Journal of Computer and System Sciences 92 (2018) 152–170

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

A parallel computing method using blocked format with optimal partitioning for SpMV on GPU

Wangdong Yang[a,b,*], Kenli Li[a,c,*], Keqin Li[a,d]

[a] College of Computer Science and Electronic Engineering, Hunan University, Changsha, Hunan 410082, China
[b] College of Information Science and Engineering, Hunan City University, Yiyang, Hunan 413000, China
[c] The National Supercomputing Center in Changsha, Changsha, Hunan 410082, China
[d] Department of Computer Science, State University of New York, New Paltz, NY 12561, USA

A R T I C L E   I N F O

Article history:
Received 3 January 2016
Received in revised form 9 June 2017
Accepted 26 September 2017
Available online 16 October 2017

Keywords:
Blocked format
CPU/GPU
Dynamic programming
Heterogeneous parallel computing
Partitioning
Reordering
Sparse matrix–vector multiplication

A B S T R A C T

For large-scale sparse matrices, SpMV cannot be processed on GPU using the common storage formats because of the memory limitation. In addition, the parallel effect is poor using general formats for the sparse matrices with extremely uneven distribution of non-zero elements, which leads to performance deterioration. This paper presents an optimal partitioning strategy based on the distribution of non-zero elements in a sparse matrix to improve the performance of SpMV, and uses a hybrid format, which mixes CSR and ELL formats, to store the blocks partitioned from the sparse matrix. The hybrid blocked format has better compression effect and more uniform distribution of non-zero elements, which can be suitable for more types of sparse matrices. Our partitioning strategy is proven to be optimal, which can yield the minimum parallel execution time on GPU.

© 2017 Elsevier Inc. All rights reserved.

1. Introduction

1.1. Motivation

In recent years, accelerator-based computing using accelerators, such as the IBM Cell synergistic processing unit (SPU), field programmable gate array (FPGA), graphics processing unit (GPU), and application specific integrated circuit (ASIC), has achieved clear performance gains compared to CPUs. Among the accelerators, GPUs have occupied a prominent place due to their low cost and high performance-per-watt ratio along with powerful programming models. However, as CPU architectures also evolve and address challenges such as the power wall and the memory wall, and compete with these accelerators, it is imperative that CPUs should also be included in computations. It is further observed by [1] that several irregular applications such as sparse matrix–vector multiplication (SpMV) can benefit from heterogeneous algorithms that run on a CPU and GPU based heterogeneous computing platform.

SpMV is an essential operation in solving linear systems and partial differential equations. SpMV faces two challenges, which are large scales and irregular distributions of non-zero elements. With the increasing scale of sparse matrices, the sparse matrix of SpMV cannot be loaded into the GPU once to be computed. So a large-scale sparse matrix must be split into some submatrices to be computed separately. It is a challenging issue to adopt an appropriate method to split a

* Corresponding authors.
E-mail addresses: yangwangdong@163.com (W. Yang), lkl@hnu.edu.cn (K. Li).

https://doi.org/10.1016/j.jcss.2017.09.010
0022-0000/© 2017 Elsevier Inc. All rights reserved.

sparse matrix. In addition, load imbalance of parallel computing on GPU are generated, because of irregular distribution of non-zero elements in a sparse matrix, which leads to parallel efficiency decrease. The ELL format is a regular compression format for a sparse matrix, which has the same length of rows, and can avoid load imbalance. But the significant difference between the numbers of non-zero elements in rows will lead to more filling of zeroes. So those rows with similar numbers of non-zero elements are grouped into the same block from a sparse matrix to improve the effect of compression of ELL. It is also a challenging issue to determine the size of a block. If the size of the block is too large, there are more zeros to fill in the block. On the contrary, if the size of the block is too small, the number of blocks is too much, leading to an increased number of computing tasks.

How to make full use of computing resources to maximize parallel computing ability is the key to improve the performance of SpMV. Firstly, load balancing between the threads is the basis of improving performance for streaming multiprocessor (SM) on GPU. Secondly, improving the efficiency of data access is very important to improve the parallel efficiency of GPU.

1.2. Our contributions

The present paper makes the following unique contributions to parallel computation of SpMV on GPU and CPU.

- We develop an optimal partitioning strategy based on dynamic programming and a distribution function (DF) of non-zero elements to improve the performance of SpMV.
- We present a reordering algorithm in which the time complexity is only $O(N \log_2 k)$, where k is the number of partitions and far less than the number of rows. However, the time complexity of a general method is $O(N \log_2 N)$.
- We employ a hybrid format to store a blocked sparse matrix partitioned by our optimal partitioning strategy.

Our partitioning strategy is proven to be optimal, which can yield the minimum parallel execution time on GPU. Our partitioning strategy is based on the DF, which characterizes the distribution of non-zero elements in a sparse matrix. Our partitioning strategy consists of three steps, i.e., building the DF of a sparse matrix, partitioning the rows using dynamic programming, and reordering the rows. Firstly, the DF of a target sparse matrix is constructed according to the analysis of the distribution of non-zero elements per row. Secondly, the intervals of partitioning are determined by our partitioning algorithm based on the DF. Thirdly, these blocks for SpMV are segmented from the sparse matrix by a reordering algorithm. Furthermore, these blocks are stored in a hybrid format to obtain further performance gain.

In this paper, 20 sparse matrices, which are obtained from [2], are tested on NVIDIA K20c GPU and AMD Opteron 6376 CPU. The performance improvement of our algorithm is very effective according to our experiments. Our partitioning strategy has the best performance, which can partition for sparse matrices according to the minimum parallel processing time. According to our experiments on 20 test cases, the performance of SpMV is significantly improved when a sparse matrix is partitioned into blocks by our method, and improvement of our reordering algorithm is also effective.

This paper extends our previous work [3,4]. The current paper presents a new partitioning strategy based on processing time and using DF, and proposes a hybrid storage format and a kernel function for SpMV on GPU.

The remainder of the paper is organized as follows. In Section 2, we review related research on SpMV. In Section 3, we review the programming model of GPU and the storage formats of sparse matrices. In Section 4, we present the DF for sparse matrices. In Section 5, we analyze the parallel processing time of SpMV to develop our optimal partitioning strategy. In Section 6, we describe the implementation of SpMV in parallel using our method on GPU. In Section 7, we demonstrate our extensive experimental performance comparison results. In Section 8, we conclude the paper.

2. Related work

2.1. Parallel implementation of SpMV on GPU and partitioning strategies

Bolz et al. [5] proposed one of the first SpMV CUDA kernel implementations. [6] designed a new HYB format for SpMV in CUDA, representing the matrix in ELLPACK format (ELL) and coordinate format (COO) portions, to combine the speed of ELL and the flexibility of COO. Lee et al. [1] discussed optimization techniques for both CPU and GPU, and analyzed what architecture features contribute to performance differences between the two architectures. Stanimire et al. [7] presented a set of techniques that can be used to develop efficient dense linear algebra algorithms for hybrid multicore + GPU systems, and used asynchronous techniques to reduce the amount of communication between the hybrid components.

In large-scale scientific and engineering calculations, some very big sparse matrices are produced. These sparse matrices are too big to compute by one GPU once. So they should be partitioned into small blocks to be processed multiple times on GPU. But due to various distributions of non-zero elements in sparse matrices, there is no general partitioning method to adapt to all kinds of sparse matrices.

For the blocked compressed sparse row (BCSR) format [8] and the row-grouped CSR (GCSR) format [9], which the rows in a sparse matrix are split into blocks. For the blocked ELLPACK (BELLPACK) format [10] and the sliced ELLPACK (SELLPACK) format [11], a sparse matrix is partitioned into blocks after it is compressed by the ELL format. For segmented interleave combination (SIC) format [12], and compressed sparse blocks (CSB) format [13], a sparse matrix is partitioned into submatrices according to the distribution of non-zero elements. The padded jagged diagonals storage (pJDS) format [14] is suitable for diagonal matrices. A sparse matrix is partitioned using the above partitioning algorithms in accordance with the original order.

Other strategies partition sparse matrices according to the configuration of a computing environment, such as the cache scale of processors [15,12], the computing power of processors [16,17], and the data transmission bandwidth [13]. Furthermore, other strategies partition sparse matrices according to the characteristics of a sparse matrix, such as the proportion of non-zero elements in the sparse matrix [18], the compression effect of different formats [19], and dense blocks in the sparse matrix [20].

Refs. [21,22] proposed to perform three-way and multi-way split approaches to break down a matrix–vector product into some matrix–vector products with smaller sizes. The partitioning strategies based on the original order are sensitive to sparse distribution and cannot achieve satisfactory compression effect in most cases. The partitioning 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 submatrices by a partitioning strategy, leading to the need of accumulation of calculation results [16,10], which adds extra time overhead.

2.2. *Reordering techniques of sparse matrices for SpMV*

Reordering techniques have been a successful approach to improving the performance of SpMV. These techniques evaluate the sparsity pattern of a matrix to find an appropriate permutation of rows and columns of the original matrix. The traditional reordering techniques mainly include approximate minimum degree (AMD) [23], distance function [24], reverse Cuthill–McKee (RCM) [25], and nested dissection [26]. All the ordering techniques try to reduce the fill-in, but each one uses a different approach. The objective of AMD is to find a permutation of the original matrix that reduces the fill-in. Distance function allows to permute individual rows/columns of the original matrix or sets of consecutive rows/columns. RCM algorithm is the same algorithm as the original one but with the resulting index numbers reversed. METIS [27] included in the library computes fill-reducing orderings using a particular implementation of nested dissection algorithm, which can only be applied to matrices with symmetric pattern.

A sparse matrix should be reordered in order to make the calculated loads of the segmented blocks more balanced. The rows with similar numbers of non-zero elements should be allocated in the same block, which is suitable for parallel processing. All the rows in CSR are reordered according to their lengths in the process of some reordering [12,14], leading to calculation amount increase, whose time complexity is $O(N\log_2 N)$. Ref. [28] discussed three reordering methods: CRS block order, ACRS block order, and ZZ-CCS block order. The ZZ-CCS block order is superior to the ACRS block order, which in turn is better than the CRS block order. Ref. [29] discussed a Hilbert-ordering on the non-zeroes of a sparse matrix and found a perfectly balanced partitioning based on this ordering, which is equivalent to partitioning into equally-sized parts.

3. CUDA and storage of sparse matrices

3.1. *An introduction to CUDA*

The modern GPUs have evolved from a fixed-function graphics pipeline to a programmable parallel processor with computing power exceeding that of multicore CPUs. The basic computing unit of a GPU is SM. As a component at the bottom of the independent hardware structure, SM can be seen as a single instruction multiple data (SIMD) processing unit. Each SM contains some scalar processors (SP) and special function units (SFU). In addition, each SM contains the shared memory for threads to share data or communications in the block. Using the model explicitly to access memory, the accessed speed of the shared memory is close to that of register without bank conflict. SM contains some registers, which are allocated by each thread in the execution. All SMs share the global memory [30]. For GPU architectures, CUDA (Compute Unified Device Architecture) was provided from NVIDIA to improve the efficiency of programming on GPU. CUDA is a complete general purpose graphics processing units (GPGPU) solution that provides direct access to the hardware interface, rather than the traditional approach that must rely on the graphical interface API. A heterogeneous parallel computing system based on CPU and GPU can be built using CUDA, as shown in Fig. 1.

The number of threads used in CUDA is decided by the programmer to be executed. A collection of threads (called a block) runs on a SM at a given time. Multiple blocks can be assigned to a single SM and their execution is time-shared. Warp is a group of threads which are issued and scheduled as a basic unit on Nvidia GPUs. Currently, the size of a warp is 32, so 32 threads are issued and scheduled at the same time by a SM.

3.2. *Storage formats of sparse matrices*

The 4-by-4 sparse matrix A shown below is used as a running example in this section:

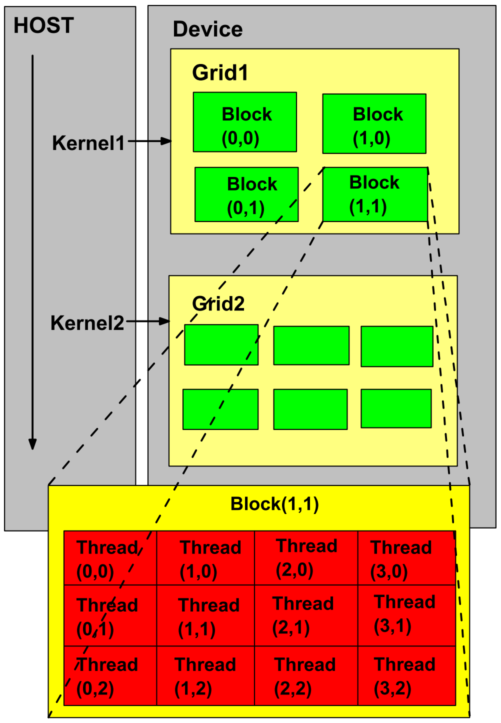


Fig. 1. A heterogeneous parallel computing system based on CUDA.

$$A = \begin{pmatrix} 1 & 0 & 2 & 0 \\ 0 & 0 & 7 & 0 \\ 4 & 5 & 0 & 8 \\ 0 & 1 & 0 & 3 \end{pmatrix}.$$

3.2.1. *Storage space of CSR*

The compressed sparse row (CSR) format is a popular and general-purpose sparse matrix representation scheme. CSR explicitly stores column indices and non-zero values in arrays **Aj** and **Av**. The third array **Ap** represents the starting position of each row in the array **Aj**. For an N-by-M matrix, **Ap** has length $N+1$ and stores the offset of the ith row in **Ap**$[i]$. The value of the last element is the *number of non-zeros* (NNZ). For the example sparse matrix S, we have

$$\mathbf{Aj} = \begin{pmatrix} 1 & 3 & 3 & 1 & 2 & 4 & 2 & 4 \end{pmatrix},$$
$$\mathbf{Av} = \begin{pmatrix} 1 & 2 & 7 & 4 & 5 & 8 & 1 & 3 \end{pmatrix},$$
$$\mathbf{Ap} = \begin{pmatrix} 0 & 2 & 3 & 6 & 8 \end{pmatrix}.$$

3.2.2. *Storage space of ELL*

Another storage scheme that is well-suited to vector architectures is the ELL format. For an N-by-M matrix with a maximum of K non-zeros per row, the ELL format stores the non-zero values in a dense N-by-K array **EData**, and rows with fewer than K non-zeros are zero-padded. Similarly, the corresponding column indices are stored in **Offset**, again with some sentinel value used for padding. For the example sparse matrix A, we have

$$\mathbf{EData} = \begin{pmatrix} 1 & 2 & 0 \\ 7 & 0 & 0 \\ 4 & 5 & 8 \\ 1 & 3 & 0 \end{pmatrix}, \qquad \mathbf{Offset} = \begin{pmatrix} 1 & 3 & * \\ 3 & * & * \\ 1 & 2 & 4 \\ 2 & 4 & * \end{pmatrix}.$$

3.2.3. *Storage space of COO*

The coordinate (COO) format is a particularly simple storage scheme of triples (**row**, **column**, **value**). The arrays **row**, **column**, and **value** store the row indices, column indices, and values of the non-zero elements in matrix respectively. For the example sparse matrix A, we have

$$\begin{aligned}\mathbf{row} &= \begin{pmatrix}1 & 1 & 2 & 3 & 3 & 3 & 4 & 4\end{pmatrix},\\ \mathbf{column} &= \begin{pmatrix}1 & 3 & 3 & 1 & 2 & 4 & 2 & 4\end{pmatrix},\\ \mathbf{value} &= \begin{pmatrix}1 & 2 & 7 & 4 & 5 & 8 & 1 & 3\end{pmatrix}.\end{aligned}$$

3.2.4. Storage space of HYB

HYB is a hybrid format of ELL and COO. Given a threshold K, the part of rows with more than K non-zeros is extracted to be stored by COO and the other part is stored by ELL with little zero-padded in ELL. A sparse matrix can be divided into two parts: COO and ELL. For the example sparse matrix A, we have

$$COO: \begin{cases}\mathbf{row} = \begin{pmatrix}3\end{pmatrix}\\ \mathbf{column} = \begin{pmatrix}4\end{pmatrix}\\ \mathbf{value} = \begin{pmatrix}8\end{pmatrix}\end{cases} ELL: \begin{cases}\mathbf{EData} = \begin{pmatrix}1 & 2\\ 7 & 0\\ 4 & 5\\ 1 & 3\end{pmatrix}\\ \mathbf{Offset} = \begin{pmatrix}1 & 3\\ 3 & *\\ 1 & 2\\ 2 & 4\end{pmatrix}\end{cases}.$$

While the ELL format is well-suited to vector architectures, its efficiency rapidly degrades when the number of non-zeroes per matrix row varies. In contrast, the storage efficiency of COO is invariant to the distribution of non-zeros per row. HYB stores the majority of matrix entries in ELL and the remaining entries in COO.

The non-zeros in rows with no more than K non-zeros are stored in ELL and the remaining entries in COO. ELL stores an N-by-K dense matrix.

3.2.5. Storage space of BSR

The only difference between the CSR and BSR formats is the format of the storage element. The former stores primitive data types whereas the latter stores a two-dimensional square block of primitive data types. The dimension of the square block is **blockDim**. An $m \times n$ sparse matrix A is equivalent to a block sparse matrix A_b with $mb = \frac{m+\mathbf{blockDim}-1}{\mathbf{blockDim}}$ block rows and $nb = \frac{n+\mathbf{blockDim}-1}{\mathbf{blockDim}}$ block columns. If m or n is not multiple of **blockDim**, then zeros are filled into A_b.

As with CSR format, (**row**, **column**) indices of BSR are stored in row-major order. The index arrays are first sorted by row indices and then within the same row by column indices. For the example sparse matrix A, if **blockDim** is 2, then mb and nb are 2, and matrix A is split into 2×2 block matrix A_b. Based on one-based indexing, the block-wise view can be represented as follows:

$$\begin{aligned}&A_b = \begin{pmatrix}A_{11} & A_{12}\\ A_{21} & A_{22}\end{pmatrix},\\ &A_{11} = \begin{pmatrix}1 & 0\\ 0 & 0\end{pmatrix}, A_{12} = \begin{pmatrix}2 & 0\\ 7 & 0\end{pmatrix}, A_{21} = \begin{pmatrix}4 & 5\\ 0 & 1\end{pmatrix}, A_{22} = \begin{pmatrix}0 & 8\\ 0 & 3\end{pmatrix}.\end{aligned}$$

4. Distribution of non-zero elements in a sparse matrix

Taking into consideration the structure of a sparse matrix can dramatically improve the performance of SpMV. However, sparse matrices arise from different domains and have distinct distribution patterns of non-zero elements. Adopting a suitable storage format according to the distribution pattern of a sparse matrix is very helpful to improve the performance of SpMV. We can accurately describe the distribution pattern of a sparse matrix by a DF, and get numerical characteristics of sparsity distribution from the DF. The suitable blocks can be partitioned from the sparse matrix by numerical characteristics of sparsity distribution.

4.1. The DF of sparse matrices

Let A be a sparse matrix with N rows and M columns. A can be viewed as a sequence of M-dimensional rows, i.e., $A = (r_1^T, r_2^T, ..., r_N^T)^T$. For convenience, we also treat A as a multiset in which members are allowed to appear more than once, i.e., $A = \{r_1, r_2, ..., r_N\}$, since some row vectors may be identical. Without loss of generality, we assume that there is no zero-vector (i.e., all components of a vector are zero) in A; otherwise, we can simply remove the zero-vectors from A, and add corresponding zeros to the result vector of SpMV.

Let R_m represent the multiset of row vectors in A, in which all vectors have m non-zero components, where $1 \le m \le M$. Thus, we have $A = R_1 \cup R_2 \cup \cdots \cup R_M$. We call $R_1, R_2, ..., R_M$ as *row vector sets* (RVS). Define the DF of A as

$$f_A : \{1, 2, ..., M\} \to \{0, 1, 2, ..., N\},$$

where $f_A(m) = |R_m| = b_m$ is the number of row vectors in R_m. It is clear that $b_1 + b_2 + \cdots + b_M = N$. Also, $R_m = \emptyset$ if $b_m = 0$.

4.2. The characteristics of the DF

Let $C = \{r_{n_1}, r_{n_2}, ..., r_{n_b}\}$ be any set of row vectors. The number of row vectors in C is

$$N(C) = |C| = b.$$

The width of C is expressed as

$$W(C) = \max\{NNZ(r_{n_1}), NNZ(r_{n_2}), ..., NNZ(r_{n_b})\},$$

where $NNZ(r)$ is the number of non-zero elements in a row vector r. The number of non-zero elements in C is

$$NNZ(C) = NNZ(r_{n_1}) + NNZ(r_{n_2}) + \cdots + NNZ(r_{n_b}).$$

The total number of elements in a minimum dense matrix (i.e., in the ELL format) that includes C is

$$E(C) = N(C) \times W(C).$$

The density of C is the proportion of non-zero elements in $E(C)$, expressed as

$$D(C) = \frac{NNZ(C)}{E(C)}.$$

As a special case, we use R_{M_1,M_2} to denote the union $R_{M_1} \cup R_{M_1+1} \cup \cdots \cup R_{M_2}$. Hence, we have

$$N(R_{M_1,M_2}) = \sum_{m=M_1}^{M_2} b_m,$$

and

$$W(R_{M_1,M_2}) = \max_{M_1 \le m \le M_2} \{m\,(b_m \neq 0)\},$$

and

$$NNZ(R_{M_1,M_2}) = \sum_{m=M_1}^{M_2} m b_m,$$

and

$$E(R_{M_1,M_2}) = N(R_{M_1,M_2}) \times W(R_{M_1,M_2}),$$

and

$$D(R_{M_1,M_2}) = \frac{NNZ(R_{M_1,M_2})}{E(R_{M_1,M_2})}.$$

5. An optimal partitioning algorithm

5.1. Construction of DF

The number of non-zero elements in each row are scanned to store the number of rows with the same NNZ by the array B, which has the length M. The $B[m] = b_m$ will be added 1 if a row has m non-zero elements. The number of scanning will be N if the storage format of the sparse matrix is CSR. For the following 10×10 sparse matrix A,

$$A = \begin{pmatrix} 3 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 9 & -1 & 0 & 0 & 7 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 12 & 3 & 0 & 0 & 0 & 0 & 0 \\ -1 & 8 & 0 & 2 & 0 & 5 & 0 & 2 & 7 & 9 \\ 0 & 0 & 0 & 0 & 0 & 0 & -6 & 0 & 0 & 0 \\ 0 & 0 & 6 & 4 & 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 2 & 0 & 5 & 0 & 0 & 8 & 0 & 1 & 0 \\ 2 & 1 & 0 & 5 & 0 & 3 & 7 & 0 & 0 & 4 \\ 0 & 0 & 3 & 0 & 0 & 7 & 0 & 0 & 0 & 0 \end{pmatrix},$$

we have $A = \{r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8, r_9, r_{10}\}$ and

$$B = [b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9, b_{10}] = [2, 3, 2, 1, 0, 1, 1, 0, 0, 0].$$

The RVS's are $R_1 = \{r_3, r_6\}$, $R_2 = \{r_1, r_4, r_{10}\}$, $R_3 = \{r_2, r_7\}$, $R_4 = \{r_8\}$, $R_6 = \{r_9\}$, $R_7 = \{r_5\}$, and $R_5 = R_8 = R_9 = R_{10} = \emptyset$.

5.2. Types of partitioning methods

There are two types of partitioning strategies to split sparse matrices. The first type splits a sparse matrix into various sets of row vectors which do not change the original order of rows in A, and the second type uses the technique of reordering to split. The algorithm complexity of the first type is relatively low. But the number of non-zero elements in rows of a set of row vectors may be unbalanced, because the non-zero element distribution in a sparse matrix can be arbitrary. A sparse matrix is firstly reordered and then split into RVS's in the second type of strategies. The algorithm complexity of the second type is relatively high, because the rows in the sparse matrix must be reordered, but the performance of SpMV is improved. Our partitioning strategy based on DF belongs to the second type.

The ELL format of a sparse matrix is very suitable for parallel computing, because the lengths of all rows are the same. In particular, SpMV based on ELL has good performance on GPU. But some zeros will be padded in the ELL format, leading to redundant computation. Improving the density of the non-zero elements can reduce the filling ratio of zero for ELL [31]. But the computable scale of blocks for SpMV will go down and the number of blocks will increase, leading to performance degradation of the whole SpMV.

5.3. Analysis for parallel execution time

A *partition* of a sparse matrix A is a set $\mathcal{P} = \{C_1, C_2, ..., C_k\}$ of disjoint subsets of row vectors of A, where $C_i \subseteq A$ for all $1 \le i \le k$, $C_i \cap C_j = \emptyset$ for all $i \neq j$, and $C_1 \cup C_2 \cup \cdots \cup C_k = A$.

The processing of SpMV on GPU includes three parts, i.e., transferring data into GPU, computing on GPU, and returning result from GPU.

For $A\mathbf{x} = \mathbf{y}$, the sparse matrix A and vector $\mathbf{x}$ with N elements are transferred into GPU. The computing results are sent back from GPU by vector $\mathbf{y}$ with N elements. The transmission time of S elements between CPU and GPU is calculated by S/TW, where TW is the bandwidth of PCIe. The transmission arrays includes the **Ap**, **Aj**, **Av**, and right vector if the sparse matrix is stored by CSR format. So the number of elements that need to be transmitted should be $2N + 1 + 2NNZ$ (approximated as $2N + 2NNZ$) if it doesn't consider datatypes. The total transmission time T_t of SpMV is expressed as

$$T_t = \frac{2N + 2NNZ}{TW}.$$

Hence, the transmission time is the same for different partitioning schemes. Because the ELL format stores C_i in a dense $N(C_i)$-by-$W(C_i)$ array, the computing amount of C_i is $E(C_i)$. The computing time for C_i on GPU should be $E(C_i)/f_p$, where f_p is the computation power of a SM in GPU, because each block is assigned to a SM to process. In addition, there is a lower limit L for the number of threads in the block assigned to a SM, which is related to the number of SPs and registers in the SM. If the number of threads in the block assigned to the SM is less than L, this means that the SM execution resources will likely be underutilized, because there will be fewer warps to schedule around long-latency operations [32]. L is about 192 for K20c GPU, which is used in our experiments. The number of rows in C_i should be more than L, because each thread computes a row data once for the ELL format. For C_i, the computing time $T_c(C_i)$ is calculated as

$$T_c(C_i) = \begin{cases} (W(C_i) \times L)/f_p, & N(C_i) \le L; \\ E(C_i)/f_p, & N(C_i) > L. \end{cases}$$

To summarize, the total processing time T of SpMV is calculated by

$$T = T_t + \sum_{i=1}^{k} T_c(C_i)/s,$$

where s is the number of SMs in GPU. It is easily observed that T is largely determined by $E(C_1) + E(C_2) + \cdots + E(C_k)$.

5.4. An optimal partitioning problem

Let us consider the following *optimal partitioning problem*. Given a sparse matrix $A = \{r_1, r_2, ..., r_N\}$ with DF f_A, find an optimal partition $\mathcal{P} = \{C_1, C_2, ..., C_k\}$ of A, such that

$$\mathrm{Cost}(\mathcal{P}) = \sum_{i=1}^{k} T_c(C_i)$$

is minimized. Such an optimal partition yields the minimum T, i.e., the total processing time of SpMV on GPU.

For a fixed k, minimizing $\mathrm{Cost}(\mathcal{P})$ is equivalent to minimizing

$$E(\mathcal{P}) = E(C_1) + E(C_2) + \cdots + E(C_k)$$

if $N(C_i) > L$ for $i = 1, 2, ..., k$. We prove two theorems which give some important properties of an optimal partition which minimizes $E(\mathcal{P})$.

The following theorem states that in an optimal partition, all row vectors of the same RVS must stay in the same C_i.

Theorem 5.1. *For any partition $\mathcal{P} = \{C_1, C_2, ..., C_k\}$, there is a partition $\mathcal{P}' = \{C'_1, C'_2, ..., C'_k\}$, such that each RVS R_m is entirely included in some C'_i, i.e., $R_m \subseteq C'_i$ for some i, where $1 \le m \le M$; and that $E(\mathcal{P}') \le E(\mathcal{P})$.*

Proof. Assume that there are two row vectors $r_{n_1}, r_{n_2} \in R_m$, such that $r_{n_1} \in C_i$ and $r_{n_2} \in C_j$, where $W(C_i) \le W(C_j)$. Let us move r_{n_2} from C_j to C_i, and the new partition is

$$\mathcal{P}' = \{C_1, ..., C'_i, ..., C'_j, ..., C_k\},$$

where $C'_i = C_i \cup \{r_{n_2}\}$ and $C'_j = C_j - \{r_{n_2}\}$, which implies that $N(C'_i) = N(C_i) + 1$ and $N(C'_j) = N(C_j) - 1$. The above operation does not change the width of C_i, i.e., $W(C'_i) = W(C_i)$, but may reduce the length of C_j (e.g., when $NNZ(r_{n_2}) = W(C_j)$), i.e., $W(C'_j) \le W(C_j)$. It is clear that

$$\begin{aligned}
&E(\mathcal{P}') - E(\mathcal{P}) \\
&= N(C'_i)W(C'_i) + N(C'_j)W(C'_j) - N(C_i)W(C_i) - N(C_j)W(C_j) \\
&\le (N(C_i)+1)W(C_i) + (N(C_j)-1)W(C_j) - N(C_i)W(C_i) - N(C_j)W(C_j) \\
&= W(C_i) - W(C_j).
\end{aligned}$$

Since $W(C_i) \le W(C_j)$, we have $E(\mathcal{P}') \le E(\mathcal{P})$. In fact, if there are p row vectors in C_j which belong to R_m, we can move all of them from C_j to C_i simultaneously, which gives rise to $N(C'_i) = N(C_i) + p$ and $N(C'_j) = N(C_j) - p$, and

$$E(\mathcal{P}') - E(\mathcal{P}) \le p(W(C_i) - W(C_j)).$$

By repeating the above operation, we can put all row vectors of R_m into the same C_i for some i, where $1 \le m \le M$, without increasing $E(\mathcal{P})$. The theorem is proven. □

The following theorem states that in an optimal partition, each C_j is a set of consecutive RVS's.

Theorem 5.2. *For any partition $\mathcal{P} = \{C_1, C_2, ..., C_k\}$ which satisfies Theorem 5.1, there is a partition $\mathcal{P}' = \{C'_1, C'_2, ..., C'_k\}$, such that if $R_{m_1} \subseteq C_i$ and $R_{m_2} \subseteq C_j$, where $i < j$, then $m_1 < m_2$; and that $E(\mathcal{P}') \le E(\mathcal{P})$.*

Proof. Without loss of generality, we assume that $W(C_1) < W(C_2) < \cdots < W(C_k)$; otherwise, it is only a matter of re-ordering of $C_1, C_2, ..., C_k$. Assume that there are two RVS's, R_{m_1} and R_{m_2}, such that $R_{m_1} \subseteq C_i$ and $R_{m_2} \subseteq C_j$, where $m_2 < m_1$ and $i < j$. Let us move R_{m_2} from C_j to C_i, and the new partition is $\mathcal{P}' = \{C_1, ..., C'_i, ..., C'_j, ..., C_k\}$, where $C'_i = C_i \cup R_{m_2}$ and $C'_j = C_j - R_{m_2}$. The above operation does not change the width of C_i and C_j. It is clear that

$$E(\mathcal{P}') - E(\mathcal{P}) = b_{m_2}(W(C_i) - W(C_j)).$$

Since $W(C_i) \le W(C_j)$, we have $E(\mathcal{P}') \le E(\mathcal{P})$. By repeating the above operation, we can arrange the RVS's without increasing $E(\mathcal{P})$, such that if $R_{m_1} \subseteq C_i$ and $R_{m_2} \subseteq C_j$, where $i < j$, then $m_1 < m_2$. The theorem is proven. □

5.5. An optimal partitioning algorithm (OPA) based on DF

An immediate consequence of Theorem 5.2 is that for a fixed k, in an optimal partition $\mathcal{P} = \{C_1, C_2, ..., C_k\}$ which minimizes $\mathrm{Cost}(\mathcal{P})$, there must exist $0 = M_0 < M_1 < M_2 < \cdots < M_{k-1} < M_k = M$, such that $C_j = R_{M_{j-1}+1, M_j}$, for all $1 \le j \le k$. That is, each C_j is a set of consecutive RVS's. Hence, the optimal partitioning problem is to determine k and the values of $M_1, M_2, ..., M_{k-1}$, such that

$$\mathrm{Cost}(\mathcal{P}) = \sum_{j=1}^{k} T_c(C_j) = \sum_{j=1}^{k} T(R_{M_{j-1}+1, M_j})$$

is minimized, where $T(R_{M_{j-1}+1, M_j})$ is the computing time of $R_{M_{j-1}+1, M_j}$.

We define $\mathrm{Cost}(m, k)$ to be the $\mathrm{Cost}(\mathcal{P})$ of an optimal partition $\mathcal{P} = \{C_1, C_2, ..., C_k\}$ of $R_1 \cup R_2 \cup \cdots \cup R_m$, where $1 \le m \le M$ and $1 \le k \le m$.

The following theorem provides a dynamic programming formulation of $\mathrm{Cost}(m, k)$.

Theorem 5.3. $\mathrm{Cost}(m, k)$ *satisfies the following recurrence relation:*

$$\begin{aligned}
&\mathrm{Cost}(m, 1) = T(R_{1,m}), \ 1 \le m \le M; \\
&\mathrm{Cost}(m, k) = \min_{k-1 \le M_{k-1} \le m-1} \left\{ \mathrm{Cost}(M_{k-1}, k-1) + T(R_{M_{k-1}+1, m}) \right\}, \\
&2 \le k \le m \le M.
\end{aligned}$$

Algorithm 1: Calculating the optimal cost matrix and the optimal division matrix.

```
Input: The DF $f_A$ of a sparse matrix $A$.
Output: The optimal cost matrix $C[m,k]$ and the optimal division matrix $D[m,k]$.
for $m \leftarrow 1$ to $M$ do
    $C[m,1] \leftarrow T(R_{1,m})$;
end
for $k \leftarrow 2$ to $M$ do
    for $m \leftarrow k$ to $M$ do
        $C[m,k] \leftarrow \infty$;
        for $j \leftarrow k-1$ to $m-1$ do
            $q \leftarrow C[j,k-1] + T(R_{j+1,m})$;
            if $q < C[m,k]$ then
                $C[m,k] \leftarrow q$;
                $D[m,k] \leftarrow j$;
            end
        end
    end
end
```

Algorithm 2: Calculating the output of the optimal partitioning problem.

```
Input: The optimal cost matrix $C[m,k]$ and the optimal division matrix $D[m,k]$.
Output: The values of $k$ and the values of $M_1, M_2, ..., M_{k-1}$.
Cost $\leftarrow \infty$;
for $k' \leftarrow 1$ to $M$ do
    if $C[M,k'] <$ Cost then
        Cost $\leftarrow C[M,k']$;
        $k \leftarrow k'$;
    end
end
$i \leftarrow M$;
$j \leftarrow k$;
while $j \geq 2$ do
    $M_{j-1} \leftarrow D[i,j]$;
    $i \leftarrow D[i,j]$;
    $j \leftarrow j-1$;
end
```

Proof. The base case when $k=1$ is trivial, since the only partition is $\mathcal{P}=\{R_{1,m}\}$. When $k \geq 2$, the possible value of M_{k-1} is in the range $k-1 \leq M_{k-1} \leq m-1$. For each M_{k-1}, we have $\text{Cost}(m,k) = \text{Cost}(M_{k-1}, k-1) + T(R_{M_{k-1}+1,m})$. Hence, $\text{Cost}(m,k)$ takes the minimum value of all these possibilities. □

Theorem 5.3 suggests a dynamic programming algorithm to compute $\text{Cost}(m,k)$, and thus, solving our optimal partitioning problem. Our *optimal partitioning algorithm* (OPA) is presented in Algorithms 1 and 2. Algorithm 1 (where j stands for M_{k-1}) calculates an array $C[m,k]$ which saves the value of $\text{Cost}(m,k)$, and an array $D[m,k]$ which saves the value of M_{k-1} that gives the minimum value of $\text{Cost}(m,k)$. Such a dynamic programming algorithm can be completed in $O(M^3)$ time.

The optimal value of k is obtained by

$$\text{Cost}(M,k) = \min_{1 \leq k' \leq M} \{\text{Cost}(M,k')\}.$$

The optimal value of M_{k-1} can be recorded when $\text{Cost}(m,k)$ is calculated and recovered after Algorithm 1 is completed. The output of the optimal partitioning problem, i.e., the values of k and $M_1, M_2, ..., M_{k-1}$, can be generated by using Algorithm 2.

In fact, if $f_A(m) = b_m = 0$, then R_m can be removed from the computation of Algorithms 1 and 2. Assume that the number of RVS's in $\{R_m | f_A(m) \neq 0\}$ is M'. For a sparse matrix, M' is far less than the number of columns M. The computing complexity of OPA is actually $O((M')^3)$.

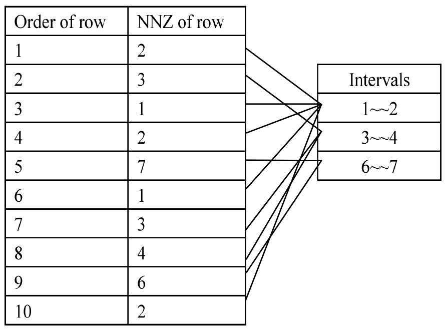
However, the processing time of OPA is unbearable if $M' > 100$. For most test cases, the number of RVS's partitioned from the sparse matrices is less than 100. Furthermore, we can observe that the sparse matrices with $M' > 100$ have many RVS's with very few rows. Some RVS's with similar width should be merged into one RVS. Then, the number of RVS's can be reduced to less than 100 by merging.

5.6. A reordering algorithm

A sparse matrix is split into k partitions $C_1, C_2, ..., C_k$ defined by $M_1, M_2, ..., M_{k-1}$ using OPA according to the *NNZ* of the rows. The rows of the sparse matrix are assigned to different C_j's by the *NNZ* of the rows. The *NNZ* of the rows in C_j are about the same. The classic binary search algorithm can be used to find out $C_j = R_{M_{j-1}+1,M_j}$ such that row r_i belongs to C_j,

Algorithm 3: The reordering algorithm.

```
Input: The values of $k$ and $M_0, M_1, ..., M_k$; the $NNZ$ of $N$ rows: $NNZ(r_1)$, $NNZ(r_2)$, ..., $NNZ(r_N)$.
Output: The index array $Index_1, Index_2, ..., Index_N$, where if $Index_i = j$, then $r_i$ belongs to $C_j = R_{M_{j-1}+1, M_j}$, where $1 \le j \le k$.
1  //The beginning and ending indices of the binary search algorithm
2  int $index_b, index_e$;
3  for $i \leftarrow 1$ to $N$ do
4      if $NNZ(r_i) > 0$ then
5          $index_b \leftarrow 0$;
6          $index_e \leftarrow k$;
7          repeat
8              $middle \leftarrow \lfloor (index_b + index_e)/2 \rfloor$;
9              if $NNZ(r_i) = M_{middle}$ then
10                 $Index_i \leftarrow middle$;
11                 return;
12             end
13             if $NNZ(r_i) < M_{middle}$ then
14                 $index_e \leftarrow middle$;
15             end
16             if $NNZ(r_i) > M_{middle}$ then
17                 $index_b \leftarrow middle + 1$;
18             end
19         until $index_b = index_e$;
20         $Index_i \leftarrow index_b$;
21     end
22 end
```



Order of row	NNZ of row
1	2
2	3
3	1
4	2
5	7
6	1
7	3
8	4
9	6
10	2

Fig. 2. Reordering of rows.

because $0 = M_0 < M_1 < M_2 < \cdots < M_{k-1} < M_k = M$. The reordering algorithm is presented in Algorithm 3. The computing complexity of the binary search algorithm is $O(\log_2 k)$, and k is far less than the number of columns M. So the computing complexity of assigning N rows to k partitions is $O(N \log_2 k)$. The process of rows reordering is to assign these rows to the corresponding partitions. So the computing complexity of Algorithm 3 is $O(N \log_2 k)$.

The reordering step is shown in Fig. 2, where a row r belongs to an interval if the $NNZ(r)$ is in the range of the interval. For the sparse matrix A, we have $k = 3$, $M_1 = 2$, $M_2 = 4$ by partitioning using OPA, and $C_1 = \{r_1, r_3, r_4, r_6, r_{10}\}$, $C_2 = \{r_2, r_7, r_8\}$, and $C_3 = \{r_5, r_9\}$ by reordering.

Furthermore, the dimension and size of blocks per grid and the dimension and size of threads per block are both important factors. Latency hiding and occupancy depend on the number of active warps per SM, which is implicitly determined by the execution parameters along with resource (register and shared memory) constraints [33]. When choosing the block size, it is important to remember that multiple concurrent blocks can reside on a SM, so occupancy is not determined by block size alone. In particular, a larger block size does not imply a higher occupancy. Note that when a thread block allocates more registers than those available on a SM, the kernel launch fails, as too much shared memory or too many threads are requested [33]. So the blocks obtained by Algorithm 3 should be divided into some sub-blocks if the sizes of them are too large. These sub-blocks can be equally partitioned from the blocks, because the rows in the blocks have similar widths.

6. A parallel implementation of SpMV on GPU

6.1. Parallel computing model on GPU

A sparse matrix is usually stored as CSR format before it is processed, and each block partitioned from the sparse matrix will be converted to ELL format for SpMV using other formats. Then, the sparse matrix is used to perform SpMV by CUDA using OPA, which includes four steps shown in Fig. 3.

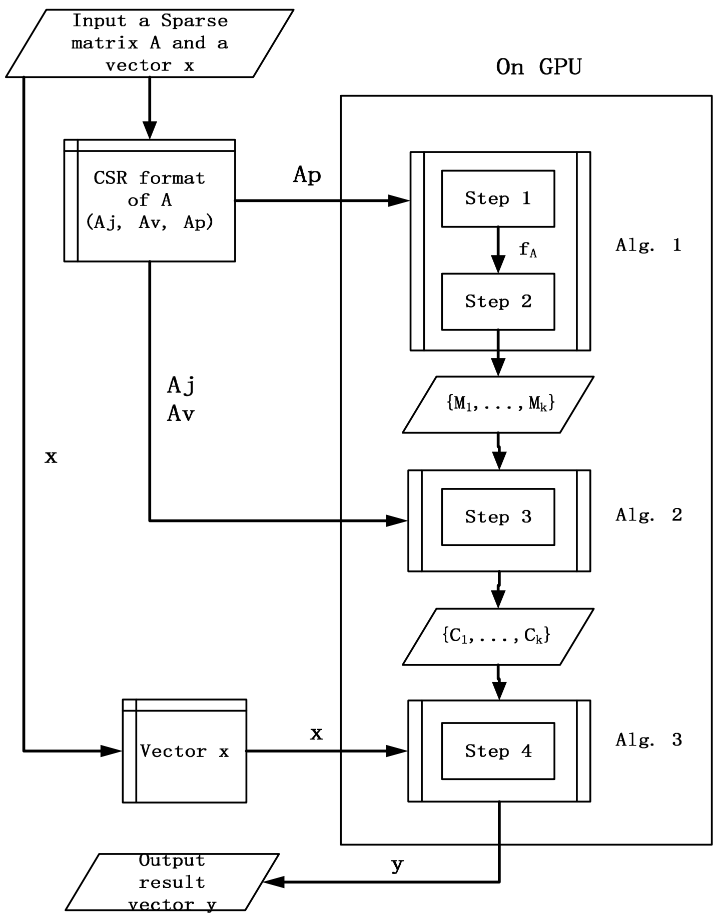


Fig. 3. The computing steps of SpMV using our approach on GPU.

1. Step 1: Construct the DF of the sparse matrix.
2. Step 2: Obtain the boundaries $M_1, M_2, ..., M_{k-1}$ of partitioning by OPA.
3. Step 3: Partition the sparse matrix into collections of RVS's using the reordering algorithm.
4. Step 4: Execute SpMV.

As shown in Fig. 3, Steps 1 and 2 can be processed by Algorithm 1 on GPU, whose input parameters are array **Ap** of CSR format and the output parameters are the boundary array $(M_0, M_1, ..., M_k)$ obtained using OPA. Step 3 is processed by the Algorithm 2 on GPU, whose input parameters are the output parameters of the Algorithm 1 and arrays **Aj** and **Av** of CSR format, and the output parameters are RVS's using the reordering algorithm. Step 4 is processed by the Algorithm 3, whose input parameters are the output parameters of the Algorithm 2 and vector x, and the output parameters are the result vector y. Each element of array **Ap** in Step 1 is assigned to a thread of CUDA, which needs to perform atomic addition. Then element $f_A[m]$ in the array f_A obtained by Step 1 is the number of row vectors whose number of non-zero elements is m. Assume that the length of array f_A is q if all elements with value 0 are removed from f_A. Then q is far less then N.

There are some parallel implementations of dynamic programming algorithms on GPU, which have better performance compared with that of CPU for Algorithms 1 and 2. Ref. [34] presented a framework for dynamic programming algorithms on GPU and reported speedups ranging from 6.1 to 25.8 on a Nvidia GTX 280 through the CUDA libraries. Ref. [35] presented an efficient parallel implementation of $O(n^3)$-time dynamic programming algorithm on the GPU, which attained a speedup factor of 247.5 on the NVIDIA GeForce GTX 580. Our implementation uses the sliding and mirroring arrangements method [35] and is to arrange the temporary data for coalesced access of the global memory in the GPU to minimize the memory access overhead, because the sizes of dynamic programming for the test cases are small and can be stored in the global memory on GPU.

For the outer loop (lines 3–22) in Algorithm 3, each i is performed on a thread of CUDA. The boundary array $(M_0, M_1, ..., M_k)$ is stored in the shared memory in GPU to reduce the access latency. The optimal partition $\mathcal{P} = \{C_1, C_2, ..., C_k\}$ obtained by Algorithm 2 is stored in a blocked ELL format, where C_i, $i = 1, 2, ..., k$, is stored in ELL format. The C_i blocks are assigned on a thread block of CUDA to perform SpMV, and each thread block is scheduled on a SM.

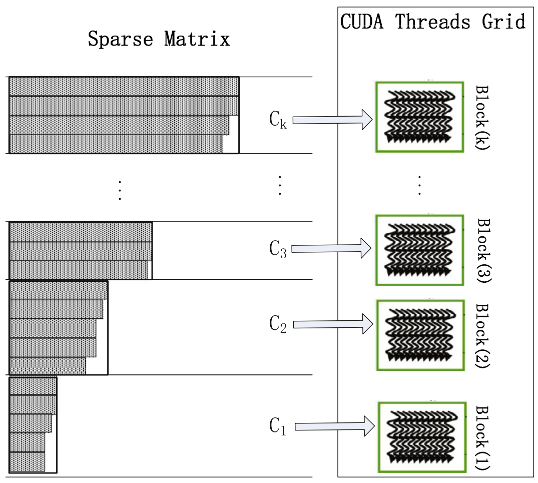


Fig. 4. Mapping of RVS's to thread blocks of CUDA.

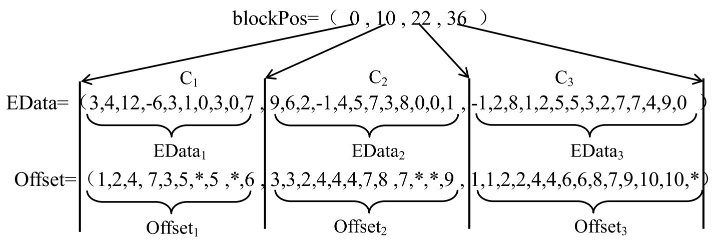

blockPos= (0 , 10 , 22 , 36)
C_1
C_2
C_3
EData= (3,4,12,-6,3,1,0,3,0,7 , 9,6,2,-1,4,5,7,3,8,0,0,1 , -1,2,8,1,2,5,5,3,2,7,7,4,9,0)
$EData_1$
$EData_2$
$EData_3$
Offset= (1,2,4, 7,3,5,*,5 ,*,6 , 3,3,2,4,4,4,7,8 ,7,*,*,9 , 1,1,2,2,4,4,6,6,8,7,9,10,10,*)
$Offset_1$
$Offset_2$
$Offset_3$

Fig. 5. The BCE format for the sparse matrix A.

Although the number of rows in C_i may not be the same, imbalance of partitions has little impact on the performance, because computing tasks between different SMs are independent and do not need to be synchronized.

6.2. The blocked stored format mixed CSR and ELL (BCE)

An $n \times n$ sparse matrix A is partitioned into the optimal partition $\mathcal{P} = \{C_1, C_2, ..., C_k\}$ by the boundary array $(M_0, M_1, ..., M_k)$ using OPA. Define an array **rowNo**, which stores the original indices of rows in A. If each block of $\mathcal{P}$ is a submatrix, the optimal partition $\mathcal{P} = \{C_1, C_2, ..., C_k\}$ can be stored in CSR format, which is shown in Figs. 4 and 5. Define an array **Ap** with $k+1$ elements, which stores the number of rows of C_i for $i = 1, 2, ..., k$. Then $\mathbf{Ap}[0] = 0$ and $\mathbf{Ap}[i] = \sum_{j=1}^{i} N(C_j)$ for $i = 1, 2, ..., k$. The block C_i is stored in two arrays $\mathbf{EData}_i$ and $\mathbf{Offset}_i$ using ELL format, whose lengths are $E(C_i)$. The ELL format stores the related block by columns, in order to assure the coalescent access to the data. The widths of all blocks are stored in an array **Aw** with k elements, where $\mathbf{Aw}[i] = W(C_i)$ for $i = 1, 2, ..., k$. Define an array **blockPos**, which stores the start positions of the blocks in the arrays **EData** and **Offset**, which are combined with all $\mathbf{EData}_i$ and $\mathbf{Offset}_i$ for $i = 1, 2, ..., k$. The length of **blockPos** is $k+1$, where $\mathbf{blockPos}[0] = 0$ and $\mathbf{blockPos}[i] = \sum_{j=1}^{i} E(C_i)$ for $i = 1, 2, ..., k$.

For the sparse matrix A in Section 5.1, if the boundary array (M_1, M_2) is $(2, 4)$, then A is split into three blocks $\{C_1, C_2, C_3\}$ using Algorithm 3. Assume that the original indices of rows $\{r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8, r_9, r_{10}\}$ in A are $(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)$. Moreover, C_1, C_2, and C_3 are $\{r_1, r_3, r_4, r_6, r_{10}\}$, $\{r_2, r_7, r_8\}$, and $\{r_5, r_9\}$ respectively using Algorithm 3. **rowNo** is $(1, 3, 4, 6, 10, 2, 7, 8, 5, 9)$. **Aw** is $(2, 4, 7)$ and **Ap** is $(0, 5, 8, 10)$. **blockPos**, **EData**, and **Offset** are shown in Fig. 5.

The kernel function based on BCE format on GPU is shown in Algorithm 4.

The index of the row r_i, which is computed by the current thread, is obtained in line 1 in Algorithm 4. Variables *start* represent the start position of the row r_i in the arrays **Edata** and **Offset**. SpMV is performed in line 8 for the row r_i.

If the block of BCE format is stored by CSR format, the BCE format will become GCSR format. If the sparse matrix is not reordered, the BCE format will become BELLPACK and SELLPACK formats. It has been theoretically proved that the performance of BCE is better than that of GCSR, BELLPACK, and SELLPACK formats.

Algorithm 4: The kernel function based on BCE format on GPU.

```
Input: The arrays blockPos, Ap, Aw, EData, and Offset; the vector $x$; the number of rows $N$ in the sparse matrix $A$.
Output: The result vector $y$.
1  int $r_i \leftarrow$ Ap[$blockIdx.x$] + $threadIdx.x$ ;
2  if $r_i < N$ then
3      int $start \leftarrow$ blockPos[$blockIdx.x$] + $threadIdx.x$ ;
4      int $width \leftarrow$ Aw[$blockIdx.x$] ;
5      int $rownum \leftarrow$ Ap[$blockIdx.x + 1$] $-$ Ap[$blockIdx.x$] ;
6      $y[r_i] \leftarrow 0$ ;
7      for $i \leftarrow 0$ to $width$ do
8          int $idx \leftarrow start + i \times rownum$ ;
9          if |Edata[$idx$])| $> 0$ then
10             $y[r_i] \leftarrow$ Edata[$idx$] $\times$ $x$[Offset[$idx$]] + $y[r_i]$ ;
11         end
12     end
13 end
```

Table 1
Parameters of the test computer.

Parameters	Descriptions	Values
S_i	the size of integer	4 Byte
S_s	the size of single	4 Byte
S_d	the size of double	8 Byte
C	the number of SPs	2496
f_s	the clock speed of SPs	0.705 GHz
f_a	the clock speed of the global memory	2.6 GHz
AW	the bus width of the global memory	320 bits
TW	the bandwidth of PCIe	8 GiB/s
s	the number of SMs	13

7. Experimental performance evaluation

7.1. Experiment settings

The following test environment has been used for all benchmarks. The test computer is equipped with two AMD Opteron 6376 CPUs running at 2.30 GHz and a NVIDIA K20c GPU. Each CPU has 16 cores. The GPU has 2496 CUDA processor cores, working on 0.705 GHz clock and 4 GB global memory with 320 bits bandwidth at 2.6 GHz clock, with CUDA compute capacity 3.5. The computing power f_p of a SM in K20c GPU is about 157.2 Gflop/s for single precision and about 89.4 Gflop/s for double precision. As for software, the test machine runs the 64bit Windows 7 and NVIDIA CUDA toolkit 7.0. The hardware parameters of the testing computer are shown in Table 1.

All benchmarks are chosen from the UF Sparse Matrix Collection [2], whose features are shown in Table 2, where W is the maximum width of the sparse matrix. 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 these libraries provide CUDA development tools and source codes.

CuBLAS [36] offers three levels of library functions for dense matrices.

CuSparse [37] provides three levels of functions for sparse matrices, with the first level for sparse vector and dense vector operations, the second level for sparse matrix and dense vector operations and the third level for sparse matrix and dense matrix operations. It includes three functions of SpMV, which use the CSR, BSR, and HYB formats respectively. The general BSR format has two parameters, *rowBlockDim* and *colBlockDim*. *rowBlockDim* is number of rows within a block and *colBlockDim* is number of columns within a block. If $rowBlockDim = colBlockDim$, general BSR format is the same as BSR format. If $rowBlockDim = colBlockDim = 1$, general BSR format is the same as CSR format. There is an analytical procedure for the sparse matrix at the beginning of the BSR function of cuSparse library, and the procedure will not be executed again after the BSR function is called for the first time for the same sparse matrix. In order to get rid of the analysis time in the single version test time, the BSR function is called once in advance, and then the single and double precision versions are called. The run time of the single version function does not contain the analysis time in the test to obtain the exact computation time of the single version function. HYB is a hybrid format of ELL and COO, such as the corresponding function for SpMV has a parameter, which has four values: 0, *AUTO*, *USER*, and *MAX*. The function will automatically select a segmentation threshold to divide 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 is tested using *AUTO*, *MAX*, and 0 respectively. A sparse matrix stored in the CSR format must be loaded into GPU by PCIe bus and the result vector is returned after SpMV has been performed for all stored

Table 2
General information of all sparse matrices used in the evaluation.

No.	Sparse Matrices	N	W	NNZ
1	Schenk_ISEI/ohne2	181343	3411	11063545
2	Janna/CoupCons3D	416800	76	22322336
3	Oberwolfach/bone010	986703	42	36326514
4	Janna/Serena	1391349	228	32961525
5	Janna/ML_Geer	1504002	74	110879972
6	Schenk_AFE/af_shell10	1508065	251	27090195
7	Janna/Flan_1565	1564794	69	59485419
8	Gleich/wikipedia-20051105	1634989	75547	19753078
9	Janna/Cube_Coup_dt0	2164760	52	64685452
10	Freescale/Freescale1	3428755	25	18920347
11	Rajat/rajat31	4690002	1252	20316253
12	DIMACS10/channel-500x100x100-b050	4802000	12	85362744
13	vanHeukelum/cage15	5154859	47	99199551
14	Freescale/circuit5M	5558326	1290501	59524291
15	DIMACS10/adaptive	6815744	4	27248640
16	DIMACS10/delaunay_n23	8388608	19	50331568
17	DIMACS10/road_central	14081816	8	33866826
18	DIMACS10/hugetrace-00020	16002413	3	47997626
19	DIMACS10/delaunay_n24	16777216	23	100663202
20	DIMACS10/road_usa	23947347	8	57708624

formats. So the transmission time of all stored formats is the same. But CSR format must be converted to the corresponding compression formats for HYB, ELL, COO, BSR, and BCE.

NVIDIA provides another library, CUSP [38], to offer SpMV for the GPU platform, which 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. But for most test cases, the DIA format cannot be performed, because the diagonal features of the test cases are not obvious.

Intel Math Kernel Library (Intel MKL) accelerates math processing routines that increase application performance and reduce development time. Intel MKL includes highly vectorized and threaded Linear Algebra, Fast Fourier Transforms (FFT), Vector Math and Statistics functions [39]. MKL has 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 tests and performance evaluation

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

(1) SpMV is tested using our method and compared to that using BSR, HYB, ELL, CSR, and COO on GPU.
(2) Linear solver is tested using SpMV on GPU.
(3) SpMV is tested using our method on GPU and compared to that using MKL on multi-core CPU.

7.2.1. Test of SpMV on GPU

The process of SpMV includes three steps, which are transmission, pretreatment, and computation. A sparse matrix A and a vector x must be loaded into the global memory of GPU by PCIe bus before SpMV is executed on GPU, and then a result vector x is returned from GPU after SpMV has been executed. The sparse matrix A is usually stored in CSR format before A is processed because of better compression efficiency. So the transmission time is the same for different computing methods. The pretreatment process includes matrix partitioning and format conversion for BSR, HYB, ELL, and COO functions of cuSparse. The pretreatment process includes matrix partitioning using OPA, reordering using Algorithm 3, and format conversion of BCE for our method. But there is no pretreatment process for CSR.

For the 20 test cases, the computing time of SpMV using our method, BSR, HYB, ELL, COO, and CSR are shown in Table 3, where *BCE* represents the computing time of SpMV using our method and NA means that the sparse matrix cannot be computed using the format.

For the 20 test cases, the performance improvements in percentage of computing time using our method over BSR, HYB, ELL, COO, and CSR are shown in Table 4. The performance improvement in percentage is calculated by $(t_1 - t_2)/t_1 \times 100$, where t_2 is the computing time of SpMV using our method and t_1 is that of BSR, HYB, ELL, COO, or CSR. The test cases expect Schenk_AFE/af_shell10 cannot be computed using ELL format for double precision. For all test cases, it is observed from Table 4 that the average performance improvements in percentage of single precision are 12.3%, 11.7%, 16.5%, 27.7%, and 21.7% using our method compared with BSR, HYB, ELL, COO, and CSR respectively, and those of double precision are 12.3%, 12.2%, 20.5%, 28.1%, and 21.7% respectively.

For vanHeukelum/cage15, SpMV cannot be processed using BSR, because the storage spaces using BSR are more than the memory of GPU. ELL format has poor adaptability for big sparse matrices, because some rows are padded zero according to the longest row, leading to the size of data too big to be processed on GPU. The long rows are divided into

Table 3
Computing time of SpMV using our method, BSR, HYB, ELL, COO, and CSR (unit: milliseconds).

No.	Single Precision						Double Precision					
	BCE	*BSR*	*HYB*	*ELL*	*COO*	*CSR*	*BCE*	*BSR*	*HYB*	*ELL*	*COO*	*CSR*
1	3.2	4.1	3.2	NA	3.9	3.3	3.3	4.8	3.3	NA	4.0	3.4
2	4.5	4.7	4.6	5.1	5.3	5.2	2.8	2.9	4.7	NA	6.0	5.4
3	7.2	7.4	7.2	8.2	11.0	8.4	6.5	6.6	7.4	NA	11.0	8.5
4	7.7	8.9	7.8	10.2	10.8	9.6	8.1	9.6	8.2	NA	11.0	9.9
5	9.0	9.2	13.9	14.9	25.4	15.8	13.2	13.4	14.0	NA	27.0	16.0
6	8.1	8.5	8.5	8.2	21.6	8.9	6.6	6.7	8.6	8.3	31.0	9.1
7	7.1	7.3	11.1	13.1	17.1	13.0	10.3	10.4	11.4	NA	17.3	13.0
8	15.8	27.6	16.5	NA	16.2	33.5	17.1	35.7	17.4	NA	17.7	34.6
9	6.5	6.6	13.8	14.4	19.4	14.1	8.0	8.5	14.0	NA	20.0	15.0
10	11.2	13.8	12.4	12.1	14.7	12.7	12.2	14.1	14.5	NA	15.0	13.2
11	13.3	17.5	17.8	NA	18.5	14.4	15.1	20.6	18.2	NA	19.0	15.2
12	16.8	19.3	17.3	17.7	24.0	19.4	16.9	20.1	17.4	NA	24.0	19.5
13	24.9	29.6	25.1	27.2	59.9	28.9	25.1	NA	27.0	NA	57.0	30.0
14	23.4	45.7	24.4	NA	33.2	353.8	25.2	47.9	25.4	NA	34.8	362.0
15	21.9	23.9	23.8	24.0	22.6	24.2	20.1	20.5	26.0	NA	23.0	24.5
16	27.9	29.3	28.4	31.6	29.6	31.2	29.2	30.2	29.4	NA	29.8	31.8
17	48.0	49.8	56.2	50.1	49.2	49.1	45.1	46.3	53.0	NA	49.5	49.2
18	49.6	51.6	61.0	61.3	55.2	54.6	51.7	52.2	62.3	NA	55.4	55.0
19	55.8	57.9	62.3	59.2	59.4	59.4	57.7	58.4	62.5	NA	59.6	59.6
20	71.3	75.5	72.8	74.0	75.3	72.2	72.5	77.6	73.3	NA	76.0	75.0

Table 4
Performance improvements of SpMV Using our method over BSR, HYB, ELL, COO, and CSR (unit: %).

No.	Single Precision					Double Precision				
	BSR	*HYB*	*ELL*	*COO*	*CSR*	*BSR*	*HYB*	*ELL*	*COO*	*CSR*
1	22.0	0.0	NA	17.9	3.0	31.3	0.0	NA	17.5	2.9
2	4.3	2.2	11.8	15.1	13.5	3.4	40.4	NA	53.3	48.1
3	2.7	0.0	12.2	34.5	14.3	1.5	12.2	NA	40.9	23.5
4	13.5	1.3	24.5	28.7	19.8	15.6	1.2	NA	26.4	18.2
5	2.2	35.3	39.6	64.6	43.0	1.5	5.7	NA	51.1	17.5
6	4.7	4.7	1.2	62.5	9.0	1.5	23.3	20.5	78.7	27.5
7	2.7	36.0	45.8	58.5	45.4	1.0	9.6	NA	40.5	20.8
8	42.8	4.2	NA	2.5	52.8	52.1	1.7	NA	3.4	50.6
9	1.5	52.9	54.9	66.5	53.9	5.9	42.9	NA	60.0	46.7
10	18.8	9.7	7.4	23.8	11.8	13.5	15.9	NA	18.7	7.6
11	24.0	25.3	NA	28.1	7.6	26.7	17.0	NA	20.5	0.7
12	13.0	2.9	5.1	30.0	13.4	15.9	2.9	NA	29.6	13.3
13	15.9	0.8	8.5	58.4	13.8	NA	7.0	NA	56.0	16.3
14	48.8	4.1	NA	29.5	93.4	47.4	0.8	NA	27.6	93.0
15	8.4	8.0	8.8	3.1	9.5	2.0	22.7	NA	12.6	18.0
16	4.8	1.8	11.7	5.7	10.6	3.3	0.7	NA	2.0	8.2
17	3.6	14.6	4.2	2.4	2.2	2.6	14.9	NA	8.9	8.3
18	3.9	18.7	19.1	10.1	9.2	1.0	17.0	NA	6.7	6.0
19	3.6	10.4	5.7	6.1	6.1	1.2	7.7	NA	3.2	3.2
20	5.6	2.1	3.6	5.3	1.2	6.6	1.1	NA	4.6	3.3

ELL and COO using HYB format to reduce zero-padded, so HYB has better adaptability than that of ELL. For Rajat/rajat31, Schenk_ISEI/ohne2, Gleich/wikipedia-20051105, and Freescale/circuit5M, the performance of our method improves significantly compared to that of CSR and BSR, because the *NNZ* of rows has big deviation, leading to very low parallel efficiency of GPU using CSR and BSR, and the performance using HYB is good, because the long rows are divided into ELL and COO to reduce deviation.

7.2.2. Test of linear solver using SpMV on GPU

For our method, BSR, HYB, ELL, and COO, there are a pretreatment time in the total processing time, and there is no pretreatment time for CSR because it does not need format conversion. However, since the coefficient matrix is fixed in the process of solving a sparse linear system, matrix partitioning is processed only once, and SpMV is executed many times for solving a large-scale sparse linear system using an iterative method, so it has little impact on solving a large-scale sparse linear system using pretreatment technique. The linear equations, whose coefficient matrices are the test cases, are solved by the iterative solving algorithms of Lis library [40] in the experiments, and then the numbers of iterations are obtained by solving these linear equations. The same initial vector was used for all cases. For the 20 test cases, the numbers *Iters* of iterations are shown in Table 5. Assume that the pretreatment time and computing time are represented by T_p and T_c respectively. Then processing time of SpMV for solving the test cases is calculated by $T_p + Iters \times T_c$. For the 20 test cases,

Table 5
Processing time of SpMV using our method, BSR, HYB, ELL, COO, and CSR for solving linear equations (unit: second).

No.	*Iters*	Single Precision						Double Precision				
		BCE	*BSR*	*HYB*	*ELL*	*COO*	*CSR*	*BCE*	*BSR*	*HYB*	*COO*	*CSR*
1	9811	31.1	40.2	31.9	NA	38.7	32.3	32.3	47.3	32.4	39.3	33.2
2	5542	25.1	26.1	25.3	28.5	29.3	28.8	15.4	16.0	25.9	33.4	30.1
3	9554	68.9	70.7	69.0	78.2	105.2	79.9	62.4	63.3	71.0	105.3	81.0
4	3865	29.8	34.4	30.3	39.5	41.9	37.2	31.3	37.2	31.8	42.7	38.2
5	9267	84.2	85.8	128.7	137.9	236.2	146.9	123.5	124.5	129.9	250.8	148.3
6	6902	55.8	58.8	58.6	56.4	149.5	61.6	45.7	46.3	59.3	214.1	62.8
7	10001	70.8	72.9	110.7	131.1	171.1	129.6	103.1	104.0	113.7	173.0	130.0
8	9827	155.35	271.6	162.6	NA	159.7	329.3	168.1	351.4	170.8	174.0	340.2
9	7051	46.3	46.6	97.2	101.4	137.0	99.8	56.8	59.6	98.9	141.4	105.8
10	8670	96.9	120.3	107.2	104.8	127.6	110.2	106.3	122.6	125.4	130.1	114.0
11	4362	58.3	76.5	77.9	NA	80.7	62.7	66.0	89.9	79.6	83.0	66.3
12	5513	93.2	106.5	95.4	97.8	132.3	107.1	93.2	110.7	95.9	132.5	107.8
13	24	1.3	1.4	0.8	0.8	1.9	0.7	1.4	NA	0.9	1.8	0.7
14	7113	167.9	305.0	174.9	NA	237.6	2516.3	181.0	341.9	182.0	249.0	2574.9
15	4998	109.5	119.5	118.9	119.9	112.8	120.7	100.5	102.7	130.1	115.1	122.5
16	6115	170.6	179.3	174.0	193.1	181.0	190.6	178.7	184.7	179.7	182.1	194.3
17	8585	412.18	427.7	482.4	429.9	422.5	421.4	387.2	397.9	455.3	424.9	422.5
18	8006	397.1	413.3	488.6	491.0	442.1	437.2	413.9	418.6	499.1	443.5	440.2
19	10000	558.0	579.2	623.4	592.3	594.7	594.2	577.6	584.4	624.8	596.7	595.7
20	9260	660.3	699.5	674.2	685.8	697.6	668.7	671.4	719.5	679.3	704.1	694.5

Table 6
Performance improvements of SpMV using our method over BSR, HYB, ELL, COO, and CSR for solving linear equations (unit: %).

No.	Single Precision					Double Precision				
	BSR	*HYB*	*ELL*	*COO*	*CSR*	*BSR*	*HYB*	*ELL*	*COO*	*CSR*
1	22.6	2.5	NA	19.6	3.7	31.7	0.3	NA	17.8	2.7
2	3.8	0.8	11.9	14.3	12.8	3.8	40.5	NA	53.9	48.8
3	2.5	0.1	11.9	34.5	13.8	1.4	12.1	NA	40.7	23.0
4	13.4	1.7	24.6	28.9	19.9	15.9	1.6	NA	26.7	18.1
5	1.9	34.6	38.9	64.4	42.7	0.8	4.9	NA	50.8	16.7
6	5.1	4.8	1.1	62.7	9.4	1.3	22.9	20.1	78.7	27.2
7	2.9	36.0	46.0	58.6	45.4	0.9	9.3	NA	40.4	20.7
8	42.8	4.5	NA	2.7	52.8	52.2	1.6	NA	3.4	50.6
9	0.6	52.4	54.3	66.2	53.6	4.7	42.6	NA	59.8	46.3
10	19.5	9.6	7.5	24.1	12.1	13.3	15.2	NA	18.3	6.8
11	23.8	25.2	NA	27.8	7.0	26.6	17.1	NA	20.5	0.5
12	12.5	2.3	4.7	29.6	13.0	15.8	2.8	NA	29.7	13.5
13	7.1	−62.5	−62.5	31.6	−85.7	NA	−55.6	NA	22.2	−100.0
14	45.0	4.0	NA	29.3	93.3	47.1	0.5	NA	27.3	93.0
15	8.4	7.9	8.7	2.9	9.3	2.1	22.8	NA	12.7	18.0
16	4.9	2.0	11.7	5.7	10.5	3.2	0.6	NA	1.9	8.0
17	3.6	14.6	4.1	2.4	2.2	2.7	15.0	NA	8.9	8.4
18	3.9	18.7	19.1	10.2	9.2	1.1	17.1	NA	6.7	6.0
19	3.7	10.5	5.8	6.2	6.1	1.2	7.6	NA	3.2	3.0
20	5.6	2.1	3.7	5.3	1.3	6.7	1.2	NA	4.6	3.3

the processing time of SpMV for solving the test cases is shown in Table 5, but the solving time using ELL is not listed in Table 6 for double precision, because the test cases expect that Schenk_AFE/af_shell10 cannot be computed using ELL format for double precision. The solving time of Schenk_AFE/af_shell10 using ELL is 57.2 seconds.

For the 20 test cases, the performance improvements in percentage using our method over BSR, HYB, ELL, COO, and CSR for solving the test cases are shown in Table 6. The performance improvement in percentage is calculated by $(t_1 - t_2)/t_1 \times 100$, where t_2 is the processing time of SpMV using our method and t_1 is that of BSR, HYB, ELL, COO, or CSR. We have the following observations from Table 6.

For the test cases, the average performance of single precision is improved by 11.7% and 26.4% by using our method compared with BSR and COO, and that of double precision is improved by 12.2% and 26.4%. The average performance of single precision is improved by 12.3% and 22.0% using our method compared with HYB and CSR except vanHeukelum/cage15, and that of double precision is improved by 12.4% and 21.8% except vanHeukelum/cage15. The average performance of single precision is improved by 16.9% by using our method compared with ELL except Schenk_ISEI/ohne2, Gleich/wikipedia-2005110, Rajat/rajat31, vanHeukelum/cage15, and Freescale/circuit5M, and that of double precision is improved by 20.1% for Schenk_AFE/af_shell10 only. The transformation time of the test cases is shown in Table 7 for SpMV. Although in a single SpMV operation, the transformation time occupied a large proportion, it has little impact on the performance of solving equations except vanHeukelum/cage15, because the number of iterations is large. For vanHeukelum/cage15, the per-

Table 7
Transformation time of BCE, BSR, HYB, ELL, COO from CSR (unit: milliseconds).

No.	Single Precision					Double Precision				
	BCE	*BSR*	*HYB*	*ELL*	*COO*	*BCE*	*BSR*	*HYB*	*ELL*	*COO*
1	26.2	23.6	25.7	NA	65.9	68.8	66.3	34.0	NA	66.0
2	45.8	15.4	37.1	34.0	27.3	50.2	19.4	41.0	NA	26.0
3	77.9	17.8	71.0	55.6	212.5	54.4	52.1	52.0	NA	204.0
4	87.0	46.4	78.4	53.1	169.5	112.6	106.5	54.0	NA	165.0
5	574.1	564.7	183.0	168.6	637.6	774.1	459.5	140.0	NA	624.0
6	164.3	163.4	51.3	39.4	98.9	173.2	166.9	83.1	63.1	76.1
7	127.1	105.8	111.6	90.7	340.6	74.5	70.3	92.0	NA	330.0
8	338.4	335.7	125.8	NA	122.8	339.2	339.5	126.0	NA	131.0
9	155.0	73.9	120.6	97.4	360.3	135.5	34.0	127.0	NA	367.0
10	66.1	64.7	56.3	39.9	66.9	135.6	111.6	75.0	NA	71.0
11	153.1	143.0	67.9	NA	74.8	212.9	207.1	97.0	NA	82.0
12	127.5	74.4	107.5	70.9	163.0	212.7	201.7	115.0	NA	153.0
13	694.2	643.2	188.0	139.5	418.5	765.1	NA	206.0	NA	465.0
14	1519.2	1272.1	1444.0	NA	1475.3	1561.7	1253.3	1354.0	NA	1417.0
15	98.4	71.0	95.2	72.4	84.4	157.2	146.0	122.0	NA	93.0
16	163.1	119.4	154.6	92.1	128.8	223.8	218.2	177.0	NA	131.0
17	206.7	203.5	192.4	147.4	168.5	311.0	306.4	229.0	NA	179.0
18	244.5	232.8	227.8	161.4	185.0	331.8	326.9	263.0	NA	187.0
19	316.2	244.7	294.0	233.9	282.7	403.8	395.9	280.0	NA	253.0
20	473.5	367.5	434.5	255.9	366.2	506.6	506.6	405.0	NA	318.0

Table 8
Performance comparison of SpMV using our method on GPU over MKL on CPU.

No.	Single Precision			Double Precision		
	MKL (s)	*BCE* (s)	Improvement (%)	*MKL* (s)	*BCE* (s)	Improvement (%)
1	0.194	0.062	68.118	0.219	0.121	44.725
2	0.390	0.114	70.823	0.421	0.151	64.142
3	0.643	0.196	69.520	0.718	0.231	67.844
4	0.564	0.189	66.496	0.640	0.280	56.275
5	2.302	0.891	61.294	2.340	1.268	45.796
6	0.634	0.264	58.438	0.609	0.430	29.450
7	1.047	0.302	71.156	1.123	0.356	68.316
8	0.492	0.436	11.346	0.484	0.475	1.760
9	1.151	0.344	70.122	1.280	0.432	66.289
10	0.377	0.136	64.011	0.406	0.245	39.692
11	0.409	0.231	43.629	0.437	0.336	23.117
12	0.795	0.271	65.893	0.843	0.469	44.394
13	2.206	1.000	54.662	2.277	1.248	45.180
14	1.994	1.814	9.015	1.917	1.877	2.089
15	0.357	0.169	52.705	0.375	0.263	29.765
16	0.548	0.274	50.073	0.593	0.434	26.823
17	0.550	0.372	32.285	0.608	0.516	15.090
18	0.693	0.424	38.755	0.748	0.591	20.999
19	1.115	0.536	51.941	1.186	0.774	34.738
20	0.797	0.660	17.205	0.873	0.811	7.119

formances of HYB, ELL, and CSR are better than that of our method, because its iteration number is less. For Janna/Serena, the computing time of HYB is close to that of our method, and the pretreatment time of HYB is half of that of our method. So the performance of HYB is better than that of our method for double precision.

7.2.3. Test of SpMV on GPU and CPU

The total processing time of SpMV using our method on GPU is the sum of transmission time, pretreatment time, and computing time. But there are no transmission time and pretreatment time using the MKL library on a multi-core CPU. Assume that the processing time of using our method and MKL are represented by *BCE* and *MKL* respectively. For the 20 test cases, *BCE* and *MKL* are shown in Table 8. Then *BCE* and *MKL* are calculated by $T_t + T_p + T_c$, where T_t, T_p, and T_c are the transmission time, pretreatment time, and computing time respectively. The performance improvement in percentage is calculated by $(MKL - BCE)/MKL \times 100$.

For all test cases, it is observed from Table 8 that the average performance improvement in percentage of single precision using our method on GPU is 51.374% for MKL on CPU, and that of double precision is 36.680%.

8. Conclusions

In this paper, we use a DF to build an optimal partitioning strategy and a reordering algorithm 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 and reordering algorithm are based on the distribution characteristics of non-zeros in a sparse matrix. We propose a blocked stored format mixing CSR and ELL for a sparse matrix. Although the sizes of the blocks partitioned from the sparse matrix may be different, the storage spaces using BCE format have better compression effect, because the widths of rows in the same block are the similar. According to the experimental results, it is found that the average performance improvements in percentage of single precision are 12.3%, 11.7%, 16.5%, 27.7%, and 21.7% using our method compared with BSR, HYB, ELL, COO, and CSR respectively, and those of double precision are 12.3%, 12.2%, 20.5%, 28.1%, and 21.7% respectively. In future work, we will explore the new solving methods based on SpMV for large scale sparse linear systems of equations.

Acknowledgments

The research was partially funded by the National Natural Science Foundation of China (Grant Nos. 61572175, 61370095, 61472124), and the Key Program of National Natural Science Foundation of China (Grant No. 61432005), and the National Outstanding Youth Science Program of National Natural Science Foundation of China (Grant No. 61625202), and the International (Regional) Cooperation and Exchange Program of National Natural Science Foundation of China (Grant No. 61661146006).

References

[1] V.W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A.D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, P. Dubey, Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU, in: Proceedings of the 37rd. International Symposium on Computer Architecture, ISCA '10, IEEE Press, Saint-Malo, France, 2010.
[2] T. Davis, Y. Hu, The University of Florida sparse matrix collection, ACM Trans. Math. Softw. 38 (1) (2011) 1–25.
[3] W. Yang, K. Li, Y. Liu, L. Shi, C. Wang, Optimization of quasi diagonal matrix–vector multiplication on GPU, Int. J. High Perform. Comput. Appl. 28 (2) (2014) 181–193.
[4] K. Li, W. Yang, K. Li, Performance analysis and optimization for SpMV on GPU using probabilistic modeling, IEEE Trans. Parallel Distrib. Syst. 26 (1) (2015) 196–205.
[5] J. Bolz, I. Farmer, E. Grinspun, P. Schroder, Sparse matrix solvers on the GPU: conjugate gradients and multigrid, ACM Trans. Graph. 22 (3) (2003) 917–924.
[6] N. Bell, M. Garland, Implementing sparse matrix–vector multiplication on throughput-oriented processors, in: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, ACM, 2009, p. 18.
[7] T. Stanimire, J. Dongarra, M. Baboulin, Towards dense linear algebra for hybrid GPU accelerated manycore systems, Parallel Comput. 36 (5) (2010) 232–240.
[8] E.-J. Im, K. Yelick, R. Vuduc, Sparsity: optimization framework for sparse matrix kernels, Int. J. High Perform. Comput. Appl. 18 (1) (2004) 135–158.
[9] B.C. Lee, R.W. Vuduc, J.W. Demmel, K.A. Yelick, Performance models for evaluation and automatic tuning of symmetric sparse matrix–vector multiply, in: Parallel Processing, 2004, International Conference on, ICPP 2004, IEEE, 2004, pp. 169–176.
[10] J.W. Choi, A. Singh, R.W. Vuduc, Model-driven autotuning of sparse matrix vector multiply on GPUs, in: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '10, ACM, 2010, pp. 115–126.
[11] A. Monakov, A. Lokhmotov, A. Avetisyan, Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures, DBLP, 2010.
[12] X. Feng, H. Jin, R. Zheng, K. Hu, J. Zeng, Z. Shao, Optimization of sparse matrix vector multiplication with variant CSR on GPUs, in: Proceedings of the IEEE 17th International Conference on Parallel and Distributed Systems, ICPADS '11, IEEE, 2011, pp. 165–172.
[13] A. Buluc, S. Williams, L. Oliker, J. Demmel, Reduced-bandwidth multithreaded algorithms for sparse matrix–vector multiplication, in: Parallel and Distributed Processing Symposium, IPDPS '11, IEEE, 2011, pp. 721–733.
[14] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, A. Basermann, A.R. Bishop, Sparse matrix vector multiplication on GPGPU clusters: a new storage format and a scalable implementation, in: Proceedings of the IEEE 26th International in Parallel and Distributed Processing Symposium Workshops and PhD Forum, IPDPSW '12, IEEE, 2012, pp. 1696–1702.
[15] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, J. Demmel, Optimization of sparse matrix vector multiplication on emerging multicore platforms, Parallel Comput. 35 (3) (2009) 178–194.
[16] B. Boyer, J.G. Dumas, P. Giorgi, Exact sparse matrix vector multiplication on GPU's and multicore architectures, in: Proceedings of the 4th International Workshop on Parallel and Symbolic Computation, IPSC '10, ACM, 2010, pp. 80–88.
[17] P. Guo, L. Wang, P. Chen, A performance modeling and optimization analysis tool for sparse matrix vector multiplication on GPUs, IEEE Trans. Parallel Distrib. Syst. 25 (5) (2014) 1112–1123.
[18] J.C. Pichel, F.F. Rivera, Sparse matrix–vector multiplication on the single-chip cloud computer many-core processor, J. Parallel Distrib. Comput. 73 (12) (2013) 1539–1550.
[19] V. Karakasis, T. Gkountouvas, K. Kourtis, G. Goumas, N. Koziris, An extended compression format for the optimization of sparse matrix vector multiplication, IEEE Trans. Parallel Distrib. Syst. 24 (10) (2013) 1930–1940.
[20] B. Schmidt, H. Aribowo, H.V. Dang, Iterative sparse matrix vector multiplication for accelerating the block Wiedemann algorithm over GF(2) on multi-graphics processing unit systems, Concurr. Comput., Pract. Exp. 25 (4) (2013) 586–603.
[21] M. Cenk, C. Nègre, M.A. Hasan, Improved three-way split formulas for binary polynomial and Toeplitz matrix vector products, IEEE Trans. Comput. 62 (7) (2013) 1345–1361.
[22] M.A. Hasan, C. Nègre, Multiway splitting method for Toeplitz matrix vector product, IEEE Trans. Comput. 62 (7) (2013) 1467–1471.
[23] P.R. Amestoy, T.A. Davis, I.S. Duff, An approximate minimum degree ordering algorithm, SIAM J. Matrix Anal. Appl. 17 (4) (1996) 886–905.
[24] J.C. Pichel, D.E. Singh, J. Carretero, Reordering algorithms for increasing locality on multicore processors, in: Proc. of the IEEE Int. Conf. on High Performance Computing and Communications, HPCC '08, IEEE, 2008, pp. 123–130.
[25] E. Cuthill, Several Strategies for Reducing the Bandwidth of Matrices, 1972.

[26] A. George, Nested dissection of a regular finite element mesh, SIAM J. Numer. Anal. 10 (2) (1973) 345–363.
[27] G. Karypis, V. Kumar, METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices, Tech. rep., Minneapolis, USA, 1998.
[28] A.N. Yzelman, R.H. Bisseling, Two-dimensional cache-oblivious sparse matrix vector multiplication, Parallel Comput. 37 (12) (2011) 806–819.
[29] A.N. Yzelman, D. Roose, High-level strategies for parallel shared-memory sparse matrix vector multiplication, IEEE Trans. Parallel Distrib. Syst. 25 (1) (2014) 116–125.
[30] NVIDIA, Nvidia cuda c programming guide, Tech. rep., 2013.
[31] W. Yang, K. Li, Z. Mo, K. Li, Performance optimization using partitioned SPMV on GPUs and multicore CPUs, IEEE Trans. Comput. 64 (9) (2015) 2623–2636.
[32] K. David, H. Wen-mei, Programming Massively Parallel Processors: A Hands-on Approach, vol. 11, Morgan Kaufmann of Elsevier, 2013.
[33] NVIDIA, Cuda c best practices guide, Tech. rep., March 2015.
[34] P. Steffen, R. Giegerich, M. Giraud, GPU parallelization of algebraic dynamic programming, in: International Conference on Parallel Processing and Applied Mathematics, 2009, pp. 290–299.
[35] K. Nishida, K. Nakano, Y. Ito, Accelerating the dynamic programming for the optimal polygon triangulation on the GPU, in: Algorithms and Architectures for Parallel Processing, 2012.
[36] NVIDIA, cuBLAS library, Tech. rep., March 2015.
[37] NVIDIA, cuSPARSE library, Tech. rep., March 2015.
[38] NVIDIA, CUSP library, Tech. rep., 2014.
[39] Intel, Intel Math Kernel Library, Tech. rep., March 2007.
[40] A. Nishida, Experience in Developing an Open Source Scalable Software Infrastructure in Japan, Springer, Berlin, Heidelberg, 2010.