Performance Optimization Using Partitioned SpMV on GPUs and Multicore CPUs

Wangdong Yang, Kenli Li, Zeyao Mo, and Keqin Li, *Fellow, IEEE*

Abstract—This paper presents a sparse matrix partitioning strategy to improve the performance of SpMV on GPUs and multicore CPUs. This method has wide adaptability for different types of sparse matrices, and is different from existing methods which only adapt to some particular sparse matrices. In addition, our partitioning method can obtain dense blocks by analyzing the probability distribution of non-zero elements in a sparse matrix, and result in very low proportion of zero padded. We make the following significant contributions. (1) We present a partitioning strategy of sparse matrices based on probabilistic modeling of non-zero elements in a row. (2) We prove that our method has the highest mean density compared with other strategies according to certain given ratios of partition obtained from the computing powers of heterogeneous processors. (3) We develop a CPU-GPU hybrid parallel computing model for SpMV on GPUs and multicore CPUs in a heterogeneous computing platform. Our partitioning strategy has balanced load distribution and the performance of SpMV is significantly improved when a sparse matrix is partitioned into dense blocks using our method. The average performance improvement of our solution for SpMV is about 15.75 percent on multicore CPUs, compared to that of the other solutions. By considering the rows of a matrix in a unique order based on the probability mass function of the number of non-zeros in a row, the average performance improvement of our solution for SpMV is about 33.52 percent on GPUs and multicore CPUs of a heterogeneous computing platform, compared to that of the partitioning methods based on the original row order of a matrix.

Index Terms—GPU, matrix partition, multicore CPU, probability distribution, sparse matrix-vector multiplication

1 Introduction

1.1 Motivation

In recent years, accelerator-based computing using accelerators such as the IBM Cell SPUs [1], FPGAs, GPUs, and ASICs has achieved noticeable performance gain compared to CPUs [2]. 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. However, as CPU architectures also evolve and address challenges such as the power wall and the memory wall, and compete with accelerators, it is imperative that CPUs should also be included in the computations. It is further observed by Lee et al. [3] that for several irregular applications, including sparse matrix-vector multiplication (SpMV), multicore CPUs can provide comparable performance to that of GPUs. This suggests that such irregular applications can benefit from heterogeneous algorithms that run on a CPUs- and GPUs-based heterogeneous computing platform. For instance, Pedram et al. [4] have studied fundamental tradeoffs and limits in efficiency (as measured in energy per operation) that can be achieved for an important class of kernels, namely the level-3 Basic Linear Algebra Sub-routines (BLAS), and established a baseline by studying general matrix-matrix multiplication (GEMM) on a variety of custom and general-purpose CPU and GPU architectures.

SpMV is an essential operation in solving linear systems and partial differential equations [10]. For many scientific and engineering applications, the matrices can be very large and sparse, and these sparse matrices may have various sparsity characteristics. It is a challenging issue to adopt an appropriate algorithm to implement and optimize SpMV. Especially, for GPUs and multicore CPUs based heterogeneous computing platforms, how to make full use of the available computing resources to maximize the parallel computing ability is the key to improve performance of SpMV. First, reasonable partition of a matrix to match the characteristics of parallel computing is the basis of improving performance. Second, the load balance between multiple processors is the key to the use of parallel computing capability. The loads on the processors depend on the distribution of computing tasks. The scale of a computational task is determined by the size of the block to be processed. Therefore, an optimal partitioning strategy of sparse matrices is a key step for SpMV on GPUs and multicore CPUs based heterogeneous computing platforms.

1.2 Our Contributions

The present paper makes the following unique contributions to partitioning for SpMV on GPUs and multicore CPUs based heterogeneous computing platforms. (1) We present a partitioning strategy of sparse matrices based on probabilistic modeling of non-zero elements in a row. (2) We prove that our method has the highest mean density

- *W. Yang and K. Li are with the College of Information Science and Engineering, Hunan University, Changsha, Hunan 410008, China, and are also with the National Supercomputing Center in Changsha, Changsha, Hunan 410008, China. E-mail: yangwangdong@163.com, lkl@hnu.edu.cn.*
- *Z. Mo is with the Institute of Applied Physics and Computational Mathematics, Beijing 100094, China. E-mail: zeyao_mo@iapcm.ac.cn.*
- *K. Li is with the College of Information Science and Engineering, Hunan University, Changsha, Hunan 410008, China, with the National Supercomputing Center in Changsha, Changsha, Hunan 410008, China, and is also with the Department of Computer Science, State University of New York, New Paltz, NY 12561, USA. E-mail: lik@newpaltz.edu.*

Manuscript received 1 Feb. 2014; revised 7 Sept. 2014; accepted 18 Oct. 2014. Date of publication 23 Nov. 2014; date of current version 12 Aug. 2015. Recommended for acceptance by R. F. DeMara.
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/TC.2014.2366731

compared with other strategies according to certain given ratios of partition obtained from the computing powers of heterogeneous processors. (3) We develop a CPU-GPU hybrid parallel computing model for SpMV on GPUs and multicore CPUs in a heterogeneous computing platform.

Our partitioning strategy is based on a probability mass function (PMF), which characterizes the distribution of non-zero elements in a sparse matrix. Our matrix partitioning and performance optimization strategy consists of three steps. First, the PMF of the target matrix is built according to the analysis of the distribution of non-zero elements per row. Second, the sparse matrix is partitioned into row vectors sets (RVS's) according to the ratios obtained from the heterogeneous computing powers. Lastly, these RVS's are assigned to GPUs and multicore CPUs to execute SpMV in a CPU-GPU hybrid parallel computing environment.

We use a probabilistic method to partition a sparse matrix. Our matrix partitioning algorithm considers the rows of a matrix in a unique order based on the probability mass function of the number of non-zeros in a row. This method has wide adaptability for different types of sparse matrices and is different from existing methods which only adapt to some particular sparse matrices. Some methods also use some distribution characteristics of a sparse matrix to partition, such as the number of non-zeros. However, these methods lack of quantitative techniques and analytical results. On the contrary, we rigorously prove the optimality of our partitioning method compared with any other algorithms.

In this paper, we use SpMV CUDA kernels developed by NVIDIA [5] on NVIDIA GTX 460 GPU and MKL BLAS functions developed by Intel [6] on Intel Xeon E5506 CPU for our experiments. According to our extensive experiments on 10 representative matrices (totally 50 tested cases), the performance improvement of our algorithm is very effective and noticeable. The average performance improvement of our solution for SpMV is about 12.76 percent on multicore CPUs for 10 representative matrices, compared to that of the automatic solution provided by Intel MKL lib, and that is about 15.75 percent for all tested cases. The average performance improvement of our solution for SpMV is about 33.19 percent on GPUs and multicore CPUs of a heterogeneous computing platform for 10 representative matrices, compared to that of the partitioning methods based on the original row order of a matrix, and that is about 33.52 percent for all tested cases.

This paper extends our previous work [7], [34]. The current paper includes a refined formalization for sparse matrices and a new partitioning strategy for SpMV analysis and optimization. The proposed approach in this paper is general, which is not limited by any specific GPU-CPU architecture, because it is based on a mathematical and analytical model. For different multicore and manycore architectures, only the benchmarks need to be performed to get performance parameters, and the algorithms are the same.

The remainder of the paper is organized as follows. In Section 2, we review related research on SpMV. In Section 3, we present a probabilistic model for sparse matrices. In Section 4, we develop our partitioning strategies. In Section 5, we analyze the optimality of our partitioning algorithm and the execution time of SpMV on GPUs and multicore CPUs. In Section 6, we describe our GPU and CPU hybrid parallel computing method for SpMV. In Section 7, we demonstrate our extensive experimental performance comparison results. In Section 8, we conclude the paper.

2 Related Work

2.1 Implementation of SpMV on GPUs and Multicore CPUs

Bolz et al. [8] proposed one of the first SpMV CUDA [7] kernel implementations. Vazquez et al. [11], [13] proposed new implementations of SpMV for GPUs called ELLR-T, to achieve high performance on GPUs. Williams et al. [14] presented several optimization strategies especially effective for the multicore environments, and demonstrated significant performance improvement compared to existing state-of-the-art sequential and parallel SpMV implementations. Lee et al. [3] discussed optimization techniques for both CPUs and GPUs, and analyzed the architecture features that contribute to performance difference between the two architectures. Boyer et al. [15] proposed a new SpMV library providing good results on Z/mZ rings to attain this efficiency. It has been mandatory to augment the complexity of the SpMV algorithms, since OpenMP and CUDA provided multi-GPU and hybrid GPU/CPU implementations. Buluc et al. [16] explored two techniques (bitmasked register blocks and symmetry) that tap into the unused computational capability and used them to reduce the memory bandwidth requirements for sparse-matrix vector multiplication on two state-of-the-art multicore processors. Sun et al. [17] presented a scalable and efficient FPGA-based SMVM architecture which can handle arbitrary matrix sparsity patterns without excessive preprocessing or zero padding and can be dynamically expanded based on the available I/O bandwidth. Pichel and Rivera [18] studied the behavior of an important irregular application such as SpMV on the Single-Chip Cloud Computer (SCC) processor in terms of performance and power efficiency and an architectural comparison of the SCC processor with several leading multicore processors and GPUs, including the new Intel Xeon Phi coprocessor.

2.2 Partition Strategies of Sparse Matrices for SpMV

For uneven distribution of non-zero elements in a sparse matrix, some partition formats were proposed to improve the load balance for SpMV on GPUs and multicore of CPUs, such as blocked compressed sparse row (BCSR) format [19], row-grouped CSR (GCSR) format [20], blocked ELLPACK (BELLPACK) format [21], sliced coordinate (SCOO) format [14], sliced ELLPACK (SELLPACK) format [23], fixed scale blocked format [24], Segmented Interleave Combination (SIC) format [25], compressed sparse blocks (CSB) [16], padded jagged diagonals storage (pJDS) format [26], doubly separated block diagonal (DSBD) format [27], and Compressed Sparse eXtended (CSX) format [28].

In addition, a sparse matrix must be partitioned into small blocks to be assigned to multi-GPUs and multicore CPUs, because they are too big to be computed or the performance is poor on one processor. For the characteristics of a sparse matrix and the configuration of a computing environment, many partitioning strategies were proposed.

Usually there are two basic types of strategies for partitioning a sparse matrix, which are based on rows and submatrices. The strategies based on rows further include two types. In the first type, a sparse matrix is partitioned into some blocks according to the number of rows. In the second type, a sparse matrix is partitioned into some blocks according to the numbers of non-zeros (NNZ) of rows. Both types do not split a row into different blocks. A row may be split into different submatrices using the strategies based on submatrices, leading to the need of accumulation of computing results from different threads. Other methods are generally improvement and extension of these three basic strategies.

Some strategies partition a sparse matrix into BCSR, GCSR, BELLPACK, sliced COO, and sliced ELLPACK using the strategy based on rows. Some strategies partition sparse matrices according to the characteristics of a sparse matrix, such as the proportion of non-zero elements of the sparse matrix [18], the compression effect of different formats [28], and dense block in the sparse matrix [30]. Refs. [31] and [32] proposed to perform three-way split and multi-way split approaches to breaking down each matrix-vector product into a number of smaller size matrix-vector products respectively. Yzelman and Roose [33] proposed a new parallelization technique to obtain parallel efficiency of 90 percent, which is based on partition using hypergraph models. Refs. [31], [32], and [33] improved the partitioning strategy based on submatrices to achieve better distribution of parallel tasks. In order to optimize the size of the block, some strategies partition sparse matrices according to the configuration of a computing environment, such as the cache scale of processors [14], [25], the computing power of processors [15], [29], and data transmission bandwidth [12], [16].

The partition strategies based on rows are sensitive to sparsity distribution and cannot achieve satisfactory compression effect in most cases. The optimization strategies based on the configuration of a computing environment have poor generality. The partition strategies based on the characteristics of a sparse matrix need an effective analysis method to analyze quantitatively the sparse characteristics results in computing complexity. In addition, a row may be split into different blocks by some partition strategies based on submatrices, leading to the calculation of collective results, which adds extra time overhead. In addition, for GPU-CPU heterogeneous systems, the cross data access between different processors and the synthesis of calculations using the strategies based on submatrices are costly and lead to performance degradation of SpMV.

3 A Probabilistic Model

Sparse matrices arise from various application domains and their distribution patterns of non-zero elements can be very specific and diversified. Furthermore, for sophisticated scientific computations and engineering applications, the scale of a sparse matrix can increase significantly, and such a huge matrix should be split into blocks to be processed in parallel. Due to irregularity of the distribution of non-zero elements in a sparse matrix, it is hard to find a partitioning method for all types of sparse matrices. However, we can accurately describe the distribution pattern of a sparse matrix by a PMF, and get numerical characteristics of sparsity distribution by a probabilistic method. Then, a suitable partition of a sparse matrix can be found based on the PMF.

3.1 PMF of Sparse Matrices

The PMF of a sparse matrix has been defined in [34], which is given below in order to improve readability. Let A be a sparse matrix, which has N rows and M columns. The discrete random variable X represents the number of non-zeros in one row of A. The range of values of X is $\Omega_X = \{0, 1, 2, \ldots, M\}$. For each $i = 0, 1, 2, \ldots, M$, let $\{X = i\}$ represent the event that the value of X is i. In particular, $\{X = 0\}$ represents the event that there is no non-zero in one row. Define another set $B = \{b_0, b_1, b_2, \ldots, b_M\}$. Each b_i, $i = 0, 1, 2, \ldots, M$, represents the number of rows, each of which contains exactly i non-zeros.

A PMF is a function that gives the probability that a discrete random variable is exactly equal to some value [35]. For each $i = 0, 1, 2, \ldots, M$, $p_i = b_i/N$ is the probability of the event $\{X = i\}$. $p_k = 0$ means that a row with k non-zeros does not exist in the sparse matrix A.

Definition 1. *The PMF P of the discrete random variable X is mathematically characterized by the following expression:*

$$\begin{aligned} &P(X = i) = p_i = b_i/N,\ i = 0, 1, 2, \ldots, M,\ \textit{where} \\ &\quad (i)\ p_i \geq 0, i = 0, 1, 2, \ldots, M; \\ &\quad (ii) \sum_{i=0}^{M} p_i = \sum_{i=0}^{M} (b_i/N) = \frac{1}{N} \sum_{i=0}^{M} b_i = N/N = 1. \end{aligned} \tag{1}$$

When A is partitioned into blocks, each block is essentially a set of row vectors.

Definition 2. *A is split into K row vector sets (RVS), which are $A_1, A_2, \ldots, A_K$ and satisfy the following properties:*

$$A_i \cap A_j = \emptyset,\ i \neq j. \tag{2}$$

We call $(A_1, A_2, \ldots, A_K)$ a partition of A.

3.2 Numerical Characteristics of RVS

Let A_i be an RVS, which is split from A. Let r denote a row vector of A_i. Define $NNZ(r)$ to be the number of non-zero elements in r. The number of non-zero elements in A_i is

$$NNZ(A_i) = \sum_{r \in A_i} NNZ(r). \tag{3}$$

The width of A_i is defined as

$$W(A_i) = \max(NNZ(r) \mid r \in A_i). \tag{4}$$

Let $N(A_i) = |A_i|$ be the number of row vectors in A_i. The total number of elements in A_i as a dense matrix is

$$E(A_i) = N(A_i) \times W(A_i). \tag{5}$$

The density $D(A_i)$ of A_i is the proportion of the number of non-zero elements in all elements of A_i, expressed as

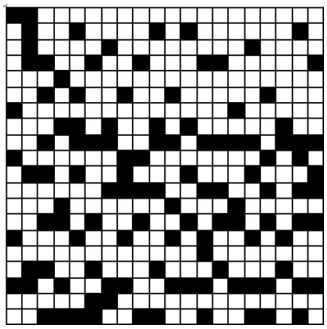


Fig. 1. A sparse matrix (black entry = non-zero element).

$$D(A_i) = \frac{NNZ(A_i)}{E(A_i)}. \tag{6}$$

Define $F(A_i)$ to be the proportion of filled zero in A_i, which is $F(A_i) = 1 - D(A_i)$.

Assume that a sparse matrix A is partitioned to K RVS's $A_1, A_2, \ldots, A_K$, where A_i is an $N(A_i) \times W(A_i)$ submatrix. The *mean density* of the partition $(A_1, A_2, \ldots, A_K)$ is

$$D(A_1, A_2, \ldots, A_K) = \frac{NNZ(A)}{E(A_1) + E(A_2) + \cdots + E(A_K)}, \tag{7}$$

where

$$NNZ(A) = \sum_{i=1}^{K} NNZ(A_i). \tag{8}$$

The *mean proportion of filled zero* is

$$F(A_1, A_2, \ldots, A_K) = 1 - D(A_1, A_2, \ldots, A_K). \tag{9}$$

4 Partitioning Algorithms

The partitioning of a sparse matrix based on probabilistic modeling has two steps, which are (1) establishing a probability model for the sparse matrix; (2) partitioning the sparse matrix according to the computing powers of processors.

4.1 A Probability Model

The number of non-zero elements in each row is examined and the information is stored. The number of rows with the same *NNZ* is saved in an array B, which has length $M + 1$, where $B[i]$ is the number of rows which have i non-zero elements, $i = 0, 1, 2, \ldots, M$.

For the sparse matrix A shown in Fig. 1, we have $N = M = 20$. The number of non-zero elements of each row in A, i.e., $Ap[1..N]$, is shown in Fig. 2. The PMF of the random variable X, i.e., $(p_0, p_1, \ldots, p_M)$, is shown in Fig. 3. A sample partition of A includes 9 RVS's, which are $B_1 = \{5, 16\}$, $B_2 = \{1, 7, 19\}$, $B_3 = \{3, 13\}$, $B_5 = \{2, 6, 10\}$, $B_6 = \{8, 17\}$, $B_7 = \{4, 11, 15\}$, $B_9 = \{12\}$, $B_{10} = \{14, 20\}$, and $B_{13} = \{9, 18\}$, where all row vectors in the same RVS have the same *NNZ*. Notice that $NNZ(A) = 117$, and the mean density is $117/(20 \times 13) = 117/260 = 45\%$.

4.2 Partitioning Strategies

Assume that there are K processors, whose computing powers are $CP_1, CP_2, \ldots, CP_K$ respectively. The total computing power is calculated as

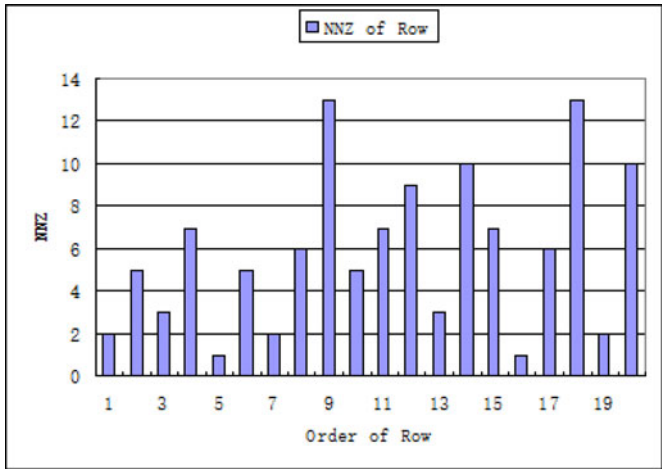


Fig. 2. The *NNZ* of each row.

$$CP = \sum_{i=1}^{K} CP_i. \tag{10}$$

How to split the SpMV into K tasks to be assigned to the K processors? It is clear that the scale of a task assigned to a processor is measured by the number of non-zero elements in the RVS, which should be linearly proportional to the computing power of the processor.

In this section, we present four sparse matrix partitioning algorithms. The first algorithm is based on the number of rows. The second algorithms is based on the *NNZ* of the rows. Both algorithms split a sparse matrix A into RVS's according to the original order of rows in A. Our third algorithm does not follow the original order of rows, but splits A according to its PMF. The fourth algorithm is based on submatrices.

4.2.1 Partitioning Based on the Number of Rows

Algorithm 1 splits a sparse matrix A into K RVS's in such a way that the number of rows in an RVS A_i is linearly proportional to the computing power CP_i of a processor (line 7). All the row vectors are considered in their original order in A (lines 8 and 12). However, Algorithm 1 does not consider the *NNZ* of the rows, i.e., $NNZ(A_i)$ may not be linearly proportional to the computing power CP_i. This may result in unbalanced load distribution. Some simple strategies such as BCSR usually use Algorithm 1.

Let us assume that there are $K = 3$ processors and the computing powers are $CP_1 = 1$, $CP_2 = 2$, and $CP_3 = 6$, with the ratios of three partitions being 11, 22, and 67 percent. For the sparse matrix A in Fig. 1, by using Algorithm 1, the three RVS's are $A_1 = \{1, 2\}$, $A_2 = \{3, 4, 5, 6\}$, and $A_3 = \{7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20\}$, with $N(A_1) = 2$, $W(A_1) = 5$, $N(A_2) = 4$, $W(A_2) = 7$, $N(A_3) = 14$, $W(A_3) = 13$, The *NNZ*'s of the three RVS's are $NNZ\ (A_1) = 7$, $NNZ(A_2) = 16$, and $NNZ(A_3) = 94$, with percentages of total *NNZ* 6, 14, and 80 percent respectively. The density of these RVS are $D(A_1) = 7/(2 \times 5) = 70\%$, $D(A_2) = 16/(4 \times 7) = 57\%$, and $D(A_3) = 94/(14 \times 13) = 52\%$ respectively, and the mean density is $D(A_1, A_2, A_3) = 117/(10 + 28 + 182) = 117/220 = 53\%$. The numbers of zero padded are 3, 12, and 88 respectively if these RVS are stored using the

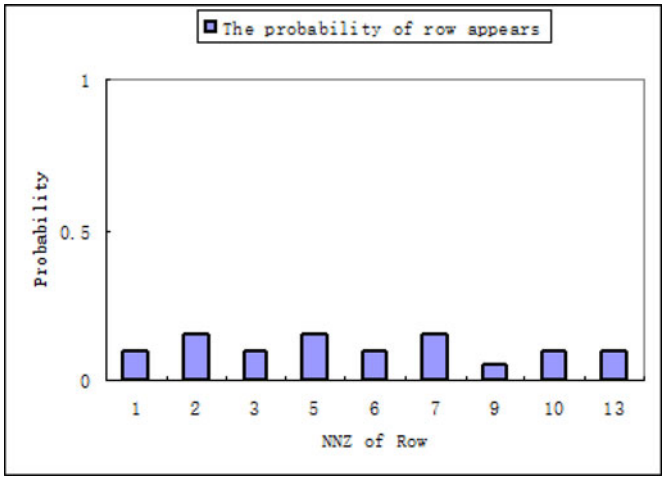


Fig. 3. The probability mass function ($p_i \neq 0$ is shown).

ELL format. The total number of zero padded is $3 + 12 + 88 = 103$, and the mean proportion of filled zero is $F(A_1, A_2, A_3) = 103/220 = 47\%$.

Algorithm 1. Partitioning based on the number of rows

Require: The row vectors of A, i.e., $Av[1..N]$; the K computing powers $CP_1, CP_2, \ldots, CP_K$.
Ensure: The K RVS's, i.e., $A_1, A_2, \ldots, A_K$.
1: **for** each $i \in [1..K]$ **do**
2: $A_i \leftarrow \emptyset$;
3: **end for**
4: $i \leftarrow 1$;
5: $index \leftarrow 0$;
6: **while** $(i \leq K)$ **do**
7: $row \leftarrow (CP_i/CP) \times N$;
8: **for** each $j \in [1..row]$ **do**
9: $A_i \leftarrow A_i \cup \{Av[index + j]\}$;
10: **end for**
11: $i \leftarrow i + 1$;
12: $index \leftarrow index + row$;
13: **end while**
14: **return** $A_1, A_2, \ldots, A_K$.

4.2.2 Partitioning Based on NNZ

Algorithm 2 splits a sparse matrix A into K RVS's in such a way that $NNZ(A_i)$ is linearly proportional to the computing power CP_i of a processor (line 7). All the row vectors are considered in their original order in A (lines 5 and 12). Some traditional partitioning strategies usually use Algorithm 2, such as BELLPACK, sliced COO, and sliced ELLPACK.

For the sparse matrix A in Fig. 1, by using Algorithm 2, the three RVS's are $\{1, 2, 3\}$, $\{4, 5, 6, 7, 8\}$, and $\{9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20\}$. The NNZ's of the three RVS's are 10, 21, and 86, with percentages of total NNZ 9, 18, and 73 percent respectively. The density of these RVS are $10/(3 \times 5) = 67\%$, $21/(5 \times 7) = 60\%$, and $86/(12 \times 13) = 55\%$ respectively, and the mean density is $117/(15 + 35 + 156) = 117/206 = 57\%$, higher than that of Algorithm 1. The numbers of zero padded are 5, 14, and 70 respectively if these RVS are stored using the ELL format. The total number of zero padded is $5 + 14 + 70 = 89$, and the mean proportion of filled zero is $89/206 = 43\%$, lower than that of Algorithm 1.

Algorithm 2. Partitioning based on NNZ

Require: The row vectors of A, i.e., $Av[1..N]$; the number of non-zeros NNZ; the number of non-zero elements of each row in A, i.e., $Ap[1..N]$; the K computing powers $CP_1, CP_2, \ldots, CP_K$.
Ensure: The K RVS's, i.e., $A_1, A_2, \ldots, A_K$.
1: **for** each $i \in [1..K]$ **do**
2: $A_i \leftarrow \emptyset$;
3: **end for**
4: $i \leftarrow 1$;
5: $j \leftarrow 1$;
6: **while** $(i \leq K)$ **do**
7: $NNZ_C \leftarrow (CP_i/CP) \times NNZ$;
8: $NNZ_T \leftarrow 0$;
9: **while** $(NNZ_T < NNZ_C)$ **do**
10: $NNZ_T \leftarrow NNZ_T + Ap[j]$;
11: $A_i \leftarrow A_i \cup \{Av[j]\}$;
12: $j \leftarrow j + 1$;
13: **end while**
14: $i \leftarrow i + 1$;
15: **end while**
16: **return** $A_1, A_2, \ldots, A_K$.

4.2.3 Partitioning Based on PMF

For a sparse matrix, only the non-zero elements need to be computed. So it is more reasonable that the computing scale of the sparse matrix is calculated in accordance with NNZ. The ELL format of a sparse matrix is suitable to be computed on GPU and the performance of ELL is usually better than CSR and COO, because the ELL format is well-suited to vector architectures. The NNZ of rows in the block partitioned from the sparse matrix by Algorithms 2 may have large deviation, resulting in decline in performance of ELL. Algorithm 3 attempts to improve Algorithm 2 based on the PMF of a sparse matrix. Assume that A is split into $B_1, B_2, \ldots, B_q$, where all row vectors in the same B_i have the same NNZ. Furthermore, we have $W(B_1) < W(B_2) < \cdots < W(B_q)$. Algorithm 3 generates K RVS's, i.e., $A_1, A_2, \ldots, A_K$, based on $B_1, B_2, \ldots, B_q$. Notice that the row vectors are no longer considered in their original order in A, but rather in the order of $B_1, B_2, \ldots, B_q$. The strongest feature of Algorithm 3 is that row vectors with the same or similar NNZ are grouped together, thus increasing the density of the RVS's and reducing the zeros padded.

For the sparse matrix A in Fig. 1, by using Algorithm 3, the three RVS's are $\{1, 3, 5, 7, 13, 16, 19\}$, $\{2, 6, 8, 10, 17\}$, and $\{4, 9, 11, 12, 14, 15, 18, 20\}$. The NNZ's of the three RVS's are 14, 27, and 76, with percentages of total NNZ 12, 23, and 65 percent respectively, very close to the required 11, 22, and 67 percent. The density of these RVS are $14/(7 \times 3) = 67\%$, $27/(5 \times 6) = 90\%$, and $76/(8 \times 13) = 73\%$ respectively, and the mean density is $117/(21 + 30 + 104) = 117/155 = 75\%$, much higher than that of Algorithm 2. The numbers of zero padded are 7, 3, and 28 respectively if these RVS are stored using the ELL format. The total number of zero padded is $7 + 3 + 28 = 38$, and the mean proportion of filled zero is $38/155 = 25\%$, much lower than that of Algorithm 2.

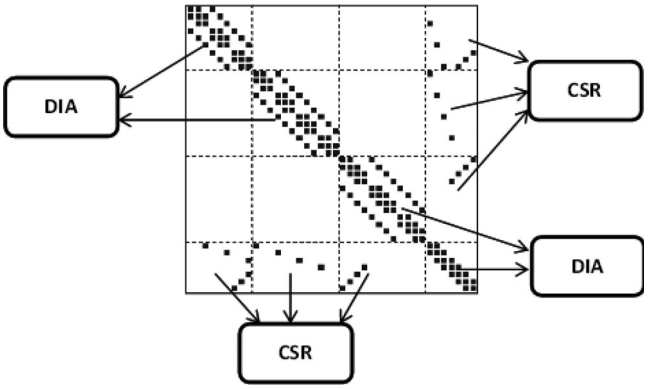


Fig. 4. Partitioning based on submatrices along the main diagonal.

Algorithm 3. Partitioning based on PMF

Require: The number of rows of the sparse matrix, N; the row vectors of A, i.e., $Av[1..N]$; the number of non-zeros NNZ; the RVS's by PMF, i.e., $B_1, B_2, \ldots, B_q$; the probability of $B_1, B_2, \ldots, B_q$, i.e., $p_1, p_2, \ldots, p_q$; the K computing powers $CP_1, CP_2, \ldots, CP_K$.
Ensure: The K RVS's, i.e., $A_1, A_2, \ldots, A_K$.

```
for each i ∈ [1..K] do
    A_i ← ∅;
end for
i ← 1;
j ← 1;
while (i ≤ K) do
    NNZ_C ← (CP_i/CP) × NNZ;
    NNZ_T ← 0;
    while (NNZ_T < NNZ_C) do
        if j × p_j × N ≤ (NNZ_C − NNZ_T) then
            A_i ← A_i ∪ B_j;
            NNZ_T ← NNZ_T + j × p_j × N;
            j ← j + 1;
        else
            row ← ⌈(NNZ_C − NNZ_T)/j⌉;
            B'_j ← the first row vectors from B_j;
            A_i ← A_i ∪ B'_j;
            B_j ← B_j − B'_j;
            p_j ← p_j − row/N;
            NNZ_T ← NNZ_T + row × j;
        end if
    end while
    i ← i + 1;
end while
return A_1, A_2, ..., A_K.
```

4.2.4 Partitioning Based on Submatrices

The DIA format is used usually for the diagonal-like matrices. However, for most diagonal-like matrices with some non-zeros outside the diagonal, the compression effect of DIA will be deteriorating and lead to the decrease of the computing performance. So for the diagonal-like matrices, some blocks are split along the main diagonal from the sparse matrix using the submatrix-based algorithms. We improve the partitioning strategies based on submatrices to adapt to the GPUs and CPUs heterogeneous environment, which is shown Algorithm 4. The blocks set $R1$ is stored by the DIA format and $R2$ is stored by the CSR format, as shown in Fig. 4. $BLOCK_{SIZE}$ usually takes 32, because the number of threads in a warp is 32, and can also take 16, because the half warp can be scheduled for GPU. For Test 1 in Section 7.2.1, $R1$ and $R2$ are assigned into multicore CPUs to be processed. For Test 2 in Section 7.2.2, $R1$ is assigned into GPUs to be processed and $R2$ is assigned into multicore CPUs.

Algorithm 4. Partitioning based on submatrices

Require: The row vectors of A, i.e., $Av[1..N]$; the size of block, $BLOCK_{SIZE}$.
Ensure: The blocks set along the main diagonal, $R1$; the blocks set outside the main diagonal, $R2$.

```
R1 ← ∅;
R2 ← ∅;
for each i ∈ [1..N] do
    r_i ← the row index of Av[i];
    c_i ← the column index of Av[i];
    if c_i > (r_i mod BLOCK_SIZE) × BLOCK_SIZE and
       c_i < (r_i mod BLOCK_SIZE + 1) × BLOCK_SIZE then
        R1 ← R1 ∪ {Av[i]};
    else
        R2 ← R2 ∪ {Av[i]};
    end if
end for
store R1 by DIA;
store R2 by CSR;
return R1, R2.
```

The partitioning strategy is tested and compared to Algorithms 1, 2, 3. For Test 1 in Section 7.2.1, the performance of the partitioning strategy is poorer than that of Algorithms 2 and 3, and is better than that of Algorithm 1 for the sparse matrices with very obvious diagonal feature. However, the performance of the partitioning strategy is poorer for irregular matrices, because the DIA format is more zero-padded. For Test 2 in Section 7.2.2, the performance of the partitioning strategy is poorer than that of Algorithm 3, and is better than that of Algorithm 1 for small number of sparse matrices with very obvious diagonal feature, and there are only two sparse matrices in all tested cases, for which the performance is better than that of Algorithm 2.

5 Performance Analysis

In this section, we analyze the optimality of the partitioning algorithm based on PMF. We also analyze the execution time of SpMV on GPUs and multicore CPUs.

5.1 Optimality Analysis

The effect of a partitioning method determines two important factors, i.e., the uniformity and the compression ratio of the RVS's. Increased density of the RVS's will improve the uniformity, while reduced zero padded will improve the compression ratio of the sparse matrix.

Let the total number of elements in a partition $(A_1, A_2, \ldots, A_K)$ be

$$E(A_1, A_2, \ldots, A_K) = E(A_1) + E(A_2) + \cdots + E(A_K). \quad (11)$$

The following theorem shows that Algorithm 3 always gives the minimum $E(A_1, A_2, \ldots, A_K)$, for all partitions with identical NNZ's.

Theorem 1. *Assume that a sparse matrix A is partitioned into K RVS's $A'_1, A'_2, \ldots, A'_K$ according to certain method, and into $A_1, A_2, \ldots, A_K$ by Algorithm 3, where $NNZ(A'_i) = NNZ(A_i)$, for all $i = 1, 2, \ldots, K$. Then, we have $E(A_1, A_2, \ldots, A_K) \leq E(A'_1, A'_2, \ldots, A'_K)$.*

Proof. Let $A_{[i,j]}$ represent the union $A_i \cup \cdots \cup A_j$. Assume that $(A'_1, A'_2, \ldots, A'_K) \neq (A_1, A_2, \ldots, A_K)$. This means that there exists i, $1 \leq i < K$, such that $A'_{[1,i]} \neq A_{[1,i]}$ and $A'_{[i+1,K]} \neq A_{[i+1,K]}$. For example, i can be the smallest index such that $A'_i \neq A_i$. Notice that $NNZ(A'_{[1,i]}) = NNZ(A_{[1,i]})$ and $NNZ(A'_{[i+1,K]}) = NNZ(A_{[i+1,K]})$. Furthermore, we have $W(A'_{[i+1,K]}) = W(A_{[i+1,K]})$, because $W(A'_{[i+1,K]}) = \max(NNZ(r) \mid r \in A) = W(A_{[i+1,K]})$.

Let $C = A'_{[1,i]} \cap A_{[i+1,K]}$, and $C' = A'_{[i+1,K]} \cap A_{[1,i]}$. It is clear that $C \neq \emptyset$ and $C' \neq \emptyset$. Let $B = A'_{[1,i]} \cap A_{[1,i]}$. Since

$$A'_{[1,i]} = B \cup C,\ A_{[1,i]} = B \cup C',$$

we have

$$NNZ(A'_{[1,i]}) = NNZ(B) + NNZ(C),$$

and

$$NNZ(A_{[1,i]}) = NNZ(B) + NNZ(C').$$

By $NNZ(A'_{[1,i]}) = NNZ(A_{[1,i]})$, we get $NNZ(C) = NNZ(C')$. Since $W(A_1) \leq W(A_2) \leq \cdots \leq W(A_K)$, we know that $W(C) \geq W(C')$ and $N(C) \leq N(C')$.

Notice that $A'_{[1,i]} = A_{[1,i]} \cup C - C'$, which implies that

$$N(A'_{[1,i]}) = N(A_{[1,i]}) + N(C) - N(C'),$$

and

$$W(A'_{[1,i]}) = W(A_{[1,i]} \cup C - C') = W(C) \geq W(A_{[1,i]}).$$

Hence, we obtain

$$\begin{aligned} E(A'_{[1,i]}) &= W(A'_{[1,i]}) \times N(A'_{[1,i]}) \\ &= W(A'_{[1,i]}) \times (N(A_{[1,i]}) + N(C) - N(C')) \\ &= W(C) \times (N(A_{[1,i]}) + N(C) - N(C')) \\ &\geq W(A_{[1,i]}) \times N(A_{[1,i]}) + W(C) \times (N(C) - N(C')) \\ &= E(A_{[1,i]}) + W(C) \times (N(C) - N(C')). \end{aligned}$$

Also, notice that $A'_{[i+1,K]} = A_{[i+1,K]} \cup C' - C$, which implies that

$$N(A'_{[i+1,K]}) = N(A_{[i+1,K]}) + N(C') - N(C),$$

and

$$\begin{aligned} E(A'_{[i+1,K]}) &= W(A'_{[i+1,K]}) \times N(A'_{[i+1,K]}) \\ &= W(A_{[i+1,K]}) \times (N(A_{[i+1,K]}) + N(C') - N(C)) \\ &= E(A_{[i+1,K]}) + W(A_{[i+1,K]}) \times (N(C') - N(C)). \end{aligned}$$

Finally, we have

$$\begin{aligned} &E(A'_1, A'_2, \ldots, A'_K) = E(A'_{[1,i]}) + E(A'_{[i+1,K]}) \\ &\geq E(A_{[1,i]}) + W(C) \times (N(C) - N(C')) \\ &\quad + E(A_{[i+1,K]}) + W(A_{[i+1,K]}) \times (N(C') - N(C)) \\ &= E(A_1, A_2, \ldots, A_K) \\ &\quad + (W(A_{[i+1,K]}) - W(C)) \times (N(C') - N(C)) \\ &\geq E(A_1, A_2, \ldots, A_K), \end{aligned}$$

where we notice that $W(A_{[i+1,K]}) \geq W(C)$. □

The following corollary claims that Algorithm 3 always gives the highest mean density.

Corollary 1. *Assume that a sparse matrix A is partitioned into K RVS's $A'_1, A'_2, \ldots, A'_K$ according to certain method, and into $A_1, A_2, \ldots, A_K$ by Algorithm 3, where $NNZ(A'_i) = NNZ(A_i)$, for all $i = 1, 2, \ldots, K$. Then, we have $D(A_1, A_2, \ldots, A_K) \geq D(A'_1, A'_2, \ldots, A'_K)$.*

Proof. According to Eq. (7), we have

$$D(A_1, A_2, \ldots, A_K) = \frac{NNZ(A)}{E(A_1, A_2, \ldots, A_K)},$$

and

$$D(A'_1, A'_2, \ldots, A'_K) = \frac{NNZ(A)}{E(A'_1, A'_2, \ldots, A'_K)}.$$

According to Theorem 1, we have $E(A_1, A_2, \ldots, A_K) \leq E(A'_1, A'_2, \ldots, A'_K)$, which implies that $D(A_1, A_2, \ldots, A_K) \geq D(A'_1, A'_2, \ldots, A'_K)$. □

The following corollary claims that Algorithm 3 always gives the lowest mean proportion of filled zero.

Corollary 2. *Assume that a sparse matrix A is partitioned into K RVS's $A'_1, A'_2, \ldots, A'_K$ according to certain method, and into $A_1, A_2, \ldots, A_K$ by Algorithm 3, where $NNZ(A'_i) = NNZ(A_i)$, for all $i = 1, 2, \ldots, K$. Then, we have $F(A_1, A_2, \ldots, A_K) \leq F(A'_1, A'_2, \ldots, A'_K)$.*

Proof. According to Eq. (9), we have

$$F(A_1, A_2, \ldots, A_K) = 1 - D(A_1, A_2, \ldots, A_K),$$

and

$$F(A'_1, A'_2, \ldots, A'_K) = 1 - D(A'_1, A'_2, \ldots, A'_K).$$

According to Corollary 1, we have $D(A_1, A_2, \ldots, A_K) \geq D(A'_1, A'_2, \ldots, A'_K)$, which implies that $F(A_1, A_2, \ldots, A_K) \leq F(A'_1, A'_2, \ldots, A'_K)$. □

5.2 Execution Time Analysis

In order to implement SpMV in such a way that it can be computed in parallel on GPUs and CPUs, a big sparse matrix should be partitioned into multiple RVS's to be computed on CPUs and/or GPUs respectively. According to the ratios of computing powers of processors, a sparse matrix A is partitioned to K RVS's $A_1, A_2, \ldots, A_K$. First, in order to achieve load balance, the computing scale of an RVS should match

the computing power of a processor, and the computing scale of an RVS depends on the *NNZ* of the RVS. The load balance using Algorithms 2 and 3 is better than that using Algorithm 1, because the RVS's using Algorithms 2 and 3 are generated according to *NNZ*, while the RVS's using Algorithm 1 are produced according to the number of rows. Second, in order to improve the utilization of computing resources, the densities of the RVS's should be increased.

If the RVS's are processed on a GPU, the computing time of SpMV includes two parts, i.e., the time T_t of data transmission between GPU and CPU, and the execution time T_e on the GPU. Due to the features of GPU with a thread grid, the ELL format with row vectors of equal length has better performance than that of the COO and CSR formats for SpMV. The size of data put into the GPU is $\sum_{i=1}^{K} E(A_i)$, if the sparse matrix is stored by the ELL format. Assume that the storage space of RVS A_i is $DS(A_i)$ using ELL, which can be calculated by Eq. (12):

$$DS(A_i) = E(A_i) \times (S_s + S_i) + N(A_i) \times S_i, \tag{12}$$

where S_i is the size of an integer, and S_s is the size of a single-precision floating point number. T_t is expressed by Eq. (13):

$$\begin{aligned} T_t &= \frac{\sum_{i=1}^{K} DS(A_i)}{PCI_e} \\ &= \frac{\sum_{i=1}^{K} (E(A_i) \times (S_s + S_i) + N(A_i) \times S_i)}{PCI_e} \\ &= \frac{(S_s + S_i) \times \sum_{i=1}^{K} E(A_i) + S_i \times N}{PCI_e}, \end{aligned} \tag{13}$$

where PCI_e is the transmission rate of the PCIe bus which connects CPU and GPU. According to Theorem 1, for the ELL format, the $\sum_{i=1}^{K} E(A_i)$ is the smallest by using the partitioning Algorithm 3 and T_t is the smallest.

The threads on CPU are scheduled separately without synchronization. So the row vectors with various lengths in an RVS do not wait for each other during computing. If the RVS is computed on CPU, the COO or CSR format should be used for SpMV, because the two formats do not need to fill zero.

When A is computed on multiple processors in parallel, the computing time of an RVS assigned to each processor should be the same by using Algorithm 3, so that the largest parallel efficiency can be achieved.

Assume that the size of the global memory of GPU is MEM_G. Then, the storage space assigned to SpMV cannot be more than MEM_G. If $DS(A_i) > MEM_G$, the RVS A_i cannot be loaded into GPU, and A should be partitioned using Algorithm 3 again.

For SpMV, the mean density of A determines the *effective computing ratio*, which is the proportion of the amount of useful computing to the total amount of computing. Processor utilization is improved with the increased effective computing ratio. For a sparse matrix A, the mean proportion of filled zero is the *storage space vacancy ratio*, which is the proportion of the storage space of zero in the total storage space. The storage space vacancy ratio should be reduced to increase the effective computing ratio.

5.3 The Computing Power of CPU and GPU Based on Benchmarks

Warp is a scheduling unit on GPU, which has 32 threads. All threads of a warp are running at the same time on a streaming multiprocessor (SM). So the running time of a warp is the time of the thread with the longest running time. For the thread grid on GPU, assume that the block can contain at most T threads, and the T threads are scheduled to run on SM by warp. The utilization of SM is the proportion of

$$\frac{\text{the valid running time } (Time_v)}{\text{the total running time } (Time_t)}.$$

For the ELL format, a thread processes a row of the sparse matrix. So the running time of a thread can be represented by the NNZ of the row. $Time_v$ of a warp is the sum of the NNZ of the rows run on the warp, and $Time_t$ is

$$32 \times MAX(\text{NNZ of the rows run on the warp}).$$

The computing efficiency of the warp is the density of the subset assigned into the warp. The block with T threads is split into $T/32$ warps to be processed on SM. So the computing efficiency of SM depends on the density of the block. To improve the utilization of SM, we must improve the density of the block. We provide a method to calculate the lower limit of the density of the block for given a lower limit of the utilization of SM. For the block partitioned using PMF, the rows are ordered by the NNZ in an ascending order. We assume that the NNZ of the first row is 1 in order to simplify calculation. To get the lower limit of the density of the block, the width of the block should be as small as possible. Assume that the width of the the ith warp in the block is W_i. The NNZ of rows 1–31 should be $W_{i-1} + 1$ in order to reduce the width of the ith warp. Since the density of the warp should be more than 0.9, the upper limit NNZ of the 32nd row in the ith warp can be calculated by Eq. (14):

$$NNZ(Row_{32}) = \left\lceil \frac{31W_{i-1} + 31}{27.8} \right\rceil. \tag{14}$$

If the utilization of SM is greater than 0.9 and $T = 1024$, the lower limit of the density of the block is 0.2494 by Eq. (15):

$$D = \frac{\sum_{i=1}^{T/32} \left(31(W_{i-1} + 1) + \left\lceil \frac{31W_{i-1}+31}{27.8} \right\rceil\right)}{32 \sum_{i=1}^{T/32} W_i}, \; W_0 = 0. \tag{15}$$

We get the blocks from the sparse matrix by different density using PMF to test. The utilization of multicore CPU is less affected by the sparsity pattern of the sparse matrix, because each core in CPU is independently scheduled. The threads between cores do not need synchronization if there is no dependency between threads. The computing power of GPU is improving faster than that of CPU with the improvement of density of blocks. The tested results remain stable at about 75:1, if the density of the blocks is more than 30 percent and the standard deviation is within the acceptable range, as shown in Table 1 of Section 1 in the supplementary material. The tested values and calculated values have good consistency. The density of the blocks partitioned using Algorithm 3 for GPU are more than 30 percent, and the proportion of the computing power of GPU over that of CPU is valid for Algorithm 3.

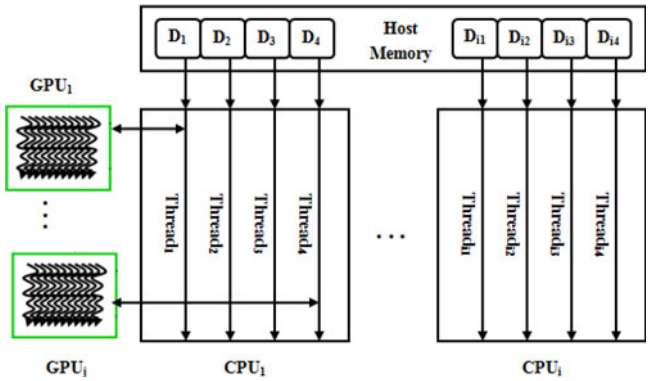


Fig. 5. A CPU-GPU hybrid parallel computing model.

6 Implementation of SpMV on GPUs and Multicore CPUs

6.1 GPUs and Multicore CPUs Hybrid Parallel Programming

More and more applications have been running on GPU-CPU heterogeneous systems. (Section 2 of the supplementary material provides more detailed discussion.) A parallel algorithm must be designed for the architectures and instructions of GPU and CPU to improve the computing performance. But hybrid parallel programming will face some challenges on a GPU-CPU heterogeneous system, because of the different architectures and instructions.

The GPU does not have the process control capability as a device in CUDA, which is controlled by the CPU. The data is transported from the host memory to the global memory of the GPU. Then the CPU invokes the calculation process of the GPU by calling the kernel function.

OpenMP provides a simple and easy-to-use parallel computing capability of multi-threads on multicore CPUs [20]. A hybrid programming model can be established by openMP and CUDA in a CPU-GPU heterogeneous computing environment. OpenMP dedicates one thread for controlling the GPU, while the other threads are used to share the workload among the remaining CPU cores. The CPU-GPU hybrid parallel computing model is illustrated in Fig. 5, where D_1, D_2, D_3, $D_4, \ldots, D_{i1}$, D_{i2}, D_{i3}, D_{i4} are sub-tasks which are stored in the host memory, and $Thread_1$, $Thread_2$, $Thread_3$, $Thread_4, \ldots, Thread_{i1}$, $Thread_{i2}$, $Thread_{i3}$, $Thread_{i4}$ are multi-threads which are assigned to cores of CPUs.

First, a computing task must be divided into multiple sub-tasks, so that these sub-tasks are assigned to multi-cores of CPUs to perform respectively. Second, in the process of execution, two groups of threads are created in the parallel section of OpenMP, where a group of threads are dedicated to controlling the GPUs, while other threads undertake the CPU workload by utilizing the remaining CPU cores.

6.2 Parallel Implementation of SpMV Based on Partitioning

A sparse matrix is partitioned into some RVS's, which are assigned to GPUs and multicore CPUs to perform SpMV by a hybrid parallel computing model. According to the partition, the computing time of a task assigned to each core on CPUs should be the same in order to optimize the computing performance. If SpMV can be computed in parallel only once on all GPUs and CPUs, then these RVS's partitioned from the sparse matrix are assigned to GPUs and the cores of CPUs according to their computing powers. If SpMV cannot be computed in parallel only once because some RVS's are too big to be loaded into the GPU once, the sparse matrix should be partitioned many times.

The implementation of SpMV includes three steps: (1) get the ratios of partition according to the computing powers of GPUs and CPUs by testing benchmarks; (2) partition the sparse matrix; (3) assign the RVS's to GPUs and CPUs to process in parallel. Notice that the computed results do not need to be summarized, because the sparse matrix is partitioned by rows.

7 Experimental Evaluation

7.1 Experiment Settings

The following test environment has been used for all benchmarks. The test computer is equipped with two Intel Xeon Core E5506 CPUs running at 2.13 GHz and two NVIDIA Geforce GTX 460 GPUs. Each CPU has 4 cores. Each GPU has 336 CUDA processor cores, working on 1.5 GHz clock and 1 GB global memory with 256 bits bus width and 1.9 GHz clock, with CUDA compute capacity 2.1. As for software, the test machine runs the 64 bit Windows 7 and NVIDIA CUDA toolkit 5.0. The hardware parameters of the testing computer are shown in Table 2 of Section 3.1 in the supplementary material which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TC.2014.2366731. All the evaluation results are averaged after running 100 times. The time of SpMV includes the time of partitioning for all algorithms in test.

All benchmarks are chosen from the UF Sparse Matrix Collection [36]. We chose 50 sparse matrices to test for Algorithms 1, 2, 3, and 4. The experimental results of 10 representative tested sparse matrices are shown in the main paper and that of all tested cases are shown in the supplementary material, available online. Tables 3, 4, and 5 of Section 3.2 in the supplementary material provide more detailed information of the ten representative sparse matrices. Most of these matrices are derived from scientific computing and real engineering applications.

NVIDIA Corporation provides three libraries (CUBLAS, CUSPARSE and CUSP) to support matrix computation. All libraries provide CUDA development tools and source codes [5]. CUBLAS offers three levels of library functions, where the second level supports SpMV of dense matrices.

CUSPARSE provides three levels of functions for sparse matrices, with the first level for ADD operation, second for MUL operation of SpMV, and the third level for MUL operation of sparse matrices. It uses both the CSR and HYB formats. HYB is a hybrid format of ELL and COO. ELL decides the column width of data matrix according to the maximum number of non-zero elements in each row, which means that ELL's efficiency of compression will be reduced by the negative effect of the sparse matrix. HYB function for SpMV has a parameter, which has tree values: *AUTO*, *USER*,

TABLE 1
The Relative Difference (%) of Partitioning in Test 1

Sparse Matrix	Algori. 1	Algori. 2	Algori. 3
Fluorem/PR02R	1.5137	0.0050	0.0075
Boeing/pwtk	4.2082	0.0044	0.0038
TSOPF/TSOPF_RS_b300_c3	11.7656	0.1280	0.0845
Schenk_AFE/af_shell10	0.0210	0.0001	0.0001
Rajat/rajat31	0.0058	0.0000	0.0000
Schenk_ISEI/ohne2	0.5187	0.0007	0.0006
Schenk/nlpkkt120	11.2423	0.0000	0.0000
Schenk/nlpkkt160	11.3389	0.0000	0.0000
Schenk/nlpkkt200	11.3976	0.0000	0.0000
vanHeukelum/cage15	1.5706	0.0000	0.0000
Average relative difference	5.3580	0.0138	0.0096

MAX. A function will automatically select a segmentation threshold to divided the sparse matrix into ELL and COO if the parameter is *AUTO*. The caller must provide a segmentation threshold if the parameter is *USER*. If the threshold is 0, HYB will become COO. If the parameter is *MAX*, HYB will become ELL. The HYB function are tested using *AUTO*, *MAX* and 0 respectively. We find that the performance of *AUTO* and *MAX* is almost the same in our experiments, because the rows in the same block have the same or similar NNZ for the blocks partitioned using Algorithm 3, and the time of segmentation by the threshold can be saved for *MAX*. But the performance of the parameter 0 is worse than that of *AUTO* and *MAX*. So we select *MAX* to test for Algorithm 3. But we select *AUTO* to test for Algorithms 1 and 2, because some sparse matrices cannot be processed by *MAX* using Algorithms 1 and 2. NVIDIA provides another library, CUSP (2012), to offer SpMV for the GPU platform. CUSP supports a variety of compression formats such as COO, DIA, CSR, ELL, and HYB. The COO, CSR, and HYB from CUSP show worse performance than CUSPARSE. We chose the DIA format in CUSP to test for Algorithm 4.

The Intel Math Kernel Library provides developers of scientific and engineering software with a set of linear algebra, fast Fourier transforms, and vector math functions optimized for the latest Intel processors. MKL contains LAPACK, the basic linear algebra subprograms (BLAS), and the extended BLAS (sparse) [6], which have higher performance compared to the other lib functions for most of the processors, and they have been parallelized and require no alterations of your application to gain the performance enhancements of multiprocessing. We test SpMV using CSR function of MKL with better performance than COO function.

7.2 SpMV Test and Performance Evaluation

We define *flop-rate* as the ratio of computing scale to computing time. The scale of the computation for SpMV is $2 \times NNZ$, because each non-zero element should perform a multiplication and an addition operations. The *flop-rate* is calculated by $2 \times NNZ/T \times 10^{-9}$, where T is the computing time of SpMV (the unit is second). The unit of *flop-rate* is Gops, which is giga operations per second.

The computing power of CPU and GPU are obtained by benchmarks, where the tested sparse matrices are obtained from the UF Sparse Matrix Collection and the tested functions of SpMV are from Intel MKL and NVIDIA CUSPARSE. We choose 30 sparse matrices to test, where the number of non-zero elements covers millions to hundreds of millions. We get the average value of 30 testing matrices. Assume that the computing power of a core on CPU is as 1. The computing power of the other cores on CPU and GPU are calculated by ratios. 30 sparse matrices of benchmarks are tested on the test computer to obtain the results, where the computing power of GPU is 75 times that of a core of CPU for the tested computer.

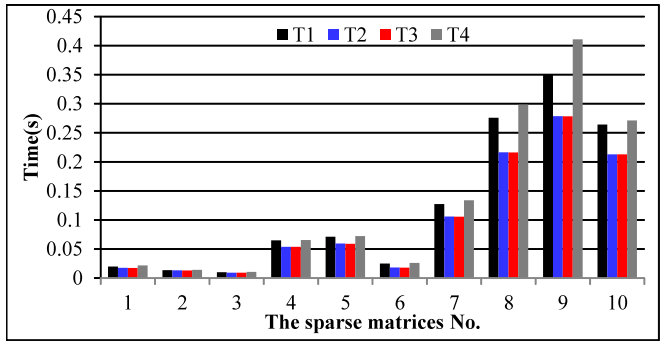


Fig. 6. The time of Algorithms 1, 2, 3, and 4 in Test 1 (single-precision).

We have performed the following two experiments for comparative performance evaluation.

1) SpMV is tested on two multicore CPUs using Algorithms 2, 3 and 4, and the performance is compared to the testing using the automatic parallelization strategy of MKL function according to the partition ratios of (1,1,1,1,1,1,1,1), because the computing power of each core on CPU is the same.
2) SpMV is tested on GPUs and multicore CPUs using Algorithm 3, and the performance is compared to the testing using Algorithms 1, 2 and 4 according to the partition ratios of (75,75,1,1,1,1,1,1), because the computing power of GPU is about 75 times that of a core on CPU.

7.2.1 Test 1

The automatic parallelization strategy of MKL function uses openMP parallel mechanism, where the sparse matrix is partitioned using Algorithm 1 for the CSR format. The density of partitioning has little influence on the performance of SpMV in Test 1, because CSR has no zero padded. However, the balance of computing tasks between cores of CPU has greater influence, and the load scale of cores for SpMV depends on the number of non-zero elements assigned on cores. The relative difference of a partition over computing powers is calculated by

$$\sum_{i=1}^{K} (NNZ(A_i) - CP_i \times NNZ)/(CP_i \times NNZ) \times 100\%,$$

where CP_i is the ratio which the ith RVS should take in a partition. The average relative difference of the ten tested matrices using Algorithms 1, 2, and 3 are 5.3580, 0.0138, and 0.0096 percent respectively, as shown in Table 1, and that of all tested cases are 17.3600, 0.0348, and 0.0119

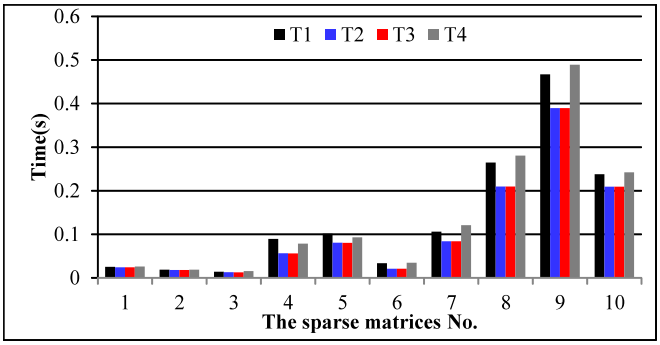


Fig. 7. The time of Algorithms 1, 2, 3, and 4 in Test 1 (double-precision).

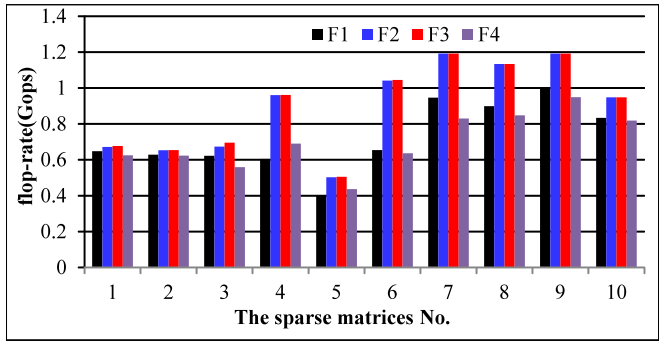


Fig. 9. The flop-rate of Algorithms 1, 2, 3, and 4 in Test 1 (double-precision).

percent respectively, as shown in Table 6 of Section 3.3 in the supplementary material. We can find that the *NNZ* of the RVS's by using Algorithms 2 and 3 match with the computing powers of CPUs very well, while Algorithm 1 does not, leading to load imbalance between cores.

In Test 1, SpMV is tested on the multicore CPU using Algorithms 2, 3, 4, and the automatic parallelization strategy of MKL function. Algorithm 1 is used to partition in the automatic parallelization strategy of MKL function. In Figs. 6 and 7, $T1$, $T2$, $T3$, and $T4$ are the time of SpMV using the automatic parallelization strategy of MKL function, Algorithm 2, Algorithm 3, and Algorithm 4. In Figs. 8 and 9, $F1$, $F2$, $F3$, and $F4$ are the flop-rate. The load on the multicore for SpMV may not be balanced using the automatic parallelization strategy of MKL function itself, because of the imbalance of the number of non-zero elements between the rows. But the computing tasks of multicore on CPUs are split evenly using Algorithms 2 and 3, and the load on the multicore for SpMV can be balanced. The load balancing problem is not considered in Algorithm 4, leading to poor performance. However, it is better than that of Algorithm 1 for the sparse matrices with very obvious diagonal feature, such as Rajat/rajat31 and Schenk_ISEI/ohne2.

It is observed from Figs. 6, 7, 8, and 9 that (1) For the ten representative tested sparse matrices, the average execution time of SpMV reduces by 16.60, 0.61, and 20.31 percent for single precision, 18.39, 0.47, and 20.23 percent for double precision, by using Algorithm 3 compared with the automatic parallelization strategy of MKL function, Algorithm 2, and Algorithm 4; and for all tested cases, the reduction is 27.37, 2.77, and 6.83 percent for single precision, 27.89, 0.39, and 29.23 percent for double precision. (2) For the 10 representative tested sparse matrices, the average flop-rate of SpMV reduces by 20.67, 0.62, and 27.65 percent for single precision, 25.08, 0.48, and 27.52 percent for double precision, by using Algorithm 3 compared with the automatic parallelization strategy of MKL function, Algorithm 2, and Algorithm 4; and for all tested cases, the reduction is 42.51, 2.99, and 62.03 percent for single precision, 68.83, 0.38, and 72.51 percent for double precision.

7.2.2 Test 2

For the 10 representative tested sparse matrices, the average relative difference of the ten tested matrices using Algorithms 1, 2, and 3 are 38.1100, 0.0091, and 0.0093 percent respectively, as shown in Table 2, and that of all tested cases are 25.5616, 0.0326, and 0.0054 percent respectively. We can find that the *NNZ* of the RVS's by using Algorithms 2 and 3 match with the computing powers of GPUs and CPUs very well. For matrices PR02R, pwtk, af_shell10, rajat31, ohne2, and cage15, the *NNZ* of the RVS's by using Algorithm 1 match reasonably, but for TSOPF_RS_b300_c3, nlpkkt120, nlpkkt160, and nlpkkt200, the match is very poor, leading to load imbalance between processors.

For GPU, the density of the RVS's has greater influence on the performance of SpMV using the ELL format. The densities of the ten representative tested matrices using Algorithms 1, 2, and 3 are shown in Fig. 10, where *E1*, *E2*, and *E3* are the densities using Algorithms 1, 2, and 3,

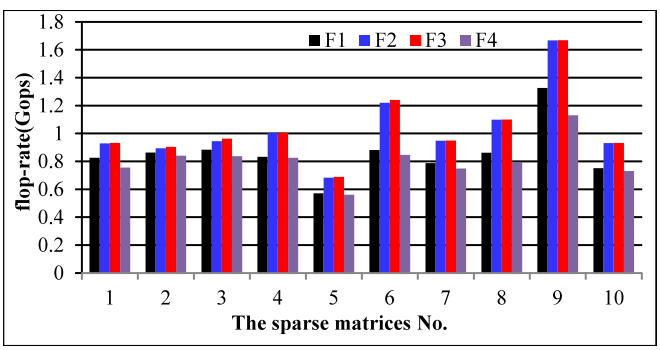


Fig. 8. The flop-rate of Algorithms 1, 2, 3, and 4 in Test 1 (single-precision).

TABLE 2
The Relative Difference (%) of Partitioning in Test 2

Sparse Matrix	Algori. 1	Algori. 2	Algori. 3
Fluorem/PR02R	2.2046	0.0094	0.0090
Boeing/pwtk	7.1559	0.0058	0.0113
TSOPF/TSOPF_RS_b300_c3	93.0066	0.0621	0.0631
Schenk_AFE/af_shell10	0.3845	0.0015	0.0009
Rajat/rajat31	0.1071	0.0007	0.0002
Schenk_ISEI/ohne2	3.5364	0.0093	0.0070
Schenk/nlpkkt120	85.1116	0.0008	0.0005
Schenk/nlpkkt160	85.3570	0.0002	0.0002
Schenk/nlpkkt200	85.5064	0.0002	0.0003
vanHeukelum/cage15	18.7229	0.0005	0.0003
Average relative difference	38.1100	0.0091	0.0093

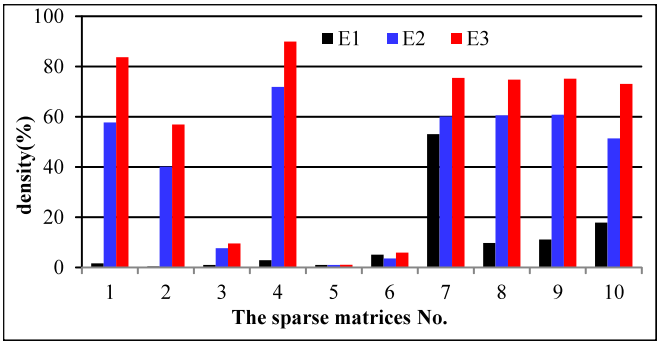


Fig. 10. Comparison of density of Algorithms 1, 2, and 3 in Test 2.

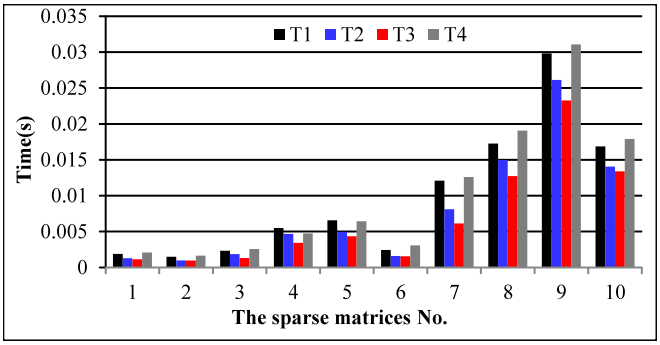


Fig. 11. The time of Algorithms 1, 2, 3, and 4 in Test 2 (single-precision).

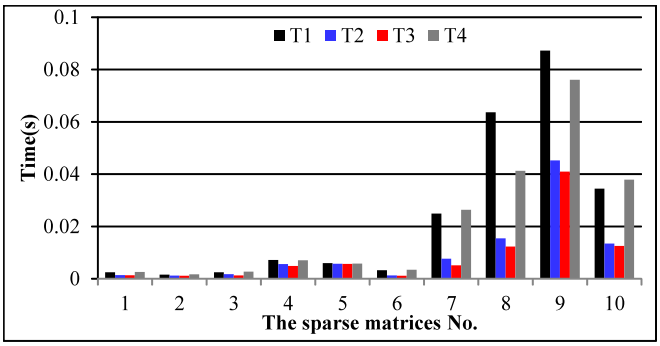


Fig. 12. The time of Algorithms 1, 2, 3, and 4 in Test 2 (double-precision).

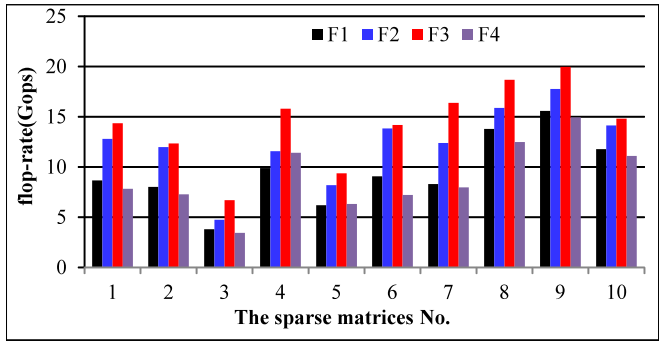


Fig. 13. The flop-rate of Algorithms 1, 2, 3, and 4 in Test 2 (single-precision).

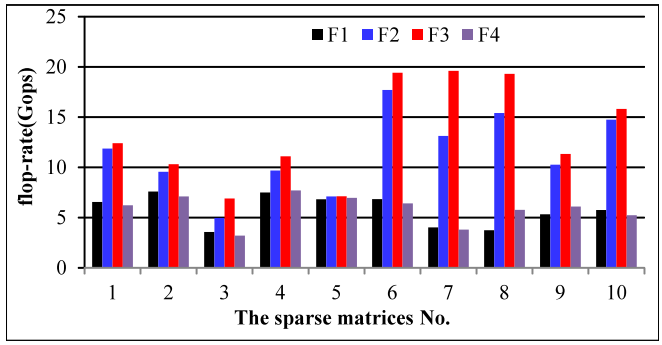


Fig. 14. The flop-rate of Algorithms 1, 2, 3, and 4 in Test 1 (double-precision).

respectively. The average densities of the ten tested matrices using Algorithms 1, 2, and 3 are 10, 41, and 55 percent respectively, and that of all tested cases are 11, 40, and 61 percent respectively.

In Test 2, SpMV is tested on GPUs and multicore CPUs using Algorithms 1, 2, and 3. In Figs. 11 and 12, $T1$, $T2$, $T3$, and $T4$ are the $time$ of SpMV using Algorithm 1, Algorithm 2, Algorithm 3, and Algorithm 4 respectively, and $F1$, $F2$, $F3$, and $F4$ are the flop-rate in Figs. 13 and 14. For the sparse matrices with uneven distribution, such as af_shell10, nlpkkt120, nlpkkt160, nlpkkt200, and cage15, the RVS's obtained from the original row order have low density, leading to high zero filling proportion and low effective computing ratio. The performance of most test cases is poor by using Algorithm 4, because the computation efficiency of GPU is not good for the DIA format, but it is better than that of Algorithms 1 or 2 for the sparse matrices with very obvious diagonal feature, such as Rajat/rajat31, Schenk_ISEI/ohne2, Janna/Serena, Oberwolfach/bone010, Bodendiek/CurlCurl_3, BenElechi/BenElechi1, and CEMW/t2em.

It is observed from Figs. 11, 12, 13, and 14 that (1) For the ten representative tested sparse matrices, the average execution time of SpMV reduces by 34.31, 13.95, and 37.94 percent for single precision, 50.00, 13.10, and 49.79 percent for double precision, by using Algorithm 3 compared with Algorithms 1, 2, and 4; and for all tested cases, the reduction is 35.59, 22.29, and 40.17 percent for single precision, 43.50, 16.60, and 42.93 percent for double precision. (2) For the ten representative tested sparse matrices, the average flop-rate of SpMV reduces by 54.99, 17.64, and 65.35 percent for single precision, 154.68, 16.84, and 144.63 percent for double precision, by using Algorithm 3 compared with Algorithms 1, 2, and 4; and for all tested cases, the reduction is 65.22, 36.17, and 76.15 percent for single precision, 97.27, 22.14, and 93.80 percent for double precision.

7.3 Additional Experimental Results

Due to space limitation, additional experimental results are presented in Sections 3.4 and 3.5 of the supplementary material, available online.

8 Conclusions

In this paper, we use a probabilistic model to develop a partitioning strategy for sparse matrices. This method has wide adaptability for different types of sparse matrices, and is different from existing methods which only adapt to some particular sparse matrices. Our partitioning strategy is based on the PMF of a sparse matrix, which characterizes the distribution of non-zeros in a sparse matrix.

Our partitioning strategy consists of two steps, i.e., distribution analysis and RVS partitioning. The proposed approach in this paper is general, and is neither limited by any GPU programming language nor restricted to any parallel architecture, because it is based on a mathematical and analytical model. In future work, we will consider solving large-scale sparse linear equations using the current SpMV method.

ACKNOWLEDGMENTS

The authors deeply appreciate five anonymous reviewers for their comments and suggestions to improve the manuscript. The research was partially funded by the Key Program of National Natural Science Foundation of China (Grant Nos. 61133005, 61432005), and the National Natural Science Foundation of China (Grant Nos. 61370095, 61472124). The Scientific and Technological Project of Hunan Science and Technology Commission (Grant No. 2013sk2014). The Scientific Research Fund of Hunan Provincial Education Department (Grant No. 13A011).

REFERENCES

[1] D. P. Scarpazza, O. Villa, and F. Petrini, "Efficient breadth-first search on the cell/BE processor," *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, no. 10, pp. 1381–1395, Oct. 10, 2008.

[2] Y. Shan, W. Tianji, Y. Wang, B. Wang, Z. Wang, N. Xu, and H. Yang, "FPGA and GPU implementation of large scale SpMV," in *Proc. 8th Symp. Appl. Sp.*, 2010, pp. 64–70.

[3] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100X GPU versus CPU myth: An evaluation of throughput computing on CPU and GPU," in *Proc. Int. Symp. Comput. Arch.*, vol. 38, no. 3, 2010, pp. 451–460.

[4] A. Pedram, R. A. van de Geijn, and A. Gerstlauer, "Codesign tradeoffs for high-performance, low-power linear algebra architectures," *IEEE Trans. Comput.*, vol. 61, no. 12, pp. 1724–1736, Oct. 2012.

[5] NVIDIA. (2012). The NVIDIA CUDA Sparse Matrix library (cuSPARSE), 2nd ed. [Online]. Available: http://docs.nvidia.com/cuda/cusparse/index.html

[6] J. Gustafson and B. Greer, "Clearspeed whitepaper: Accelerating the intel math kernel library," Tech. Rep., 2007. [Online]. Available: http://www.clearspeed.com/docs/resources/ClearSpeedIntelWhitepaperFeb07.pdf

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

[8] J. Bolz, I. Farmer, E. Grinspun, and P. Schroder, "Sparse matrix solvers on the GPU: Conjugate gradients and multigrid," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 917–924, 2003.

[9] NVIDIA CUDA C Programming Guide, Version 5.0, May 2012.

[10] Z. Fu, T. J. Lewis, R. M. Kirby, and R. T. Whitaker, "Architecting the finite element method pipeline for the GPU," *J. Comput. Appl. Math.*, vol. 257, pp. 195–211, Feb. 2014.

[11] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proc. Conf. High Performance Comput. Netw., Storage Anal.*, 2009, p. 18.

[12] W. T. Tang, W. J. Tan, R. Ray, Y. W. Wong, W. Chan, S. H. Kuo, R. S. M. Goh, S. J. Turner, and W. F. Wong, "Accelerating sparse matrix-vector multiplication on GPUs using bit-representation-optimized schemes," in *Proc. Int. Conf. High Performance Comput., Netw., Storage Anal.*, 2013.

[13] F. Vazquez, G. Ortega, J. J. Fernandez, and E. M. Garzon, "Improving the performance of the sparse matrix vector product with GPUs," in *Proc. 10th IEEE Int. Conf. Comput. Inform. Technol.*, ser. CIT, 2010, pp. 1146–1151.

[14] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick, "Optimization of a lattice boltzmann computation on state-of-the-art multicore platforms," *J. Parallel Distrib. Comput.*, vol. 69, no. 9, pp. 762–777, 2009.

[15] B. Boyer, J. G. Dumas, and P. Giorgi, "Exact sparse matrix vector multiplication on GPU's and multicore architectures," in *Proc. 4th Int. Workshop Parallel Symbolic Comput.*, Jul. 2010, pp. 80–88.

[16] A. Buluc, S. Williams, L. Oliker, and J. Demmel, "Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, pp. 721–733, May 2009.

[17] S. Sun, M. Monga, P. H. Jones, and J. Zambreno, "An I/O bandwidth-sensitive sparse matrix vector multiplication engine on FPGAs," *IEEE Trans. Circuits Syst. I: Reg. Papers*, vol. 59, no. 1, pp. 113–123, Jan. 2012.

[18] J. C. Pichel and F. F. Rivera, "Sparse matrix-vector multiplication on the single-chip cloud computer many-core processor," *J. Parallel Distrib. Comput.*, vol. 73, no. 12, pp. 1539–1550, 2013.

[19] L. Buatois, G. Caumon, and B. Levy, "Concurrent number cruncher: A GPU implementation of a general sparse linear solver," *Int. J. Parallel Emerg. Distrib. Syst.*, vol. 24, no. 3, pp. 205–223, 2009.

[20] T. Oberhuber, A. Suzuki, and J. Vacata, "New row-grouped CSR format for storing the sparse matrices on GPU with implementation in CUDA," arXiv preprint arXiv:1012.2270, 2010.

[21] M. Belgin, G. Back, and C. J. Ribbens, "Pattern-based sparse matrix representation for memory-efficient SMVM kernels," in *Proc. 23rd Int. Conf. Supercomput.*, Jun. 2009, pp. 100–109.

[22] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix vector multiplication on emerging multicore platforms," *Parallel Comput.*, vol. 35, no. 3, pp. 178–194, 2009.

[23] A. Monakov, A. Lokhmotov, and A. Avetisyan, "Automatically tuning sparse matrix vector multiplication for GPU architectures," *High Performance Embedded Architectures and Compilers*. Berlin, Germany: Springer, 2010, pp. 111–125.

[24] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix vector multiply on GPUs," in *Proc. 15th ACM SIGPLAN Symp. Principles Practice Parallel Programming*, 2010, pp. 115–126.

[25] X. Feng, H. Jin, R. Zheng, K. Hu, J. Zeng, and Z. Shao, "Optimization of sparse matrix vector multiplication with variant CSR on GPUs," in *Proc. IEEE 17th Int. Conf. Parallel Distrib. Syst.*, 2011, pp. 165–172.

[26] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, A. Basermann, and A. R. Bishop, "Sparse matrix vector multiplication on GPGPU clusters: A new storage format and a scalable implementation," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp. Workshops PhD Forum*, May 2012, pp. 1696–1702.

[27] A. N. Yzelman and R. H. Bisseling, "Two-dimensional cache-oblivious sparse matrix vector multiplication," *Parallel Comput.*, vol. 37, no. 12, pp. 806–819, 2011.

[28] V. Karakasis, T. Gkountouvas, K. Kourtis, G. Goumas, and N. Koziris, "An extended compression format for the optimization of sparse matrix vector multiplication," *IEEE Trans. Parallel Distrib. Syst. Sep.* vol. 24, no. 10, pp. 1930–1940, Oct. 2013.

[29] P. Chen, "A performance modeling and optimization analysis tool for sparse matrix vector multiplication on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 5, pp. 1112–1123, May 2014.

[30] B. Schmidt, H. Aribowo, and H.-V. Dang, "Iterative sparse matrix vector multiplication for accelerating the block Wiedemann algorithm over GF(2) on multi-graphics processing unit systems," *Concurrency Comput.: Practice Experience*, vol. 25, no. 4, pp. 586–603, 2013.

[31] M. Cenk, C. Negre, and M. A. Hasan, "Improved three-way split formulas for binary polynomial and Toeplitz matrix vector products," *IEEE Trans. Comput.*, vol. 62, no. 7, pp. 1345–1361, Jul. 2013.

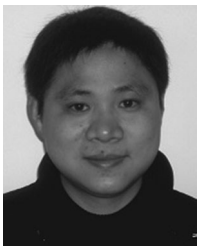
[32] M. A. Hasan and C. Negre, "Multiway splitting method for toeplitz matrix vector product," *IEEE Trans. Comput.*, vol. 62, no. 7, pp. 1467–1471, May 2013.

[33] A.-J. N. Yzelman and D. Roose, "High-level strategies for parallel shared-memory sparse matrix vector multiplication," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 1, pp. 116–125, Jan. 2014.

[34] 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.

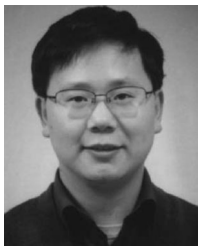
[35] N. L. Johnson, S. Kotz, and A. Kemp, *Univariate Discrete Distributions*. 2nd Ed., New York, NY, USA: Wiley, ISBN 0-471-54897-9, 1993 p. 36.

[36] T. A. Davis and Y. Hu, University of Florida sparse matrix collection[J], 2009.



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

Kenli Li is currently a Full Professor of computer science and technology at Hunan University, Changsha, China, and Deputy Director of National Supercomputing Center in Changsha. He has published more than 100 papers in international conferences and journals. He is an outstanding member of CCF.



Zeyao Mo is currently a Professor with the Institute of Applied Physics and Computational Mathematics, Beijing, China. His research interests include parallel algorithms and software for large-scale science and engineering simulations.

Keqin Li is currently a distinguished Professor of computer science at Stony Brook University, New York, NY. He has over 300 research publications. He is currently or has served on the editorial boards of the *IEEE Transactions on Parallel and Distributed Systems*, the *IEEE Transactions on Computers*, and the *IEEE Transactions on Cloud Computing*. He is a Fellow of the IEEE.