Research Article

A Heterogeneous Parallel LU Factorization Algorithm Based on a Basic Column Block Uniform Allocation Strategy

Rongteng Wu ^{1,2} and Xiaohong Xie ³

¹College of Computer and Control Engineering, Minjiang University, Fuzhou 350108, China

²Fujian Provincial Key Laboratory of Information Processing and Intelligent Control, Minjiang University, Fuzhou 350108, China

³Department of Teaching Affairs, Minjiang University, Fuzhou 350108, China

Correspondence should be addressed to Rongteng Wu; rongtengwu@163.com

Received 11 October 2018; Revised 19 January 2019; Accepted 10 February 2019; Published 25 February 2019

Academic Editor: Fazal M. Mahomed

Copyright © 2019 Rongteng Wu and Xiaohong Xie. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Most supercomputers are shipped with both a CPU and a GPU. With the powerful parallel computing capability of GPUs, heterogeneous computing architecture produces new challenges for system software development and application design. Because of the significantly different architectures and programming models of CPUs and GPUs, conventional optimization techniques for CPUs may not work well in a heterogeneous multi-CPU and multi-GPU system. We present a heterogeneous parallel LU factorization algorithm for heterogeneous architectures. According to the different performances of the processors in the system, any given matrix is partitioned into different sizes of basic column blocks. Then, a static task allocation strategy is used to distribute the basic column blocks to corresponding processors uniformly. The idle time is minimized by optimized sizes and the number of basic column blocks. Right-looking ahead technology is also used in systems configured with one CPU core to one GPU to decrease the wait time. Experiments are conducted to test the performance of synchronization and load balancing, communication cost, and scalability of the heterogeneous parallel LU factorization in different systems and compare it with the related matrix algebra algorithm on a heterogeneous system configured with multiple GPUs and CPUs.

1. Introduction

Traditional computer architecture primarily depends on homogeneous computing resources to manage complexity and to facilitate programming productivity and portability. However, homogeneous computing is not a good choice for supplying the performance demanded by emerging applications because of their inherent inefficiency and the stringent power and thermal constraints imposed by the fabrication technologies [1, 2]. Heterogeneous parallel computing has undergone great development in the past few years and will be increasingly important in future computing architectures [3, 4]. Currently, computer architects, programmers, and researchers have turned from the CPU versus GPU debate into a CPU and GPU paradigm where the best features of both can be intelligently combined to achieve even further computational gains [5–8]. Computer architecture is transitioning from the multicore era into the heterogeneous

era [9, 10]. Most supercomputers are shipped with CPUs and GPUs. However, heterogeneous computing systems have introduced new challenges to algorithm design and system software development because of the significantly different architectures and programming models of CPUs and GPUs. Conventional optimization techniques for CPUs may not work well in a heterogeneous system with multiple CPUs and GPUs [11, 12]. Hence, it is necessary to present novel techniques to exploit the computing potentiality of heterogeneous computing. This paper presents a parallel block LU factorization algorithm based on a basic column block uniform allocation strategy for heterogeneous computing architecture. According to the different performances of different processors, the matrix is partitioned into different-sized basic column blocks that are allocated to corresponding processors by using a cyclic distribution strategy. The experiment tests the algorithm performance including parallel execution time,

load balancing, synchronization and communication, and scalability.

The remainder of this paper is organized as follows: Section 2 reviews related works. Section 3 presents the proposed algorithm for heterogeneous computing architecture. Section 4 gives the experimental results, followed by a brief conclusion in Section 5.

2. Related Works

Gaussian elimination is commonly used to solve dense linear algebra systems and is widely employed in scientific and engineering models [13–15]. LU factorization routines are included in almost all popular linear algebra libraries, such as the Linear Algebra Package (LAPACK), the Scalable Linear Algebra Package (ScaLAPACK), and Matrix Algebra on GPU and Multicore Architectures (MAGMA). As de facto standard software for high-performance dense linear algebra computations, LAPACK and ScaLAPACK have been developed for shared-memory and distributed-memory architectures [16]. LAPACK [17] is a software library for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems. It is designed to be efficient on a wide range of modern high-performance computers and provides the associated matrix factorizations. ScaLAPACK [18] is a continuation of the LAPACK project for distributed-memory message-passing MIMD computers and networks of workstations supporting PVM and MPI. However, they do not apply to heterogeneous systems with GPUs. MAGMA [19] is a collection of linear algebra libraries for hybrid manycore and GPU systems. MAGMA uses a hybridization methodology where algorithms of interest are split into tasks of varying granularity, and their execution is scheduled over the available hardware components. To improve the performance of the algorithm, small nonparallelizable tasks are often scheduled on the CPU, and larger more parallelizable tasks are often scheduled on the GPU. Their exploitation of parallelism is based on the availability of parallel BLAS. CUBLAS [20] and CULA [21] have implemented the standard BLAS and LAPACK subroutine libraries, respectively, on multiple GPUs. For the works on performance optimization for matrix factorization, Volkov et al. [22] implemented the right-looking algorithm for LU, Cholesky, and QR for GPUs. Van De Velde [23] presented an LU decomposition algorithm for multicomputers with implicit pivoting. Kurzak et al. [24] used partial pivoting for multicore systems to accelerate LU factorization. A partial pivoting strategy is introduced to LU factorization in popular linear algebra libraries such as LAPACK. The LU factorization with partial pivoting is a pivotal numerical decomposition for the solution of dense systems of linear equations appearing in scientific and engineering applications [25]. Tomov et al. [19] presented an LU factorization with fewer pivots for hybrid manycore and GPU systems. Deisher et al. [26] described an algorithm by using a dynamic load balancing strategy on a multicore platform. Architecture-specific optimizations are carefully designed and used to obtain higher performance. A load balancing method was proposed in [27] to utilize all processors

updating the trail matrix. Choi et al. [28] developed a parallel Cholesky block factorization algorithm, but it cannot be used in heterogeneous systems with multiple CPUs and multiple GPUs. Wang et al. [29] optimized the load distribution on a heterogeneous system with CPU and GPU. Lastovetsky and Reddy [30] developed a static data distribution for dense matrix factorization. In these works, the researchers did not take advantage of the computing capability of both CPUs and GPUs. A row block cyclic distribution Cholesky factorization algorithm was described in [31] for heterogeneous multiple CPUs and multiple GPUs architectures. To take advantage of the computing capability of both CPUs and GPUs, the matrix is partitioned into different-sized row blocks based on the performance of the CPU and GPU and then statically allocated to corresponding processors. Dong et al. [32] designed the Cholesky factorization and QR factorization for multicores and multiple GPUs systems. The heterogeneous algorithm with different tile sizes is designed to address processor heterogeneity, and a static data distribution is used to realize load balancing. However, those studies did not include the LU factorization.

Considering the capability of both CPUs and GPUs, load balancing, communication cost, scalability, etc., we present a heterogeneous parallel LU factorization based on a basic column block cyclic uniform distribution strategy for heterogeneous architectures. The experiment results show that the performance of our algorithm can compete with the related hybrid algorithms in MAGMA.

3. Heterogeneous Parallel LU Block Factorization Algorithm

3.1. LU Factorization of Matrices with Different Size Blocks. Given an $N \times N$ matrix A to be partitioned into $n \times n$ blocks:

$$A = \begin{pmatrix} A_{11} & \cdots & A_{1n} \\ \cdots & \cdots & \cdots \\ A_{n1} & \cdots & A_{nn} \end{pmatrix} \quad (1)$$

where the submatrix A_{ii} is of size $b_i \times b_i$ ($i = 1, \dots, n$). LU factorization of matrix A uses a series of Gaussian eliminations to form $PA = LU$, where P is a permutation matrix, L is a unit lower triangular matrix, and U is an upper triangular matrix. Generally, the column block LU factorization algorithm will be implemented in n iterations, and no extra storage is required for L_{ij} and U_{ij} because they will overwrite the corresponding A_{ij} blocks of the original matrix A actually. Assume that after the $(k-1)$ -th iteration,

$$A = A^{(k-1)} = \begin{pmatrix} L_{11}U_{11} & \cdots & U_{1k} & \cdots & U_{1n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ L_{k1}^{(k-1)} & \cdots & A_{kk}^{(k-1)} & \cdots & A_{kn}^{(k-1)} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ L_{n1}^{(k-1)} & \cdots & A_{nk}^{(k-1)} & \cdots & A_{nn}^{(k-1)} \end{pmatrix} \quad (2)$$

The k -th iteration will involve the following operation steps.

(1) Use Gaussian eliminations on $(A_{kk}^{(k-1)} \dots A_{nk}^{(k-1)})^T$ to compute

$$P_k (A_{kk}^{(k-1)} \dots A_{nk}^{(k-1)})^T = (L_{kk}^{(k)} \dots L_{nk}^{(k)})^T U_{kk} \quad (3)$$

Obviously, $L_{kk}^{(k)} = L_{kk}$.

(2) Apply row interchanges to the left column blocks and the right column blocks on the submatrix $(A_{kk}^{(k-1)} \dots A_{nk}^{(k-1)})^T$ according to permutation matrix P_k ; i.e.,

$$P_k \begin{pmatrix} L_{k1}^{(k-1)} & \dots & L_{k(k-1)}^{(k-1)} \\ L_{(k+1)1}^{(k-1)} & \dots & L_{(k+1)(k-1)}^{(k-1)} \\ \dots & \dots & \dots \\ L_{n1}^{(k-1)} & \dots & L_{n(k-1)}^{(k-1)} \end{pmatrix} \Rightarrow \begin{pmatrix} L_{k1} & \dots & L_{k(k-1)} \\ L_{(k+1)1}^{(k)} & \dots & L_{(k+1)(k-1)}^{(k)} \\ \dots & \dots & \dots \\ L_{n1}^{(k)} & \dots & L_{n(k-1)}^{(k)} \end{pmatrix} \quad (4)$$

and

$$P_k \begin{pmatrix} A_{k(k+1)}^{(k-1)} & \dots & A_{kn}^{(k-1)} \\ A_{(k+1)(k+1)}^{(k-1)} & \dots & A_{(k+1)n}^{(k-1)} \\ \dots & \dots & \dots \\ A_{n(k+1)}^{(k-1)} & \dots & A_{nn}^{(k-1)} \end{pmatrix} \Rightarrow \begin{pmatrix} A_{k(k+1)}^{(k-1)} & \dots & A_{kn}^{(k-1)} \\ A_{(k+1)(k+1)}^{(k-1)} & \dots & A_{(k+1)n}^{(k-1)} \\ \dots & \dots & \dots \\ A_{n(k+1)}^{(k-1)} & \dots & A_{nn}^{(k-1)} \end{pmatrix} \quad (5)$$

(3) Compute $(U_{k(k+1)} \dots U_{kn})$ from $(A_{k(k+1)}^{(k-1)} \dots A_{kn}^{(k-1)})$, according to

$$L_{kk} (U_{k(k+1)} \dots U_{kn}) = (A_{k(k+1)}^{(k-1)} \dots A_{kn}^{(k-1)}) \quad (6)$$

(4) Update the rest of matrix A ; i.e.,

$$\begin{pmatrix} A_{(k+1)(k+1)}^{(k-1)} & \dots & A_{(k+1)n}^{(k-1)} \\ \dots & \dots & \dots \\ A_{n(k+1)}^{(k-1)} & \dots & A_{nn}^{(k-1)} \end{pmatrix} = \begin{pmatrix} A_{(k+1)(k+1)}^{(k-1)} & \dots & A_{(k+1)n}^{(k-1)} \\ \dots & \dots & \dots \\ A_{n(k+1)}^{(k-1)} & \dots & A_{nn}^{(k-1)} \end{pmatrix} - \begin{pmatrix} L_{(k+1)k}^{(k)} \\ \vdots \\ L_{nk}^{(k)} \end{pmatrix} (U_{k(k+1)} \dots U_{kn}) \quad (7)$$

And the submatrix block $\begin{pmatrix} A_{(k+1)(k+1)}^{(k-1)} & \dots & A_{(k+1)n}^{(k-1)} \\ A_{n(k+1)}^{(k-1)} & \dots & A_{nn}^{(k-1)} \end{pmatrix}$ will be written as $\begin{pmatrix} A_{(k+1)(k+1)}^{(k)} & \dots & A_{(k+1)n}^{(k)} \\ A_{n(k+1)}^{(k)} & \dots & A_{nn}^{(k)} \end{pmatrix}$.

3.2. Matrix Partition and Data Distribution. Given an $N \times N$ matrix A to be partitioned into different size column blocks based on the different performance of the processors in the system, the one-dimensional column block cyclic uniform allocation strategy will be used to distribute the matrix data. Assume that the system has p processors involved in computing LU factorization, and the performance ratio of these p processors is $perf_1 : \dots : perf_p$. Ideally, if $(perf_1 + \dots + perf_p) \times m = N$, where $perf_1, \dots, perf_p$, and m are integer, processors $p_1, \dots, p_p$ will be allocated with $m \cdot perf_1, \dots, m \cdot perf_p$ matrix columns, respectively. Provided the best size of column blocks is $b_1, \dots, b_p$ for processors $p_1, \dots, p_p$ computing a submatrix with sizes $m \cdot perf_1 \times N, \dots, m \cdot perf_p \times N$, respectively, these submatrices will be partitioned into $m \cdot perf_1 / b_1, \dots, m \cdot perf_p / b_p$ column blocks, and we call these column blocks the basic column block. Then, according to the numbers of column block $m \cdot perf_1 / b_1, \dots, m \cdot perf_p / b_p$ and the sizes of the basic column blocks $b_1, \dots, b_p$, the column blocks of matrix A are circularly allocated to processors $p_1, \dots, p_p$. Obviously, the numbers and sizes of the column blocks are determined by the performance of the processors and the matrix size. The k -th column block is allocated to processor p_z , where z is determined by k as shown in

$$z = \begin{cases} 1, & \text{if } (k-1) \bmod \left(\sum_{j=1}^p \frac{m \cdot perf_j}{b_j} \right) \in \left[0, \frac{m \cdot perf_1}{b_1} \right) \\ i, & \text{if } (k-1) \bmod \left(\sum_{j=1}^p \frac{m \cdot perf_j}{b_j} \right) \in \left[\sum_{j=1}^{i-1} \frac{m \cdot perf_j}{b_j}, \sum_{j=1}^i \frac{m \cdot perf_j}{b_j} \right) \end{cases} \quad (8)$$

Figure 1 shows an example of the matrix partition and data distribution in a system with three processors. Suppose

the performance ratio of three processors p_1, p_2 , and p_3 is $perf_1 : perf_2 : perf_3 = 1:4:8$, and the best column block

FIGURE 1: Example of the matrix partition and data distribution strategy.

size is b_1 , b_2 , and b_3 , where $b_2=2b_1$ and $b_3=4b_1$. The matrix A will be partitioned into three types of basic column blocks with column sizes b_1 , $b_2 (=2b_1)$, and $b_3 (=4b_1)$ matching with processors p_1 , p_2 , and p_3 . Then, these basic column blocks are uniformly allocated to p_1 , p_2 , and p_3 circularly. Each $4b_1$ column belonging to p_2 is divided into 2 basic column blocks that include $b_2 (=2b_1)$ columns. Similarly, every $8b_1$ columns belonging to p_3 is divided into 2 basic column blocks that include $b_3 (=4b_1)$ columns. In other words, every $13b_1$ column will be divided into 5 different basic column blocks, the sizes of which are $b_1, 2b_1, 2b_1, 4b_1$, and $4b_1$. In addition, the first b_1 columns are allocated to p_1 , the subsequent two $2b_1$ to p_2 , and the last two $4b_1$ to p_3 .

3.3. Heterogeneous Parallel LU Factorization Algorithm Design. Suppose that an $N \times N$ matrix is partitioned into $n \times n$ different blocks and that the corresponding basic column blocks have been allocated to all processors. Then, a heterogeneous parallel LU factorization algorithm based on the basic column blocks uniform data allocation contains n iterations, and the k -th iteration will be designed as follows.

Step 1. The process p_z which is allocated with the k -th column block of A based on (8) factorizes the k -th column block by computing $P_k \begin{pmatrix} A_{kk}^{(k-1)} & \dots & A_{nk}^{(k-1)} \end{pmatrix}^T = \begin{pmatrix} L_{kk}^{(k)} & \dots & L_{nk}^{(k)} \end{pmatrix}^T U_{kk}$.

Step 2. The process p_z broadcasts P_k and $\begin{pmatrix} L_{kk}^{(k)} & \dots & L_{nk}^{(k)} \end{pmatrix}^T$ to every process p_y .

Step 3. The process p_z runs row exchange on the left columns and right columns of the k -th column block belonging to it. In addition, every other process p_y runs a row exchange on all columns belonging to it.

Step 4. Every process p_x computes its corresponding U_{ki} ($k < i \leq n$) according to (6).

Step 5. Every process p_x updates the corresponding column blocks based on (7).

Figure 2 illustrates the operation steps of the k -th iteration of the algorithm run on a system with two different processors p_1 and p_2 . Suppose that the sizes of the basic column blocks

for processors p_1 and p_2 are b_1 and b_2 . According to the different performance of the two processors, a matrix is partitioned into 9×9 blocks that include two different basic sizes b_1 and b_2 , as shown in (a). The algorithm completes 9 iterations. Figure 2(b) shows the state at the end of the $(k-1)$ -th step, and Figure 2 from (c) to (f) shows the operations of the end of Steps 1, 3, 4, and 5 of the k -th iteration.

In the actual implementation of the algorithm, we will use the LAPACK routines `dgetrf()`, `dlaswp()`, `dtrsm()`, and `dgemm()` for CPU cores and the CUBLAS routines `cublasDswap`, `cublasDtrsm()`, and `cublasDgemm()` for GPUs to factorize the double precision matrix. We denote the submatrix composed of all column blocks distributed to process p as S . These routines can be described as follows.

(1) Assuming that the k -th column block of A is the k' -th column block of S in the current process p_z , then $\begin{pmatrix} A_{kk} & \dots & A_{nk} \end{pmatrix}^T$ is equivalent to $\begin{pmatrix} S_{kk'} & \dots & S_{nk'} \end{pmatrix}^T$ which we denote as $\overrightarrow{S_{kk'}}$. Routine `dgetrf`($\overrightarrow{S_{kk'}}, P_k$) computes LU factorization of the column block $\overrightarrow{S_{kk'}}$ for the current CPU core process p_z and routine `gpu_dgetrf`($\overrightarrow{S_{kk'}}, P_k$) computes LU factorization of the column block $\overrightarrow{S_{kk'}}$ for the current GPU process p_z by using a series of Gaussian eliminations.

(2) Routine `dlaswp` (`left`($\overrightarrow{S_{kk'}}$), `right`($\overrightarrow{S_{kk'}}$), P_k) or `cublasDswap`(`left`($\overrightarrow{S_{kk'}}$), `right`($\overrightarrow{S_{kk'}}$), P_k) is used to make the row exchange on the left columns and the right columns of the k' -th column block $\overrightarrow{S_{kk'}}$ of the submatrix S for the current CPU or GPU process p_z . And `dlaswp`(S, P_k) or `cublasDswap`(S, P_k) is used to make the row exchange on the submatrix S for other CPU or GPU processes p_y .

(3) Routine `dtrsm`(L_{kk} , `right`(A_{kk})) or `cublasDtrsm`(L_{kk} , `right`(L_{kk})) computes the corresponding blocks of the row panel $\begin{pmatrix} U_{k(k+1)} & \dots & U_{kn} \end{pmatrix}$ on the right of the block A_{kk} for all CPU or GPU processes p_x .

(4) Routine `dgemm`($\overrightarrow{S_{(k+1)k'}}$, `right`(A_{kk}), `rest`(A_{kk})) or `cublasDgemm`($\overrightarrow{S_{(k+1)k'}}$, `right`(A_{kk}), `rest`(A_{kk})) updates the corresponding column blocks of the remaining part of matrix A for all CPU or GPU processes p_x . For process p_x , `right`(A_{kk}) is the blocks A_{kj} which is allocated to p_x and satisfied with $k < j \leq n$, and `rest`(A_{kk}) is the blocks A_{ij} which is allocated to p_x and satisfied with $k < i \leq n$ and $k < j \leq n$.

```

(1) for  $k=1, \dots, n$  do
(2)   if (process_id=z) then
(3)      $\text{dgetrf}(\overrightarrow{S_{kk'}}, P_k)$  or  $\text{gpu\_dgetrf}(\overrightarrow{S_{kk'}}, P_k)$ 
(4)   end if
(5)    $\text{broadcast}(\overrightarrow{S_{kk'}}, P_k, k')$ 
(6)   if (process_id=z) then
(7)      $\text{dlaswp}(\text{left}(\overrightarrow{S_{kk'}}), \text{right}(\overrightarrow{S_{kk'}}), P_k)$  or  $\text{cublasDswap}(\text{left}(\overrightarrow{S_{kk'}}), \text{right}(\overrightarrow{S_{kk'}}), P_k)$ 
(8)   else
(9)      $\text{dlaswp}(S, P_k)$  or  $\text{cublasDswap}(S, P_k)$ 
(10)  end if
(11)   $\text{dtrsm}(L_{kk}, \text{right}(A_{kk}))$  or  $\text{cublasDtrsm}(L_{kk}, \text{right}(A_{kk}))$ 
(12)   $\text{dgemm}(\overrightarrow{S_{(k+1)k'}}, \text{right}(A_{kk}), \text{rest}(A_{kk}))$  or  $\text{cublasDgemm}(\overrightarrow{S_{(k+1)k'}}, \text{right}(A_{kk}), \text{rest}(A_{kk}))$ 
(13) end for

```

ALGORITHM 1: Heterogeneous parallel LU factorization algorithm based on a basic column block uniform allocation strategy for a multiple CPU/GPU system.

FIGURE 2: The operations of the heterogeneous parallel column block LU factorization algorithm. (a) Matrix partition and task distribution. (b) The state at the end of the $(k-1)$ -th iteration. (c) Step 1 of the k -th iteration. (d) Step 3 of the k -th iteration. (e) Step 4 of the k -th iteration. (f) Step 5 of the k -th iteration.

Assume that all the basic column blocks of matrix A have been distributed to the corresponding processes p_x and form submatrix S in process p_x . The heterogeneous parallel LU factorization algorithm based on a basic column block uniform allocation strategy for a multiple

CPU and/or multiple-GPU system can be described as in Algorithm 1.

In a CPU/GPU hybrid heterogeneous system, the right-looking ahead strategy can also be used to reduce the wait time. For every GPU, a CPU core is assigned as a match. In

FIGURE 3: Parallel execution time of a 16-core system related to different column block sizes.

the k -th iteration, after broadcasting the related data, the corresponding GPU first runs `cublasDswap()`, `cublasDtrsm()`, and `cublasDgemm()` on the $(k+1)$ -th column block so that its matching CPU core can run `dgetrf()` on the $(k+1)$ -th column block ahead of time. In this way, it will be completely parallel for GPU runs `cublasDswap()`, `cublasDtrsm()`, and `cublasDgemm()` on the other column blocks in the k -th iteration and CPU core runs `dgetrf()` on the $(k+1)$ -th column block in the $(k+1)$ -th iteration. The wait time for synchronization will be reduced further.

4. Experimental Results

We tested the heterogeneous parallel LU factorization algorithm on an AMAX XG-4820IGK that contained two CPUs and eight GPUs. Table 1 shows the main hardware resources in our experiments.

4.1. Parallel Execution Time. Each iteration of the heterogeneous parallel LU factorization algorithm based on basic block uniform allocation includes three section computation times. For the k -th iteration of the algorithm, the first section is factorization of the current column block run by the current process to form (3); in the second section, the current process broadcasts related data obtained from the first section to all processes; and in the third section, all processes swap corresponding rows and compute the corresponding column blocks based on (6) and (7). The smaller the size of the column block is, the lower the execution time of the first section is. However, a smaller column block means more iterations n , which will lead to an execution time increase. Hence, the size of the column block is neither too large nor too small. For different performance processors, the size of the column block will be different to achieve better parallel performance. Figure 3 shows the parallel execution time of a 16-core system when the size of the column block changes from 4 to 256. For a 16-core system, the algorithm has better performance when the size of the block is between 8 and 32, and the best

FIGURE 4: Parallel execution time of an 8-GPU system related to different column block sizes.

size is approximately 8 for $N=8192$, 12288, and 16384. Figure 4 shows the parallel execution time of an 8-GPU system related to the sizes of the column block. For an 8-GPU system, the algorithm has better performance when the size of the block is between 64 and 128 for $N=16384$, 24576, and 32768.

Parallel execution time of the algorithm is related to the load balancing strategy, synchronization, communication, etc. We compare our algorithm with the two representative LU factorization algorithms, “`magma_dgetrf`” and “`magma_dgetrf_mgpu`” in MAGMA. The `magma_dgetrf` is a hybrid CPU/GPU routine where the matrix is initially in the CPU host memory, and the `magma_dgetrf_mgpu` is a hybrid CPU /multiple-GPU routine where the matrix is distributed across multiple GPUs' device memories. Both of them are the right-looking Level 3 BLAS version of the algorithm to compute an LU factorization of a general $M \times N$ matrix A using partial pivoting with row interchanges.

Figure 5 shows the parallel performance comparison of our algorithm with `magma_dgetrf` and `magma_dgetrf_mgpu` for different size matrices in different computational capability systems. The parallel execution time gradually decreases when the GPU is added, from 1 GPU to 8 GPUs for a larger matrix. However, the parallel execution time does not change evidently for `magma_dgetrf_mgpu` and our algorithm when the matrix size is smaller, such as $N=8192$, and even increases evidently for `magma_dgetrf` when the matrix size is less than 16384. The main reason is that the cost of the data transfer between the CPU and GPU accounts for a considerable proportion of the parallel execution time for a smaller matrix that does not have sufficient computational capability. However, when the matrix size is larger than 32768, the cost of data transfer, compared with computation time, is trivial, and the runtime is reduced obviously when the GPU is added. However, the algorithm `magma_dgetrf_mgpu` returns the error “cannot allocate memory on GPU device” when the size of the matrix is greater than 32768, and only 1 GPU is used, as shown in Figure 5(d).

TABLE 1: Experiment Environment.

	CPU	GPU
Processor type	Intel Xeon E5-2620 v4	NVIDIA Titan XP
Frequency	2.1 GHz	1582 MHz
Processors' number	2	8
Cores' number	8 Cores /CPU	3840 Cores /GPU
Memory	16GB*8	12G /GPU

FIGURE 5: Parallel performance comparison of three heterogeneous parallel block LU algorithms.

4.2. Synchronization and Load Balancing. Because of the triangular decrease in computing tasks, we use static basic column block cyclic uniform allocation strategy to achieve load balancing. Since the first step of every iteration is run by a single processor and then the operation is broadcasting related data, the wait time would be considerable for a system with more processors. The right-looking ahead strategy can be used for CPU-GPU system to reduce the processors' wait time further. To evaluate the wait time for synchronization

and the load balancing, we define *Average idle ratio* as follows:

$$\text{Average idle ratio} = \frac{\text{Average idle time}}{\text{Parallel execution time}} \quad (9)$$

If an algorithm has a smaller *Average idle ratio*, it would have better load balancing performance and a lower wait time for synchronization. The best is the *Average idle ratio* equal to 0, which means no synchronization wait time and

FIGURE 6: The Average idle ratio in the system configured with 8 cores or 16 cores.

the best load balancing, and the worst is the Average idle ratio approach to 1.

In our experiment, we test our algorithm in different computing configurations. Figure 6 shows the smaller Average idle ratio of the algorithm in different multicore systems. The Average idle ratios are less than 10% for an 8-core configuration and less than 15% for a 16-core configuration. The Average idle ratios of the algorithm in the system configured with 8 cores or 16 cores decrease when the matrix sizes change from 8192 to 36384, and the matrix values are less than 5% when the size of the matrix is larger than 20480. The reason is that when the matrix is larger, the parallel execution will be larger and the synchronization wait time will be smaller and can be ignored. Obviously, the algorithm runs on a 16-core system with larger Average idle ratios than on an 8-core system. Generally, when the CPU cores increase, the parallel execution time will decrease, and the synchronization wait time will increase. In contrast with that for a multicore system, the Average idle ratio is larger for a multi-GPU system, as shown in Figure 7. Most of the Average idle ratios of the algorithm are more than 20% in the system configured with 2 GPUs, 4 GPUs, or 8 GPUs. Because of the powerful parallel computing capability of the GPU, the synchronization wait time accounts for a considerable proportion of the computing time even if the matrix is larger, such as $N=36384$. However, when a right-looking ahead strategy is introduced, the Average idle ratio of the algorithm decreases to below 10%.

The Average idle ratio also changes with the sizes of the column block. Figure 8 shows the relationship between the Average idle ratio and the sizes of the column block in an 8-core system and a 16-core system. Figure 8(a) shows that the Average idle ratio gradually increases when the sizes of the column block change from 4 to 256 in an 8-core system. Figure 8(b) shows the same rule in a 16-core system except with a smaller matrix, such as $N=8192$.

4.3. Communication Cost. At the k -th iteration, the operation on the data transfer is broadcasting the partial of the k -th

FIGURE 7: The Average idle ratio in the system configured with 2 GPUs, 4 GPUs, or 8 GPUs.

column block $(A_{kk} \cdots A_{nk})^T$ and the permutation vector P_k . The total transferred data of all iterations is approximately broadcasting a permutation vector P and half a matrix A . In our experiments, we test the ratio of the average communication time to parallel execution time, which is defined as follows:

$$\text{Communication ratio} = \frac{\text{Average communication time}}{\text{Parallel execution time}} \quad (10)$$

Figure 9 shows the communication ratio of the algorithm run on a system with an 8-core configuration and a 16-core configuration. Evidently, the communication ratio is larger for a 16-core system than for an 8-core system. When the matrix size changes from 8192 to 36864, the communication ratio decreases from 3.6% to below 1.5% for the 16-core system and 1.1% to below 0.5% for the 8-core system. The increase in the calculation is more than the increase in the communication followed by an increase in the matrix size. Compared with parallel execution time, the communication time is trivial for larger matrices in 8-core or 16-core systems. Figure 10 shows that the communication ratio decreases when the matrix size changes from 8192 to 36384 and increases when the system configuration changes from 2GPU or 4GPU to 8GPU for a given matrix. However, the communication ratio is greater than 4% even for a larger matrix in a multiple-GPU system because of the powerful parallel computing capability of GPUs.

4.4. Scalability. Scalability is often used to evaluate the capability of an algorithm to solve potentially larger problems and the acceleration of an algorithm to solve a specific problem when more computing resources are available. First, we fix a matrix size to test how this changes the parallel execution time when computing resources gradually increase. Then, we measure the parallel execution time by increasing the matrix size and adding computing resources to test the scalability.

FIGURE 8: The *Average idle ratio* related to the sizes of the column block in a multicore system.

FIGURE 9: The *communication ratio* related to the different matrix sizes in the system with 8 cores or 16 cores.

In Figure 11, the x -axis represents the computing resources from 1 GPU to 8 GPUs, the y -axis shows the sizes of a double precision matrix from 8192 to 36384, and the z -axis is the parallel execution time in seconds on a logarithmic scale. When the sizes are larger than 16384, the parallel execution time decreases evidently when the computing resources are added from 1 GPU to 8 GPUs for a larger matrix such as $N > 20480$. However, the parallel execution time has no evident decrease for a smaller matrix such as $N=8192$ because the workloads provided are insufficient for more GPUs to compute.

Theoretically, the calculation of the algorithm is directly proportional to the matrix size on a logarithmic scale. Figure 12 shows the parallel execution time of the algorithm when the computing resources change from 1 GPU to 8 GPUs and the matrix sizes change from 4096 to 32768 simultaneously. The curve follows a nearly straight line in a logarithmic scale when N is larger than 12288.

However, when the matrix size is smaller, such as 4096 or 8192, the curve does not follow this straight line. The reason could be that the smaller size matrix has insufficient calculations to take advantage of the GPU's computation capability.

5. Conclusions

Heterogeneous parallel computing has undergone great development in the past few years and will be increasingly important in future computing architectures. Traditional homogeneous computing is not a good choice for the heterogeneous system with GPUs because of the GPU's powerful computing capability and different architecture. Computer architecture is transitioning from the multicore era into the heterogeneous era, which poses new challenges for algorithm design and software optimization. In this paper, we present a parallel LU factorization based on a basic column block cyclic uniform allocation strategy for heterogeneous architectures. First, any given matrix is partitioned into different sizes of column blocks according to the different performance of the processors in the system. Then, the one-dimensional static basic column block distribution strategy is used to distribute the matrix data. Right-looking ahead technology is also used in the system configured with one CPU core to one GPU to diminish the wait time. The idle time is further minimized by optimized sizes and the numbers of column blocks. Experiments on parallel execution time show that the algorithm performance can compete with the related algorithm of MAGMA. The experimental results also show the good performance of the algorithm on scalability, load balancing, and communication cost for heterogeneous systems. The *Average idle ratios* of the algorithm in the multicore systems are less than 5%, and the *communication ratios* are less than 1.5% when the size of the matrix is larger. However, the *communication ratio* is greater than 4% even for larger matrices in a multiple-GPU system because of the powerful parallel computing capability of GPUs. Future research

FIGURE 10: The *communication ratio* related to the different matrix sizes in the system with 2 GPUs, 4 GPUs, or 8 GPUs.

FIGURE 11: Scalability of the heterogeneous parallel LU factorization algorithm.

FIGURE 12: Parallel execution time change for computing resources and matrix sizes.

should attempt to further diminish the communication cost and wait time for systems with more GPUs.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This research was funded by the National Natural Science Foundation of China (Grant no. 61871204), the National Natural Science Foundation of China (Grant no. 61703195),

the Natural Science Foundation of Fujian Province (Grant no. 2015J01659), the Fujian Provincial Leading Project (Grant no. 2017H0030), and the Natural Science Foundation of Fujian Province (Grant no. 2018J01544).

References

- [1] N. S. Kim, D. Chen, J. Xiong, and W.-M. W. Hwu, "Heterogeneous computing meets near-memory acceleration and high-level synthesis in the post-moore era," *IEEE Micro*, vol. 37, no. 4, pp. 10–18, 2017.
- [2] L. Zhang, K. Li, C. Li, and K. Li, "Bi-objective workflow scheduling of the energy consumption and reliability in heterogeneous computing systems," *Information Sciences*, vol. 379, pp. 241–256, 2015.
- [3] M. Zahran, "Heterogeneous computing: Here to stay," *Communications of the ACM*, vol. 60, no. 3, pp. 42–45, 2017.
- [4] J. Wang, T. Wang, Z. Yang, N. Mi, and B. Sheng, "eSplash: Efficient speculation in large scale heterogeneous computing systems," in *Proceedings of the IEEE 35th International Performance Computing and Communications Conference (IPCCC '16)*, pp. 1–8, Las Vegas, NV, USA, December 2016.
- [5] C. Gregg and K. Hazelwood, "Where is the data? Why you cannot debate CPU vs. GPU performance without the answer," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '11)*, pp. 134–144, Austin, TX, USA, April 2011.
- [6] V. W. Lee, P. Hammarlund, R. Singhal et al., "Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 451–460, 2010.
- [7] S. Mittal and J. S. Vetter, "A survey of methods for analyzing and improving GPU energy efficiency," *ACM Computing Surveys*, vol. 47, no. 2, pp. 19:1–19:23, 2015.
- [8] S. Mittal and J. S. Vetter, "A survey of CPU-GPU heterogeneous computing techniques," *ACM Computing Surveys*, vol. 47, no. 4, pp. 69:1–69:36, 2015.
- [9] P. Rogers and A. C. Fellow, "Heterogeneous system architecture overview," in *Proceedings of the IEEE Hot Chips 25 Symposium (HCS '13)*, pp. 1–41, Stanford, Calif, USA, August 2013.
- [10] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, "Gem5-gpu: A heterogeneous CPU-GPU simulator," *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 34–36, 2015.
- [11] G. Kan, T. Lei, K. Liang et al., "A multi-core CPU and many-core GPU based fast parallel shuffled complex evolution global optimization approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 2, pp. 332–344, 2017.
- [12] Z. Cheng, E. Shaffer, R. Yeh, G. Zagaris, and L. Olson, "Efficient parallel optimization of volume meshes on heterogeneous computing systems," *Engineering with Computers*, vol. 33, no. 4, pp. 717–726, 2017.
- [13] C. Chen, J. Fang, T. Tang, and C. Yang, "LU factorization on heterogeneous systems: an energy-efficient approach towards high performance," *Computing: Archives for Scientific Computing*, vol. 99, no. 8, pp. 791–811, 2017.
- [14] L. Grigori, S. Cayrols, and J. W. Demmel, "Low rank approximation of a sparse matrix based on LU factorization with column and row tournament pivoting," *SIAM Journal on Scientific Computing*, vol. 40, no. 2, pp. C181–C209, 2018.
- [15] T. Dong, A. Haidar, P. Luszczek, J. A. Harris, S. Tomov, and J. Dongarra, "LU factorization of small matrices: Accelerating batched DGETRF on the GPU," in *Proceedings of the 16th IEEE International Conference on High Performance Computing and Communications (HPCC '14), 11th IEEE International Conference on Embedded Software and Systems (ICESS '14) and 6th International Symposium on Cyberspace Safety and Security (CSS '14)*, pp. 157–160, August 2014.
- [16] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Computing. Systems & Applications*, vol. 35, no. 1, pp. 38–53, 2009.
- [17] E. Anderson, Z. Bai, C. Bischof et al., *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, 1999.
- [18] J. Choi, J. Demmel, I. Dhillon et al., "ScaLAPACK: a portable linear algebra library for distributed memory computers—design issues and performance," *Computer Physics Communications*, vol. 97, no. 1–2, pp. 1–15, 1996.
- [19] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parallel Computing*, vol. 36, no. 5–6, pp. 232–240, 2010.
- [20] NVIDIA, "DU-06702-001_v9.2 CUBLAS library user guide," Tech. Rep., 2018, Available: https://docs.nvidia.com/cuda/archive/9.2/pdf/CUBLAS_Library.pdf.
- [21] J. R. Humphrey, E. J. Kelmelis, D. K. Price, K. E. Spagnoli, A. L. Paolini, and E. J. Kelmelis, "CULA: hybrid GPU accelerated linear algebra routines," in *Proceedings of the SPIE Defense, Security, and Sensing*, p. 770502, Orlando, FL, USA, 2010.
- [22] V. Volkov, J. W. Demmel, and LU, "QR and Cholesky factorizations using vector capabilities of GPUs," Technical Report UCB/EECS-2008-49, EECS Department, University of California, Berkeley, Calif, USA, 2008.
- [23] E. F. van de Velde, "Experiments with multicomputer LU-decomposition," *Concurrency: Practice and Experience*, vol. 2, no. 1, pp. 1–26, 1990.
- [24] J. Kurzak, P. Luszczek, M. Faverge, and J. Dongarra, "LU factorization with partial pivoting for a multicore system with accelerators," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 8, pp. 1613–1621, 2013.
- [25] S. Catalán S, R. Rodríguez-Sánchez, E. S. Quintana-Ortí et al., "Static versus dynamic task scheduling of the lu factorization on arm big. little architectures," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW '17)*, pp. 733–742, Lake Buena Vista, FL, USA, May 2017.
- [26] M. Deisher, M. Smelyanskiy, B. Nickerson, V. W. Lee, M. Chuvelev, and P. Dubey, "Designing and dynamically load balancing hybrid LU for multi/many-core," *Computer Science - Research and Development*, vol. 26, no. 3–4, pp. 211–220, 2011.
- [27] T. Endo, S. Matsuoka, A. Nukada, and N. Maruyama, "Linpack evaluation on a supercomputer with heterogeneous accelerators," in *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing (IPDPS '10)*, pp. 1–8, Atlanta, GA, USA, April 2010.
- [28] J. Choi, J. J. Dongarra, L. S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley, "Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines," *Scientific Programming*, vol. 5, no. 3, Article ID 483083, pp. 173–184, 1996.
- [29] F. Wang, C.-Q. Yang, Y.-F. Du, J. Chen, H.-Z. Yi, and W.-X. Xu, "Optimizing linpack benchmark on GPU-accelerated petascale

- supercomputer,” *Journal of Computer Science and Technology*, vol. 26, no. 5, pp. 854–865, 2011.
- [30] A. Lastovetsky and R. Reddy, “Data distribution for dense factorization on computers with memory heterogeneity,” *Parallel Computing. Systems & Applications*, vol. 33, no. 12, pp. 757–779, 2007.
- [31] R. Wu, “A heterogeneous parallel cholesky block factorization algorithm,” *IEEE Access*, vol. 6, pp. 14071–14077, 2018.
- [32] F. Dong, S. Tomov, and J. Dongarra, “Efficient support for matrix computations on heterogeneous multi-core and multi-GPU architectures,” Technical Report UT-CS-11-668, University of Tennessee Knoxville, Knoxville, TN, USA, 2011.

Submit your manuscripts at
www.hindawi.com