A Hybrid Parallel Solving Algorithm on GPU for Quasi-Tridiagonal System of Linear Equations

Kenli Li, *Member, IEEE*, Wangdong Yang, and Keqin Li, *Fellow, IEEE*

Abstract—There are some quasi-tridiagonal system of linear equations arising from numerical simulations, and some solving algorithms encounter great challenge on solving quasi-tridiagonal system of linear equations with more than millions of dimensions as the scale of problems increases. We present a solving method which mixes direct and iterative methods, and our method needs less storage space in a computing process. A quasi-tridiagonal matrix is split into a tridiagonal matrix and a sparse matrix using our method and then the tridiagonal equation can be solved by the direct methods in the iteration processes. Because the approximate solutions obtained by the direct methods are closer to the exact solutions, the convergence speed of solving the quasi-tridiagonal system of linear equations can be improved. Furthermore, we present an improved cyclic reduction algorithm using a partition strategy to solve tridiagonal equations on GPU, and the intermediate data in computing are stored in shared memory so as to significantly reduce the latency of memory access. According to our experiments on 10 test cases, the average number of iterations is reduced significantly by using our method compared with Jacobi, GS, GMRES, and BiCG respectively, and close to those of BiCGSTAB, BiCRSTAB, and TFQMR. For parallel mode, the parallel computing efficiency of our method is raised by partition strategy, and the performance using our method is better than those of the commonly used iterative and direct methods because of less amount of calculation in an iteration.

Index Terms—Execution time, GPU, hybrid parallel algorithm, linear equation, quasi-tridiagonal matrix

1 Introduction

1.1 Motivation

The tridiagonal solver is an important core tool in wide range of engineering and scientific applications, such as computer graphics, fluid dynamics, Poisson solvers, cubic spline calculation, and semi-coarsening for multi-grid method [1]. How to solve tridiagonal systems is a very common task in numerical simulations for many science and engineering problems [2]. Some non-zero elements in a matrix may distribute outside the diagonals if the data grid has an irregular boundary, and the coefficient matrices are termed quasi-tridiagonal matrices, which have the characteristic of diagonal dominance. The quasi-tridiagonal matrices are sparse and their majorities of non-zero elements concentrate in the three diagonals.

With the expansion of the scale of problems, the dimensions of the equations are also increased dramatically. Thus, many commonly used algorithms encounter great challenge for solving quasi-tridiagonal system of linear equations with more than millions of dimensions. There are two main approaches to solving the quasi-tridiagonal system of linear equations, which are direct and iterative methods. The direct methods include Gauss elimination, LU, Thomas, and cyclic reduction (CR), etc. Gauss elimination is not well suited to sparse matrices because some zero elements may become non-zero elements in the process of elimination for sparse matrices. Some LU methods can employ some good ordering strategies to keep the L and U factors sparse for the sparse matrices, such as KLU solver [3]; however the ordering strategies may produce a lot of zero-padded which increase density rapidly for some quasi-tridiagonal matrices. If the scale of a sparse matrix is very big, the performance of the direct methods will deteriorate rapidly, because they do not take into account the sparse characteristic of the matrix. Although Thomas and CR have stable performance, they are only suitable for tridiagonal equations. The iterative methods are suitable for large sparse matrices, such as Jacobi, Gauss-Seidel (GS), GMRES, and BiConjugate gradient (BiCG), etc. But these iterative methods do not use the tridiagonal characteristics of quasi-tridiagonal system of linear equations to improve the solving performance. So a new solving method which mixes direct methods and iterative methods should be explored, which can overcome the limitations of the direct and iterative methods.

In recent years, accelerator-based computing using accelerators such as the IBM Cell SPUs, FPGAs, GPUs, and ASICs has achieved clear performance gains compared to CPUs. Among the accelerators, GPUs have occupied a prominent place due to their low cost and high performance-per-watt ratio along with powerful programming models. So a new parallel scheme using GPUs should be explored to accelerate the process of solving the quasi-tridiagonal system of linear equations.

1.2 Our Contributions

For quasi-tridiagonal system of linear equations, we present a solving method which mixes direct and iterative methods. Our method generates a sequence of approximate solutions

- K. Li and W. Yang are with the College of Computer Science and Electronic Engineering, Hunan University, Changsha, Hunan 410082, China, and the National Supercomputing Center in Changsha, Changsha, Hunan 410082, China. E-mail: lkl@hnu.edu.cn, yangwangdong@163.com.
- K. Li is with the College of Computer Science and Electronic Engineering, Hunan University, Changsha, Hunan 410082, China, and the National Supercomputing Center in Changsha, Changsha, Hunan 410082, China, and the Department of Computer Science, State University of New York, New Paltz, New York 12561. E-mail: lik@newpaltz.edu.

Manuscript received 29 Apr. 2015; revised 15 Dec. 2015; accepted 25 Dec. 2015. Date of publication 12 Jan. 2016; date of current version 14 Sept. 2016. Recommended for acceptance by G. Tan.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2016.2516988

1045-9219 © 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

$\{x^k\}$ in the same way as the conventional iteration methods, where x^k expresses the approximate solutions of kth iteration. The conventional iteration methods essentially involve a matrix A only in the context of matrix-vector multiplication and do not make full use of tridiagonal characteristics of quasi-tridiagonal matrices, so as to result in slow convergence. In our method, a quasi-tridiagonal matrix is split into a tridiagonal matrix and a sparse matrix, and then the tridiagonal equation is solved by the direct methods in the iteration processes. Because the approximate solutions obtained by the direct methods are closer to the exact solutions, the convergence speed of solving the quasi-tridiagonal system of linear equations can be improved. Some direct methods have good performance in solving tridiagonal equations, such as Thomas and CR. We present an improved CR algorithm using a partition strategy to solve tridiagonal equations on GPU, and the intermediate data in computing are stored in shared memory, so as to significantly reduce the latency of memory access. So the computational complexity of the hybrid method is not increased and the convergence speed can be accelerated.

According to our experiments on 10 test cases, the performance improvement using our algorithm is very effective and noticeable. The average number of iterations is reduced by 69.23, 15.79, 39.22, and 47.42 percent by using our method compared with Jacobi, GS, GMRES (5), and BiCG of Lis library respectively, and the performance using our method is better than those of the commonly used iterative and direct methods because of less amount of calculation in an iteration and fast convergence speed.

The remainder of the paper is organized as follows. In Section 2, we review related research on solving quasi-tridiagonal system of linear equations. In Section 3, we present an introduction to CUDA. In Section 4, we develop the method of solution for solving quasi-tridiagonal matrices. In Section 5, we describe parallel implementation of our method on GPU. In Section 6, we demonstrate the performance comparison results in our extensive experiments. In Section 7, we conclude the paper.

2 Related Work

One of the best known algorithms for solving tridiagonal systems is the Thomas algorithm [4]. It is a simplified version of Gaussian elimination without pivoting to solve dominant diagonal systems of equations in O(n) steps by performing an LU decomposition. Unfortunately, this algorithm is not well suited to parallel and vector architectures, such as multi-cores and many-cores processers. Ref. [2] analyzed the projection of four known parallel tridiagonal system solvers: cyclic reduction [5], [6], recursive doubling [7], Bondeli's divide and conquer algorithm [8], and Wang's partition method [9]. Cyclic reduction and recursive doubling focus on a line grain of parallelism, where each processor computes only one equation of the system. Ref. [9] developed a partition method with a coarser grain of parallelism. Ref. [8] had also made efforts in this direction including divide and conquer approximations.

GPUs are widely used in parallel numerical computing, such as parallel accelerating for matrix-matrix multiplication [10] and solvers of linear systems [11], [12]. The solution of tridiagonal systems on the GPU is a problem which has been studied in the literatures recently. Ref. [11] presented the first GPU implementation of the alternating direction implicit tridiagonal solver (ADI-TRIDI) based on shading languages. Ref. [13] presented the application of the split-and-merge technique to the following parallel tridiagonal system solvers on GPU: cyclic reduction and recursive doubling, which could efficiently exploit the memory hierarchy of GPU. More recently, a methodology for reducing communication on the GPU for algorithms with a down sweep pattern was applied to cyclic reduction [14]. Ref. [12] presented a new implementation of cyclic reduction for the parallel solution of tridiagonal systems and employed this scheme as a line relaxation smoother in the GPU-based multigrid solver. Some parallelization approaches are based on cyclic reduction in [6], [7], [15]. Recently, Ref. [16] discussed the applicability of these algorithms on modern GPUs. They concluded that cyclic reduction suffers from bank conflicts of shared memory and poor thread utilization in lower stages of the solution process, while parallel cyclic reduction is not asymptotically optimal, and recursive doubling is not optimal and additionally exhibits numerical stability issues.

Currently, Krylov subspace methods are considered to be among the most important iterative techniques available for solving large sparse linear systems. These techniques are based on projection processes, both orthogonal and oblique, onto Krylov subspaces [17]. Many iterative solving algorithms based on Krylov subspace have been proposed for various sparse linear systems, such as Generalized Minimum Residual Method (GMRES), Conjugate Gradient algorithm (GC), biconjugate gradient algorithm (BiCG), biconjugate gradient stabilized algorithm (BiCGSTAB), transpose-free quasi-minimal residual algorithm (TFQMR), and stabilized biconjugate residual method (BiCRSTAB). BiCGSTAB, TFQMR, and BiCRSTAB have better adaptability and stability, so that they are widely used in numerical simulations.

There are three main types of operations for the iterative algorithms, which are sparse matrix-vector multiplication (SpMV), vector inner product, and scalar and vector multiplication. SpMV occupies about half of the total amount of calculation. Many parallel methods of SpMV on GPU or multi-core CPU have been proposed to improve the performance [18], [19], [20], [21]. Furthermore, both efficiency and robustness of iterative techniques can be improved by preconditioning. Ref. [22] investigated the convergence characteristics of effective preconditioners and pursued the performance of some preconditioners such as IC and symmetric Gauss-Seidel (SGS). Ref. [23] presented numerical results with a variable block multilevel incomplete LU factorization preconditioner for solving sparse linear systems. Ref. [24] proposed a parallel conjugate gradient method and a parallel square root method with preconditioners for solving SLAE with block-tridiagonal matrices arising from geoelectric problems on Intel multi-core processors and NVIDIA graphics processors. Furthermore, high-performance heterogeneous computers that employ field programmable gate arrays (FPGAs) as computational elements are known as high-performance reconfigurable computers (HPRCs). Ref. [25] illustrated some of the issues associated with mapping floating-point kernels onto HPRCs, but the performance is not better than that of GPUs.

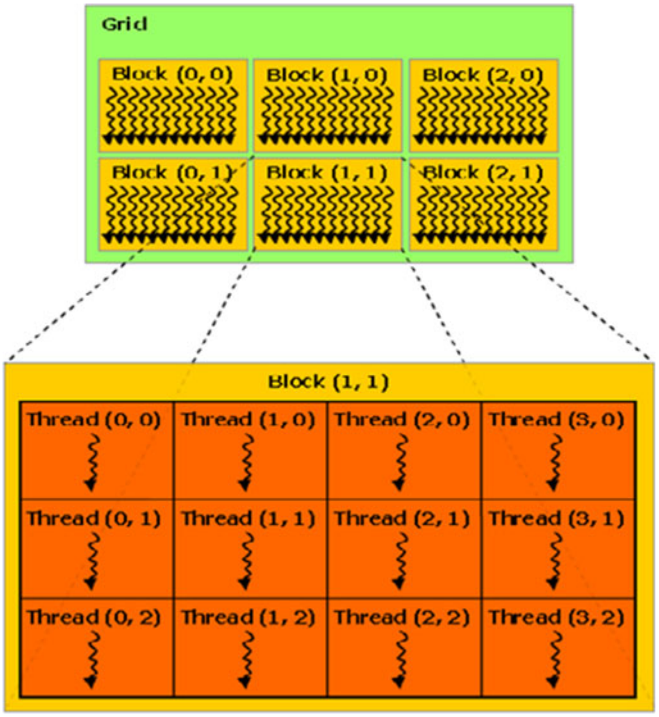


Fig. 1. The CUDA execution model.

Recently, for quasi-diagonal matrices, Ref. [26] presented a new diagonal storage format, which mixes the diagonal format (DLA) and the compressed sparse row format (CSR), and overcomes the inefficiency of DLA in storing irregular matrices and the imbalance of CSR in storing non-zero elements. Furthermore, there is much interest in hybrid solvers with some combination features mixing direct and iterative methods. Typically, they partially factor a matrix using direct and iterative methods on the remaining Schur complement [27], such as HIPS [28], MaPhys [29], and PDSLin [30]. Ref. [31] presented ShyLU, a hybrid-hybrid solver for general sparse linear systems, that is hybrid in two ways. First, it combines direct and iterative methods. Second, the solver uses two-level parallelism via hybrid programming (MPI+threads) in the shared memory environments and on largely parallel computers with distributed memory.

3 An Introduction to CUDA

GPUs are provided by NVIDIA, which provides CUDA (Compute Unified Device Architecture) for improving the efficiency of developing parallel program [32]. The CUDA execution model shown in Fig. 1 includes a collection of threads running in parallel. The number of threads to be executed is decided by programmers. A collection of threads (called a thread block) runs on a multiprocessor at a given time. Multiple blocks can be assigned to a single multiprocessor and their execution is time-shared. For heterogeneous computing systems with CPUs and GPUs, each core of CPU can independently perform its instructions and access its data, which is called the MIMD model. If there are no dependencies, the threads on different cores of CPU do not need to be synchronized. For multi-core CPUs, each core can be scheduled independently to perform threads. But the basic computing unit of a GPU is called streaming multiprocessor (SM). As a component at the bottom of the independent hardware structure, SM can be seen as an SIMD processing unit. Each SM contains some scalar processors (SP) and special function units (SFU). The threads in the same SM need to be synchronized. So the inequality of the load of different threads will have larger impact on performance.

4 The Method of Solution for Quasi-Tridiagonal Matrices

A quasi-tridiagonal matrix A is shown below:

$$A = \begin{pmatrix} d_1 & u_1 & & & * & \\ l_2 & d_2 & u_2 & & & * \\ & l_3 & d_3 & u_3 & & * \\ * & & \ddots & \ddots & \ddots & * \\ * & & & l_{n-1} & d_{n-1} & u_{n-1} \\ & & * & & l_n & d_n \end{pmatrix}.$$

Matrix A has three diagonals, which are $l_2, l_2, \ldots, l_n$, d_1, $d_2, \ldots, d_n$ and $u_1, u_2, \ldots, u_{n-1}$. The $*$ represents a non-zero element outside the diagonals.

4.1 Division of Quasi-Tridiagonal Matrices

Matrix A is divided into two parts, which are the tridiagonal matrix T and another sparse matrix S with non-zero elements outside the three diagonals, as follows:

$$\begin{aligned} A &= T + S \\ &= \begin{pmatrix} b_1 & c_1 & & & & \\ a_2 & b_2 & c_2 & & & \\ & a_3 & b_3 & c_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix} \\ &+ \begin{pmatrix} 0 & 0 & & & * & \\ 0 & 0 & 0 & & & * \\ & 0 & 0 & 0 & & * \\ * & & \ddots & \ddots & \ddots & * \\ * & & & 0 & 0 & 0 \\ & & * & & 0 & 0 \end{pmatrix}. \end{aligned}$$

The three diagonals of the tridiagonal matrix T are stored in three arrays L, D and U, where $L = (l_2, l_3, \ldots, l_n)$, $D = (d_1, d_2, \ldots, d_n)$, and $U = (u_1, u_2, \ldots, u_{n-1})$. The sparse matrix S is stored as a compressed format, such as COO or CSR. The storage space used as dense matrix format is more than the space of T and S, because the non-zero elements outside the diagonals are very sparse.

4.2 Hybrid Iterative Solving Algorithm (HISA)

When A is partitioned into T and S, the quasi-tridiagnoal equation

$$Ax = b \tag{1}$$

is given by $(T + S)x = b$, and we have $Tx = b - Sx$. Let $b' = b - Sx$. Then,

$$Tx = b'. \tag{2}$$

For the tridiagonal equations, there are some solution algorithms with good performance, such as CR and Thomas. So Eq. (2) can be solved quickly by direct methods to get the approximate solutions of Eq. (1) if $Sx = Sj$, where j is the eigenvector of A. Because it is difficult to get the eigenvector of A, j needs to be approached using an iterative method.

Define the iteration as

$$Tx^{i+1} = b - Sx^i,\ x_0 = 0. \tag{3}$$

The iterative solving algorithm is shown as Algorithm 1. The tridiagonal equation $Tx^m = b'$ is solved using a direct method in Algorithm 1. Thomas has better performance if the tridiagonal matrix has diagonal dominance, and the tridiagonal equation can be solved in parallel using CR also. The solution x^m of $Tx^m = b - Sx^{m-1}$ can be used as an approximation to that of $Tx^m = b - Sx^m$ if β approaches 0 using Algorithm 1. So x^m is the solution of Eq. (3) if $\beta = 0$, and x^m is that of Eq. (1) also.

Algorithm 1. The Iterative Solving Algorithm for Quasi-Tridiagonal System of Linear Equations.

```
Require: A tridiagnoal matrix and a sparse matrix partitioned
    from $A$, i.e., $T$ and $S$; the vector $b$.
Ensure: The solution $x^m$.
 1: $x^0 \leftarrow 0$;
 2: $b' \leftarrow b$;
 3: for $m \leftarrow 1, 2, 3, \ldots$, until convergence do
 4:     //The tridiagonal equation is solved by a direct method,
        such as CR or Thomas;
 5:     Solving $Tx^m = b'$;
 6:     $b' \leftarrow b - Sx^m$;
 7:     $\beta \leftarrow \|x^m - x^{m-1}\|$;
 8:     if $\beta = 0$ then
 9:         Stop;
10:     end if
11: end for
```

4.3 Iteration Convergence Analysis

Lemma 1. *Assume $Ax = b$, and $A = T + S$, where A is a nonsingular matrix and T is a nonsingular tridiagonal matrix. The necessary and sufficient condition of convergence for solving $Ax = b$ using Algorithm 1 is $\rho(T^{-1}S) < 1$, where $\rho(T^{-1}S)$ is the spectral radius of $T^{-1}S$.*

Lemma 2. *The solving process of $Ax = b$ using Algorithm 1 is convergent, if A is a diagonally dominant matrix.*

Due to space limitation, the proofs of Lemmas 1 and 2 are provided in the supplementary file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2016.2516988.

4.4 Computational Complexity Analysis

For solving linear equations, the main operation is vector-vector multiplication. For two vectors with n elements, there are n multiplication operations and $n-1$ addition operations respectively. Furthermore, there are multiple adding and multiplying units in a processor, such as CPU and GPU, and addition and multiplication operations can be executed in parallel. So the performance of solving linear equations mainly depends on the number of multiplication operations. Assume that the number of iterations for solving Eq. (1) using Algorithm 1 is m, and Eq. (2) is solved using Thomas. The number of operations belonging to Thomas is $5n-4$. Assume that the number of non-zero elements of A is NNZ, and the number of non-zero elements of S is $NNZ - 3n + 2$. So the number of operations belonging to Sx is $NNZ - 3n + 2$. The sum of operations of solving can approximately be calculated as

$$((5n-4) + (NNZ - 3n + 2)) \times m = (NNZ + 2n + 2) \times m$$
$$= NNZ \times m + 2n \times m + 2m.$$

TABLE 1
The Number of Operations Using the Iterative Algorithms in an Iteration Process

HISA	Jacobi	GS	GMRES
$NNZ + 2n$	$NNZ + 2n$	$NNZ + 2n$	$NNZ + (2j+1)n$
BiCG	BiCGSTAB	TFQMR	BiCRSTAB
$2NNZ + 8n$	$2NNZ + 11n$	$2NNZ + 8n$	$3NNZ + 11n$

Assume that the number of iterations for solving Eq. (1) using Jacobi is m_J. Then, the number of operations of solving approximately is $(NNZ + 2n) \times m_J$, because there are one SpMV and two scalar and vector multiplications in an iteration process. The number of operations using Gauss-Seidel is close to that of Jacobi, but its convergence speed is usually faster than that of Jacobi. The number of operations of HISA is similar to those of Jacobi and Gauss-Seidel and its convergence speed is usually faster than those of Jacobi and Gauss-Seidel.

There are one SpMV, j vector inner products, and $j+1$ scalar and vector multiplications in an iteration process for GMRES algorithm [17], where j is increased as the number of iterations increases. There are two SpMVs, three vector inner products, and five scalar and vector multiplications in an iteration process for BiCG algorithm [17]. There are two SpMVs, five vector inner products, and six scalar and vector multiplications in an iteration process for BiCGSTAB algorithm [17]. There are two SpMVs, two vector inner products, and six scalar and vector multiplications in an iteration process for TFQMR algorithm [17]. There are three SpMVs, five vector inner products, and six scalar and vector multiplications in an iteration process for BiCRSTAB [33]. The number of operations using the iterative algorithms are shown in Table 1.

The NNZ is between $4n$ and $6n$ in most cases, because the non-zero elements outside three diagonals are very sparse for the quasi-tridiagonal matrix, and the mean $5n$ is chosen to calculate. Because m is far less than n, the computational complexity of HISA is approximately $7nm$, and that of Jacobi is approximately $7nm_J$, and that of Gauss-Seidel is approximately $7nm_S$, where m_S is the number of iterations for solving Eq. (1) using Gauss-Seidel. The computational complexities of the iterative methods largely depend on the number of iterations, and the convergence speeds of HISA, BiCG, BiCGSTAB, TFQMR, and BiCRSTAB are usually faster than those of Jacobi and Gauss-Seidel.

The computational complexities of the iterative algorithms are shown in Table 2, where m is the number of iterations.

TABLE 2
The Computational Complexities of the Iterative Algorithms

HISA	Jacobi	GS	GMRES
$7mn$	$7mn$	$7mn$	$(6mn + m^2n)$
BiCG	BiCGSTAB	TFQMR	BiCRSTAB
$18mn$	$21mn$	$18mn$	$26mn$

4.5 Storage Analysis

T is stored in three arrays and S is stored as CSR format. CSR explicitly stores column indices and non-zero values in arrays **Aj** and **Av**, and the starting position of each row in the array **Aj** is stored in the third array **Ap**. So the storage space of S using CSR is $(NNZ - 3n + 2) \times 2 + n + 1$. In addition, three vectors, i.e., x^{i+1}, x^i, and b, need to be stored in the iteration process. So the amount of storage using HISA is

$$\begin{aligned} &3n - 2 + (NNZ - 3n + 2) \times 2 + (n + 1) + 3n \\ &= 2NNZ + n + 3 \approx 2NNZ + n. \end{aligned}$$

For Jacobi, the iteration matrix needs to be stored using CSR format whose storage requirement is $2NNZ + n + 1$. In addition, three vectors x^{i+1}, x^i, and b need to be stored in the iteration process also. So the storage requirement using Jacobi is

$$2NNZ + n + 1 + 3n = 2NNZ + 4n + 1 \approx 2NNZ + 4n.$$

The storage space using Gauss-Seidel is

$$2NNZ + n + 1 + 2n = 2NNZ + 3n + 1 \approx 2NNZ + 3n,$$

because x^{i+1} and x^i can share the same array.

The storage requirements of GMRES, BiCG, BiCGSTAB, TFQMR, and BiCRSTAB are approximately $NNZ + 2mn + m^2 + 1$, $NNZ + 8n$, $NNZ + 7n$, $NNZ + 7n$, and $NNZ + 7n$ respectively [17], [33]. The storage requirements of the iterative algorithms are shown in Table 3 if the sparse matrix is stored as CSR format, where m is the number of iterations. For BiCG and BiCRSTAB, more swap spaces should be occupied, because there is a transposed operation of the iteration matrix in the iteration process.

5 Parallel Implementation Using CR for HISA

Thomas algorithm has fewer operations and better performance for serial computing mode, but it is not suitable for parallel processing. We require a parallel algorithm for quasi-tridiagonal system of linear equations to realize parallel processing. There are some parallel algorithms for solving tridiagonal equations, such as cyclic reduction and recursive doubling, and CR algorithm has good performance for parallel algorithms of tridiagonal equations. But they cannot be adapted to solve quasi-tridiagonal system of linear equations. The main computing process of iterative algorithms is SpMV, and the solving process can be computed in parallel by the parallel algorithms of SpMV. But the tridiagonal characteristics of the

TABLE 3
The Storage Requirements of the Iterative Algorithms

HISA	Jacobi	GS	GMRES
$2NNZ + n$	$2NNZ + 4n$	$2NNZ + 3n$	$2NNZ + 2mn + m^2$
BiCG	BiCGSTAB	TFQMR	BiCRSTAB
$2NNZ + 8n$	$2NNZ + 7n$	$2NNZ + 7n$	$2NNZ + 7n$

tridiagonal matrices have not been fully considered to accelerate the convergence speed for some iterative algorithms. Gauss-Seidel cannot be processed in parallel, because x_{j-1}^{i+1} must be computed synchronously and cannot be obtained in the same iteration process.

5.1 Cyclic Reduction

Cyclic reduction is a divide-and-conquer algorithm, which was proposed in [6]. Cyclic reduction mainly has two phases, which are forward reduction and backward substitution. After a forward reduction, all odd-indexed unknowns are eliminated while even-indexed ones remain. Then, a new tridiagonal equation can be obtained, which has half of the size of the previous equation. The odd-indexed unknowns in the reduced system of equations are then recursively eliminated until an equation with 1 unknowns is obtained. The last equation can be directly solved. For backward substitution, we plug the solved values into the tridiagonal equations from last to first and obtain the solutions of remaining unknowns in these equations.

For a tridiagonal linear system $Tx = b$, these tridiagonal equations can be written as $Eq_i^0 : l_i x_{i-1} + d_i x_i + u_i x_{i+1} = b_i$ $(i = 1, 2, \ldots, n)$, where l_1 and u_n are 0. Assume that a group of equations includes three consecutive equations of the system as follows:

$$\begin{pmatrix} l_{2i-1} & d_{2i-1} & u_{2i-1} & & \\ & l_{2i} & d_{2i} & u_{2i} & \\ & & l_{2i+1} & d_{2i+1} & u_{2i+1} \end{pmatrix} x = \begin{pmatrix} b_{2i-1} \\ b_{2i} \\ b_{2i+1} \end{pmatrix},$$

where $x = (x_{2i-2}, x_{2i-1}, x_{2i}, x_{2i+1}, x_{2i+2})^T$. x_{2i-1} and x_{2i+1} are eliminated by multiplying equation Eq_{2i-1}^0 by $-l_{2i}/d_{2i-1}$, equation Eq_{2i+1}^0 by $-u_{2i}/d_{2i+1}$, and then adding them to equation Eq_{2i}^0. After that, a new equation Eq_i^1 is obtained:

$$(l_i^1, 0, d_i^1, 0, u_i^1)x = b_i^1. \tag{4}$$

But there are only two equations as shown in the following in the last group of equations if n is an even number:

$$\begin{pmatrix} l_{n-1} & d_{n-1} & u_{n-1} \\ & l_n & d_n \end{pmatrix} x = \begin{pmatrix} b_{n-1} \\ b_n \end{pmatrix}.$$

After the elimination, the last equation $Eq_{\lfloor n/2 \rfloor}^1$ can be obtained as $(l_{\lfloor (n-2)/2 \rfloor}^1, 0, d_{\lfloor n/2 \rfloor}^1, 0)x = b_{\lfloor n/2 \rfloor}^1$. After all odd-indexed unknowns x_{2i-1} are eliminated for each i from 1 to $\lfloor n/2 \rfloor$, a new tridiagonal system of linear equations can be formed as follows:

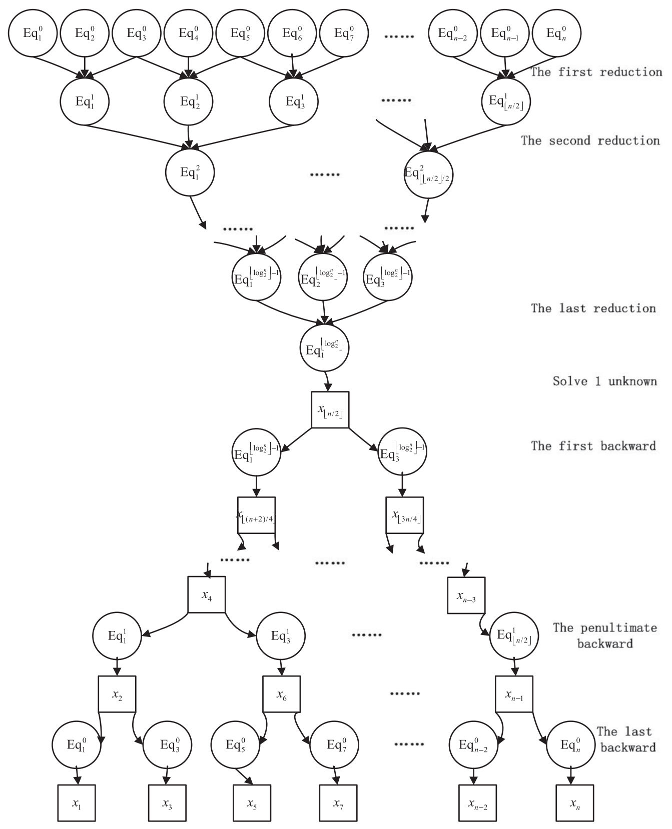


Fig. 2. The solving process of the cyclic reduction for the tridiagonal system containing equations labeled Eq_1^0 to Eq_n^0. The segments in the boxes represent equations with unknowns, and the segments in the circles represent equations with the resolved variables.

$$\begin{pmatrix} d_1^1 & u_1^1 & & & & \\ l_2^1 & d_2^1 & u_2^1 & & & \\ & l_3^1 & d_3^1 & u_3^1 & & \\ & & \ddots & \ddots & \ddots & \\ & & l^1_{\lfloor\frac{n}{2}\rfloor-1} & d^1_{\lfloor\frac{n}{2}\rfloor-1} & u^1_{\lfloor\frac{n}{2}\rfloor-1} & \\ & & & l^1_{\lfloor\frac{n}{2}\rfloor} & d^1_{\lfloor\frac{n}{2}\rfloor} & \end{pmatrix} \begin{pmatrix} x_2 \\ x_4 \\ x_6 \\ \vdots \\ x_{2\lfloor\frac{n}{2}\rfloor-2} \\ x_{2\lfloor\frac{n}{2}\rfloor} \end{pmatrix}$$
$$= \begin{pmatrix} b_1^1 \\ b_2^1 \\ b_3^1 \\ \vdots \\ b^1_{\lfloor\frac{n}{2}\rfloor-1} \\ b^1_{\lfloor\frac{n}{2}\rfloor} \end{pmatrix}.$$

If there are K equations before the ith reduction, the number of equations is $\lfloor\frac{K}{2}\rfloor$ after the K equations are reduced. There is only one equation when the reduction process is completed. So the total number N_{eq} of equations in the process of reduction can be calculated by Eq. (5):

$$N_{eq} = n + \left\lfloor\frac{n}{2}\right\rfloor + \left\lfloor\frac{\lfloor\frac{n}{2}\rfloor}{2}\right\rfloor + \cdots + 1 \approx 2(n-1). \tag{5}$$

N_{eq} is about $2(n-1)$ according to geometric series summation. For the sake of simplicity, we diagrammatize the solving process of this tridiagonal system. Fig. 2 shows the solving pattern of the algorithm. The figure can be partitioned into two portions. The upper half portion of the figure is forward reduction, and the lower half portion is backward substitution.

There are two steps in CR, which are the forward reduction and backward substitution. The forward reduction is processed only once in the iteration processes if the quasi-tridiagonal system of linear equations are solved by the iterative algorithms, and the vector b must be recalculated in each iteration process. So CR can be split into three steps:

1) (Step 1) Coefficient reduction of consecutive equations.
2) (Step 2) Calculation of the vector b in reduction.
3) (Step 3) Backward substitution.

For Step 1, there are 4 multiplications when three consecutive equations are reduced into an equation. The number of operations of Step 1 can be calculated as

$$4\left(\left\lfloor\frac{n}{2}\right\rfloor + \left\lfloor\frac{\lfloor\frac{n}{2}\rfloor}{2}\right\rfloor + \cdots + 1\right) = 4(N_{eq} - n) \approx 4n.$$

In Step 2, there are 2 multiplications when three consecutive equations are reduced into an equation. The number of operations of Step 2 is $2(N_{eq} - n) \approx 2n$. In Step 3, there are 2 multiplications and 1 division in the process of a backward substitution. The number of operations of Step 3 is about $3n$. On a parallel computer with $n/2$ processors, the cyclic reduction algorithm only requires $9\lfloor\log_2 n\rfloor$ steps for the solving process using CR. Algorithm 2 is an iterative algorithm using CR for quasi-tridiagonal system of linear equations. Step 1 is processed only once for the whole solving process, and Steps 2 and 3 must be processed in each iteration process.

Algorithm 2. The Iterative Solving Algorithm for Quasi-Tridiagonal System of Linear Equations Using CR.

```
Require: A tridiagnoal matrix and a sparse matrix partitioned
    from $A$, i.e., $T$ and $S$; the vector $b$.
Ensure: The solution $x^m$.
 1: $x^0 \leftarrow 0$;
 2: $b' \leftarrow b$;
 3: $Eqs \leftarrow$ coefficient_reduction($T$);
 4: for $m \leftarrow 1, 2, 3, \ldots,$ until convergence do
 5:     $bs \leftarrow$ calculated_reduction_b($Eqs$,$b'$);
 6:     $x^m \leftarrow$ backward_substitution($Eqs$,$bs$);
 7:     $b' \leftarrow b - Sx^m$;
 8:     $\beta \leftarrow \|x^m - x^{m-1}\|$;
 9:     if $\beta = 0$ then
10:         Stop;
11:     end if
12: end for
```

Assume that the number of iterations using Algorithm 2 is m. The total number of operations using Algorithm 2 can be calculated as

$$\begin{aligned} &4n + m(2n + 3n + (NNZ - 3n + 2)) \\ &= NNZ \times m + 2n \times m + 4n + 2m. \end{aligned}$$

The coefficients and vector b of all equations in the process of reduction are stored in Eqs and bs respectively.

The number of elements in Eqs is N_{eq}, which can be calculated by Eq. (5).

Define $Eqs = \{e_1^0,\ e_2^0, \ldots,\ e_n^0,\ e_1^1,\ e_2^1, \ldots,\ e_{\lfloor n/2 \rfloor}^1,\ e_1^2, \ldots,\ e_{\lfloor (\lfloor n/2 \rfloor)/2 \rfloor}^2, \ldots,\ e_1^{\lfloor \log_2 n \rfloor}\}$, and $bs = \{b_1^0, b_2^0, \ldots,\ b_n^0,\ b_1^1,\ b_2^1, \ldots,\ b_{\lfloor n/2 \rfloor}^1, b_1^2, \ldots, b_{\lfloor (\lfloor n/2 \rfloor)/2 \rfloor}^2, \ldots,\ b_1^{\lfloor \log_2 n \rfloor}\}$. The elements $e_1^0, e_2^0, \ldots, e_n^0$ are (l_1, d_1, u_1), $(l_2, d_2, u_2), \ldots, (l_n, d_n, u_n)$ in T and $b_1^0, b_2^0, \ldots, b_n^0$ are $b_1, b_2, \ldots, b_n$ in b of Eq. (1). The elements e_i^1 in Eqs is a triple (l_i^1, d_i^1, u_i^1), where l_i^1, d_i^1, u_i^1 are the coefficients of $x_{2i-2}, x_{2i}, x_{2i+2}$ in Eq. (4), and b_i^1 of bs is b_i^1 in Eq. (4). For double precision, the storage spaces S_{eq} of Eqs and bs can be calculated by Eq. (6):

$$S_{eq} = N_{eq} \times (3 \times 8 + 8) = 32N_{eq} \approx 64n(bytes). \tag{6}$$

Each group of equations is reduced by a thread of CUDA if CR is processed on GPU. All threads need to access Eqs and bs frequently, which cannot be accessed in alignment, because Eqs and bs are irregular data set. If Eqs and bs are stored in the global memory, the performance of CR will decline because of the access latency of global memory. The tridiagonal is partitioned into some blocks and each block is processed by a thread block of CUDA. Then Eqs and bs of a block can be stored in the shared memory of the thread block, and it has little impact on the performance of CR, because the shared memory has very little access latency.

5.2 HISA Based on Blocks (BHISA)

The quasi-tridiagonal matrix A is shown as follows:

$$A = \begin{pmatrix} b_1 & c_1 & & * & & & & \cdots \\ a_2 & b_2 & c_2 & & & & & \cdots \\ * & \ddots & \ddots & \ddots & * & & & \cdots \\ & & a_k & b_k & c_k & * & & \cdots \\ & & * & a_{k+1} & b_{k+1} & c_{k+1} & & \cdots \\ & * & & \ddots & \ddots & \ddots & & \cdots \\ & & * & a_{(q-1)k} & b_{(q-1)k} & c_{(q-1)k} & & \cdots \\ & * & & \cdots & a_{(q-1)k+1} & b_{(q-1)k+1} & c_{(q-1)k+1} & \cdots \\ * & & & \cdots & \ddots & \ddots & \ddots & \cdots \\ & & & \cdots & * & & a_{qk} & b_{qk} \end{pmatrix}.$$

The tridiagonal in A can be partitioned into q blocks and each block has k rows. So the quasi-tridiagonal matrix A is partitioned into q tridiagonal submatrices and a matrix without tridiagonal. Define T_i to be the ith tridiagonal submatrix for $i = 1, 2, \ldots, q$. For $i = 1, \ldots, q-1$, T_i is

$$T_i = \begin{pmatrix} b_{(i-1)k+1} & c_{(i-1)k+1} & & \\ a_{(i-1)k+2} & b_{(i-1)k+2} & c_{(i-1)k+2} & \\ & \ddots & \ddots & \ddots \\ & a_{(i-1)k+k-1} & b_{(i-1)k+k-1} & c_{(i-1)k+k-1} \\ & & a_{ik} & b_{ik} \end{pmatrix}.$$

T_q is

$$T_q = \begin{pmatrix} b_{(q-1)k+1} & c_{(q-1)k+1} & & \\ a_{(q-1)k+2} & b_{(q-1)k+2} & c_{(q-1)k+2} & \\ & \ddots & \ddots & \ddots \\ & a_{(q-1)k+k-1} & b_{(q-1)k+k-1} & c_{(q-1)k+k-1} \\ & & a_{qk} & b_{qk} \end{pmatrix}.$$

Define S to be a submatrix of A without tridiagonal, which is shown as follows:

$$S = \begin{pmatrix} 0 & 0 & & * & & & & \cdots \\ 0 & 0 & 0 & & & & & \cdots \\ * & \ddots & \ddots & \ddots & * & & & \cdots \\ & & 0 & 0 & c_k & * & & \cdots \\ & & * & a_{k+1} & 0 & 0 & & \cdots \\ & * & & \ddots & \ddots & \ddots & & \cdots \\ & & * & 0 & 0 & c_{(q-1)k} & & \cdots \\ & * & & \cdots & a_{(q-1)k+1} & 0 & 0 & \cdots \\ * & & & \cdots & \ddots & \ddots & \ddots & \cdots \\ & & & \cdots & * & & 0 & 0 \end{pmatrix}.$$

And A can be expressed as follows:

$$A = \begin{pmatrix} T_1 & & & & \\ & T_2 & & & \\ & & \ddots & & \\ & & & T_{q-1} & \\ & & & & T_q \end{pmatrix} + S.$$

Define x_j and b_j for $j = 1, 2, \ldots, q$ to be the sub-vectors of x and b respectively. x_j includes the $((j-1) \times k)$th to $(j \times k)$th elements of x and b_j includes the $((j-1) \times k)$th to $(j \times k)$th elements of b. So the iteration equation for the jth block is

$$T_j x_j^{i+1} = b_j - Sx^i,\ x_j^0 = 0,\ j = 1, 2, \ldots, q. \tag{7}$$

Eq. (7) is an independent computing task for each j, and can be executed in parallel. The iterative solving algorithm using CR based on blocks for the quasi-tridiagonal equation is expressed as Algorithm 3, which can be processed in parallel.

In lines 7 to 9 of Algorithm 3, each T_j, $j = 1, 2, \ldots, q$, is assigned into a block of thread grid in CUDA and each block is solved in parallel in the thread block using CR algorithm. Each thread solves an odd-indexed equation in Eqs of T_j to get the solution of the unknowns x by backward substitution. The solving process of CR on GPU is shown in Fig. 3, where Eq_i^j expresses the equation $e_i^j x = b_i^j$, and e_i^j and b_i^j are stored in Eqs and bs respectively.

5.3 Optimization of Parallel Algorithm on GPU

The Eqs and bs of a block can be stored in the shared memory of the corresponding thread block, because the threads of the block do not access the Eqs and bs in the other blocks. x_j should be accessed frequently in lines 9 and 10 of Algorithm 3, and x_j is not accessed by the threads of the other blocks, so x_j of the block can be stored in the shared memory also. Lines 13 to 15 are implemented by sum reduction on GPU. The main operations of line 20 are sparse matrix

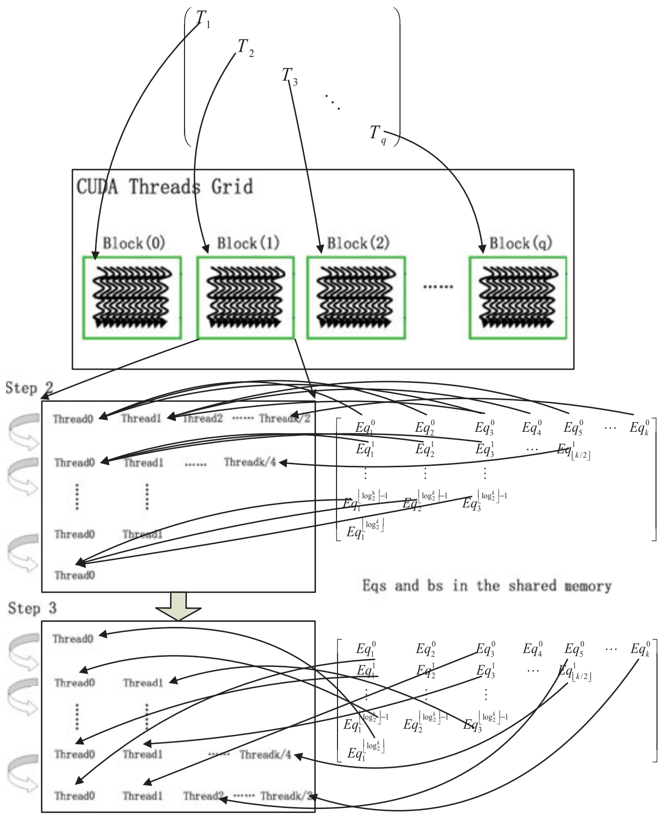


Fig. 3. The solving process of the cyclic reduction on GPU. Each thread of the block processes three consecutive equations using a new x to get a new b according to Eq. (4) in each reduction of Step 2. Then an unknown is solved by a thread of the block according to backward substitution in Step 3.

vector multiplication, which can be implemented by SpMV function on GPU.

Algorithm 3. The Iterative Solving Algorithm Using CR Based on Partition for Quasi-Tridiagonal System of Linear Equations.

```
Require: The blocks partitioned from $A$, i.e., $T_1, T_2, \ldots, T_q$ and
    $S$; the vector $b$.
Ensure: The solution $x_m$.
 1: $x^0 \leftarrow 0$; $b' \leftarrow b$;
 2: for $j \leftarrow 1$ to $q$ do
 3:     $Eqs_j \leftarrow$ coefficient_reduction($T_j$);
 4: end for
 5: for $i \leftarrow 1, 2, 3, \ldots,$ until convergence do
 6:     for $j \leftarrow 1$ to $q$ do
 7:         $bs_j \leftarrow$ calculated_reduction_b($Eqs_j$,$b'_j$);
 8:         $x_j^i \leftarrow$ backward_substitution($Eqs_j$,$bs_j$);
 9:         $\beta_j \leftarrow \|x_j^i - x_j^{i-1}\|$;
10:     end for
11:     $\beta \leftarrow 0$;
12:     for $j \leftarrow 1$ to $q$ do
13:         $\beta \leftarrow \beta + \beta_j$;
14:     end for
15:     if $\beta = 0$ then
16:         $m \leftarrow i$; Stop;
17:     end if
18:     $b' \leftarrow b - Sx^i$;
19: end for
```

The convergence speed of HISA may decline using partition strategy, because the non-zero elements of S will increase. The bigger the size of blocks is, the less the non-zero elements of S increase, and the less the convergence speed decreases. But the proportion of the increased number of iterations is very little. The bigger the size of blocks is, the more the storage spaces of the Eqs and bs of blocks need. The Eqs and bs cannot be loaded into the shared memory if the sum of the Eqs and bs is beyond the limit of the shared memory of thread block. Furthermore, if the number of threads in the block is too large, more threads will be idle in the process of reduction. Assume that the number of rows in T is n and the size of the block is k. Then a threads block resides at least $\lfloor k/2 \rfloor$ threads, because we can find from Fig. 3 that the number of threads residing in the threads block should be half of the size of T_j. The number of reductions in Step 2 and backwards in Step 3 is $\lfloor \log_2 k \rfloor$. The number of available computing time units in the thread block is $2\lfloor k/2 \rfloor \lfloor \log_2 k \rfloor$, and the number of used computing time units is $2(\lfloor k/2 \rfloor - 1) + k$ according to Eq. (5). So the number of idle time units in Steps 2 and 3 of solving Eq. (7) using CR can be calculated as

$$\frac{n}{k} \times \left(2 \times \left\lfloor \frac{k}{2} \right\rfloor \times \lfloor \log_2 k \rfloor - \left(2 \times \left(\left\lfloor \frac{k}{2} \right\rfloor - 1 \right) + k \right) \right)$$
$$\approx n \left(\log_2 k + \frac{2}{k} \right).$$

With the increase of the size of blocks, we can find that the number of idle time units will increase also, leading to the efficiency decrement in parallel computing.

The threads are loaded into an SM on GPU to be processed in accordance with warp unit. The size of thread block should be multiple of 32, because the number of threads in a warp is 32. The size of T_j is at least 64 because the number of threads in one block is at least 32; otherwise, some threads will be idle.

There are few non-zero elements in the sparse matrix S divided from A. We observe that the number of non-zero elements in some rows of S divided from benchmarks is less than 2. The performance of SpMV of S using CSR or ELL directly is poor, because some threads in the grid have very low load, so as to waste computing resources. The performance analysis method using partition in [19], [20] for SpMV can be used and S can be partitioned into some blocks with similar number of non-zero elements. These blocks can be split into some groups according to the size of thread blocks of CUDA, and these groups processed by the same thread block have similar number of non-zero elements. So the loads of these threads in the same block are more balanced and fuller.

6 Experimental Evaluation

6.1 Experiment Settings

The following test environment has been used for all benchmarks. The test computer is equipped with two AMD Opteron 6,376 CPUs running at 2.30 GHz and a NVIDIA Tesla K20c GPUs. Each CPU has 16 cores. The GPU has 2,496 CUDA processor cores, working at 0.705 GHz clock, and possessing 4 GB global memory with 320 bits

TABLE 4
Parameters of the Test Computer

Parameters	Description	Values
S_i	the size of integer	4 Byte
S_s	the size of single	4 Byte
S_d	the size of double	8 Byte
C	the number of stream processor	2,496
f_s	the clock speed of SP	0.705 GHz
f_a	the clock speed of the global memory	2.6 GHz
AW	the bus width of the global memory	320 bits
TW	the bandwidth of PCIe	8 GiB/s

bandwidth at 2.6 GHz clock, and the CUDA compute capacity is 3.5. As for software, the test machine runs the 64 bit Windows 7 and NVIDIA CUDA toolkit 7.0. The hardware parameters of the testing computer are shown in Table 4. The tested matrices are derived from some scientific computing applications, such as gas discharges. The ten tested matrices shown in Table 5 are the quasi-tridiagonal matrices, which have nonpositive definite, asymmetric, and diagonally dominant characteristics. Further characteristics of these matrices are given in the supplementary file, available online.

For the test using HISA on CPU, the tridiagonal matrix partitioned from the coefficient matrix is solved by Thomas algorithm, because it has better performance for serial algorithms. For the test using BHISA on GPU, the tridiagonal matrix partitioned from the coefficient matrix is solved by our parallel CR algorithm. The solving functions of KLU library [3] are used to solve the test cases for LU algorithm, and AMD reordering technique [34] is used in the solving functions on CPU. But KLU function does not support parallel mode on multi-core CPU. SuiteSparse library provides a sparse direct solver SuiteSparseQR for GPU [35], [36], but the performance of SuiteSparseQR is worse than that of cuSparse provided by CUDA tools [37]. The solving functions of Lis library [38] are used to solve the test cases for iterative algorithms, such as Jacobi, GS, GMRES(m), BiCG, BICGSTAB, BiCRSTAB, and TFQMR, and these iterative algorithms of Lis library are tested using both serial mode and parallel mode. The storage spaces of GMRES are more than those of the other iterative algorithms in an iteration process from Table 2. So Gas10000000a, Gas10000000b, and Gas10000000c cannot be solved using GMRES, because too much storage spaces are occupied, leading to memory overflow. So we tested restarted version GMRES(m) for parallel mode and serial mode, where m is 5. The last three matrices can be solved by GMRES(5). The cases are tested on single core of multi-core CPU for the serial mode and they are tested on all cores of multi-core CPU using openMP for the parallel mode. The cuSparse library provided by CUDA tools includes three sparse direct solvers, which use BSR, CSR, and HYB storage formats respectively [37], and the solver using HYB format has better performance, so the HYB solver is used in our experiments on GPU. Furthermore, two preconditioners are provided by cuSparse, which are the incomplete-Cholesky factorization and the incomplete-LU factorization. The incomplete-Cholesky factorization is sutable for Hermitian/symmetric positive definite sparse matrix. But the tesed cases are nonpositive

TABLE 5
General Information of the Quasi-Tridiagonal Matrices used in the Evaluation

Quasi-tridiagonal Matrices	n	NNZ	Ratios of NNZ over n	NNZ outside tridiagonals
Gas2000	2,000	9,989	4.9945	3,991
Gas10000a	10,000	49997	4.9997	19,999
Gas10000b	10,000	59,989	5.9989	29,991
Gas100000a	100,000	499,991	4.9999	199,993
Gas100000b	100,000	599,976	5.9998	299,978
Gas1000000a	1,000,000	3,999,997	4.0000	999,999
Gas1000000b	1,000,000	4,999,972	5.0000	1,999,974
Gas10000000a	10,000,000	39,999,997	4.0000	9,999,999
Gas10000000b	10,000,000	49,999,707	4.9999	19,999,709
Gas10000000c	10,000,000	59,999,104	5.9999	29,999,106

definite sparse matrices. So the incomplete-LU factorization is used in the HYB solver. Furthermore, CUDA tools also provide cuSolver library, which is a high-level package based on the cuBLAS and cuSPARSE libraries [39]. The cuSolverSP of cuSolver library provides a new set of sparse routines based on a sparse QR factorization. But cuSolverSP cannot solve the test cases except Gas2000 matrix. Some zero entries will become non-zero entries when a sparse matrix is factorized. For QR factorization, non-zero elements will increase sharply because of irregular distributions of non-zero elements. The processing of QR factorization need more storage spaces, exceeding the limits of the memory on GPU for large sparse matrices. So the sparse matrices bigger than Gas2000 cannot be factorized on GPU because of the limits of the memory on GPU. So the cuSolver library is not used in experiments. The cases were solved using the iterative solvers of PARALUTION library for parallel and serial mode on CPUs and GPU [40], which used restart version GMRES(5) and BiCGSTAB methods.

The convergence tolerance is set to 0.0000001 for the iterative algorithms in the test. The performance of the iterative algorithms depends on two factors, which are the convergence speed and the number of operations in each iteration process. The number of operations in each iteration process is analyzed in Section 4.4, and the convergence speeds are tested for the iterative algorithms before the performance analysis. The cases are solved by the tested algorithms using double precision.

6.2 Convergence Test

The number of iterations for solving the test cases is shown in Table 6, where *No* represents that the case cannot be solved using the algorithm and $Percentage$ represents the reduction percentage of the number of iterations using HISA compared with other iterative methods. We have the following observations from Table 6.

(1) The convergence speed of iterative method depends on the degree of diagonally dominant characteristics for the sparse matrix, which is defined $|a_{ii}| > \sum_{j\neq i} a_{ij}$. The bigger $|a_{ii}| - \sum_{j\neq i} |a_{ij}|$ is, the more obvious the diagonally dominant characteristics is. If the number of non-zeros (NNZ) outside tridiagonal is greater, $\sum_{j\neq i} a_{ij}$ will be bigger, leading the

TABLE 6
The Number of Iterations for Solving the Cases using Iterative Algorithms

Quasi-tridiagonal	HISA	BHISA	Lis lib							PARALUTION lib	
Matrices			Jacobi	GS	GMRES(5)	BiCG	BiCGSTAB	TFQMR	BiCRSTAB	GMRES(5)	BiCGSTAB
Gas2000	11	11	31	13	18	19	11	10	11	10	10
Gas10000a	11	12	31	13	18	19	11	11	11	10	10
Gas10000b	12	13	49	13	19	22	11	12	12	11	10
Gas100000a	11	11	31	13	18	20	11	11	11	10	10
Gas100000b	12	13	49	14	20	21	11	11	11	11	11
Gas1000000a	10	11	24	12	17	20	10	10	11	11	10
Gas1000000b	11	11	33	13	19	22	11	11	11	12	11
Gas10000000a	10	11	25	13	18	21	11	11	11	10	10
Gas10000000b	11	11	35	14	20	23	11	11	11	11	11
Gas10000000c	13	13	56	15	23	26	12	12	12	12	12
Percentage(%)		−4.27	−69.23	−15.79	−39.22	−47.42	1.82	0	1.82	3.57	6.25

diagonally dominant characteristics less obvious. So the convergence speed of iterative method may not be reduced when the size of matrix increases, and the convergence speed may be reduced in general when the NNZ outside diagonal increases.

(2) For Lis library, the convergence speeds of HISA are faster than those of Jacobi, Gauss-Seidel, GMRES(5) and BiCG. The convergence speeds of Gauss-Seidel and HISA are faster than that of Jacobi, and the number of iterations using HISA and BHISA is very close to those of BiCGSTAB, BiCRSTAB, and TFQMR, and the deviation of the average number of iterations is 1.82, 0, and 1.82 percent by using HISA compared with BiCGSTAB, BiCRSTAB, and TFQMR. For the 10 test cases, the average number of iterations is reduced by 69.23, 15.79, 39.22, and 47.42 percent by using HISA compared with Jacobi, GS, GMRES (5), and BiCG. The convergence speeds of GMRES (5) and BiCGSTAB of PARALUTION library are faster than those of Lis library and HISA, but the number of iterations using HISA is very close to those of GMRES(5) and BiCGSTAB of PARALUTION library, and the deviation of the average number of iterations is 3.57 and 6.25 percent by using HISA compared with GMRES(5) and BiCGSTAB of PARALUTION library.

(3) The number of iterations using BHSIA is similar to that of HISA, and the average number of iterations of BHISA is increased by 4.27 percent compared with that of HISA for the test cases.

6.3 Performance Evaluation

6.3.1 Test on Serial Mode

We have the following observations from Table 7.

(1) For Gas10000a, Gas10000b, Gas100000a, and Gas100000b, the performance using iterative algorithms is far better than that of KLU, because the padded-zeros are too much, resulting in the performance of KLU deteriorated sharply. Especially Gas1000000a, Gas1000000b, Gas10000000a, Gas10000000b, and Gas10000000c cannot be solved using KLU, because the sizes of the matrices are too big. But for Gas2000, KLU has good performance because of its small data size.

(2) Although the number of iterations using Jacobi is more than those of the other iterative algorithms, the performance of Jacobi is close to those of the other iterative algorithms, because the number of operations in each iteration process is fewer. But the performance of Jacobi will decline rapidly when the size of matrix increases. Furthermore, the process of solving includes pretreatment, computing, and data access. For Jacobi and GS, although the computing complexity of GS is less than that of Jacobi, the time of pretreatment of GS is more than that Jacobi. For small matrices, the pretreatment has more influence

TABLE 7
The Performance of Solving the Cases on Serial Mode (Unit: Second)

Quasi-tridiagonal	HISA	Lis lib							PARALUTION lib		KLU
Matrices		Jacobi	GS	GMRES(5)	BiCG	BiCGSTAB	TFQMR	BiCRSTAB	GMRES(5)	BiCGSTAB	
Gas2000	0.31	1.88	3.51	2.43	3.26	4.85	3.87	3.27	0.34	0.35	1.00
Gas10000a	0.58	1.91	3.61	3.19	3.32	4.49	4.07	3.07	0.55	0.56	98.35
Gas10000b	0.92	2.84	4.53	2.98	3.81	4.90	4.81	3.50	0.66	0.66	182.86
Gas100000a	1.23	2.47	5.75	3.05	4.23	4.91	4.58	3.68	1.38	1.37	1,526.05
Gas100000b	1.56	3.69	6.11	3.78	4.44	5.14	5.05	3.72	1.64	1.62	2,214.83
Gas1000000a	1.69	3.67	7.53	3.54	7.78	7.23	7.00	5.75	1.92	1.74	No
Gas1000000b	1.89	5.48	8.49	4.08	9.59	8.03	8.35	6.21	2.55	2.03	No
Gas10000000a	16.86	24.59	24.04	27.25	50.25	27.92	28.47	27.89	20.53	18.66	No
Gas10000000b	20.01	44.09	31.29	36.82	70.83	35.72	38.94	35.94	30.02	27.78	No
Gas10000000c	25.92	85.87	39.06	48.30	98.43	41.97	65.14	44.10	42.74	34.15	No

TABLE 8
The Performance Improvement of Solving Using HISA on Serial Mode ()

Quasi-tridiagonal	Lis lib							PARALUTION lib		KLU
Matrices	Jacobi	GS	GMRES(5)	BiCG	BiCGSTAB	TFQMR	BiCRSTAB	GMRES(5)	BiCGSTAB	
Gas2000	83.43	91.11	87.16	90.43	93.56	91.94	90.47	8.23	10.85	68.92
Gas10000a	69.79	84.06	81.96	82.63	87.17	85.86	81.23	−4.72	−2.85	99.41
Gas10000b	67.63	79.68	69.08	75.84	81.19	80.85	73.65	−39.54	−39.54	99.50
Gas100000a	50.19	78.61	59.63	70.92	74.96	73.17	66.59	10.87	10.22	99.92
Gas100000b	57.69	74.46	58.73	64.83	69.66	69.09	58.11	4.88	3.70	99.93
Gas1000000a	54.06	77.60	52.29	78.30	76.64	75.87	70.62	14.31	2.99	No
Gas1000000b	65.49	77.74	53.72	80.29	76.47	77.35	69.57	25.88	6.90	No
Gas10000000a	31.46	29.89	38.15	66.46	39.63	40.79	39.57	17.90	9.67	No
Gas10000000b	54.63	36.07	45.67	71.76	43.99	48.62	44.33	33.36	27.99	No
Gas10000000c	69.81	33.64	46.33	73.66	38.23	60.20	41.22	39.35	24.09	No

on the overall performance, but the pretreatment has less influence for big matrices. So the performance of Jacobi is better than that of GS for small matrices, and is poorer than that of GS for big matrices.

The performance improvement precentages by using HISA compared with Jacobi, GS, GMRES(5), BiCG, BiCGSTAB, TFQMR, BiCRSTAB of Lis library, GMRES(5) and BiCGSTAB of PARALUTION library, and KLU in Table 8 are calculate by $(T_i - T_H)/T_i \times 100$ $(i = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)$, where T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9, and T_{10} are the performance of Jacobi, GS, GMRES(5), BiCG, BiCGSTAB, TFQMR, BiCRSTAB of Lis library, GMRES(5) and BiCGSTAB of PARALUTION library, and KLU respectively and T_H is the performance of HISA. It is observed from Table 8 that the average execution time of HISA reduces by 60.42, 66.29, 59.27, 75.51, 68.15, 70.37, 63.54, and 93.54 percent by using HISA compared with Jacobi, GS, GMRES(5), BiCG, BiCGSTAB, TFQMR, BiCRSTAB of Lis library, and KLU respectively. The performance of PARALUTION library is better than that of Lis library for all test cases. Furthermore, the performance of PARALUTION library is better than that of HISA for Gas10000a and Gas10000b, but the performance of PARALUTION library is worse than that of HISA for other matrices. The average performance improvements using HISA are 11.05 and 5.40 percent compared with GMRES(5) and BiCGSTAB of PARALUTION library.

6.3.2 Test on Parallel Mode

The size and number of blocks have influence on the performance of solving using BHISA. For K20c GPU used in the experiments, the size of blocks is at least 64 in order to make full use of computing power of the SMs. The total storage spaces of Eqs, bs, and x_j of each data block will be 4,096 bytes according to Eq. (6) if the size of each block is 64. The limit of available shared memory of each thread block is 49,152 bytes for K20c GPU. So we choose 64 to test all the cases, and each block is 64 × 64 tridiagonal sub-matrix. The block numbers of all the cases are calculated by $n/(blocksize)$, where n is the number of rows in the matrix and $blocksize$ is 64. So the block numbers of Gas2000, Gas10000a, Gas10000b, Gas100000a, Gas100000b, Gas1000000a, Gas1000000b, Gas1000000a, Gas1000000b, and Gas10000000c are 32, 157, 157, 1,563, 1,563, 15,625, 15,625, 156,250, 156,250, and 156,250. The test cases are solved using OpenMP and CUDA for BHISA method on multi-core CPUs and GPUs. Each block is processed on a threads block of CUDA and each block is processed on a thread of openMP because the number of threads of CPU is far less than that of GPU.

The performance of solving the test cases using openMP and CUDA are shown in Tables 9 and 11 respectively, where No represents that the case cannot be solved using the algorithm.

We analyzed the two implementations for BiCG and BiCRSTAB in Lis library and found that they are not thread-safe for parallel mode. There are memory access conflicts for large-scale data operation. So some large matrices cannot be solved by BiCG and BiCRSTAB methods of Lis library for parallel mode. However, there is a thread-safe implementation in Lis library for BiCG algorithm, which is

TABLE 9
The Performance of Solving the Cases on Multi-Core CPUs (Unit: Second)

Quasi-tridiagonal	BHISA	Lis lib							PARALUTION lib	
Matrices		Jacobi	GS	GMRES(5)	BiCGSafe	BiCGSTAB	TFQMR	BiCRSTAB	GMRES(5)	BiCGSTAB
Gas2000	0.025	0.11	0.112	1.463	2.768	1.403	2.93	2.77	0.046	0.047
Gas10000a	0.053	0.11	0.124	2.219	3.889	2.801	2.94	2.60	0.050	0.058
Gas10000b	0.062	0.17	0.142	2.264	3.547	3.074	4.00	2.95	0.055	0.060
Gas100000a	0.121	0.15	0.166	2.206	2.988	2.795	10.68	2.85	0.136	0.158
Gas100000b	0.146	0.24	0.182	3.511	3.555	2.968	11.79	2.81	0.146	0.176
Gas1000000a	1.274	1.37	1.63	3.983	3.946	3.990	1.51	4.21	1.747	1.693
Gas1000000b	1.732	2.37	2.07	4.773	4.907	4.675	12.72	4.79	1.902	1.795
Gas10000000a	11.914	15.59	13.84	16.090	16.767	16.620	24.20	No	13.846	12.806
Gas10000000b	16.572	33.35	36.27	23.428	25.156	25.441	34.40	No	23.025	21.760
Gas10000000c	22.175	72.40	32.82	34.625	30.989	32.921	41.55	No	34.249	30.500

TABLE 10
The Performance Improvement of Solving the Cases Using BHISA on Multi-Core CPUs (%)

Quasi-tridiagonal	Lis lib							PARALUTION lib	
Matrices	Jacobi	GS	GMRES(5)	BiCGSafe	BiCGSTAB	TFQMR	BiCRSTAB	GMRES(5)	BiCGSTAB
Gas2000	77.41	78.04	98.32	99.11	98.25	99.16	99.11	45.96	47.43
Gas10000a	51.64	56.90	97.58	98.62	98.08	98.17	97.94	−6.89	7.96
Gas10000b	63.46	56.28	97.25	98.24	97.97	98.44	97.89	−12.86	−4.05
Gas100000a	21.66	27.49	94.52	95.95	95.67	98.87	95.75	11.20	23.69
Gas100000b	39.24	20.21	95.85	95.90	95.09	98.76	94.82	0.24	17.29
Gas1000000a	7.17	21.70	68.01	67.71	68.07	15.58	68.74	27.08	24.73
Gas1000000b	26.92	16.16	63.71	64.70	62.94	86.38	63.84	8.92	3.47
Gas10000000a	23.56	13.96	25.95	28.86	28.32	50.76	No	13.95	6.96
Gas10000000b	50.30	54.31	29.27	34.12	34.86	51.82	No	28.03	23.84
Gas10000000c	69.37	32.45	35.96	38.38	32.64	46.63	No	35.44	27.05

TABLE 11
The Performance of Solving the Cases on GPU

Quasi-tridiagonal	BHISA	PARALUTION lib				cuSparse	
Matrices		GMRES(5)		BiCGSTAB			
		Time(s)	Improvement(%)	Time(s)	Improvement(%)	Time(s)	Improvement(%)
Gas2000	0.021	0.034	39.33	0.034	38.48	0.035	40.86
Gas10000a	0.031	0.049	37.21	0.045	31.55	0.078	60.26
Gas10000b	0.046	0.052	11.28	0.047	2.07	0.063	26.98
Gas100000a	0.103	No	No	No	No	0.812	87.32
Gas100000b	0.107	No	No	No	No	0.842	87.29
Gas1000000a	1.220	No	No	No	No	7.831	84.42
Gas1000000b	1.306	No	No	No	No	7.519	82.63
Gas10000000a	8.166	No	No	No	No	78.811	89.64
Gas10000000b	10.640	No	No	No	No	79.622	86.64
Gas10000000c	12.939	No	No	No	No	80.059	83.84

BiCGSafe. We tested all the cases using BiCGSafe for parallel mode, whose flexibility is better than that of BiCG.

The performance of Jacobi and GS are close to those of the other iterative algorithms, because the number of operations in each iteration process is fewer. But the performance improvement rates of Jacobi and GS will be less than those of the other iterative algorithms with faster convergence speeds when the compute nodes increase. The operations in the same iteration process can be relatively easy to be computed in parallel, and the operations in different iterations cannot be computed in parallel.

For the matrices without very large sizes, the performance of cuSparse is better than that of most iterative algorithms, because the solver of cuSparse is a direct solving algorithm and it is suitable for small linear systems, such as Gas2000, Gas10000a, Gas10000b, Gas100000a, and Gas100000b. The performance of the direct solving algorithm deteriorates sharply when the sizes of linear systems increase rapidly. So the performance of cuSparse is worse than those of the iterative algorithms for Gas1000000a, Gas1000000b, Gas10000000a, Gas10000000b, and Gas10000000c.

GMRES(5) and BiCGSTAB of PARALUTION library have good performance for parallel mode, but have poor robustness, because matrices with large sizes are not computed using CUDA.

The performance improvement precentages by using BHISA on multi-core CPUs compared with Jacobi, GS, GMRES(5), BiCGSafe, BiCGSTAB, TFQMR, BiCRSTAB of Lis library, GMRES(5) and BiCGSTAB of PARALUTION library in Table 10 are calculate by $(T_i - T_H)/T_i \times 100$ $(i = 1, 2, 3, 4, 5, 6, 7, 8, 9)$, where T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8 and T_9 are the performance of Jacobi, GS, GMRES(5), BiCGSafe, BiCGSTAB, TFQMR, BiCRSTAB of Lis library, GMRES (5) and BiCGSTAB of PARALUTION library respectively and T_H is the performance of BHISA.

It is observed from Table 10 that the average execution time of BHISA on multi-core CPUs is reduced by 43.07,

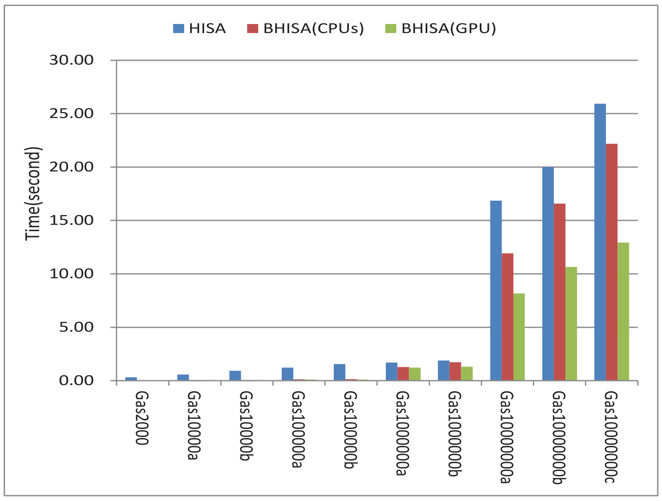


Fig. 4. The performance of using HISA, BHISA on multi-core CPUs, and BHISA on K20c GPU.

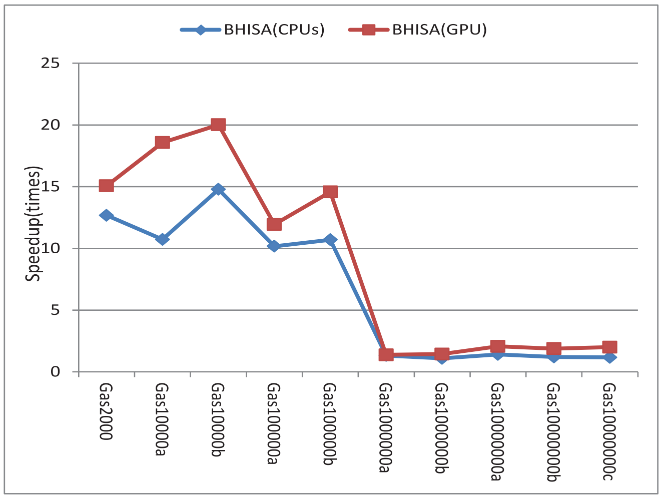


Fig. 5. The speedups using BHISA on multi-core CPU and K20c GPU compared with HISA.

37.75, 70.64, 72.16, 71.19, 74.46, 88.44, 15.11, and 17.84 percent by using BHISA compared with Jacobi, GS, GMRES(5), BiCGSafe, BiCGSTAB, TFQMR, BiCRSTAB of Lis library, and GMRES(5) and BiCGSTAB of PARALUTION library on multi-core CPUs.

It is observed from Table 11 that the average execution time of BHISA on K20c is reduced by 29.28, 24.03, and 72.99 percent by using BHISA compared with GMRES(5) and BiCGSTAB of PARALUTION library, and cuSparse on K20c.

Parallel computation efficiency of solving the quasi-tridiagonal system of linear equations is not high because of the irregularity of matrices and data dependencies between the iterative processes, and we found from Tables 7, 9, and 11 that performance improvements using iterative methods are not very prominent for parallel mode. The performance of using HISA, BHISA on multi-core CPUs, and BHISA on K20c GPU is shown in Fig. 4. It is observed from Fig. 5 that the average speedups of BHISA on multi-core CPUs and K20c GPU are 6.52 and 8.90 respectively compared with HISA on serial mode.

7 Conclusion

In this paper, a parallel hybrid solving algorithm is proposed for quasi-tridiagonal system of linear equations, whose performance is better than those of the other iterative algorithms and direct algorithms, and the storage space of the hybrid algorithm is less than those of the other solving algorithms. Furthermore, we implement the parallel CR algorithm on GPU using partitioning strategy so as to significantly reduce the latency of memory access. The computing performance of some numerical simulations problems can be improved by using our method. Other sparse linear systems arising from numerical simulation may be quasi-block-diagonals equations, and how to solve them quickly will be our next step of investigation.

Acknowledgments

The research was partially funded by the Key Program of National Natural Science Foundation of China (Grant Nos. 61133005, 61432005), the National Natural Science Foundation of China (Grant Nos. 61370095, 61472124, 61572175), and the International Science & Technology Cooperation Program of China (Grant No. 2015DFA11240), and Science and technology project of Hunan Province (Grant No. 2015SK20062).

References

[1] H. S. Kim, S. Wu, L. W. Chang, and W. M. W. Hwu, “2011 international conference on parallel processing a scalable tridiagonal solver for GPUs,” in *Proc. 42nd Int. Conf. Parallel Process.*, 2011, pp. 444–453.
[2] J. Lamas-Rodrguez, D. B. Heras, M. Bóo., and F. Argúello, “Tridiagonal system solvers internal report,” Dept. of Electron. and Comput. Sci., Univ. of Santiago de Compostela, Spain, Internal Rep. 01/2011, Feb. 2011.
[3] E. P. Natarajan, “Klu-a high performance sparse linear solver for circuit simulation problems,” Ph.D. dissertation, Univ. of Florida, 2005.
[4] L. H. Thomas, “Elliptic problems in linear difference equations over a network,” Columbia Univ., New York, USA, Watson Sci. Comput. Lab. Rep. 01/1949, 1949.
[5] B. L. Buzbee, G. H. Golub, and C. W. Nielson, “On direct methods for solving Poisson’s equations,” *SIAM J. Numerical Anal.*, no. 7, pp. 627–656, 1970.
[6] R. W. Hockney, “A fast direct solution of Poisson’s equation using fourier analysis,” *J. ACM*, vol. 12, pp. 95–113, 1965.
[7] H. S. Stone, “An efficient parallel algorithm for the solution of a tridiagonal linear system of equations,” *J. ACM*, vol. 20, pp. 27–38, 1973.
[8] S. Bondeli, “Divide and conquer: A parallel algorithm for the solution of a tridiagonal linear system of equations,” in *Proc. Joint Int. Conf. Vector Parallel Process.*, 1990, vol. 4, pp. 419–434.
[9] H. H. Wang, “A parallel method for tridiagonal equations,” *ACM Trans. Math. Softw.*, vol. 7, pp. 170–183, 1981.
[10] G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao, and N. Sun, “Fast implementation of DGEMM on fermi GPU,” in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2011, pp. 1–11.
[11] A. Lefohn, J. D. Owens, and M. Kass, “Interactive depth of field using simulated diffusion,” Pixar Animation Studios, Emeryville, California, USA, Pixar Tech. Memo 06-01, 2011.
[12] G. Dominik and S. Robert, “Cyclic reduction tridiagonal solvers on GPUs applied to mixed-precision multigrid,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 1, pp. 22–32, Jan. 2011.
[13] J. Lamas-Rodríguez, F. Arguello, D. B. Heras, and M. Bóo, “Memory hierarchy optimization for large tridiagonal system solvers on GPU,” in *Proc. IEEE 10th Int. Symp. Parallel Distrib. Process. with Appl.*, 2012, pp. 87–94.
[14] A. Davidson and J. D. Owens, “Register packing for cyclic reduction: A case study,” in *Proc. 4th Workshop General Purpose Process. Graph. Process. Units*, Mar. 2011, vol. 4, pp. 1–6.
[15] R. Hockney and C. Jesshope, *Parallel Computers*, 1st ed. Bristol, England, U.K.: Adam Hilger, 1981.
[16] Y. Zhang, J. Cohen, and J. Owens, “Fast tridiagonal solvers on the GPU,” in *Proc. 15th ACM SIGPLAN Symp. Principles Practice Parallel Programm.*, Aug. 2010, pp. 97–106.
[17] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Philadelphia, PA, USA: SIAM, 2003.
[18] C. Aykanat, O. Selvitopi, and M. Ozdal, “A novel method for scaling iterative solvers: Avoiding latency overhead of parallel sparse-matrix vector multiplies,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 3, pp. 632–645, Mar. 2015.
[19] K. Li, W. Yang, and K. Li, “Performance analysis and optimization for SPMV on GPU using probabilistic modeling,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 1, pp. 196–205, Jan. 2015.
[20] W. Yang, K. Li, Z. Mo, and K. Li, “Performance optimization using partitioned SPMV on GPUs and multicore CPUs,” *IEEE Trans. Comput.*, vol. 64, no. 9, pp. 2623–2636, Sep. 2015.
[21] J. Li, G. Tan, M. Chen, and N. Sun, “SMAT: An input adaptive auto-tuner for sparse matrix-vector multiplication,” *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 117–126, 2013.
[22] T. Tsuburaya, Y. Okamoto, K. Fujiwara, and S. Sato, “Performance of preconditioned linear solvers based on minimum residual for complex symmetric linear systems,” *IEEE Trans. Magnetics*, vol. 50, no. 2, pp. 557–560, Feb. 2014.
[23] B. Carpentieri, J. Liao, and M. Sosonkina, “Variable block multilevel iterative solution of general sparse linear systems,” in *Proc. Parallel Process. Appl. Math.*, 2014, pp. 520–530.

[24] E. N. Akimovaa and D. V. Belousova, "Parallel algorithms for solving linear systems with block-tridiagonal matrices on multi-core CPU with GPU," *J. Comput. Sci.*, vol. 3, pp. 445–449, 2012.

[25] G. R. Morris and K. H. Abed, "Mapping a Jacobi iterative solver onto a high-performance heterogeneous computer," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 1, pp. 85–91, Jan. 2013.

[26] W. Yang, K. Li, Y. Liu, L. Shi, and L. Wan, "Optimization of quasi-diagonal matrix–vector multiplication on GPU," *Int. J. High Perform. Comput. Appl.*, vol. 28, no. 2, pp. 183–195, 2014.

[27] C. D. E and H. E. V, "An identity for the schur complement of a matrix," *Proc. Am. Math. Soc.*, vol. AMS-22, pp. 364–366, 1969.

[28] J. Gaidamour and P. Henon, "A parallel direct/iterative solver based on a schur complement approach," in *Proc. IEEE Int. Conf. Comput. Sci. Eng.*, 2008, vol. 1, pp. 98–105.

[29] L. Giraud and A. Haidar, "Parallel algebraic hybrid solvers for large 3d convection-diffusion problems," *Numerical Algorithms*, vol. 51, pp. 151–171, 2009.

[30] I. Yamazaki and X. S. Li, "On techniques to improve robustness and scalability of a parallel hybrid linear solver," in *Proc. 9th Int. Conf. High Perform. Comput. Comput. Sci.*, 2011, vol. 1, pp. 421–434.

[31] S. Rajamanickam, E. G. Boman, and M. A. Heroux, "Shylu: A hybrid-hybrid solver for multicore platforms," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp.*, 2012, pp. 631–643.

[32] NVIDIA, "Nvidia cuda c programming guide," 2701 San Tomas Expressway Santa Clara, USA, Tech. Rep. PG-02829-001_v7.5, 2015.

[33] K. Abe, T. Sogabe, S. Fujino, and S. Zhang, "A product-type Krylov subspace method based on conjugate residual method for non-symmetric coefficient matrices," *IPSJ Trans. Adv. Comput. Syst.*, vol. 48, pp. 11–21, 2007.

[34] P. R. Amestoy, T. A. Davis, and I. S. Duff, "An approximate minimum degree ordering algorithm," *SIAM J. Matrix Anal. Appl.*, vol. 17, no. 4, pp. 886–905, 1996.

[35] T. A. Davis, "Algorithm 915, suitesparseqr: Multifrontal multithreaded rank-revealing sparse QR factorization," *ACM Trans. Math. Softw.*, vol. 38, no. 1, p. 8, 2011.

[36] T. A. Davis, S. N. Yeralan, and S. Ranka, "Algorithm 9xx: Sparse qr factorization on the GPU," *ACM Trans. Math. Softw.*, no. 1, pp. 1–28, Jan. 2015.

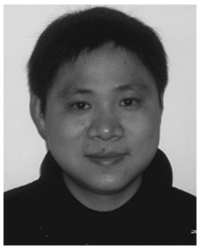
[37] NVIDIA, "cusparse library," 2701 San Tomas Expressway Santa Clara, USA, Tech. Rep. DU-06709-001_v7.5, Mar. 2015.

[38] A. Nishida, "Experience in developing an open source scalable software infrastructure in Japan," in *Proc. Int. Conf., Comput. Sci. Its Appl.*, 2010, pp. 448–462.

[39] NVIDIA, "cusolver library," 2701 San Tomas Expressway Santa Clara, USA, Tech. Rep. DU-06709-001_v7.5, Sep. 2015.

[40] P. L. U. (haftungsbeschrankt), and C. KG., "Paralution - user manual," Am Hasensprung 6, 76571 Gaggenau, Germany, Tech. Rep. version 1.0.0, 2015.

Kenli Li is a full professor of computer science and technology at Hunan University and a deputy director of the National Supercomputing Center in Changsha. He has published more than 130 papers in international conferences and journals. He is currently served on the editorial boards of *IEEE Transactions on Computers*. He is an outstanding member of CCF and a member of the IEEE.



Wangdong Yang is a professor of computer science and technology at Hunan City University, China. His research interests include modeling and programming for heterogeneous computing systems, parallel algorithms, grid and cloud computing.

Keqin Li is a SUNY distinguished professor of computer science. He has more than 370 research publications. He is currently or has served on the editorial boards of *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, *and IEEE Transactions on Cloud Computing*. He is a fellow of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.