm is published in [36].

REFERENCES

[1] R. W. Hockney, "A fast direct solution of Poisson's equation using Fourier analysis," Journal of the ACM (JACM), vol. 12, pp. 95–113, January 1965.

[2] J. Zhu, Solving partial differential equations on parallel computers. World Scientific, 1994.

[3] D. Göddeke and R. Strzodka, "Cyclic reduction tridiagonal solvers on GPUs applied to mixed-precision multigrid," IEEE Transactions on Parallel and Distributed Systems, vol. 22, pp. 22–32, 2011.

[4] L.-W. Chang, M.-T. Lo, N. Anssari, K.-H. Hsu, N. E. Huang, and W.-m. W. Hwu, "Parallel implementation of multi-dimensional ensemble empirical mode decomposition," in International Conference on Acoustics, Speech, and Signal Processing, May 2011, pp. 1621–1624.

[5] CUDA Programming Guide 5.5, NVIDIA Corporation, 2013.

[6] The OpenCL C Specification Version: 2.0, Khronos OpenCL Working Group, 2013.

[7] A. Davidson and J. D. Owens, "Register packing for cyclic reduction: A case study," in Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units, 2011.

[8] D. Egloff, "GPUs in financial computing part II: Massively parallel solvers on GPUs," Wilmott, vol. 50, pp. 50–53, 2010.

[9] D. Egloff, "GPUs in financial computing part III: ADI solvers on GPUs with application to stochastic volatility," Wilmott, vol. 52, pp. 51–53, 2011.

[10] N. Sakharnykh, "Tridiagonal solvers on the GPU and applications to fluid simulation," in NVIDIA GPU Technology Conference, 2009.

[11] N. Sakharnykh, "Efficient tridiagonal solvers for ADI methods and fluid simulation," in NVIDIA GPU Technology Conference, 2010.

[12] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for GPU computing," in Graphics Hardware 2007, 2007, pp. 97–106.

[13] Y. Zhang, J. Cohen, and J. D. Owens, "Fast tridiagonal solvers on the GPU," ACM Sigplan Notices, vol. 45, no. 5, pp. 127–136, 2010.

[14] F. Argüello, D. B. Heras, M. Bóo, and J. Lamas-Rodríguez, "The split-and-merge method in general purpose computation on GPUs," Parallel Computing, vol. 38, no. 6-7, pp. 277–288, June 2012.

[15] CUSPARSE Library, NVIDIA Corporation, 2013.

[16] A. Davidson, Y. Zhang, and J. D. Owens, "An auto-tuned method for solving large tridiagonal systems on the GPU," in Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium, May 2011.

[17] H.-S. Kim, S. Wu, L.-W. Chang, and W.-m. W. Hwu, "A scalable tridiagonal solver for GPUs," in Parallel Processing (ICPP), 2011 International Conference on, 2011, pp. 444–453.

[18] R. A. Sweet, "A generalized cyclic reduction algorithm," SIAM Journal on Numerical Analysis, vol. 11, no. 3, pp. 506–520, 1974.

[19] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects," in Journal of Physics: Conference Series, vol. 180, no. 1. IOP Publishing, 2009, p. 012037.

[20] J. B. Erway, R. F. Marcia, and J. A. Tyson, "Generalized diagonal pivoting methods for tridiagonal systems without interchanges," IAENG International Journal of Applied Mathematics, vol. 40, no. 4, pp. 269–275, 2010.

[21] A. H. Sameh and D. J. Kuck, "On stable parallel linear system solvers," Journal of the ACM, vol. 25, no. 1, pp. 81–91, 1978.

[22] E. Polizzi and A. H. Sameh, "A parallel hybrid banded system solver: The SPIKE algorithm," Parallel Computing, vol. 32, no. 2, pp. 177–194, Feb. 2006.

[23] P. P. de Groen, "Base-*p*-cyclic reduction for tridiagonal systems of equations," Applied Numerical Mathematics, vol. 8, no. 2, pp. 117–125, 1991.

[24] "Math kernel library," 2014, Intel. [Online]. Available: http://developer.intel.com/software/products/mkl/

[25] MATLAB version 8.2 (R2013b). Natick, Massachusetts: The MathWorks Inc., 2013.

[26] Ö. Eğecioğlu, "*LU* factorization and parallel evaluation of continued fractions," in Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Systems. ACTA Press, 1998.

[27] G. E. Blelloch, "Scans as primitive parallel operations," Computers, IEEE Transactions on, vol. 38, no. 11, pp. 1526–1538, Nov. 1989.

[28] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," Computers, IEEE Transactions on, vol. 22, no. 8, pp. 786–793, Aug. 1973.

[29] H. S. Stone, "An efficient parallel algorithm for the solution of a tridiagonal linear system of equations," Journal of the ACM (JACM), vol. 20, no. 1, pp. 27–38, Jan. 1973.

[30] Ö. Eğecioğlu, C. K. Koc, and A. J. Laub, "A recursive doubling algorithm for solution of tridiagonal systems on hypercube multiprocessors," Journal of Computational and Applied Mathematics, vol. 27, no. 1, pp. 95–108, 1989.

[31] D. S. Dodson and S. A. Levin, "A tricyclic tridiagonal equation solver," SIAM Journal on Matrix Analysis and Applications, vol. 13, no. 4, pp. 1246–1254, 1992.

[32] R. A. Sweet, "A parallel and vector variant of the cyclic reduction algorithm," SIAM Journal on Scientific and Statistical Computing, vol. 9, no. 4, pp. 761–765, 1988.

[33] L.-W. Chang, J. A. Stratton, H.-S. Kim, and W.-m. W. Hwu, "A scalable, numerically stable, high-performance tridiagonal solver using GPUs," in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2012, pp. 27:1–27:11.

[34] I.-J. Sung, J. A. Stratton, and W.-M. W. Hwu, "Data layout transformation exploiting memory-level parallelism in structured grid many-core applications," in Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques. ACM, 2010, pp. 513–522.

[35] Compute Visual Profiler User Guide, NVIDIA Corporation, 2013.

[36] L.-W. Chang and W.-m. W. Hwu, "A guide for implementing tridiagonal solvers on GPUs," in Numerical Computations with GPUs. Springer, 2014, pp. 29–44.

[37] "The OpenMP API specification for parallel programming," 2014, OpenMP. [Online]. Available: http://openmp.org

[38] "Message Passing Interface (MPI) forum home page," 2014, MPI Forum. [Online]. Available: http://mpi-forum.org

[39] S. Xiao and W.-c. Feng, "Inter-block GPU communication via fast barrier synchronization," in Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on. IEEE, 2010, pp. 1–12.

[40] S. Yan, G. Long, and Y. Zhang, "StreamScan: fast scan algorithms for GPUs without global barrier synchronization," in ACM SIGPLAN Notices, vol. 48, no. 8. ACM, 2013, pp. 229–238.

[41] CUDA Dynamic Parallelism Programming Guide, NVIDIA Corporation, 2012.

[42] W.-m. W. Hwu, "HSA application programming techniques," Heterogeneous System Architecture (HSA): Architecture and Algorithms Tutorial, 2014.

[43] I. S. Dhillon, "Reliable computation of the condition number of a tridiagonal matrix in O(N) time," SIAM Journal on Matrix Analysis and Applications, vol. 19, no. 3, pp. 776–796, July 1998.

[44] G. I. Hargreaves, "Computing the condition number of tridiagonal and diagonal-plus-semiseparable matrices in linear time," SIAM Journal on Matrix Analysis and Applications, vol. 27, no. 3, pp. 801–820, July 2005.

[45] B. J. Murphy, "Solving tridiagonal systems on a GPU," in High Performance Computing (HiPC), 2013 20th International Conference on. IEEE, 2013, pp. 159–168.

[46] A. Li, A. Seidl, and D. Negrut, "SPIKE::GPU - a GPU-based banded linear system solver," University of Wisconsin-Madison, Tech. Rep., 2012.