© 2019 Copyright held by the owner/author(s). This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in ACM Transactions on Mathematical Software, https://doi.org/10.1145/3328731

Tree Partitioning Reduction: A New Parallel Partition Method for Solving Tridiagonal Systems

ADRIÁN P. DIÉGUEZ, MARGARITA AMOR, and RAMÓN DOALLO, University of A Coruña, Spain

Solving tridiagonal linear-equation systems is a fundamental computing kernel in a wide range of scientific and engineering applications, and its computation can be modeled with parallel algorithms. These parallel solvers are typically designed to compute problems whose data fit in a common shared-memory space where all the cores taking part in the computation have access. However, when the problem size is large, data cannot be entirely stored in the common shared-memory space, and a high number of high-latency communications are performed. One alternative is to partition the problem among different memory spaces. At this point, conventional parallel algorithms do not facilitate the partition of computation in independent tiles, since each reduction depends on equations which may be in different tiles. This paper proposes an algorithm based on a tree reduction, called *the Tree Partitioning Reduction* (TPR) method, which partitions the problem into independent slices, that can be partially computed in parallel within different common shared-memory spaces. The TPR method can be implemented for any parallel and distributed programming paradigm. Furthermore, in this work, TPR is efficiently implemented for CUDA GPUs to solve large size problems, providing highly competitive performance results with respect to existing packages, being, on average, 22.03x faster than CUSPARSE.

Additional Key Words and Phrases: GPU, CUDA, Tuning, Tridiagonal systems, CUSPARSE

ACM Reference Format:
Adrián P. Diéguez, Margarita Amor, and Ramón Doallo. 2019. Tree Partitioning Reduction: A New Parallel Partition Method for Solving Tridiagonal Systems. *ACM Trans. Math. Softw.* 1, 1 (May 2019), 26 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Solving systems of linear equations with tridiagonal matrices arises in many scientific, engineering and computing problems, being a very important component in different fields, such as fluid dynamics, heat conduction, diffusion equations, numerical analysis, ocean models, cubic spline approximations and real-time or interactive applications in computer graphics. The Thomas algorithm [21] is the best-known sequential algorithm for solving these systems. Since the 1960s, a wide range of parallel algorithms for solving tridiagonal systems have been developed, among which Cyclic Reduction (CR) [10], Parallel Cyclic Reduction (PCR) [9] and Recursive Doubling (RD) [20] are the most notable methods. These algorithms have been historically implemented in vector supercomputers first, and later with a message-passing paradigm as an alternative. Nowadays, these algorithms have been also implemented in *Graphics Processing Units* (GPUs), since they are used for

Authors' address: Adrián P. Diéguez; Margarita Amor; Ramón Doallo, University of A Coruña, Department of Computer Engineering, A Coruña, 15073, Spain, {adrian.perez.dieguez,margarita.amor,doallo}@udc.es.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2019 Association for Computing Machinery.
0098-3500/2019/5-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

scientific computation, providing high computational throughput and large memory bandwidth, being less expensive with rather lower power consumption than CPUs.

It should be noted that many applications require solving a number of tridiagonal systems simultaneously. Large problem sizes are the most interesting ones, and there are several reasons for solving this kind of problems [13]: (*i*) a set of small systems can be expressed as a single large problem by joining their matrices, (*ii*) solving a large problem is the most difficult case to implement, since there is no independence to compute slices of the problem separately, in addition to which the common shared-memory is limited. In the case of GPU programming, there is also another strong reason: (*iii*) although there are a great number of GPU-based solvers for small problem sizes, only few implementations can handle large problem sizes.

In this paper, a new parallel partition method for solving large-problem sizes is described, called *Tree Partitioning Reduction* (TPR). This approach can be easily implemented using any parallel and distributed programming paradigm; specifically, this work also provides an efficient GPU CUDA implementation which allows several large-size problems to be solved simultaneously with a single invocation to our method.

The rest of the paper is organized as follows, after summarizing the related work: Section 2 introduces the basics of tridiagonal systems and the best known parallel methods. Section 3 presents the new partition method we propose, the Tree Partitioning Reduction algorithm; whereas Section 4 gives an introduction to CUDA GPU programming and discusses the CUDA implementation of our algorithm. Section 5 shows the experimental results and their analysis; and finally, Section 6 presents our conclusions and future work.

2 TRIDIAGONAL SYSTEM PRELIMINARIES

A tridiagonal system is composed of N equations E_i, with $i = 1, \cdots, N$ where E_i has the form: $a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i$. The system can also be represented by its coefficient matrix, A. The b_i coefficients constitute the main diagonal of the coefficient matrix, whereas a_i and c_i are known as the lower and upper diagonals, respectively. Thus, $Ax = d$, where x and d are vectors.

$$A = \begin{pmatrix} b_1 & c_1 & & & & \\ a_2 & b_2 & c_2 & & 0 & \\ & a_3 & b_3 & c_3 & & \\ & & \ddots & \ddots & \ddots & \\ & 0 & & \ddots & \ddots & c_{N-1} \\ & & & & a_N & b_N \end{pmatrix}$$

In this matrix, a_1 and c_N values are zero. If $|b_i| \geq |a_i| + |c_i|, \forall i = 1, \cdots, N$, then the system is known as diagonally dominant. This kind of matrix guarantees numerical stability in most of the algorithms proposed in the literature.

In a system solver, as seen above, an equation E_i is composed of different unknowns depending on the step k in which: $E_i^k \equiv a_i^k x_{i-u} + b_i^k x_i + c_i^k x_{i+u} = d_i^k$ is computed where u is a function of k. For the sake of clarity, an equation E_i^k is represented in this article by three numbers $\{i-u, i, i+u\}$ which correspond to the indices of the unknowns that compose that equation in the step k.

2.1 Thomas Algorithm

The classic algorithm for solving tridiagonal systems is the Thomas algorithm [21], which is based on Gaussian elimination. The algorithm comprises two phases, *forward elimination* and *backward*

substitution. The first phase eliminates the first unknown in each equation (a_i coefficient in E_i equation) by

$$c_i^{k+1} = \frac{c_i^k}{b_i^k - c_{i-1}^{k+1} a_i^k}$$
$$d_i^{k+1} = \frac{d_i^k - d_{i-1}^{k+1} a_i^k}{b_i^k - c_{i-1}^{k+1} a_i^k} \quad \text{with} \quad i = 1, ..., N$$

The second phase solves the reduced system by back substitution:

$$x_N = d_N^{k+1}$$
$$x_i = d_i^{k+1} - c_i^{k+1} x_{i+1}, \quad i = N-1, ..., 1$$

This algorithm is inherently serial, taking $2 \cdot N$ computation steps, since c_i^{k+1}, d_i^{k+1} and x_i depend on the preceding $c_{i-1}^{k+1}, d_{i-1}^{k+1}$ and x_{i+1}.

2.2 Parallel Algorithms

There are several parallel algorithms for solving tridiagonal systems, but Cyclic Reduction (CR) [10] and Parallel Cyclic Reduction (PCR) [9] are the most popular methods [4]. Additionally, the Wang and Mou algorithm [23] is also a well-known parallel tridiagonal solver.

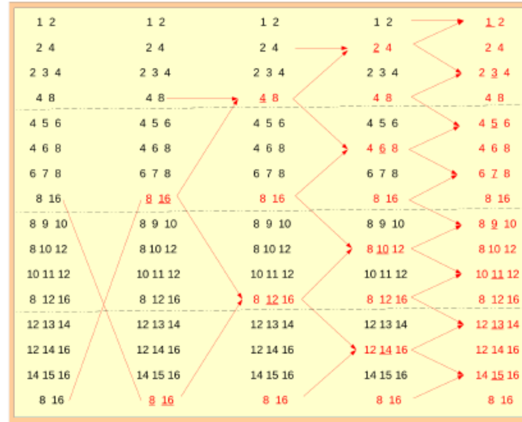
2.2.1 The Cyclic Reduction Algorithm (CR). On the one hand, CR (see Figure 1), comprises two phases, *forward reduction* (Figure 1 (a)) and *backward substitution* (Figure 1 (b)). As already mentioned, an equation $E_i \equiv a_i x_{i-u} + b_i x_i + c_i x_{i+u} = d_i$ is represented as $E_i \equiv \{i-u, i, i+u\}$ in figures. Forward reduction reduces a system to another one comprising half the number of equations, until a 2-unknown 2-equation system is reached in $log_2 N - 1$ steps. Even-indexed equations are updated in parallel as a linear combination of their adjacent equations E_i, E_{i-u} and E_{i+u}, deriving a system of only even-indexed unknowns by Eq. 1:

$$a_i^{k+1} = -a_{i-u}^k s_1, \quad b_i^{k+1} = b_i^k - c_{i-u}^k s_1 - a_{i+u}^k s_2, \quad with \ s_1 = \frac{a_i^k}{b_{i-u}^k}$$
$$c_i^{k+1} = -c_{i+u}^k s_2, \quad d_i^{k+1} = d_i^k - d_{i-u}^k s_1 - d_{i+u}^k s_2, \quad with \ s_2 = \frac{c_i^k}{b_{i+u}^k} \tag{1}$$

where k denotes the step of the algorithm; i starts from all even index and shrinks exponentially for each step k; and u starts from 1 and increases exponentially step-by-step, $u = 2^{k-1}$. After one step of the algorithm, redundant unknowns and zeros are removed, and a half-size matrix is formed by the remaining unsolved equations. As Figure 1 (a) shows for the forward reduction with $N = 16$ elements, the system is reduced to another one where the selected equations to be kept in the updated system, the even-indexed ones, are highlighted in red, building a half-size system. In this example, the process is repeated along 3 steps, until a two-unknown two-equation system is obtained.

In each step of the backward substitution, unknowns x_i are solved in parallel. To do this, the previously solved x_{i-u} and x_{i+u} values at step $k-1$ are substituted on each E_i equation in the step k; repeating this process along $log_2 N$ steps, as Eq. 2 defines:

(a) Forward Reduction



(b) Backward Substitution

Fig. 1. Pattern of Cyclic Reduction (CR) with $N = 16$ elements.

$$x_i = \frac{d_i^k - a_i^k x_{i-u} - c_i^k x_{i+u}}{b_i^k} \tag{2}$$

where u decreases exponentially step-by-step, from $N/2$ to 1 ($u^{k+1} = u^k/2$); and the domain of i also increases exponentially, employing a double-size system in each step. Figure 1 (b) shows the backward substitution when solving $N = 16$ unknowns. As can be observed, the process takes 4 steps, and each step uses the double of equations with respect to the previous step, solving the double of unknowns.

2.2.2 The Parallel Cyclic Reduction (PCR). On the other hand, PCR is a modification of CR that performs the forward reduction phase in $log_2 N - 1$ steps, and the substitution phase in only 1 step. The PCR forward reduction is performed on all equations, instead of even equations only;

thus, doing asymptotically more work per step. Therefore, the domain of i does not decrease exponentially. To do this, two systems are initially considered at step $k = 1$, one whose odd-indexed equations are updated with its adjacent equations ($u = 1$), and another one which updates its even-indexed equations. In both systems, the equation E_i^2 is updated with E_i, E_{i-u} and E_{i+u}, where $u = 1$ at this initial step. Regarding the remaining steps, each of the existing systems at step k is divided into two systems of half-size in the step $k + 1$, where the stride u increases exponentially $u = 2^{k-1}$. After $log_N - 1$ steps, there are $N/2$ two-unknown systems that can be solved in parallel in only one step as Eq.3 shows:

$$x_i = \frac{b_{i+N/2}^k d_i^k - c_i^k d_{i+N/2}^k}{b_{i+N/2}^k b_i^k - c_i^k a_{i+N/2}^k}$$

$$x_{i+N/2} = \frac{d_{i+N/2}^k b_i^k - d_i^k a_{i+N/2}^k}{b_{i+N/2}^k b_i^k - c_i^k a_{i+N/2}^k} \tag{3}$$

Figure 2 depicts the pattern of PCR with $N = 16$ elements. It performs the forward reduction along 3 steps, and the substitution phase in one step. In each step, each of the existing systems (marked with different colours) is divided into two systems of half-size, until achieving eight two-unknown two-equation systems. At this point, it is possible to solve them in one step and the algorithm is finished after $log_2 N$ steps.

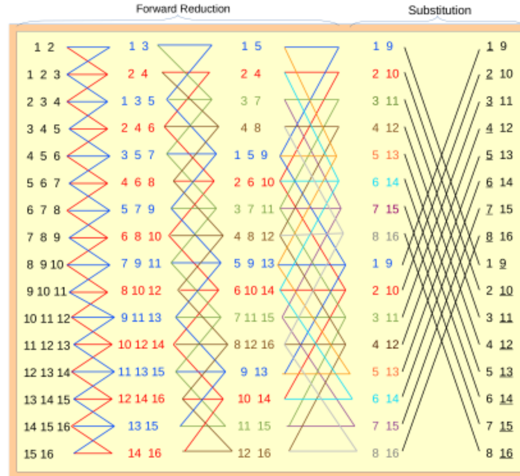


Fig. 2. Pattern of Parallel Cyclic Reduction (PCR) with $N = 16$ elements.

2.2.3 *The Wang & Mou Algorithm.* In addition to these algorithms, the Wang and Mou (WM) algorithm, presented in Figure 3 for $N = 8$ equations, divides the computation into $log_2 N$ steps. Each element of the algorithm is composed of three equations, also known as a triad of equations. For the sake of clarity, and in order to be homogeneous with the other algorithms, each element (triad) is represented as E_i, which is composed of three equations, labeled as *L*eft, *C*enter and *R*ight. As previously, each equation is formed of $a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i$.

Unlike CR and PCR, WM has an unique reduction phase, where E_i^{k+1} and E_{i+u}^{k+1} are both updated (instead of only E_i^{k+1}) using E_i^k and E_{i+u}^k with $u = 2^{k-1}$. The updating of coefficients is more complex

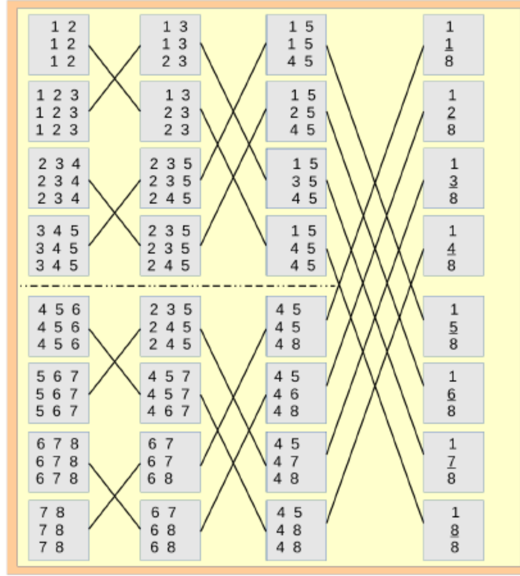


Fig. 3. Pattern of Wang& Mou algorithm (WM) with $N = 8$ elements.

in WM than in CR and PCR, as Figure 4 represents. A more detailed explanation of the coefficient updating in WM can be found in [15]. Initially, each equation in the original system is triplied in order to build L_i, C_i and R_i for each element E_i. After this, in order to reduce E_i and E_{i+u} in each step, the algorithm operates as follows: first, the middle term of equation R_i is used to reduce the first term in the three equations of E_{i+u}, using the Gaussian elimination, already explained above. Next, the middle term of the new equation in L_{i+u} is used to reduce all the other coefficients. At the end of the sequence, both left equations will be identical, and the same happens with both right equations. In the last step, the Center equation, C_i, of each element, E_i, is composed of only one unknown, thus it can be inmediately solved by Eq. 4:

$$x_i = \frac{d_i^k}{b_i^k} \tag{4}$$

Figure 3 represents the pattern of WM in order to solve a system of $N = 8$ equations. Each grey-box represents an E_i element, where its top equation is L_i, its middle one is C_i and its bottom one is R_i. After $log_2 N$ steps, each C_i equation can easily obtain the corresponding x_i value.

2.3 The Partitioning Problem

As introduced above, partitioning the system is crucial for surpassing the memory capacity restriction. In order to efficiently solve the problem on distributed platforms, each private-memory system of the distributed platform must process a subset of equations as independently as possible, to avoid communication latency. In this work, each subset of equations, which is computed in a private-memory space, is called a *slice*. However, most of the parallel algorithms cannot be easily partitioned. In the case of the Cyclic Reduction (CR) method, equations that take part in a reduction may belong to different slices. Specifically, the equation E_i^k, at step k, is the result of reducing the $[E_{i-u}^{k-1}, E_i^{k-1}, E_{i+u}^{k-1},]$ equations, with $u = 2^{k-1}$. Figure 1 showed an example of the CR method for

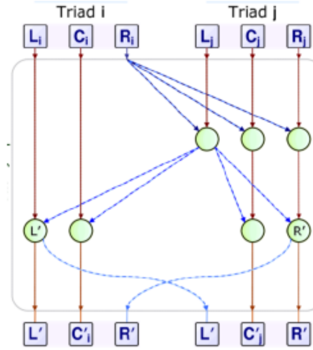


Fig. 4. Reduction of two triads in the Wang and Mou algorithm

$N = 16$ equations. Specifically, Figure 1 (a) depicts the forward reduction phase. As can be observed in this figure, the problem cannot be directly partitioned into independent slices (marked with horizontal dashed lines), as equations need from other slice equations to be reduced. The same problem happens in the subsitution backward phase (see Figure 1 (b)).

Regarding PCR, this algorithm cannot be splited into different slices. Figure 2 showed the same problem as CR for the forward reduction: the participating equations in each calculation do not belong to the same slice.

With respect to WM, as can be observed in Figure 3, the computation can be easily performed in different independent slices. This is possible due to the fact that each element is used for only one reduction; whereas equations of PCR and CR can be used in different reductions.

In terms of computation, both PCR and WM require fewer algorithmic steps than CR to solve the system, but do asymptotically more work per step. In distributed (or non shared memory) machines only WM can easily partition the problem into independent shared-memory spaces, as seen above; whereas PCR and CR have a communication pattern with strong dependences that do not allow the system to be partitioned (since equations from one shared-memory space would need equations from another shared-memory space). However, although WM can easily divide the problem into independent slices, the fact of not having access to all equations makes the application of the adjacency property impossible [7], tripling the number of equations to be stored. This is a serious issue in memory-bound problems, as the number of transactions is also triplied. Thus, it is necessary to find an algorithm that partitions well the problem into independent shared-memory spaces and does not consume a huge amount of memory bandwidth. This algorithm, the Tree Partitioning Reduction Method (TPR), is presented in Section 3.

2.4 Related Work

There are many tridiagonal system solver implementations on GPUs. Most of them solve small problems that can be stored in the GPU shared memory, such as [2, 24], where parallelism is inherent and there is no partitioning overhead. In [6], a new tridiagonal system solver based on a parallel prefix sum pattern is developed. *CUDPP* [17] is another accelerated GPU library that solves small-size tridiagonal systems and other parallel operations.

In [1], the authors first recognized that partitioning is essential for solving large matrices on GPUs, using a hybrid PCR - Thomas algorithm to do so, although this algorithm suffers from a computation overhead. Argüello et al. [3] proposed a split-and-merge method based on the CR algorithm, reducing the overhead from previous proposals. This split-and-merge approach is later refined in [5]. In [8], a

partition method based on the SPIKE [19] algorithm is presented. Additionally, a diagonal pivoting method for numerical stability is first introduced in [13]. Combining QR factorization with Given rotations in [22] improved the previous implementation. In [25], a CR-based approach for solving large-problems is also presented; whereas a asymmetric banded solver with a direct approach that scales very well across many processors is proposed in [11]. Finally, *NVIDIA* implements CUSPARSE [16], a library that uses a hybrid CR-PCR implementation with pivoting for solving large-problem sizes.

A different approach was presented in [15] for small problem sizes and extended in [7] for medium and large problem sizes. This approach adapts the Wang and Mou algorithm [23] for CUDA-enabled *GPU* architectures. The Wang and Mou algorithm is based on the same Divide-and-Conquer strategy [18] as the SPIKE algorithm; however, in contrast to the SPIKE algorithm, the diagonalization of each block is performed using the Gaussian elimination method, also reordering the equations in a different way.

3 THE TREE PARTITIONING REDUCTION METHOD

In this section, a new tridiagonal system solver, called *Tree Partitioning Reduction* (TPR), is presented. This method is based on a division of the problem into independent slices to compute large-problem sizes. The TPR algorithm has two phases: the forward reduction and the backward substitution. In contrast to most solvers, equations that take part in the TPR can be reduced independently in each slice over many steps, facilitating the computation of large systems.

The goal of the forward reduction, which is shown in Figure 5 where each i−box represents the E_i equation, is to compute as many independent steps in slices as possible; where there is no communication between slices, and finally, to integrate all the resulting equations in the lowest number of steps possible. In the backward substitution, unknowns are solved with the equations obtained in the forward reduction.

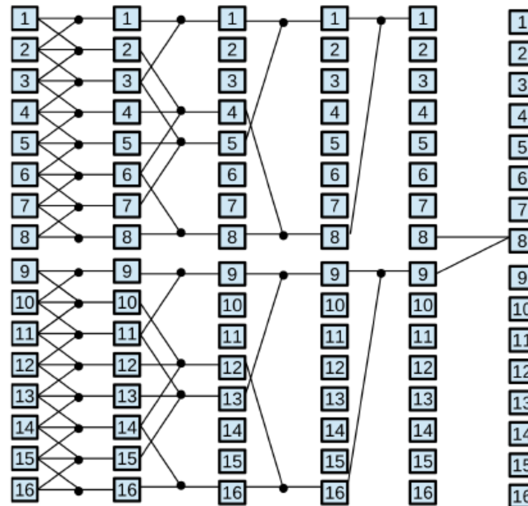
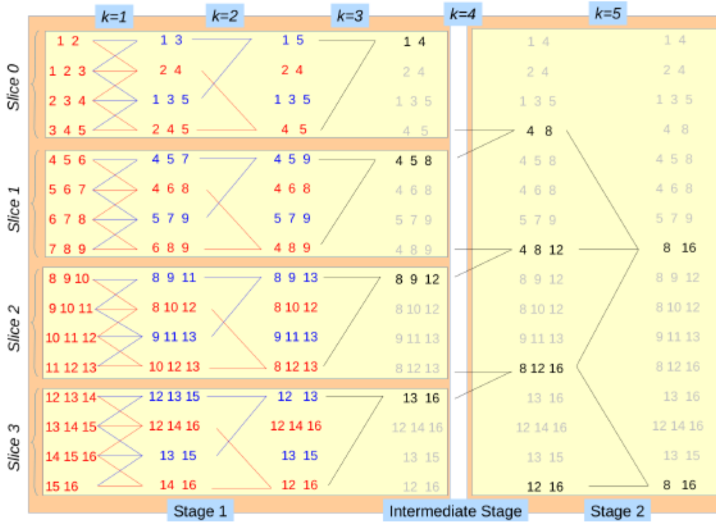
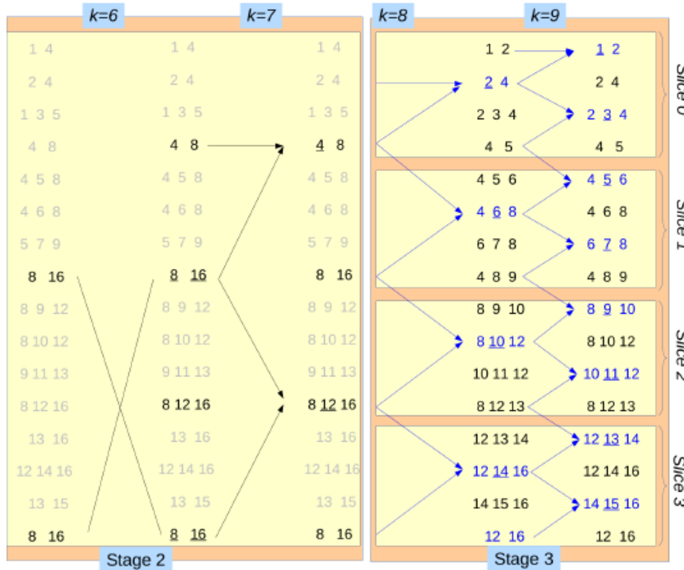


Fig. 5. Forward reduction phase for $N = 16$ elements in the TPR method, with $S = 8$.



(a) TPR forward reduction



(b) TPR backward substitution

Fig. 6. Tree Partitioning Reduction example for $N = 16$ elements with $S = 4$

3.1 The TPR Forward Reduction phase

The TPR Forward Reduction phase, see Figure 5, divides a tridiagonal system of N equations into slices of size S; i.e., a divide-and-conquer algorithm where each slice is reduced independently from other slices. Specifically, the algorithm treats each slice as two different sub-systems, one that operates over the initial odd-indexed equations (red colour), and the other that operates over the initial even-indexed equations (blue colour). This mechanism can be seen in Figure 6 (a) for $N = 16$

and $S = 4$ where each of the two sub-systems is marked with a different colour. In each step of the forward reduction, each of the two sub-systems is reduced to another with half the number of equations. In order to update the equation E_i^k in the step k, the equations E_{i-u}^{k-1}, E_i^{k-1} and E_{i+u}^{k-1} are used, with $u = 2^{k-1}$ shrinking exponentially in each step, as Figure 7 depicts, forming a tree reduction schema, following Eq. 5:

$$\begin{aligned} a_i^{k+1} &= -a_{i-u}^k s_1, \quad b_i^{k+u} = b_i^k - c_{i-u}^k s_1 - a_{i+u}^k s_2, \quad with \;\; s_1 = \frac{a_i^k}{b_{i-u}^k} \\ c_i^{k+1} &= -c_{i+u}^k s_2, \quad d_i^{k+u} = d_i^k - d_{i-u}^k s_1 - d_{i+u}^k s_2, \quad with \;\; s_2 = \frac{c_i^k}{b_{i+u}^k} \end{aligned} \tag{5}$$

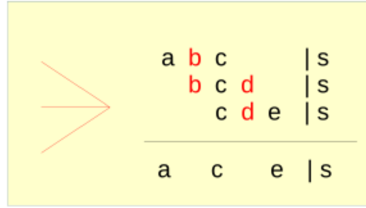


Fig. 7. Coefficient reductions in the TPR forward reduction phase for a node computation

After log_2S steps, each of the two sub-systems is composed of only one two-unknown equation. In the step $k = log_2S + 1$, the equation of the first sub-system is updated with the corresponding equation of the second sub-system, obtaining a single two-unknown equation. This procedure is repeated in every slice and, consequently, the step $k = log_2S + 2$ integrates all the slices by reducing the unique equation of each slice with the unique equation of its adjacent slice. After these $log_2S + 2$ steps, the remaining $log_2(N/S) - 1$ steps reduce the created system until obtaining a two-unknown two-equation system with a conventional tree-reduction schema: each step reduces the system to another with half of the number of equations, using the coefficient updating explained in Eq. 5 over the even-indexed equations. It should be observed that the last $log_2(N/S) - 1$ steps are not calculated in independent slices due to the dependencies between equations. It should be observed that the term *stage*, which appears in Figure 6, is related with the implementation, where each stage, or kernel, represent a set of steps performed in one computing function, as will be explained in Section 4.

More precise details about the algorithm are presented in Figure 8, which shows the pseudo-code for the TPR forward reduction phase. The first for-loop (lines 4-16), along log_2S steps, updates the coefficients of the two sub-systems. To do this, a while-loop (lines 7-15) iterates over the different S slices, where the variable p controls the slice. The two inner for-loops update the coefficients of its corresponding sub-system. Then, the step $log_2S + 1$ is performed in the lines 18-24. The next for-loop, lines 27-29, integrates the unique equation of each slice with its adjacent. Finally, the second while-loop (lines 32-39) performs the remaining $log_2(N/S) - 1$ steps. The *update* function calculates the coefficients of the i equation (E_i), following the formula of Eq.5, with the $i \pm u$ equations (E_{i-u}, E_{i+u}).

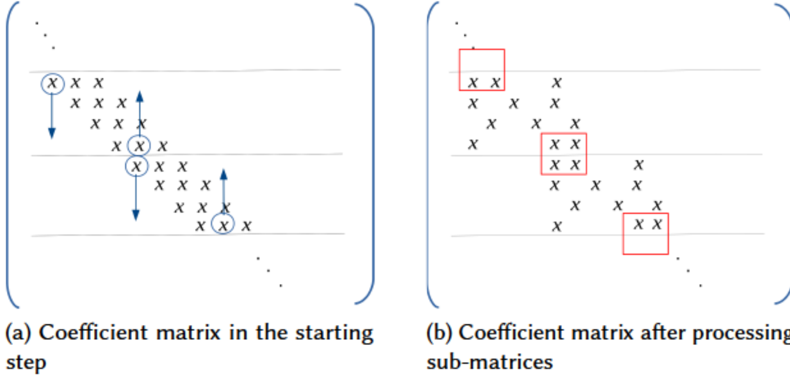
The fact of performing $log_2N + 1$ steps in independent slices, and integrating them later, is possible thanks to work with the specific columns of the coefficient matrix in each step. The coefficient matrix A is divided into $M = N/S$ sub-matrices of equal size S, $A = \{A_0, ..., A_{M-1}\}$, where each sub-matrix corresponds to an independent slice. A sub-matrix A_j is composed of the

```
1  PROCEDURE TPR_forward(A,S,N)
2  k:=0
3  //Stage 1
4  for k:=1; k≤ log2S; k:=k+1
5    u:=2^(k−1)
6    p:=1
7    while(p≤N)
8      for i:=p; i<p+S; i:= i+2^k
9        update(i,±u)
10     end
11     for i:=p+2^k −1; i<p+S; i:= i+2^k
12       update(i,±u)
13     end
14     p:= p+S
15   end
16 end
17
18 k:= k+1
19 u:= 2^(k−1)
20 p:= 1
21 while(p≤N)
22   update(p,+(u−1))
23   p:=p+S
24 end
25
26 //Intermediate Stage
27 for i:=S; i ≤ N−S; i:= i+S
28   update(i,+1)
29 end
30 //Stage 2
31 j:=0
32 while(j<log2(N/S)− 1)
33   u:=2^(k−1)
34   for i:=2^k; i≤N; i:= i+2^k
35     update(i,±u)
36   end
37   j:= j+1
38   k:=k+1
39 end
```

Fig. 8. Pseudo-code of the TPR forward reduction phase.

following set of equations $\{E_{j\cdot S+1}, \cdots, E_{j\cdot 2\cdot S}\}$, represented as its rows. The TPR method transforms the starting coefficient matrix, as shown in Figure 9 (a), into an equivalent matrix composed of sub-matrices where the bottom row (equation) of each sub-matrix has two common columns (unknowns) with respect to the top row of its lower sub-matrix, as represented in Figure 9 (b). Thanks to this process, each sub-matrix computes a high number of steps independently, and subsequently, can easily use equations from other sub-matrices to build the overall final reduction. This transformation is carried out in $log_2 S + 1$ steps, called *sliced forward reduction*, where the rows of a sub-matrix are independently reduced with other rows from the same sub-matrix, creating zeros in some columns and giving values to others. As can be observed, following the presented reduction schema guarantees that, after $log_2 S + 1$ steps, columns from the top rows of each sub-matrix are carried to its bottom rows in order to have common columns with lower sub-matrices; i.e., provide these unknowns to other sub-matrices and solve the overall system. After $log_2 S + 1$ steps, the corresponding rows from one sub-matrix share the same two columns with adjacent sub-matrices, allowing them to be reduced (integrated) with the tree-form reduction presented above.

In terms of computation, each sub-matrix represents a slice of the problem to be computed in an independent memory space. Equations of each slice are independently computed through $log_2 S + 1$ iterations, without communication among these memory spaces. In a parallel environment, each



(a) Coefficient matrix in the starting step

(b) Coefficient matrix after processing sub-matrices

Fig. 9. Coefficient matrix evolution in the TPR method

thread or *computing process* is the responsible of perfoming the coefficient updating for an equation, which is called *node computation* or *reduction*. In Figure 5, the node computations are represented with black circles. The first $log_2 S + 1$ steps are performed independently in each memory space (slice); however the last $log_2 N/S$ steps need communication between the different participating memory spaces.

To sum up, firstly the equations of each slice are independently reduced in $log_2 S + 1$ steps. These steps are also known as *sliced forward reduction*. Then, the bottom equation of each slice is updated with the top equation of its corresponding lower slice in one step. And finally, the resulting equations of previous steps are solved in a single overall matrix, in $log_2(N/S) - 1$ steps. At this point, the backward substitution phase is performed.

For the sake of clarity, Figure 6 showed an example of the TPR method for $N = 16$ equations, where the matrix is divided into $M = 4$ independent sub-matrices of size $S = 4$; i.e., the computation is divided into four slices. The sliced forward reduction phase is performed in $(log_2 S) + 2 = 4$ steps, as defined above. Specifically, the first $log_2 S + 1$ steps are computed inside each slice, with no communication with other slices. One sub-system is represented with the red colour, and the other one in blue. To perform the forward reduction, the suitable E_i equations of each sub-system are reduced with the corresponding E_{i-u} and E_{i+u} equations inside each slice, where u shrinks exponentially in each step ($u = \{1, 2, 4\}$). For example, the following operations are performed for slice 0 (sub-matrix A_0):

- $k = 1$: The equations $\{E_1, E_3\}$ in one sub-system, and $\{E_2, E_4\}$ in the other, are respectively updated with $u = 1$.
- $k = 2$: $\{E_1\}$ and $\{E_4\}$ are updated, respectively, with $u = 2$.
- $k = 3$: $\{E_1\}$ is updated with $\{E_4\}$. At this point, each slice has an unique equation.
- $k = 4$: The unique equation of each slice is updated with the unique equation of its adjacent slice. In the case of slice 0, E_4 (slice 0) is updated with E_5 (slice 1), creating an overall system that cannot be split into slices.

At this point, the created overall system is reduced through steps, as explained previously: reducing the system to another with half the number of equations in each step. This reduction is performed in $k = 5$, achieving a 2-equation 2-unknown system whose unknowns are x_8 and x_{16}. This system can be solved directly in the first step of the backward substitution phase.

Please, note that the examples of this work are given for power-of-two sizes. In the case of an arbitrary N, it can be expressed as $2^n + N'$. On the one hand, the sliced forward reduction for 2^n is calculated as explained in slices of size S. On the other hand, the smallest power of 2 able to compute N' is calculated, fulfilling the extra equations with the last equation of N'. Then, both 2^n and N' are integrated into the overall system, after the sliced forward reduction, to perform the substitution.

3.2 The TPR Backward Substitution phase

Concerning the backward substitution, a two-unknown system composed of two equations is received from the forward reduction phase and is solved in the first step of the backward substitution. As defined before, the system was divided in M partitions of size S. Thus, the whole phase takes $log_2 N$ steps: specifically, the solution of unknowns during the first $log_2 M$ steps is solved as a single system; whereas the system can be partitioned into slices in the remaining $log_2 S$ steps.

The first step solves the two-unknown two-equation system, with $u = N/2$, by Eq. 6:

$$x_i = \frac{b^k_{i+N/2} d^k_i - c^k_i d^k_{i+N/2}}{b^k_{i+N/2} b^k_i - c^k_i a^k_{i+N/2}}$$

$$x_{i+N/2} = \frac{d^k_{i+N/2} b^k_i - d^k_i a^k_{i+N/2}}{b^k_{i+N/2} b^k_i - c^k_i a^k_{i+N/2}} \tag{6}$$

Then, each step of the backward substitution solves the unknown variables in the overall matrix in the next $log_2 M - 1$ steps, by substituting the solutions obtained from the previous step in Eq. 7:

$$x_i = \frac{d_i - a_i x_{i-u} - c_i x_{i+u}}{b_i} \tag{7}$$

decreasing u exponentially step-by-step, $u^k = \frac{u^{k-1}}{2}$, whereas the domain of i increases exponentially, as Figure 6 (b) shows. After these steps, the odd-indexed E_i equations are replaced with their homologous E_i equations from the original system (step $k = 0$); whereas the even-indexed equations are replaced with their corresponding equations from the system after the sliced forward reduction, step $k = log_2 S + 1$. Thanks to this replacement, the system can be now partitioned into slices in order to solve the unknowns in the last $log_2 S$ steps, following Eq. 7. It should be observed that this computation of the last $log_2 S$ steps can be performed in independent slices, since each slice only needs to know the bottom unknown of its upper slice, which does not vary during the final steps. These steps are known as the *sliced backward substitution*. This algorithm can be implemented efficiently in any parallel programming paradigm, as Section 4 shows.

This procedure is possible thanks to the matrix operation properties and the Gaussian elimination. In the first $log_2 M$ steps of the substitution phase, after the forward reduction phase, there are M rows (with stride S) with enough common columns between them to solve the unknowns. In order to solve the remaining unknowns (i.e., having enough common columns between the participating rows), it is necessary to transform the coefficient matrix. Let us define A^0 as the initial coefficient matrix (step $k = 0$); and $\hat{A}$ as the coefficient matrix generated after the step $k = log_2 S + 1$. Replacing the odd-indexed rows of the matrix by the odd-indexed rows from A^0, and the even-indexed rows by the even-indexed ones from $\hat{A}$, there are enough common columns between the corresponding rows to be eliminated and solve the unknowns.

```
1  PROCEDURE TPR_substitution(A,S,N)
2
3  I:=N/2
4  M:=N/S
5  //Stage 2
6  u:=N/2
7  i:=I
8  x_i := (b_{i+u} d_i − c_i d_{i+u}) / (b_{i+u} b_i − c_i a_{i+u})
9
10 x_{i+u} := (d_{i+u} b_i − d_i a_{i+u}) / (b_{i+u} b_i − c_i a_{i+u})
11
12 j:=0
13 while (j<log2M − 1)
14   I:=I/2
15   u:=u/2
16   for i:=I; i≤N; i:=i+2u
17       x_i := (d_i − a_i x_{i−u} − c_i x_{i+u}) / b_i
18   end
19   j:=j+1
20 end
21
22 for i:=1; i≤N; i:=i+1
23   if (i%2=0)
24     replace(i,log2S + 1)
25   else
26     replace(i,0)
27   end
28 end
29 //Stage 3
30 for j:=0; j<log2S; j:=j+1
31   I:=I/2
32   u:=u/2
33   p:=0
34   while(p<N)
35     for i:=p+I; i≤p+S; i:=i+2u
36        x_i := (d_i − a_i x_{i−u} − c_i x_{i+u}) / b_i
37     end
38     p:=p+S
39   end
40 end
```

Fig. 10. Pseudo-code of the TPR substitution phase.

A more detailed description of the algorithm is presented in the pseudo-code of Figure 10. The first step of the backward substitution, which solves $x_{N/2}$ and x_N, is performed in line 8 and line 10, respectively. After that, the next $log_2M - 1$ steps are performed in a while-loop (lines 13-20). The offset u decreases exponentially in each step. The variable I is used to control the stride for the first equation of each iteration, which also decreases exponentially. As can be observed in the for-loop of lines 16-18, the domain of i increases in each iteration thanks to reduce I and u. The i unknowns are solved by using the formula of Eq.7. After this point, equations of the system are replaced, as explained above, in the for-loop of lines 22-28, considering even-indexed equations ($i\%2 = 0$) and odd-indexed equations. The *replace* function changes the current i-equation, indicated in the first argument of the function, by the values of this equation after the step indicated in the second argument. Finally, the for-loop of lines 30-40 perform the remaining log_2S steps in slices. To do this, the while-loop (lines 34-39) performs the solution of unknowns for each slice, controlling the slice with the variable p.

In the example of Figure 6 (b), there are $N = 16$ equations with $S = 4$; thus, the unknowns x_8 and x_{16} are solved in the first step of the backward substitution phase (global step $k = 6$), since

a 2-unknown 2-equation system was obtained from the forward reduction phase. As $M = 4$, the next step ($k = 7$) is solved in a single overall system, where the unknowns x_4 and x_{12} are solved. After this point, the remaining steps ($k = 8$ and $k = 9$) can be performed in independent slices for the last 2 steps. It should be observed that the slice j needs the unknown $x_{j \times S}$ during every step of the sliced backward substitution, which belongs to the slice $j - 1$; i.e., slice 1 needs x_4, slice 2 needs x_8 and slice 3 needs x_{12}. However, the value of $x_{j \times S}$ does not vary since the sliced backward substitution starts and, in terms of computation, storing this single value does not imply either a bottleneck or overhead.

3.3 Algorithms comparison and GPU implementation considerations

Table 1 highlights the links between the explained algorithms of this work:

Algorithm	Arithmetic Operations	Steps in Forward phase	Steps in Substitution phase	Observations
CR	$17N$	$log_2N - 1$	log_2N	Dependences for partitioning
PCR	$12N log_2N$	$log_2N - 1$	1	Dependences for partitioning
WM	$48N(log_2N - 1) + N$	$log_2N - 1$	1	Huge memory bandwidth requirements
TPR	$24M(S - 2) + 60M - 36 + 5N$	$log_2N + 1$	log_2N	Easy to partition, fewer inter-systems memory transactions

Table 1. Comparison of algorithms, where N is the number of equations, M is the number of slices and S is the slice size, and assuming these parameters power of two.

From a GPU point of view, CR and PCR algorithms are easily implemented in CUDA when the problem fits in the shared memory of one thread block. Otherwise, when the problem is bigger than the shared memory capacity, the computation needs to be divided among several thread blocks, which hinders the CUDA implementation.

Regarding the GPU performance of WM, this algorithm is well suited to GPU architectures, as demonstrated in [15]. This algorithm is easily partitioned among different thread blocks for solving large problem sizes [7] when exceeding the GPU shared-memory size. However, when distributing the equations among different thread blocks, the elements are not stored in the same shared-memory space and it is necessary to work with (and store) whole triads, decreasing the global performance due to memory bandwidth limits.

In terms of computation, as will be seen in Section 4 for a GPU implementation, the TPR algorithm matches excellently to distributed (or non-shared memory) systems. Although it performs $2log_2N + 1$ steps, most of them are performed in independent shared-memory systems; thus, the communication between systems is minimum, which is the best advantage in memory-bound problems in comparison to other paralell algorithms. Although in the sliced substitution backward phase, each slice needs one unknown from another shared-memory slice, this data is sent only once during the whole phase, which has no penalty.

4 AN EFFICIENT CUDA IMPLEMENTATION FOR THE TPR METHOD

As stated in the introduction, the TPR method can be implemented in any parallel and distributed programming paradigm. In order to show its efficiency for parallel platforms, this work provides an efficient implementation for GPU devices, since they play a huge role accelerating applications nowadays.

In order to correctly exploit the GPU parallelism, this solver has been implemented under the *BPLG* library [14]. This library is composed of *BPLG skeletons*, which are predefined and generic functions that implement common specific patterns of computation and data movements, customized with user-defined code parameters, and whose performance has been demonstrated

in [7]. These *BPLG skeletons* make extensive use of CUDA templates to create several optimized versions, depending on the problem size and the target architecture. Different static tables are built, where each entry represents a problem size indicating both the slice size, and the optimal values of the GPU tunable parameters (see Section 4.2) for the given execution. The library chooses the entry, and then kernels are built with the corresponding entry parameters at compile time. Hence, users do not have to generate kernels or worry about performance decisions. Most of the function calls, register loops and move operations are fully optimized at compile time; thus this library provides generality and usability, generating well performing kernels with little effort.

4.1 The CUDA GPU Programming Model

NVIDIA GPUs are made up of many streaming processors (SPs) organized into a set of streaming multiprocessors (SMs). Each SM has its own small high-speed on-chip programmable memory, called shared memory. It also has its own set of registers. In addition to this, the main memory of the GPU is called global memory and is accessible by different SMs simultaneously. On the software side, there are some programming interfaces, such as *OpenCL* and *CUDA*, for programming the software layer, although CUDA is the most popular when using NVIDIA GPUs. CUDA virtualizes threads, grouping them into a grid of thread blocks, enabling programmers to run thousands of threads and thread blocks regardless of the number of hardware-processors. The thread blocks compute a function (kernel) in parallel, where a stage corresponds with a kernel invocation. In the CUDA runtime, thread blocks are assigned to available SMs and, depending on the amount of required resources, each SM may execute several blocks simultaneously. Threads of the same thread block can exchange information through the thread block's shared memory, and a small number of adjacent threads, called warp, is executed at the same time (currently, one warp comprises 32 threads). Threads from the same warp can exchange information via registers with the *shuffle* instructions, avoiding shared memory communication. Finally, there is a barrier instruction to synchronize threads inside a thread block, but there is no instruction to synchronize thread blocks inside the kernel grid. However, launching a kernel involves an implicit global synchronization barrier between thread blocks, and can be used as a global synchronization mechanism. A more detailed description can be found in [12].

4.2 BPLG Implementation Parameters

In order to facilitate the understanding of our CUDA implementation, the following implementation parameters are defined. Firstly, each problem is composed of $N = 2^n$ elements. Additionally, G problems are simultaneously solved in a single invocation to the method. Each computation step has a given number of reductions, called *node computations*, which execute the reduction operation over a number of P elements. To do so, B thread blocks are executed per kernel, which can be arranged as $B = B_x \times B_y$, where B_x is the number of thread blocks used per problem, whereas B_y represents the number of batch problems being simultaneously executed. Each thread block is composed of L threads, and each thread works with P elements in private registers, whereas all threads share information through S elements in shared memory. As all data processed in registers have a copy in shared memory, $S = P \times L$ is obtained. Table 2 collects all previous parameters and their definitions.

4.3 BPLG CUDA Kernels for the TPR method

Our CUDA implementation of TPR divides the execution in three stages (kernels). The first kernel, Stage 1, is responsible for performing $log_2 S + 1$ steps of the forward reduction, where each slice (sub-matrix) is computed in one thread block. After $log_2 S + 1$ steps, the last equation of each slice uses the first equation of the next slice, thus communication among thread blocks is needed. In order

Problem Parameters

$N = 2^n$	Problem size.
G	Number of problems being solved simultaneously.
M	Number of partitions which solve the system independently.

GPU Parameters

S	Number of shared-memory elements per block.
P	Number of elements stored in registers per thread.
B	Number of thread blocks executed per GPU, where $B = B_x \times B_y$
L	Number of threads that compose a block, where $S = P \times L$ and $L = L_x \times L_y$

Table 2. Description of implementation parameters.

to do this, a second kernel, Stage 2, is launched, working as a global synchronization barrier among thread blocks. In Stage 2, each problem is represented by as many equations as the number of slices the first stage had ($M = N/S$). Stage 2 computes the last $log_2 M$ steps of the forward reduction, and the first $log_2 M$ steps of the backward substitution. Finally, Stage 3 computes the remaining steps of the backward substitution in slices of S equations, where each slice is again solved by a thread block. It should be noted that each slice needs the last equation of the upper slice to perform its substitutions, which is taken from global memory. In Figure 6, the division of stages can be seen.

Specifically, the first kernel is invoked with ($B_x = N/S,\ B_y = G$) thread blocks, and its pseudo-code is shown in Figure 11. Considering floating point single precision elements, each element is composed of four 4-byte elements, requiring 16 bytes of storage, that can be stored in a $float4$ datatype (line 3). In the case of double precision, it would be represented by a $double4$ datatype. Each thread performs a *node computation*, and although each *node computation* works with three elements, these elements are shared by two different *node computations*; thus each thread loads $P = 2$ elements in its own registers and takes the third element from shared memory (lines 7-11). Please observe that there are S *node computations* in the first step, but there are $L = S/P$ threads; considering $P = 2$, each thread has to compute two *node computations* in the first step (lines 13-19). In the following steps, the number of *node computations* shrinks exponentially; thus it is necessary to control the thread *id* to know which threads must perform the reduction (lines 26,31). Please observe that there are cases where the node computation only computes two elements. In these cases, the identity equation is assigned to the third element to avoid influencing in the computation and giving rise to produce branch divergence. Finally, the even-indexed equations, with $i\%2 = 0$, are stored in global memory, overwriting their previous values (line 38), whereas the E_i equations with $i\%2 \neq 0$ remain constant in global memory. Additionally, the bottom equation of each slice is stored in an auxiliary buffer for the next stage (lines 35-36). It should be noted that the size of this buffer corresponds to G problem times $B_x - 1$ slices per problem (the first slice of each problem can skip this storage action, since its equation is not used in future steps).

To optimize the communication among threads, the pseudo-code of Figure 11 can be improved with the use of shuffle instructions during the last four steps. Considering 32 threads/warp in current architectures, where each thread collaborates with one element, this implies four steps of computation where the comunication is entirely performed by shuffle instructions.

Regarding the second kernel (Stage 2), each problem needs as many elements as slices it had in the previous stage. As this number can be low, each thread block can compute several problems. In the first step, each element from the auxiliary buffer is reduced with its corresponding equation, as Figure 6 (a) shows. Then, a conventional reduction is applied, until being able to solve the unknowns $x_{N/2}$ and x_N; after which, the backward substitution starts.

```
template<int N> __global__ void
BPLG_TPR_Stage1(const float* __restrict__ src, float* bufferAux){
  Float4 reg[3];
  __shared__ Float4 shm[N];
  //Obtain id, offsets and strides
  ...
  //Load data from global mem2reg, reg2shm
  copy<2>(reg,src+strideId,...);
  copy<2>(shm+strideSHM, reg, ...);
  __syncthreads();
  copy<1>(reg+2,shm+strideSHM+offset,...);

  //First compute step
  Float4 aux[3];  //second node comp. in first step
  copy<1>(aux,shm+strideSHM-1,..);
  copy<1>(aux+1,reg,..);
  copy<1>(aux+2,reg+1,..);
  compute<2,MixStep>(reg);
  compute<2,MixStep>(aux);

  for(int accR=MixR; accR < N ; accR*=2) {
    __syncthreads();
    //Obtains strides and offsets
    ...
    //Reg-> Shm
    if(threadId<numThreads)
      copy<1>(shm+writeOffset, reg,..);
    __syncthreads();
    numThreads/=2;
    //Shm-> Reg
    if(threadId<numThreads)
      copy<3>(reg,shm+readOffset,..);
      compute<2>(reg); //Computation in registers
  }
  if(threadId==1)
    copy<1>(bufferAux+offset,reg+1,..);
  copy<1>(reg+1,shm+strideSHM+1,..);
  copy<1>(src+strideId+1,reg,..);
}
```

Fig. 11. Forward Reduction code for the TPR tridiagonal algorithm using BPLG.

Finally, the third kernel (Stage 3) performs the remaining substitutions that did not take place in the Stage 2. The number of thread blocks and threads per block is the same as in the first kernel, dividing the problem in the same number of slices. Observing Figure 6 (b), each slice needs the last element of its upper slice.

4.4 BPLG Tuning

The values of S and P (from which L is derived) are key in our implementation. In this work, the tuning strategy developed in [7] [15] has been followed. Basically, this strategy is composed of a set of performance premises. These premises say that it should be achieved: *(i)* a high granularity of work performed by threads (P parameter), which implies a high instruction level parallelism and a reduction of the high-latency communications, but must be aware of register consumption; *(ii)* a high warp occupancy to hide latency, also taking into account a high block parallelism rate per SM; and *(iii)* a performance trade-off between kernels, looking for a workload balance in performance. Additionally, coalescing patterns are essential for achieving the maximum memory bandwidth, especially in memory-bound problems.

Considering devices with compute capability 5.0, Table 3 shows different configurations and the corresponding parallelism achieved. The row in bold represents the configuration which maximizes

Archit.	Warps per block	Regs per thread	Shared memory per block	Warp occupancy	SM blocks
Maxwell	1	64	2048	50%	32
	2	**32**	**2048**	**100%**	**32**
	2	40	2560	75%	24
	4	32	4096	100%	16
	4	40	5120	75%	12
	8	32	8192	100%	8
	8	40	10240	75%	6
	16	32	16384	100%	4
	32	32	32768	100%	2

Table 3. Performance parameters which maximize the number of warps and blocks per SM

both warp and block occupancy. However, it is not always possible to use this configuration, since the resource consumption limits these occupancies. The idea is to maximize these values within the available resources.

As explained above, a problem of size N is solved by partitioning the data into $M = N/S$ slices of size S. The first and third kernel solve this problem with $B_x = N/S$ blocks of $L = S/P$ threads, whereas the second kernel solves N/S elements with $\frac{N}{S \cdot P}$ threads within a single block. In the case of solving G problems simultaneously, $B_y = G$ is used. In order to improve the warp occupancy and, as such, performance, each thread block of the second kernel computes L_y problems, resulting in an invocation of $B = G/L_y$ blocks.

Using $P = 2$ already implies employing 40 registers per thread; thus, higher P values would consume a huge amount of registers, resulting in inefficiency. Therefore, $P = 2$ must be used, and S is expressed as $S = 2 \cdot L$. It should be noted that the configuration marked in the row in bold cannot be applied to this case due to the register consumption; thus, an alternative configuration, which maximizes the occupancies as much as possible, must be found when consuming 40 or a higher number of registers per thread.

In the case of storing the unknowns in global memory, the amount of shared memory bytes per block (floats) is calculated as $S{\times}4\ coef/eq{\times}4\ bytes = 2{\times}L{\times}4{\times}4\ bytes$, as each equation is composed of 4 coefficients. For the first and third kernel, looking at Table 3, the row of $L = 64$ threads, 40 registers per thread and up to 2560 shared memory bytes per thread block, maximizes both the warp occupancy and the number of active thread blocks per SM, between all other possibilities that consume 40 registers or more. This configuration implies solving $N/128$ elements per problem in the second kernel. In order to maximize the second kernel performance, each block works with $L = 64 = L_y \times L_x$ threads, with the following configuration $L_x = \frac{N/128}{2}$ and $L_y = max(1, \frac{64}{L_x})$.

It should be noted that the case of $n \geq 19$ implies that the value of S is higher than 128. Otherwise, the number of threads in the second kernel would result in more than 1024 threads per threadblock. In this case, S is the minimum value higher than 64 that allows the execution of the second kernel (fewer or equal to 1024 threads for current architectures). Table 4 summarizes the tuning parameter values for each $N = 2^n$ value.

In the case of storing the unknowns in shared memory, the amount of shared memory bytes per thread block is $S \times 4\ coef/eq \times 4\ bytes + S \times 4\ bytes = 2 \times L \times (4 \times 4 + 4)\ bytes$. Due to the register consumption, the same warp and block parallelism is achieved as in the global memory case, thus the tuning values are the same.

Problem size	*Kernel 1,3*	*Kernel 2*
$n \leq 18$	$L = 64$	$L = (L_x, L_y) = (\frac{N}{128},\ max(1, \frac{64}{L_x}))$
$n = 19$	$L = 128$	$L = (L_x, L_y) = (N/256,\ 1)$
$n = 20$	$L = 256$	$L = (L_x, L_y) = (N/512,\ 1)$
$n = 21$	$L = 512$	$L = (L_x, L_y) = (N/1024,\ 1)$
$n = 22$	$L = 1024$	$L = (L_x, L_y) = (N/2048,\ 1)$

Table 4. Description of tuning parameters, being $S = P \cdot L$ and $P = 2$.

5 EXPERIMENTAL RESULTS

This section presents two different analyses for our proposal on the CUDA Maxwell platform exposed in Table 5, a performance study and a numerical-stability study. Tests were run in both floating point single and double precision, the data of which already reside in the device memory at the beginning of each test. There are no data transfers to the CPU during the benchmarks to avoid interactions with other factors in the analysis.

	Platform: Maxwell Architecture
CPU	Intel Core i7-2600 3.4 GHz
Memory	8 GB DDR3 1333
OS	Ubuntu 12.04 LTS
Compiler	GCC 4.6.3
GPU	Nvidia GeForce GTX980
Driver	384.9, SDK 8.0

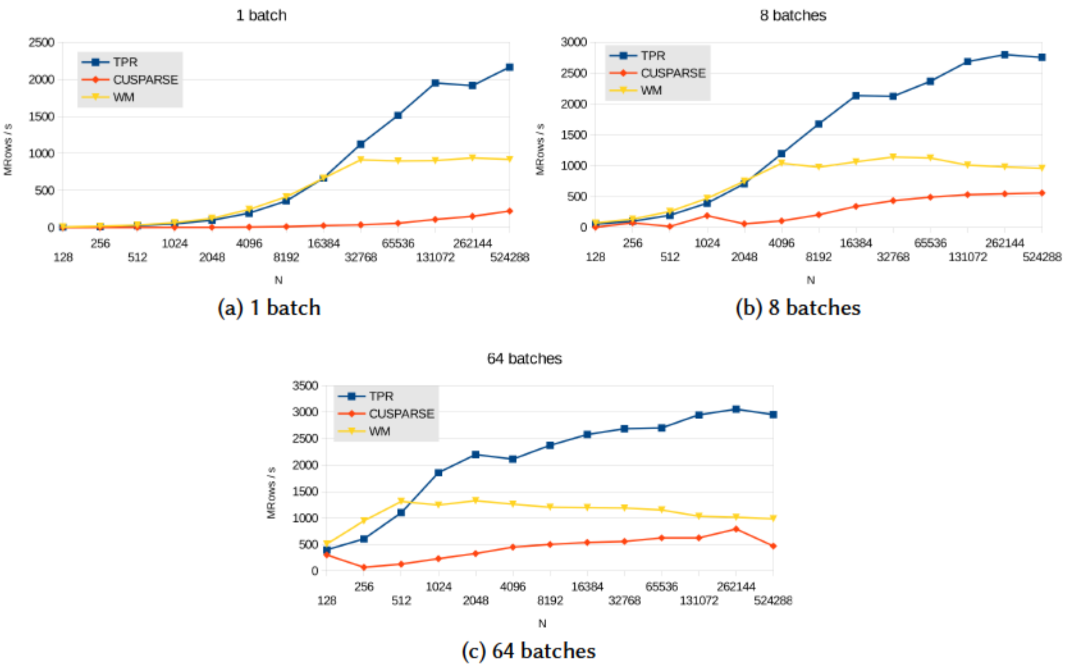
Table 5. Description of the test platform

5.1 Performance Analysis

In this subsection, a study of the performance achieved with the CUDA implementation of the TPR method is analyzed and compared with other solvers. Specifically, our proposal is compared with respect to the *CUSPARSE* library and our previous Wang&Mou BPLG approach presented in [7]. Here, we should stress that the performance results are measured in million rows computed per second, MROWS/s, using a diagonally dominant system which ensures numerical stability (Toeplitz matrix with row [-1 2 -1]). The number of batch problems being simultaneously solved in parallel, G, is studied for $G = 1$, $G = 8$ and $G = 64$, whereas the problem size range goes from $N = 128$ to $N = 524288$. Thus, the MROWS/s value is performed using the expression $N \cdot G \cdot 10^{-6}/t$.

Figure 12 shows a global overview and a comparison with respect to CUSPARSE and our previous WM BPLG implementation. In the case of a single problem being solved ($G = 1$), Figure 12 (a), the TPR method outperforms up to 30.16x the CUSPARSE library for all problem sizes, being 22.03x times faster on average. Regarding WM, TPR surpasses WM for $N \geq 32768$ values, being 1.22x faster on average, although this speedup is higher considering large problem sizes, being up to 2.35x in the case of $N = 524288$. As explained in Section 2, this Wang and Mou implementation has to store 3 equations per element for solving large problem sizes, saturating global memory bandwidth when there are many elements. Although TPR has more computing steps and invokes three kernels instead of two, it performs a better access to global memory and reduces the use of shared memory, being especially notable when solving large problem sizes. Figure 12 (b) depicts the same comparison but solving 8 problems simultaneously ($G = 8$). The TPR method again surpasses

CUSPARSE by up to 13.28x for all cases, being 7.04x faster on average. With respect to WM, as the number of batches has been increased, there are more elements being processed, thus there are more global memory transactions in the execution, saturating the global memory bandwidth for a smaller problem size. In this case, TPR outperforms WM from $N \geq 4096$, being 1.64x times faster on average and up to 2.88x in the best case. In contrast to the previous case, performance stops increasing at $N = 524288$. This lack of scalability can be explained by two factors: firstly, as Table 3 shows for this problem size, L increases and the achieved GPU occupancies drop and, secondly, the global memory bandwidth is saturated. Finally, Figure 12 (c) depicts the case of $G = 64$ batch problems. In this case, our approach is up to 5.53x faster than CUSPARSE on average, and up to 8.95x in the best case. With respect to WM, our solver achieves 1.9x on average, and up to 3x in the case of $N = 524288$.

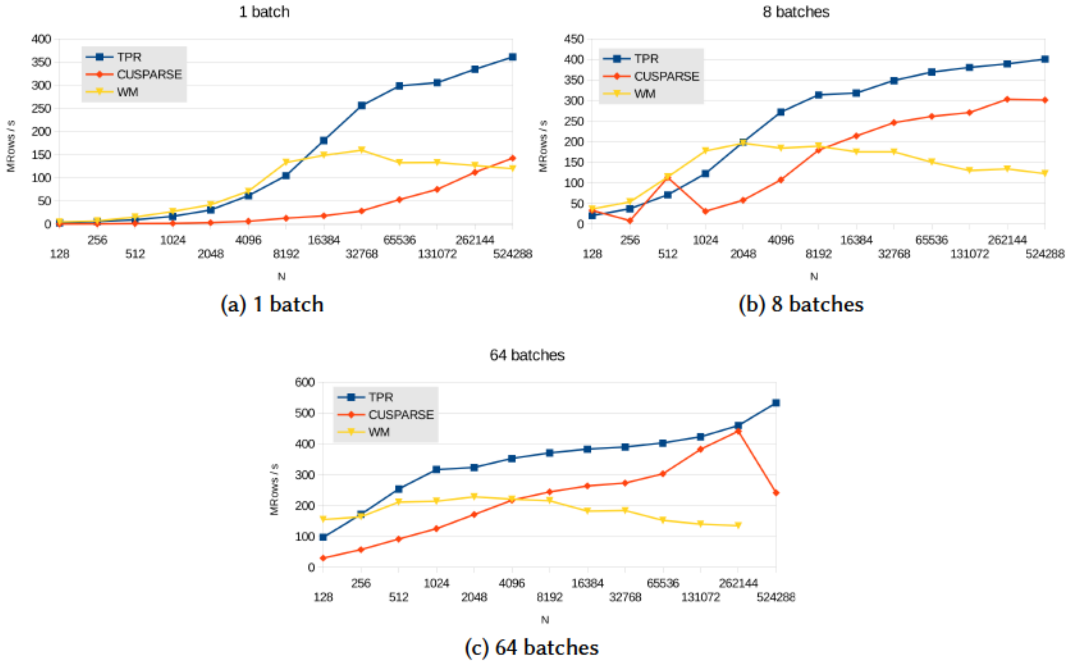


(a) 1 batch (b) 8 batches

(c) 64 batches

Fig. 12. Overall FP32 performance comparison of the TPR method

Additionally, Figure 13 depicts the same performance analysis in the case of double precision. It is easy to see how performance drops in comparison to FP32, this is due to the fact that this Maxwell platform has 128 FP32 CUDA cores but just 4 FP64 ALUS per SM, as well as the memory consumption is doubled. In the case of a single batch, our approach is 7.48x faster than CUSPARSE on average, and up to 10.44x in the case of $N = 1024$. With respect to WM, 1.38x on average and up to 3.02x. When solving $G = 8$ batches, 1.98x on average compared with CUSPARSE, and up to 4.42x in the best case. Using WM, the improvement is 1.7x on average, being up to 3.27x faster for $N = 524288$. And finally, in the case of $G = 64$, our approach is, on average, 1.93x faster than CUSPARSE and up to 3.41x than WM. It should be pointed out that the huge memory consumption of WM does not allow us to execute $N = 524288$ in the target architecture.



(a) 1 batch

(b) 8 batches

(c) 64 batches

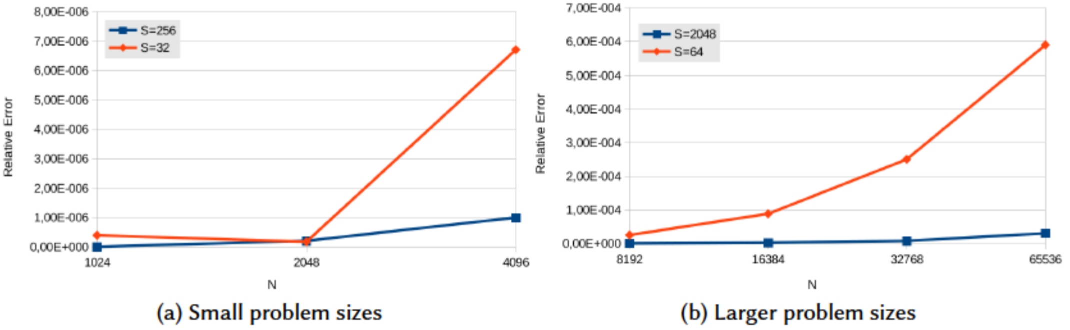
Fig. 13. Overall FP64 performance comparison of the TPR method

5.2 Numerical-Stability Analysis

Depending on the application in which the solver is being executed, the numerical stability may be essential or may have a minor role. It is impossible to provide a general solver suitable for all the applications, some of them require solving one problem, and others need to solve several problems simultaneously. The same happens with the numerical stability and the execution time. Our proposal allows the user to choose the rate performance / stability to be employed, depending on the target application, as well as the number of problems to be solved.

Figure 14 shows an analysis about the influence of S in the numerical stability of the algorithm. The chosen slice size, S, determines the numerical stability in the TPR method. Larger slice sizes allow more equations to participate in the sliced forward reduction (Stage 1), increasing the numerical stability. On the other hand, small slice sizes limit the sliced forward reduction to a reduced number of equations (see Figure 14 (a)); therefore, less information from other . It should be noted that the unacuraccy is more evident in larger problem sizes, since there are more computing steps, and the choice of the size of S is crucial for the stability, as Figure 14 (b) shows.

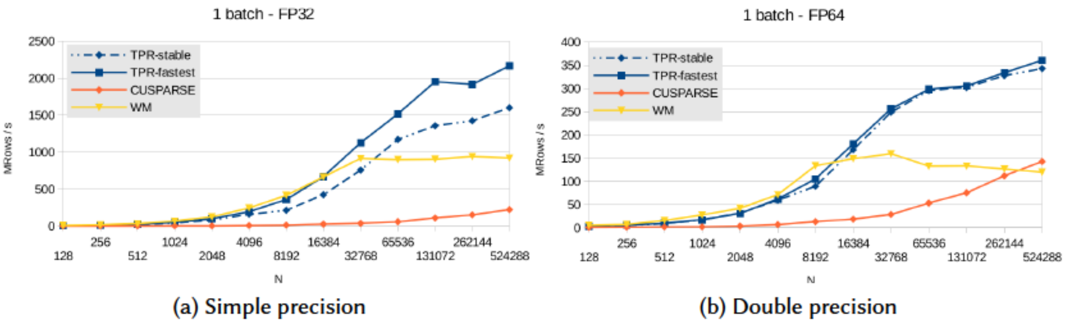
In the previous performance analysis, the given results are based on the performance configuration which achieves the best execution times. However, if a strong numerical stability is required, the said configuration can be chosen to achieve the maximum numerical stability possible (basically by increasing the slice size). Figure 15 shows a performance comparison of two configurations for our proposal in the case of $G = 1$: the one which minimizes the execution time (*TPR-fastest* in graphics) with respect to the one which maximizes the numerical stability (*TPR-stable* in graphics). Specifically, the results presented under the *TPR-stable* approach were taken using $S = 2048$. On average, the *TPR-fastest* approach obtains a speedup of 1.33x with respect to the *TPR-stable* approach. The chosen S in the *TPR-stable* approach is the most suitable for each N, following the tuning



(a) Small problem sizes

(b) Larger problem sizes

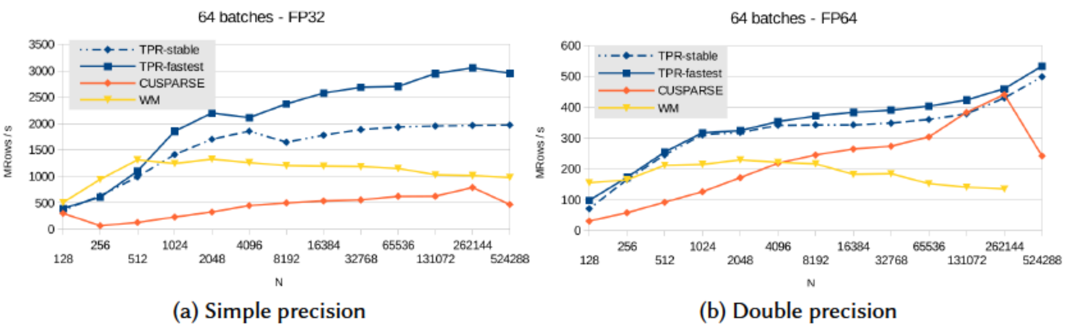
Fig. 14. Analysis of numerical stability for different S and N values



(a) Simple precision

(b) Double precision

Fig. 15. Performance comparison of two different TPR configurations: performance vs numerical stability, executing 1 batch



(a) Simple precision

(b) Double precision

Fig. 16. Performance comparison of two different TPR configurations: performance vs numerical stability, executing 64 batches

strategy of Section 4.4. The same analysis is performed in Figure 16 for $G = 64$, where *TPR-fastest* is 1.32x faster than *TPR-stable*. Table 6 shows the relative error of the previous configurations for the FP32 execution, when executing the Toeplitz matrix described in the introduction of this section

and whose unknowns have the value 1.0 as solution. As can be observed in table, *TPR-fastest* results are acceptable in most cases.

N	TPR-stable	TPR-fastest	Thomas Sequential
128	5.70E-007	1.40E-006	2.10E-006
256	0.00E+000	2.20E-006	5.70E-006
512	8.40E-007	2.40E-005	4.40E-005
1024	0.00E+000	3.90E-007	2.30E-004
2048	2.00E-007	1.70E-007	9.30E-004
4096	9.90E-007	6.70E-006	1.10E-002
8192	4.00E-007	2.50E-005	3.10E-001
16384	2.00E-006	8.80E-005	1.2E+000
32768	7.40E-006	2.50E-004	2.3E+000
65536	3.00E-005	5.90E-004	3.7E+000
131072	1.20E-004	3.20E-002	5.4E+000
262144	4.80E-004	3.20E-003	7.9E+000
524288	1.90E-003	5.50E-001	1.1E+001

Table 6. Relative error of the two FP32-TPR configurations for a Toeplitz matrix

Table 7 and 8 show a numerical-stability analysis for the different 16 input matrices of size 512 proposed in [13], whose description is shown in Table 9, in simple and double precision (using the *TPR-stable* configuration). This analysis compares the achieved stability with respect to other solvers accuracy, using the Thomas algorithm as a baseline, following the experiment of [13]. In the case of $N = 512$, TPR is quite stable for the studied matrices. The relative error for a solution $\hat{x}$ is calculated from the following equation, where x is the solution of the baseline solver:

$$\frac{\|\hat{x} - x\|_2}{\|x\|_2}$$

Matrix	TPR	WM	CUSPARSE
1	**4.20E-006**	**8.80E-006**	**3.80E-006**
2	**2.60E-009**	**2.90E-009**	**8.60E-010**
3	**8.70E-008**	**8.20E-008**	**4.40E-008**
4	**2.80E-006**	**5.30E-006**	**2.90E-006**
5	**NAN**	**NAN**	**1.10E-006**
6	**1.60E-007**	**1.60E-007**	**4.00E-008**
7	**1.00E-007**	**9.20E-008**	**1.80E-007**
8	**7.90E-007**	**1.50E-006**	**1.80E-007**
9	4.70E+005	4.70E+005	4.70E+005
10	4.40E+013	4.40E+013	4.40E+013
11	6.10E+000	**2.00E-005**	**3.90E-006**
12	**4.90E-007**	**4.70E-007**	**3.80E-007**
13	9.10E-001	1.10E+001	1.10E+001
14	**NAN**	**NAN**	**NAN**
15	**NAN**	**NAN**	**NAN**
16	**NAN**	**NAN**	**NAN**

Table 7. Relative errors for FP32

Matrix	TPR	WM	CUSPARSE
1	**1.30E-014**	**7.70E-015**	**5.20E-015**
2	**1.20E-016**	**1.20E-016**	**3.90E-017**
3	**1.60E-016**	**2.40E-016**	**8.20E-017**
4	**1.00E-014**	**2.00E-014**	**6.00E-015**
5	**NAN**	**NAN**	**1.20E-015**
6	**1.30E-016**	**1.30E-016**	**9.50E-017**
7	**3.60E-016**	**2.10E-016**	**3.40E-016**
8	**4.40E-015**	**5.50E-015**	**4.30E-015**
9	4.70E+005	4.70E+005	4.70E+005
10	4.40E+013	4.40E+013	4.40E+013
11	6.00E+000	**1.60E-015**	**7.30E-015**
12	7.60E-010	**3.90E-016**	**7.80E-016**
13	8.50E-001	**1.20E-009**	7.40E-001
14	1.00E+000	**5.40E-014**	**8.50E-015**
15	**NAN**	**NAN**	**NAN**
16	**NAN**	**NAN**	**NAN**

Table 8. Relative errors for FP64

It should be noted that these matrices were chosen to test the robustness of solvers, thus the accuracy of valid solutions varies greatly. It is not possible to compare directly these results with the ones obtained in [13], since (i) the vector d has been randomly generated, (ii) the baseline solver is a sequential version of Thomas instead of a Matlab solver, (iii) the CUDA SDK and drivers are different, and (iv) the relative error formula is slightly different. Also, the results that can be

Matrix Type	Condition number	Description
1	4.41E+04	Each matrix entry randomly generated from a uniform distribution on [-1,1] (denoted as U(-1,1))
2	1.00E+00	A Toeplitz matrix, main diagonal is 1e8, off-diagonal elements are from U(-1,1)
3	3.52E+02	gallery('lesp',512) in Matlab: eigenvalues which are real and smoothly distributed in the inversal approximately [-2*512-3.5,-4.5]
4	2.75E+03	Each matrix entry from U(-1,1), the 256th lower diagonal element is multiplied by 1e-50
5	1.24E+04	Each main diagonal element from U(-1,1), each off-diagonal entry chosen with 50% probability either 0 or from U(-1,1)
6	1.03E+00	A Toeplitz marix, 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
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	2.20E14	Each matrix entry from U(-1,1), the lower diagonal elements 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 is 1e-8, off-diagonal element are from U(-1,1)
15	6.02E+24	gallery('clement',512,0) in Matlab: All main diagonal elements are 0; eigenvalues include plus and minus 511, 509, ... ,1
16	7.1E+191	A Toeplitz matrix, main diagonal is 0, off-diagonal elements are from U=(-1,1)

Table 9. Matrix types used in the numerical evaluation from [13]

considered correct are marked in bold in tables; thus, we can conclude that in most cases, our proposal produces stable results, similar to the ones achieved by CUSPARSE.

6 CONCLUSIONS

We have developed a new tridiagonal system solver, called *Tree Partitioning Reduction* method (TPR). This algorithm is especially well-suitable for parallel and distributed computing systems, since it allows to partition the problem among several processing cores, reducing the communication latency in comparison to other parallel solvers and allowing to solve large problem-sizes efficiently. Additionally, this work also presents an efficient GPU CUDA implementation of the TPR method. This implementation is especially designed to solve several batches simultaneously, i.e. solving several equation systems at the same time, as demanded by many scientific applications. This GPU implementation surpasses other well-known libraries, such as CUSPARSE (being, on average, 22.03x times faster) for both single and double precision executions. A numerical stability study is also provided for simple and double floating point precision, obtaining stable results in most cases, similar to other libraries.

ACKNOWLEDGMENTS

This work was cofunded by the Government of Galicia and ERDF funds from the EU, under the Consolidation Programme of Competitive Reference Groups [ED431C 2017/04]; by the Ministry of Economy and Competitiveness of Spain and ERDF funds [TIN2016-75845-P]; and by the Ministry of Education of Spain (FPU14/02801). Additionally, it has been also supported by the Xunta de Galicia (Centro Singular de Investigación de Galicia accreditation 2016-2019) and ERDF funds [ED7431G/01].

REFERENCES

[1] A. Davidson, Y. Zhang and J.D. Owens. 2011. An Auto-tuned Method for Solving Large Tridiagonal Systems on the GPU. In *Proc. of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS) (2011).* 956–965.

[2] A. Davison and J. D. Owens. 2011. Register Packing for Cyclic Reduction: A Case Study. In *Proc. of the Fourth Workshop on General Purpose Processing on Graphics Processing Units.* 4:1–4:6.

[3] F. Argüello, D.B. Heras, M. Bóo, and J. Lamas-Rodríguez. 2012. The Split-and-Merge Method in General Purpose Computation on GPUs. *Parallel Comput.* 38, 6âĂŞ7 (2012), 277 – 288.

[4] Li-Wen Chang and Wen-mei Hwu. 2014. A Guide for Implementing Tridiagonal Solvers on GPUs. In *Numerical computation with GPUs*, V. Kindratenko (Ed.). Springer, Chapter 2, 29–44.

[5] L.-W. Chang and W.-W. Hwu. 2013. Mapping tridiagonal solvers to linear recurrences. *Technical Report, University of Illinois at Urbana-Champaign* (2013).

[6] A. P. Diéguez, M. Amor, and R. Doallo. 2015. New Tridiagonal Systems Solvers on GPU Architectures. In *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*. 85–94.

[7] A. P. Diéguez, M. Amor, J. Lobeiras, and R. Doallo. 2018. Solving Large Problem Sizes of Index-Digit Algorithms on GPU: FFT and Tridiagonal System Solvers. *IEEE Trans. Comput.* 67, 1 (2018), 86–101.

[8] H.-S. Kim, S. Wu, L.-W. Chang, W.W. Hwu. 2011. A Scalable Tridiagonal Solver for GPU. In *Procd. of Int. Conf. on Parallel Processing (2011)*. 444–453.

[9] R.W. Hockney and C.R. Jesshope. 1988. *Parallel Computers 2: Architecture, Programming and Algorithms.* Taylor & Francis.

[10] R. W. Hockney. 1965. A Fast Direct Solution of Poisson's Equation Using Fourier Analysis. *J. ACM* 12, n.1, 1 (1965), 95–113.

[11] Michael A. Jandron, Anthony A. Ruffa, and James Baglama. 2017. An Asynchronous Direct Solver for Banded Linear Systems. *Numer. Algorithms* 76, 1 (Sept. 2017), 211–235.

[12] D. B. Kirk and W. W. Hwu. 2012. *Programming Massively Parallel Processors: A Hands-on Approach* (2nd ed.). Morgan Kaufmann.

[13] L.-W. Chang, J.A. Stratton, H.-S. Kim, W. W. Hwu. 2012. A Scalable, Numerically Stable, High-performance Tridiagonal Solver Using GPUs. In *Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12) (2012)*. 27:1–27:11.

[14] J. Lobeiras, M. Amor, and R. Doallo. 2015. BPLG: A Tuned Butterfly Processing Library for GPU Architectures. *International Journal of Parallel Programming* 43, n.6 (2015), 1078–1102.

[15] Jacobo Lobeiras, Margarita Amor, and Ramon Doallo. 2016. Designing Efficient Index-Digit Algorithms for CUDA GPU Architectures. *IEEE Transactions on Parallel and Distributed Systems* 27, 5 (2016), 1331–1343.

[16] NVIDIA-Corporation. 2012. CUDA CUSPARSE Library.

[17] NVIDIA-Corporation. 2014. CUDPP: CUDA Data Parallel Primitives Library. (2014). http://cudpp.github.io/

[18] J.L. Larriba Pey. 1995. *Design and Evaluation of Tridiagonal Solvers for Vector and Parallel Computers.* Ph.D. Dissertation. Universitat Politecnica de Catalunya.

[19] A. H. Sameh and D. J. Kuck. 1978. On Stable Parallel Linear System Solvers. *J. ACM* 25, 1 (1978), 81–91.

[20] Harold S. Stone. 1973. An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations. *J. ACM* 20, n.1, 1 (1973), 27–38. https://doi.org/10.1145/321738.321741

[21] L. H. Thomas. 1949. Elliptic Problems in Linear Difference Equations over a Network. *Watson Sci. Comput. Lab. Rep., Columbia University* (1949).

[22] I.E. Venetis, A. Kouris, A. Sobczyk, E. Gallopoulos, and A.H. Sameh. 2015. A direct tridiagonal solver based on Givens rotations for GPU architectures. *Parallel Comput.* 49 (2015), 101 – 116.

[23] X. Wang and Z.G. Mou. 1991. A divide-and-conquer method of solving tridiagonal systems on hypercube massively parallel computers. In *Proc. of the Third IEEE Symposium on Parallel and Distributed Processing (1991)*. 810–817.

[24] Y. Zhang, J. Cohen, J.D. Owens. 2010. Fast Tridiagonal Solvers on the GPU. In *Proc. of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 127–136.

[25] Di Zhao and Jinhang Yu. 2015. Efficiently Solving Tri-diagonal System by Chunked Cyclic Reduction and single-GPU Shared Memory. *J. of Supercomputing* 71, 2 (2015), 369–390.