

Chapter 2

A Guide for Implementing Tridiagonal Solvers on GPUs

Li-Wen Chang and Wen-mei W. Hwu

2.1 Introduction

The tridiagonal solver has been recognized as a critical building block for many engineering and scientific applications [3, 8, 9, 11, 17, 18] on GPUs. However, a general high-performance tridiagonal solver for GPU is challenging, not just because the number of independent, simultaneous matrices varies greatly among applications, but also because applications may require their tridiagonal solvers to have customized requirements, such as: data with different layouts, matrices with a certain structure, or execution on multi-GPUs. Therefore, although building a tridiagonal solver library is crucial, it is very difficult to meet all demands. In this chapter, guidelines are given for customizing a high-performance tridiagonal solver for GPUs.

A wide range of algorithms for implementing tridiagonal solvers on GPUs, including both sequential and parallel algorithms, was studied. The selected algorithms were chosen for the requirement of applications, and to take the advantage of massive data parallelism of GPU architecture. Meanwhile, corresponding optimizations were proposed to compensate for some inherent limitations of the selected algorithms. In order to achieve high performance on GPUs, workloads have to be partitioned and computed in parallel on stream processors. For the tridiagonal solver, the inherent data dependence found in sequential algorithms (e.g. the Thomas algorithm [5] and the diagonal pivoting method [10]), limits the opportunities for partitioning the workload. On the other hand, parallel algorithms (e.g. Cyclic Reduction (CR) [12], Parallel Cyclic Reduction (PCR) [12], or the SPIKE algorithm [16, 19]) allow the partitioning of workloads, but suffer from the required overheads of extra computation, barrier synchronization, or communication.

L.-W. Chang • W.-m.W. Hwu (✉)
University of Illinois, 1308 W Main St, Urbana, IL 61801, USA
e-mail: lchang20@illinois.edu; w-hwu@illinois.edu

Two main kinds of components are recognized in most GPU tridiagonal solvers. (1) Partitioning methods are applied to divide workloads for parallel computing. Independent solvers compute massive independent workloads in parallel. In this chapter, we first review cutting-edge partitioning techniques for GPU tridiagonal solvers. Different partitioning techniques require different types of overheads, such as computation or memory overhead. (2) State-of-the-art optimization techniques for independent solvers are discussed. Different algorithms of independent solvers might require different optimizations. Optimization techniques might perform together for more robust independent solvers. Finally, a case study of a new algorithm, SPIKE-CR, which replaces part of the traditional SPIKE algorithm with Cyclic Reduction, is given to demonstrate how to systematically build a highly optimized tridiagonal solver by selecting the partitioning method, and by applying optimization techniques to the independent solver for each partition. *The main purpose of this chapter is to inspire readers building their own GPU tridiagonal solvers to meet their application requirement, instead of demonstrating high performance of SPIKE-CR.*

The rest of the sections in this chapter are organized as following. Section 2.2 briefly reviews the selected algorithms used by GPU tridiagonal solvers. Section 2.3 reviews and compares corresponding optimizations applied to the GPU tridiagonal solvers. Section 2.4 shows a case study of the new GPU tridiagonal solver, SPIKE-CR; discusses its partitioning and optimizations; and compares its performance to alternative methods. Section 2.5 concludes the chapter. In the following sections, we use NVIDIA CUDA [14] terminology.

2.2 Related Algorithms

In this section, we briefly cover the selected tridiagonal solver algorithms used for GPUs. Although, in general, most tridiagonal solvers may be used to solve multiple systems of equations each with its own tridiagonal matrix, for simpler explanation here, we only discuss the case of solving a single system with one tridiagonal matrix. The tridiagonal solver solves $Tx = d$, where T is a tridiagonal matrix with n rows and n columns, defined in Eq. (2.1), and x and d are both column vectors with n elements. Note that the first row of T is row 0, and the first element of x and d is element 0.

$$T = \begin{bmatrix} b_0 & c_0 & & & \\ a_1 & b_1 & c_1 & & \\ & a_2 & \ddots & \ddots & \\ & & \ddots & \ddots & c_{n-2} \\ & & & a_{n-1} & b_{n-1} \end{bmatrix} \quad (2.1)$$

2.2.1 Thomas Algorithm

The Thomas algorithm is a special case of Gaussian elimination without pivoting (or LU decomposition with LU solvers) for a tridiagonal matrix. It consists of two phases, a forward reduction and a backward substitution. The forward reduction sequentially eliminates the lower diagonal of the original matrix, while the backward substitution sequentially solves for unknown variables using known variables and the upper and main diagonals in the resultant matrix. For $Tx = d$, decompose $T = LU$ by LU decomposition, let $Ux = y$, solve $Ly = d$, and then solve $Ux = y$.

2.2.2 Diagonal Pivoting Algorithm

The diagonal pivoting algorithm for tridiagonal matrices was proposed by Erway et al. [10]. Although Gaussian elimination with partial pivoting is widely used for tridiagonal solvers on CPUs, it is not efficient on GPUs due to its inherent data dependence and expensive row interchange operations. Erway's diagonal pivoting method avoids row interchanges by dynamically selecting 1-by-1 or 2-by-2 pivots. The factorization is defined as follows:

$$T = \begin{bmatrix} P_h & B \\ C & T_r \end{bmatrix} = \begin{bmatrix} I_h & 0 \\ CP_h^{-1} & I_r \end{bmatrix} \begin{bmatrix} P_h & 0 \\ 0 & T_s \end{bmatrix} \begin{bmatrix} I_h & P_h^{-1}B \\ 0 & I_r \end{bmatrix} \quad (2.2)$$

where P_h is a 1-by-1 ($[b_0]$) or 2-by-2 pivoting block $\begin{bmatrix} b_0 & c_0 \\ a_1 & b_1 \end{bmatrix}$, and

$$T_s = T_r - CP_h^{-1}B = \begin{cases} T_r - \frac{a_1 c_0}{b_0} e_1^{(n-1)} e_1^{(n-1)T}, & \text{for 1-by-1 pivoting} \\ T_r - \frac{a_2 b_0 c_1}{\Delta} e_1^{(n-2)} e_1^{(n-2)T}, & \text{for 2-by-2 pivoting} \end{cases} \quad (2.3)$$

where $\Delta = b_0 b_1 - a_1 c_0$ and $e_1^{(k)}$ is the first column vector of the k -by- k identity matrix. Since T_s is still tridiagonal (Eq. (2.3)), it can also be factorized by the same Eq. (2.2). Therefore, a tridiagonal matrix T can be recursively factorized in LBM^T , where B only contains either 1-by-1 or 2-by-2 blocks in its diagonal. After LBM^T factorization, the tridiagonal matrix T can be solved by solving L , B , and M^T sequentially.

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

Fig. 2.1 One step 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 row 2 can form a smaller matrix

2.2.3 Cyclic Reduction

The Cyclic Reduction (CR) algorithm, also known as an odd-even reduction, contains two phases, forward reduction and backward substitution. In every step of the forward reduction, defined in Eq. (2.4), each odd (or even) equation is eliminated by using the adjacent two even (or odd) equations.

$$\begin{aligned} \alpha &= a_i / b_{i-\text{stride}}, \quad \beta = c_i / b_{i+\text{stride}}, \\ a'_i &= -\alpha a_{i-\text{stride}}, \quad b'_i = b_i - \alpha c_{i-\text{stride}} - \beta a_{i+\text{stride}}, \\ c'_i &= -\beta c_{i+\text{stride}}, \quad d'_i = d_i - \alpha d_{i-\text{stride}} - \beta d_{i+\text{stride}}, \end{aligned} \quad (2.4)$$

where the *stride* starts from 1 and increases exponentially step-by-step, and the domain of i starts from all odd and shrinks exponentially. The boundary condition can be simplified by using $a_i = c_i = 0$, and $b_i = 1$. Figure 2.1 shows a CR example for a 4-by-4 tridiagonal matrix. After a step of CR forward reduction, redundant unknown variables and zeros can be removed, and a half-size matrix is formed of the remaining unsolved equations. Each step of the backward substitution, defined in Eq. (2.5), solves for unknown variables by substituting solutions obtained from the smaller system.

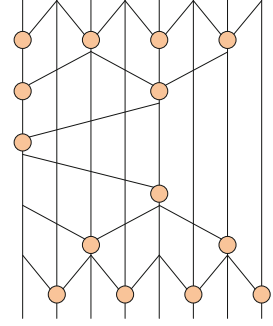
$$x_i = \frac{d'_i - a'_i x_{i-\text{stride}} - c'_i x_{i+\text{stride}}}{b'_i} \quad (2.5)$$

where the stride decreases exponentially step-by-step, and the domain of i increases exponentially. The graph representation of CR for a 8-by-8 matrix is shown in Fig. 2.2, where each vertical line represents an equation, and each circle represents forward or backwards computation.

2.2.4 Parallel Cyclic Reduction

The PCR algorithm, different from CR, only performs the forward reduction, Eq. (2.4). Also, the PCR forward reduction is performed on all equations, instead of odd (or even). That means the domain of i does not decrease exponentially, but

Fig. 2.2 The CR access pattern on a 8-by-8 matrix: each *vertical line* represents an equation, each *circle* represents forward or backwards computation, and each *edge* represents communication between two equations



the *stride* still keeps increasing exponentially step-by-step. Figure 2.3 shows a PCR example for the same 4-by-4 tridiagonal matrix. After a step of PCR, two half-size matrices are formed of the resultant new equation by reorganizing unknown variables. It also illustrates how a matrix can be partitioned after each PCR (forward reduction) step.

$$\begin{bmatrix} b_0 & c_0 & & \\ a_1 & b_1 & c_1 & \\ & a_2 & b_2 & c_2 \\ & & a_3 & b_3 \end{bmatrix} \rightarrow \begin{bmatrix} b'_0 & 0 & c'_0 & \\ 0 & b'_1 & 0 & c'_1 \\ a'_2 & 0 & b'_2 & 0 \\ & a'_3 & 0 & b'_3 \end{bmatrix} \Leftrightarrow \begin{bmatrix} b'_0 & c'_0 \\ a'_2 & b'_2 \\ b'_1 & c'_1 \\ a'_3 & b'_3 \end{bmatrix}$$

Fig. 2.3 One step PCR forward reduction on a 4-by-4 matrix: a_i and c_i on each row i are eliminated by adjacent two rows. For example, a_2 and c_2 on row 2 are eliminated by row 1 and 3. After that, row 0 and 2 can form a smaller matrix, and row 1 and 3 can form another

2.2.5 Recursive Doubling

The Recursive Doubling (RD) algorithm [21] can be considered as a reformulation of a parallel tridiagonal solver into a second-order linear recurrence, Eq. (2.6). By solving the relationship between x_0 and d_{n-1} , all unknown variables, x_i 's, can be solved.

$$\begin{bmatrix} 1 & & & & & \\ b_0/c_0 & 1 & & & & \\ a_1/c_1 & b_1/c_1 & & & & \\ & \ddots & \ddots & \ddots & & \\ & & a_{n-2}/c_{n-2} & b_{n-2}/c_{n-2} & 1 & \\ & & & a_{n-1} & b_{n-1} & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ 0 \end{bmatrix} = \begin{bmatrix} x_0 \\ d_0/c_0 \\ d_1/c_1 \\ \vdots \\ d_{n-2}/c_{n-2} \\ d_{n-1} \end{bmatrix} \quad (2.6)$$

However, in the Recursive Doubling algorithm, huge numerical errors might be produced, even for a diagonally dominant matrix, since division operations are performed on upper diagonal elements (c_i 's). Because of this shortcoming, we skip the discussion of RD in this chapter.

2.2.6 SPIKE Algorithm

The SPIKE algorithm was originally introduced by Sameh et al. [19] and the latest version described by Pollizi et al. [16]. It is a domain decomposition algorithm, that partitions a matrix into block rows containing diagonal sub-matrices, T_i , and off-diagonal elements, a_{hi} and c_{ti} . The original matrix, T , can be further defined as the product of two matrices, the block-diagonal matrix D and the spike matrix S , Fig. 2.4, where V_i and W_i of S can be solved by Eq. (2.7).

$$T_i V_i = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ c_{ti} \end{bmatrix}, \quad T_i W_i = \begin{bmatrix} a_{hi} \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \quad (2.7)$$

After the formation of the matrices D and S , the SPIKE algorithm solves $Dy = d$ for y , and then uses the special form of S to solve $Sx = y$ [16]. The spike matrix, S , can also be considered a specialized block tridiagonal matrix, and can be solved by a block tridiagonal solver algorithm, such as the block Cyclic Reduction algorithm [2].

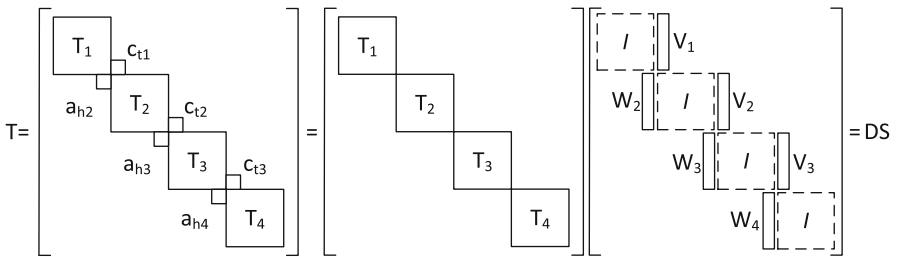


Fig. 2.4 A tridiagonal matrix T can be defined as $T = DS$, where D is a block diagonal matrix and S is a spike matrix (a specialized block tridiagonal matrix)

2.3 Optimization Techniques

As mentioned in Sect. 2.1, partitioning is necessary for high performance on GPUs. Although sequential algorithms inherently cannot be partitioned, they are widely applied to solving multiple independent systems in parallel. On the other hand, although parallel algorithms are capable of both partitioning individual systems and solving multiple independent systems, they might require high overheads. In this section, all existing optimization techniques for GPU tridiagonal solvers are examined. However, while not every optimization is discussed in detail, references are provided for each technique to satisfy readers who need more information.

2.3.1 *Partitioning Method*

Many of the early tridiagonal solvers on GPUs [6, 8, 11, 17, 20, 23] can only be applied to problems with multiple independent matrices. They simply assume no partitioning occurs, and exploit only the inherent parallelism from multiple independent matrices. This assumption works very efficiently, simply because parallelism is inherent and no partitioning overhead is required. However, when the number of independent matrices shrinks, the overall performance drops dramatically.

Partitioning is found in many studies of tridiagonal solvers for GPUs, and particularly, the PCR algorithm was widely applied to partitioning. Sakharnykh [18] first introduced PCR in his PCR-Thomas implementation to further extract more parallelism for a limited number of independent matrices. Kim et al. [13] and Davidson et al. [7] first recognized that partitioning is necessary for a tridiagonal solver to handle a single large matrix on GPUs, and they proposed PCR-Thomas tridiagonal solvers. In both papers, PCR was used to decompose one large matrix into many smaller independent matrices. The main limitation of PCR is its computation overhead. In order to minimize the computation overhead of PCR, only a few PCR steps are performed. Kim et al. further proposed the sliding window technique to reduce the requirement of scratchpad memory size for PCR, and to make PCR more efficient.

Compared to PCR, domain partitioning requires less computational overhead. The CR-PCR implementation for the non-pivoting tridiagonal solver in NVIDIA CUSPARSE [15] uses implicit domain partitioning by duplicating memory accesses between two adjacent partitions. By storing data back to global memory between two CR steps, the redundant equations of CR (see Fig. 2.1) can be removed to avoid unnecessary memory overhead. Although this naive partitioning method simplifies the source code, it may cost a large memory overhead, since each CR step requires reloading data from global memory.

Argüello et al. [1] proposed a split-and-merge method for CR by separating computation workloads into two sets, called split and merge sets. The split sets represent the independent workloads partitioned and are assigned to stream processors, while

the merge sets represent computation workloads requiring data from two or more independent split sets. Figure 2.5a illustrates the graph representation for the split-and-merge method of CR forward reduction. The independent split sets can be simply computed in parallel, while the merge sets are postponed and computed in a separate kernel later. Compared to the NVIDIA CR-PCR implementation, Argüello’s method dramatically reduces memory access overhead, since multiple steps of CR might be computed with shared data in a kernel. Chang et al. [2] further refined Argüello’s split-and-merge CR to support the larger split sets. The corresponding illustration is shown in Fig. 2.5b.

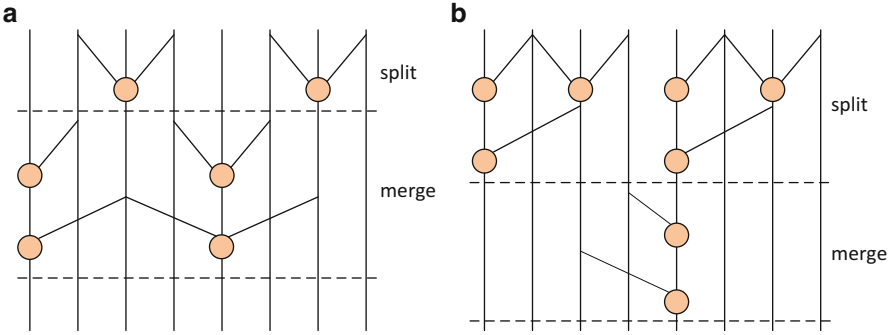


Fig. 2.5 The graph representation of a 8-by-8 matrix for CR using split-and-merge. (a) Argüello’s split-and-merge method, which has smaller splits sets and larger merge sets; (b) Chang’s split-and-merge method, which has larger splits sets and smaller merge sets

Chang et al. [4] and the pivoting tridiagonal solver in NVIDIA CUSPARSE applied the SPIKE algorithm to decompose a matrix into disjoint partitions. The SPIKE algorithm requires extra overhead for solving the spike matrix, $Sx = y$. The computation cost for solving the spike matrix is relatively small, compared to the cost for solving all of the independent partitions.

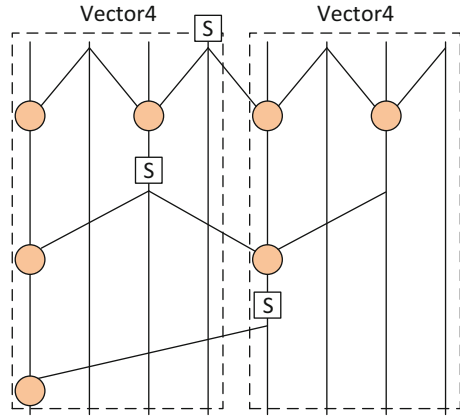
2.3.2 Algorithms and Optimizations for Independent Solver

After using a matrix partitioning method, or given multiple independent matrices, the multiple independent workloads can be computed in parallel. The Thomas algorithm was applied in [4, 7, 13, 17] simply for its low complexity and lack of warp divergence. Chang et al. [4] first introduced the diagonal pivoting method [10] for numerical stability, and the same method is also implemented in the CUSPARSE pivoting tridiagonal solver. With Chang’s dynamic tiling technique, the overhead of warp divergence in the diagonal pivoting method is dramatically reduced.

Different from the sequential algorithms, the parallel algorithms, such as CR, require more optimization techniques to reduce possible overheads and to perform efficiently. Göddeke et al. [11] eliminated bank conflict caused by the strided

access of CR, by marshaling data on scratchpad memory. Davidson et al. [6] proposed register packing for CR to hold more data in registers within a stream processor without increasing the size of scratchpad memory. Figure 2.6 illustrates an example of 4-equation register-packing CR forward reduction for an 8-by-8 matrix. A 4-equation CR forward reduction is computed locally in packed vector4 registers. The label S represents the data copied to scratchpad memory for communication among threads. Note that the needed scratchpad size is equal to the number of threads. Davidson’s optimization can potentially increase the size of each partition, and further reduce the possible overhead of partitioning, though the benefits were not explicitly mentioned in Davidson’s paper.

Fig. 2.6 The graph representation of a 8-by-8 matrix for Davidson’s 4-equation register-packing CR forward reduction: the label S represents the data copied to scratchpad memory. Davidson’s method can hold the number of thread times the number of register packing equations in a thread block. The needed scratchpad size is equal to the number of threads. In this illustration, the scratchpad size is only 2 equations



PCR can be used as an efficient independent solver for small-size matrices. Zhang et al. [23] first demonstrated it in their CR-PCR method, and CUSPARSE extended the CR-PCR method in a non-pivoting tridiagonal solver to support larger matrices. A high-performance warp-level PCR that has no barrier overhead is proposed in Sect. 2.4 and Listing 2.5.

Zhang et al. [23] first systematically introduced the hybrid methods for GPU tridiagonal solvers, by combining the Thomas, CR, PCR, and RD algorithms, to gain feasible complementary benefits. Although Zhang’s idea only worked for small matrices, implementing an independent solver using his idea is extremely efficient when running on a stream processor. The reading of Zhang’s paper is highly recommended.

2.3.3 Short Summary

Table 2.1 summarizes the above partitioning methods, and the corresponding limitation or overhead. Different applications may require different partitioning methods, and have different overheads. Another possible overhead for all methods is the data marshaling [22] overhead to glue two memory access patterns of the applied

Table 2.1 Summary of partitioning methods

Methods	Limitation or overhead
No partitioning	No overhead, but only for massive independent matrices
PCR	Heavy computation overhead
Naive domain partitioning	Heavy memory access overhead
SPIKE algorithm	Light computation/memory overhead
Split-and-merge	Light memory access overhead

Table 2.2 Optimization of independent solver

Optimization	Algorithms
Dynamic tiling	Diagonal pivoting method
Register packing	CR
Bank conflict elimination	CR
Warp-level computation	PCR
Hybrid method	All tridiagonal algorithms

partitioning method and independent solver. For example, in Chang’s SPIKE-based tridiagonal solver [4], data marshaling is used to guarantee a coalesced memory access pattern in the independent solver. The data marshaling overhead is required only if the output pattern of the partitioning method is different from the input pattern of the independent solver.

Table 2.2 categorizes applicable optimizations for the algorithms used in independent solvers. Different optimization techniques might perform better together for more robust independent solvers. The concept of Zhang’s hybrid method [23] can further enable more potential optimizations across different algorithms. For example, in the case study (Sect. 2.4), we use a hybrid of CR and PCR to enable optimizations in the both algorithms.

2.4 Case Study: SPIKE-CR

In this section, a new hybrid algorithm, SPIKE-CR, is used as a case study to demonstrate how to apply the optimization techniques that were summarized in Sect. 2.3. Using a systematic optimization analysis, the implementation of the SPIKE-CR tridiagonal solver conceptually interacts with the GPU architecture. Previous works did not discover the SPIKE-CR method. This is mainly because the previous works did not systematically analyze the partitioning methods.

In the SPIKE-CR, the SPIKE algorithm is applied to partitioning for its lower computation overhead than PCR and lower memory access overhead than the other domain partitioning methods. After the partitioning method is selected, CR is applied for the independent solver. Although the sequential algorithms are efficient with the SPIKE algorithm [4], the CR algorithm is chosen to avoid the potential data marshaling overhead from combining the SPIKE algorithm and the sequential

algorithms. SPIKE-PCR is another potential direction for a GPU tridiagonal solver. However, the computation cost of PCR is much higher than CR.

In order to implement an efficient SPIKE-CR, the following optimization techniques are applied in this case study. First, Davidson's register packing [6] is applied to hold more equations in a partition. This optimization can potentially reduce partitioning overhead of the SPIKE algorithm by reducing the number of partitions. Second, Zhang's hybrid idea [23] of CR and PCR is used to avoid the potential low utilization of vector units in CR and to further enable more options of optimization in PCR. Third, a new warp-level PCR is proposed to remove barrier synchronization overheads in PCR. Last, another level partitioning using the SPIKE algorithm is applied to minimize communication between warps within a thread block. This strategy makes partitioning become hierarchical and further reduces communication overheads.

Listing 2.1 The baseline kernel of CR forward

```

1  ...
2  tx = threadIdx.x;
3  b_dim = blockDim.x;
4  ...
5  //CR iteration within a thread block
6  active_tx = b_dim;
7  for(int i=1;i<b_dim;i*=2)
8  {
9      active_tx/=2;
10     if(tx < active_tx)
11     {
12         //CR forward computation using data in
13         scratchpad
14     }
15     __syncthreads();
16     if(tx < active_tx)
17     {
18         //update scratchpad
19     }
20     __syncthreads();
21 }
22 ...

```

Listing 2.2 The optimized kernel of CR

```

1  ...
2  tx = threadIdx.x;
3  b_dim = blockDim.x;
4  lane_id = tx % warpSize;
5  warp_id = tx / warpSize;
6  ...
7  double2 a_reg,b_reg, c_reg, d_reg; //vectorize register
8  //load data into scratchpad using vector2
9  a_reg = a[id];
10 b_reg = b[id];
11 c_reg = c[id];
12 d_reg = d[id];
13 //code fragment 1, CR forward reduction
14 //code fragment 2, warp-level PCR
15 //code fragment 3, CR backward substitution
16 //store partial results and the spike matrix
17 ...

```

Since the source codes of the SPIKE algorithm have been provided by Chang et al. [4] at <http://impact.crhc.illinois.edu>, and the computation cost for solving the spike matrix is much smaller than the cost for solving all independent partitions, we only discuss the detailed source codes of the independent solver. The reading of Chang's paper and source codes is highly recommended. Listing 2.1 shows the simplified baseline of the CR forward reduction kernel, and Listing 2.2 shows the structure of our optimized CR. The code fragments are written for NVIDIA Fermi architecture, and possible changes for NVIDIA Kepler architecture are further discussed.

Listing 2.3 The code fragment 1: CR forward reduction

```

1  ...
2  //CR forward in register
3  sh_a[tx] = a_reg.y;
4  sh_b[tx] = b_reg.y;
5  sh_c[tx] = c_reg.y;
6  sh_d[tx] = d_reg.y;
7  //up side
8  {
9      k1=c_reg.x/b_reg.y;
10     b_reg.x -= a_reg.y*k1;
11     d_reg.x -= d_reg.y*k1;
12     c_reg.x = -c_reg.y*k1;
13 }
14 // down side
15 if(lane_id>=1)
16 {
17     k1=a_reg.x/sh_b[tx-1];
18     b_reg.x -= sh_c[tx-1]*k1;
19     d_reg.x -= sh_d[tx-1]*k1;
20     a_reg.x = -sh_a[tx-1]*k1;
21 }
22 sh_a[tx] = a_reg.x;
23 sh_b[tx] = b_reg.x;
24 sh_c[tx] = c_reg.x;
25 sh_d[tx] = d_reg.x;
26 ...

```

Listing 2.4 The code fragment 3: CR backward substitution

```

1  ...
2  //CR backward in register
3  k1 = a_reg.y/b_reg.x;
4  a_reg.y = 0.0;
5  if(lane_id<warpSize-1)
6  {
7      k2 = c_reg.y/sh_b[tx+1];
8      c_reg.y = -sh_c[tx+1]*k2;
9      a_reg.y = -sh_a[tx+1]*k2;
10     d_reg.y -= sh_d[tx+1]*k2;
11 }
12 c_reg.y -= c_reg.x*k1;
13 a_reg.y -= a_reg.x*k1;
14 d_reg.y -= d_reg.x*k1;
15 ...

```

Listing 2.3 shows the portion of 2-equation register-packing CR forward reduction using Davidson's technique [6], and Listing 2.4 shows the portion of corresponding CR backward substitution. Note that Listing 2.4 is the fragment 3, and is performed after the warp-level PCR. Here, we change the order of the listings for an

easier discussion by putting the two fragments of CR together. The packed registers are defined in line 7 of Listing 2.2, and the computation of CR happens at line 3–14 of Listing 2.3 and all of Listing 2.4. Scratchpad memory, sh_a to sh_d , is used to communicate among threads only within a warp. Compared to the baseline of CR forward reduction, which contains at least two barrier synchronizations, line 14 and 19 of Listing 2.1, in a loop, the optimized CR requires no barrier synchronization, since communication only happens within a warp. Also, for NVIDIA Kepler architecture, shuffle instructions can replace those scratchpad memory accesses, since communication happens within a warp. Moreover, since Kepler provides larger register files, a larger size register packing can be applied to holding more data.

Listing 2.5 shows the warp-level PCR fragment of our CR-PCR hybrid. Similarly, since PCR only happens in a warp, no barrier synchronization is needed. Also, shuffle instructions can be used for Kepler by replacing scratchpad memory accesses. In these code fragments, our CR-PCR performs 1 CR forward reduction step, followed by 5 PCR steps in the warp-level PCR and 1 CR backward substitution step, without any barrier synchronization. After CR-PCR, the computed results are stored back to global memory, and also the formed spike matrix is explicitly stored in another space. Since each thread block is further partitioned into multiple warps, another level of domain partitioning using SPIKE algorithm is implicitly applied.

Listing 2.5 The code fragment 2: warp-level PCR

```

1  ...
2  //PCR for each warp, no barrier needed
3  for(int i=1;i<warpSize;i*=2)
4  {
5      // down side
6      if(lane_id>=i)
7      {
8          k1=sh_a[tx]/sh_b[tx-i];
9          b_reg.x -= sh_c[tx-i]*k1;
10         d_reg.x -= sh_d[tx-i]*k1;
11         a_reg.x = -sh_a[tx-i]*k1;
12     }
13     //up side
14     if(lane_id<warpSize-i)
15     {
16         k1=sh_c[tx]/sh_b[tx+i];
17         b_reg.x -= sh_a[tx+i]*k1;
18         d_reg.x -= sh_d[tx+i]*k1;
19         c_reg.x = -sh_c[tx+i]*k1;
20     }
21     sh_a[tx] = a_reg.x;
22     sh_b[tx] = b_reg.x;
23     sh_c[tx] = c_reg.x;
24     sh_d[tx] = d_reg.x;
25 }
26 ...

```

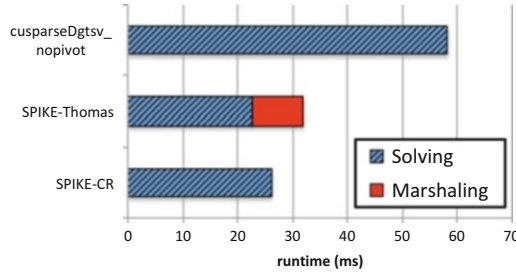


Fig. 2.7 Performance results for solving a 16M-equation double-precision matrix using CUSPARSE non-pivoting tridiagonal solver(cusparseDgtsv_nopivot), Chang’s SPIKE-Thomas, and SPIKE-CR on an NVIDIA Tesla C2050. The data marshaling overhead of Chang’s SPIKE-Thomas implementation is shown in the *right portion* of the bar

2.4.1 Performance Comparison

Figure 2.7 shows the performance comparison for solving a 16M-equation (2^{24}) double-precision matrix using CUSPARSE non-pivoting tridiagonal solver (CR-PCR), Chang’s SPIKE-Thomas [4], and SPIKE-CR on an NVIDIA Tesla C2050. Although the Thomas algorithm is extremely efficient as an independent solver, in Chang’s SPIKE-Thomas, the overhead of data marshaling, required to maintain coalescing memory access for Thomas algorithm, causes Chang’s SPIKE-Thomas performing slightly slower than SPIKE-CR. Compared to CUSPARSE CR-PCR, the domain partitioning using the SPIKE algorithm tends to have less memory access overhead than the naive domain partitioning used by CUSPARSE. The memory access overhead causes the main performance difference between the SPIKE-based methods and CUSPARSE CR-PCR. In the end, SPIKE-CR has $1.23\times$ and $2.23\times$ speedups over SPIKE-Thomas and CUSPARSE CR-PCR, respectively.

2.5 Conclusion

This chapter summarizes most cutting-edge optimization techniques, applied in both partitioning methods and independent solvers, for GPU tridiagonal solvers, and demonstrates how to apply optimization techniques for building a high-performance tridiagonal solver in our case study, SPIKE-CR. The case study, SPIKE-CR, shows $1.23\times$ and $2.23\times$ speedups, respectively, over Chang’s SPIKE-Thomas [4] and CUSPARSE non-pivoting tridiagonal solver, since SPIKE-CR has no data marshaling overhead and less memory access overhead.

As mentioned in Sect. 2.1, the main purpose of this chapter is to give readers the current status of GPU tridiagonal solvers, and further to inspire readers to customize GPU tridiagonal solvers to meet their application requirements, instead of showing high performance of SPIKE-CR. Multiple partitioning methods, such as the split-and-merge method [1, 2] and the SPIKE algorithm, tend to have very low

overheads for a limited number of large matrices, while no partitioning is required for a massive number of matrices. For independent solvers, the sequential methods usually perform very efficiently, while the parallel algorithms, such as CR, can also provide comparable performance after optimization. Therefore, the main concern of building a high-performance GPU tridiagonal solver is how the applied algorithm and its memory access pattern meet a given application.

Some unique properties, such as numerical stability, of a GPU tridiagonal solver for the application are also very critical. So far, only few previous works [4, 23] recognized the numerical stability issue of current GPU tridiagonal solvers, and even fewer ones [4] investigated it. Numerical stability becomes the most important future work for the research of GPU tridiagonal solvers.

Acknowledgements This project was partly supported by the STARnet Center for Future Architecture Research (C-FAR), the DoE Vancouver Project (DE-FC02-10ER26004/DE-SC0005515), and the UIUC CUDA Center of Excellence.

References

1. Argüello, F., Heras, D.B., Bóo, M., Lamas-Rodríguez, J.: The split-and-merge method in general purpose computation on GPUs. *Parallel Comput.* **38**(6–7), 277–288 (2012)
2. Chang, L.-W., Hwu, W.-m.W.: Mapping tridiagonal solvers to linear recurrences. Technical report, University of Illinois at Urbana-Champaign (2013)
3. Chang, L.-W., Lo, M.-T., Anssari, N., Hsu, K.-H., Huang, N.E., Hwu, W.-m.W.: Parallel implementation of multi-dimensional ensemble empirical mode decomposition. In: *International Conference on Acoustics, Speech, and Signal Processing*, pp. 1621–1624 (May 2011)
4. Chang, L.-W., Stratton, J.A., Kim, H.-S., Hwu, W.-m.W.: A scalable, numerically stable, high-performance tridiagonal solver using GPUs. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12*, pp. 27:1–27:11 (2012)
5. Conte, S.D., De Boor, C.W.: *Elementary Numerical Analysis: An Algorithmic Approach*, 3rd edn. McGraw-Hill Higher Education, New York (1980)
6. Davidson, A., Owens, J.D.: Register packing for cyclic reduction: a case study. In: *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units* (2011)
7. Davidson, A., Zhang, Y., Owens, J.D.: 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)
8. Egloff, D.: GPUs in financial computing part II: massively parallel solvers on GPUs. *Wilmott*, **50**, 50–53 (Nov 2010)
9. Egloff, D.: GPUs in financial computing part III: ADI solvers on GPUs with application to stochastic volatility. *Wilmott*, **52**, 51–53 (Mar 2011)
10. Erway, J.B., Marcia, R.F., Tyson, J.A.: Generalized diagonal pivoting methods for tridiagonal systems without interchanges. *IAENG Int. J. Appl. Math.* **40**(4), 269–275 (2010)
11. Göddeke, D., Strzodka, R.: Cyclic reduction tridiagonal solvers on GPUs applied to mixed-precision multigrid. *IEEE Trans. Parallel Distrib. Syst.* **22**, 22–32 (2011)
12. Hockney, R.W., Jesshope, C.R.: *Parallel Computers: Architecture, Programming and Algorithms*. Hilger, Bristol (1981)
13. Kim, H.-S., Wu, S., Chang, L.-W., Hwu, W.-m.W.: A scalable tridiagonal solver for GPUs. In: *2011 International Conference on Parallel Processing (ICPP)*, pp. 444–453 (2011)

14. NVIDIA Corporation: CUDA Programming Guide 5.5 (2013)
15. NVIDIA Corporation: CUSPARSE Library (2013)
16. Polizzi, E., Sameh, A.H.: A parallel hybrid banded system solver: the SPIKE algorithm. *Parallel Comput.* **32**(2), 177–194 (2006)
17. Sakharnykh, N.: Tridiagonal solvers on the GPU and applications to fluid simulation. In: NVIDIA GPU Technology Conference (September 2009)
18. Sakharnykh, N.: Efficient tridiagonal solvers for ADI methods and fluid simulation. In: NVIDIA GPU Technology Conference (September 2010)
19. Sameh, A.H., Kuck, D.J.: On stable parallel linear system solvers. *J. ACM* **25**(1), 81–91 (1978)
20. Sengupta, S., Harris, M., Zhang, Y., Owens, J.D.: Scan primitives for gpu computing. In: *Graphics Hardware 2007*, pp. 97–106 (2007)
21. Stone, H.S.: An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. *J. ACM* **20**(1), 27–38 (1973)
22. Sung, I.-J., Stratton, J.A., Hwu, W.-M.W.: Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. In: *PACT '10: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pp. 513–522. ACM, New York (2010)
23. Zhang, Y., Cohen, J., Owens, J.D.: Fast tridiagonal solvers on the GPU. In: *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '10*, pp. 127–136 (2010)

Numerical Computations with GPUs

Kindratenko, V. (Ed.)

2014, X, 405 p. 107 illus., 49 illus. in color., Hardcover

ISBN: 978-3-319-06547-2