Performance Analysis and Optimization for SpMV on GPU Using Probabilistic Modeling

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

Abstract—This paper presents a unique method of performance analysis and optimization for sparse matrix-vector multiplication (SpMV) on GPU. 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 method does not need additional benchmarks to get optimized parameters, which are calculated directly through the probability mass function (PMF). We make the following contributions. (1) We present a PMF to analyze precisely the distribution pattern of non-zero elements in a sparse matrix. The PMF can provide theoretical basis for the compression of a sparse matrix. (2) Compression efficiency of COO, CSR, ELL, and HYB can be analyzed precisely through the PMF, and combined with the hardware parameters of GPU, the performance of SpMV based on COO, CSR, ELL, and HYB can be estimated. Furthermore, the most appropriate format for SpMV can be selected according to estimated value of the performance. Experiments prove that the theoretical estimated values and the tested values have high consistency. (3) For HYB, the optimal segmentation threshold can be found through the PMF to achieve the optimal performance for SpMV. Our performance modeling and analysis are very accurate. The order of magnitude of the estimated speedup and that of the tested speedup for each of the ten tested sparse matrices based on the three formats COO, CSR, and ELL are the same. The percentage of relative difference between an estimated value and a tested value is less than 20% for over 80% cases. The performance improvement of our algorithm is also effective. The average performance improvement of the optimal solution for HYB is over 15% compared with that of the automatic solution provided by CUSPARSE lib.

Index Terms—GPU, performance modeling, probability mass function, sparse matrix-vector multiplication.

1 Introduction

Sparse matrix-vector multiplication (SpMV) is an essential operation in solving linear systems and partial differential equations. 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. This paper addresses this challenge by presenting a performance modeling and analysis method to estimate and optimize SpMV performance on GPU using a probabilistic model.

N. Bell and M. Garland [1] proposed and implemented SpMV CUDA kernels for some storage formats, including COO (coordinate format), CSR (compressed sparse row format), ELL (ELLPACK format), and HYB (hybrid format). Based on our experiments using cuSPARSE lib [2], which is developed by NVIDIA, COO is the most intuitive storage format and usually has worse performance than other formats; but is not sensitive to the distribution of non-zero elements per row. CSR usually has good performance for sparse matrices with large numbers of non-zero elements; but is sensitive to the distribution of non-zero elements per row. ELL is usually good for a sparse matrix with nearly equal and small number of non-zero elements per row. HYB has better performance when the matrix has small number of non-zero elements per row, and most rows are nearly equal but there may be a few irregular rows with much more non-zero elements, where the matrix is split into two parts, i.e., ELL and COO, such that the most rows which are nearly equal are stored by ELL and the other few irregular rows with much more non-zero elements are stored by COO. We observed that different matrices may have their own most appropriate storage formats to achieve the best performance. Besides, we also notice that the performance of HYB is effected by the proportion of the two parts. All these observations motivate us to build a mathematical model to analyze the distribution characteristics of non-zero elements in a sparse matrix and to estimate the execution times of multiple SpMV kernels, and furthermore, to help choose an optimal SpMV solution (i.e., storage format and storage strategy) for a target sparse matrix.

The present paper makes the following unique contributions to performance analysis and optimization for SpMV on GPU. (1) We present a probability mass function (PMF) to analyze precisely the distribution pattern of non-zero elements in a sparse matrix. The PMF can provide theoretical basis for the compression of a sparse matrix. (2) Compression efficiency of COO, CSR, ELL,

- *Kenli Li, Wangdong Yang, and Keqin Li are with the College of Information Science and Engineering, Hunan University, Changsha, Hunan 410082, China; and the National Supercomputing Center in Changsha, Changsha, Hunan 410082, China.*
  E-mail: lkl@hnu.edu.cn; yangwangdong@163.com; likq@hnu.edu.cn.
- *Keqin Li is also with the Department of Computer Science, State University of New York, New Paltz, New York 12561, USA.*
  E-mail: lik@newpaltz.edu.

Manuscript received Month Day, 2013; revised Month Day, 2014.

and HYB can be analyzed precisely through the PMF, and combined with the hardware parameters of GPU, the performance of SpMV based on COO, CSR, ELL, and HYB can be estimated. Furthermore, the most appropriate format for SpMV can be selected according to the estimated values of performance. Experiments prove that the theoretical estimated values and the tested values have high consistency. (3) For HYB, the optimal segmentation threshold can be found through the PMF to achieve the optimal performance for SpMV.

Our performance modeling is based on PMF, which fully reflects the distribution characteristics of non-zero elements in a sparse matrix and does not need any benchmark matrices to get the properties and parameters for SpMV. Our performance modeling consists of three steps, i.e., probability analysis, performance estimation, and strategy optimization. Firstly, the PMF for the target matrix is built according to the analysis of the distribution of non-zero elements per row. Secondly, the performance estimation formulas of COO, CSR, ELL, and HYB can be established according to the PMF for the target matrix and the storage structures of these formats. Lastly, the performance of SpMV using these formats can be estimated using the estimated formulas through the input hardware parameters of GPU. For COO, CSR, and ELL, the format with the smallest estimate will be selected for SpMV to get the best performance. For HYB, the optimal segmentation threshold can be found through performance estimation. The target matrix is split into COO and ELL by the optimal segmentation threshold to get the best performance for SpMV.

We use a probabilistic method to analyze the performance of SpMV. 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 method does not need additional benchmarks to get optimized parameters, which are calculated directly through the PMF. Some methods also use some distribution characteristics of a sparse matrix to analyze the performance of SpMV, such as the number of non-zeros. However, these methods do not combine various storage formats for comprehensive analysis of the performance of SpMV due to lack of quantitative techniques.

In this paper, we use SpMV CUDA kernels developed by NVIDIA [2] and NVIDIA GTX 645M for our performance modeling and experiments. According to our experiments on 10 representative matrices (totally 58 test cases), our performance modeling and analysis is very accurate. The order of magnitude of the estimated speedup and that of the tested speedup for each sparse matrix based on the three formats COO, CSR, and ELL are the same. The percentage of relative difference between an estimated value and a tested value is less than 20% for over 80% cases. The optimal SpMV solutions of the 10 matrices are reported by our optimal solutions for HYB. Specifically, the performance improvement of our algorithm is very effective. The average performance improvement of the optimal solution for HYB is over 15% compared with that of the automatic solution provided by CUSPARSE lib.

2 Related Work

In this section, we review related research in implementation of SpMV for different formats, optimization of SpMV on GPU, and performance modeling and prediction. Due to space limitation, this section is moved to Section 2 of the supplementary material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/00.0000/TPDS.2014.000.

3 Overview of GPU and CUDA

In this section, we provide an overview of the GPU computing architecture and parallel programming with CUDA, which is moved to Section 3 of the supplementary material, available online.

4 SpMV Performance Modeling

Sparse matrices arise form various domains and their distribution patterns of non-zero elements can be very specific. Taking into consideration the structure of a sparse matrix can dramatically improve the performance of SpMV. However, there is no general storage format that is efficient for all kinds of sparse matrices. 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 probability mass function (PMF), and get numerical characteristics of sparsity distribution by a probabilistic method. The suitable storage format can be selected by numerical characteristics of a sparsity distribution.

4.1 PMF of Sparse Matrices

A PMF is a function that gives the probability that a discrete random variable is exactly equal to some value [34].

4.1.1 *Definition of Probability Mass Function*

A is a sparse matrix. N is the number of rows in A and M is the number of columns in A. The discrete random variable X represents the number of non-zeros of one row in A. The range of values of the discrete random variable X is $\Omega_X = \{0, 1, 2, ..., M\}$. For each $i = 0, 1, 2, ..., M$, when the value of X is i, it represents the event $\{X = i\}$. Define another set $B = \{b_0, b_1, b_2, ..., b_M\}$. Each b_i, $i = 0, 1, 2, ..., M$, represents the number of rows, each of which contains i non-zeros exactly. For each $i = 0, 1, 2, ..., M$, $p_i = b_i/N$ is the probability of the event

$\{X = i\}$. Define the probability mass function of discrete random variable X as P, which is mathematically characterized by the following expression:

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

$X = 0$ represents the event that there is no non-zero in one row. $p_k = 0$ represents that the row with k non-zeros does not exist in the sparse matrix A.

4.1.2 *Relevant Probability and Numerical Characteristics*

The probability of $X \leq K$ is equal to the probability of $0 \leq X \leq K$. The probability of $X > K$ is equal to the probability of $M \geq X > K$. The probabilities are expressed by

$$P(X \leq K)=\sum_{i=0}^{K}p_i, \tag{2}$$

$$P(X > K)=\sum_{i=K+1}^{M}p_i. \tag{3}$$

The conditional probabilities are expressed as

$$P(X=k|X \leq K)=\begin{cases} p_k/\sum_{i=0}^{K}p_i, & k \leq K;\\ 0, & k > K;\end{cases} \tag{4}$$

$$P(X=k|X > K)=\begin{cases} p_k/\sum_{i=K+1}^{M}p_i, & k > K;\\ 0, & k \leq K.\end{cases} \tag{5}$$

$E(X)$ is the expectation operator. σ is the standard deviation. γ is the skewness of random variable X. The eigenvalues are expressed by

$$E(X)=\sum_{i=0}^{M}(i \times p_i), \tag{6}$$

$$E(X|X \leq K)=\sum_{i=0}^{K}(i \times p_i)/P(X \leq K), \tag{7}$$

$$E(X|X > K)=\sum_{i=K+1}^{M}(i \times p_i)/P(X > K), \tag{8}$$

$$\sigma=\sqrt{E(X-E(X))^2}, \tag{9}$$

$$\gamma=\frac{E(X-E(X))^3}{\sigma^3}. \tag{10}$$

α_i is the ith fractile of W-fractiles for X if α_i satisfies

$$P(X \leq \alpha_i)=i/W. \tag{11}$$

The number of all non-zeros (NNZ) in the sparse matrix A is expressed as

$$NNZ=E(X) \times N. \tag{12}$$

$NNZ(X \leq K)$ represents the total number of non-zeros of the rows whose numbers of non-zeros are less than or equal to K:

$$NNZ(X \leq K)=E(X|X \leq K) \times N \times P(X \leq K). \tag{13}$$

$NNZ(X > K)$ represents the total number of non-zeros of the rows whose numbers of non-zeros are greater than K:

$$NNZ(X > K)=E(X|X > K) \times N \times P(X > K). \tag{14}$$

4.2 Storage Space Analysis

In this section, we analyze S_{coo}, S_{csr}, S_{ell}, and S_{hyb}, i.e., the storage space used by four storage formats COO, CSR, ELL, and HYB, which is moved to Section 4 of the supplementary material, available online. In the following, we list the main results.

$$S_{coo}=S_s \times NNZ+2 \times S_i \times NNZ. \tag{15}$$

$$S_{csr}=S_s \times NNZ+S_i \times NNZ+S_i \times (N+1). \tag{16}$$

$$S_{ell}=S_s \times N \times K+S_i \times N \times K. \tag{17}$$

$$S_{ell}=S_s \times N_e \times K+S_i \times N_e \times K+S_i \times N_e. \tag{18}$$

$$\begin{aligned}S_{coo}=&(NNZ(X>K)-K \times P(X>K) \times N) \times S_s\\&+(NNZ(X>K)-K \times P(X>K) \times N) \times 2 \times S_i.\end{aligned} \tag{19}$$

$$\begin{aligned}S_{hyb}&=S_{coo}+S_{ell}\\&=NNZ(X>K) \times (S_s+2S_i)\\&-K \times P(X>K) \times N \times (S_s+2S_i)\\&+S_s \times N_e \times K+S_i \times N_e \times K+S_i \times N_e.\end{aligned} \tag{20}$$

4.3 Performance Analysis for SpMV

The performance of SpMV on GPU depends on two parts, i.e., data transfer time (DTT) and computing time (CT). The execution time T of SpMV on GPU can be expressed as

$$T=DTT+CT. \tag{21}$$

DTT contains two parts, i.e., from host to device and from device to host. The sparse matrix and vector are transferred from host to device and the result vector is returned from device. DTT is expressed as

$$DTT=\frac{\text{size of data}}{B}, \tag{22}$$

where B is the transfer bandwidth of PCIe, which connects CPU and GPU. CT also contains two parts, i.e., the computing time on cores (CTC) and access memory time (AM), including read and write, which mainly considers access global memory time. The computing time on cores contains two parts, i.e., multiplication (CTC_m) and addition (CTC_a). The rate of multiplication and addition on SP can be considered to be the same, because SP can

TABLE 1
Variables and Definitions

Variable	Description
T	$DTT + CT$, execution time of SpMV
DTT	Data transfer time between host and device
B	The bandwidth of PCIe
CT	$CTC + AM$, computing time on SMs
CTC	$CTC_m + CTC_a$, computing time on cores
CTC_m	Computing time of multiplication on cores
CTC_a	Computing time of addition on cores
F	The rate of multiplication or addition on SP
F_i	The rate of $\times$ or $+$ on SP for integers
F_s	The rate of $\times$ or $+$ on SP for single-precision
F_d	The rate of $\times$ or $+$ on SP for double-precision
AM	Access memory time
BW	The bus width of the global memory
CR	The clock rate of the global memory
C	The number of stream processors (SP)
W	The number of threads per warp
BS	The number of threads per block
S_i	The size of an integer
S_s	The size of a single-precision floating point number
S_d	The size of a double-precision floating point number

execute a multiplication and an addition operation with the same time. F is the rate of multiplication or addition on SP. F is divided into three types: F_i (integer), F_s (single-precision), and F_d (double-precision), depending on the data type.

Access global memory also contains two parts, i.e., read the sparse matrix and read and write vectors. BW is the bus width of the global memory on GPU. The global memory can read once for continuous data with bus width length. If the data set is continuous, the latency of access to the global memory can be hidden. The access time to the global memory of a thread can be calculated by Eq. (23):

$$AM = \left\lceil \frac{\text{size of } DS}{RW} \right\rceil \times \frac{1}{CR}, \tag{23}$$

where DS is the data set stored in the global memory, which is read from the global memory once; RW is the length of the continuous data and RW should not exceed BW; and CR is the clock rate of the global memory.

C is the number of stream processor (SP). W is the number of threads per warp. BS is the number of threads per block. The shared memory can be used in warp to reduce the access latency of the global memory. Some data can be read into the shared memory and can be accessed by each thread in the block.

All variables are described in Table 1.

4.3.1 Performance Analysis for COO

The data transferred from host to device contains three arrays of COO and the vector, whose total size is $S_{coo} + S_s \times N$. The size of the vector returned from device is $S_s \times N$. Thus, the DTT for SpMV using COO is expressed by Eq. (24) according to Eq. (22):

$$DTT = \frac{S_{coo} + 2 \times S_s \times N}{B}. \tag{24}$$

According to Eq. (15), DTT for SpMV using COO can be expressed as

$$DTT = \frac{(S_s + 2 \times S_i) \times NNZ + 2 \times S_s \times N}{B}. \tag{25}$$

The three arrays of COO can be read only one element each time in a thread, because each non-zero is assigned to a thread when SpMV is computed on GPU, leading to inappropriate use of the shared memory. The data accessed in each thread contains two parts, i.e., one element from each of the three arrays of COO and the corresponding element in the vector, where the size of each element is S_i, S_i, S_s, and S_s. So, $RW = S_i$ or $RW = S_s$ is the size of one element. The GPU has C threads to compute at the same time, because GPU has C cores. Thus, there are NNZ/C rounds of parallel computing for SpMV on GPU. AM is expressed by Eq. (26) according to Eq. (23):

$$AM = \frac{NNZ}{C} \times \left(\frac{2 \times S_i}{CR \times S_i} + \frac{2 \times S_s}{CR \times S_s} \right). \tag{26}$$

Each thread only performs one multiplication with computing time $1/F_s$ (for single-precision). The total time of multiplication is given by Eq. (27), because GPU has NNZ/C rounds of parallel computing for SpMV:

$$CTC_m = \frac{NNZ}{C \times F_s}. \tag{27}$$

The results of the above computing need to be summarized per row. The results of one row is summed by one warp. The number of warps which can perform at the same time on GPU is C/W, and there are $N \times W/C$ rounds of parallel summation on GPU. Because the number of non-zeros in a row is unknown, the mean $E(X)$ of the number of non-zeros in a row is used for the length of summation per row when the rows are summed. The parallel reduction algorithm is used when the summation of a row is calculated. Hence, CTC_a is expressed as

$$CTC_a = \frac{N \times W}{C} \times \left\lceil \frac{E(X)}{W} \right\rceil \times \frac{1}{F_s}. \tag{28}$$

Finally, CT for SpMV using COO is given by

$$\begin{aligned} CT &= AM + CTC_m + CTC_a \\ &= \frac{NNZ}{C} \times \left(\frac{2 \times S_i}{CR \times S_i} + \frac{2 \times S_s}{CR \times S_s} \right) \\ &+ \frac{NNZ}{C \times F_s} + \frac{N \times W}{C} \times \left\lceil \frac{E(X)}{W} \right\rceil \times \frac{1}{F_s}. \end{aligned} \tag{29}$$

4.3.2 Performance Analysis for CSR

The data transferred from host to device contains three arrays of CSR and the vector, whose total size is S_{csr}+$S_s \times N$. The size of the vector returned from device is $S_s \times N$. Therefore, the DTT for SpMV using CSR is expressed by Eq. (30) according to Eq. (22):

$$DTT = \frac{S_{csr} + 2 \times S_s \times N}{B}. \tag{30}$$

According to Eq. (16), DTT for SpMV using CSR can be expressed as

$$DTT = \frac{(S_s + S_i) \times \text{NNZ} + S_i \times (N+1) + 2 \times S_s \times N}{B}. \quad (31)$$

Each row is assigned to a thread when SpMV is computed on GPU using the CSR format, and there are N/C rounds of parallel computing for SpMV on GPU. Because the length of a row is a random variable, the computing time of a warp is the maximum computing time of the threads in the warp, which is determined by the longest row in the warp. We define set Q as

$$Q = \{X | X \in W\text{-fractile of } X\}. \quad (32)$$

The W-fractiles of X are break points of X, which are evenly divided into W subsets. The mean $E(Q)$ is used as the length of a row when we estimate the computing time. Hence, the computing time of multiplication in a thread is $E(Q)/F_s$ (for single-precision), and the total computing time of multiplication is expressed by

$$CTC_m = \frac{N}{C} \times \frac{E(Q)}{F_s}. \quad (33)$$

The results of multiplication must be accumulated to a value in each thread. Notice that one more addition must be performed, because the position of columns and values must be calculated by the index of rows array. We need to increase the time by $E(Q)/F_i$, because the index of the array is an integer. Thus, the total computing time of addition is expressed as

$$CTC_a = \frac{N}{C} \times \left(\frac{E(Q)}{F_s} + \frac{E(Q)}{F_i} \right). \quad (34)$$

The data set of the value and column index arrays can be read in BW length data once, because the arrays can be continuously accessible in a thread. So, the access time of the two arrays is $\left(\left\lceil \frac{S_s \times E(Q)}{BW} \right\rceil + \left\lceil \frac{S_i \times E(Q)}{BW} \right\rceil \right) \times \frac{1}{CR}$ in a thread according to Eq. (23). However, the vector X cannot be continuously accessible in a thread and RW is the size of one element. Thus, the access time is $\frac{S_s \times E(Q)}{CR \times S_s}$ according to Eq. (23). An element is read only from the array of row indices in a thread and the access time is $\frac{S_i}{CR \times S_i}$. Similarly, the write time of the vector is $\frac{S_s}{CR \times S_s}$. The effect of using the shared memory is not obvious, because the length of each row is different and access to the location of the vector X is random. Hence, the total access memory time is expressed by

$$\begin{aligned} AM = \frac{N}{C} \Bigg(\left(\left\lceil \frac{S_s \times E(Q)}{BW} \right\rceil + \left\lceil \frac{S_i \times E(Q)}{BW} \right\rceil \right) \times \frac{1}{CR} \\ + \frac{S_i}{CR \times S_i} + \frac{S_s}{CR \times S_s} + \frac{S_s \times E(Q)}{CR \times S_s} \Bigg). \end{aligned} \quad (35)$$

Finally, CT for SpMV using CSR is given by

$$\begin{aligned} CT &= AM + CTC_m + CTC_a \\ &= \frac{N}{C} \Bigg(\left(\left\lceil \frac{S_s \times E(Q)}{BW} \right\rceil + \left\lceil \frac{S_i \times E(Q)}{BW} \right\rceil \right) \times \frac{1}{CR} \\ &+ \frac{2}{CR} + \frac{S_s \times E(Q)}{CR \times S_s} + E(Q) \times \left(\frac{2}{F_s} + \frac{1}{F_i} \right) \Bigg). \end{aligned} \quad (36)$$

4.3.3 Performance Analysis for ELL

The data transferred from host to device contains three arrays of ELL and the vector, whose total size is $S_{ell}+S_s \times N$. The size of the vector returned from device is $S_s \times N$. Hence, the DTT for SpMV using ELL is expressed as

$$DTT = \frac{S_{ell} + 2 \times S_s \times N}{B}. \quad (37)$$

According to Eq. (18), DTT for SpMV using ELL can be express as

$$\begin{aligned} DTT = &\frac{P(X > 0) \times N \times K \times (S_s + S_i)}{B} \\ + &\frac{S_i \times P(X > 0) \times N + 2 \times S_s \times N}{B}. \end{aligned} \quad (38)$$

Each row is also assigned to a thread when SpMV is calculated on GPU using the ELL format, and there are $P(X > 0) \times N/C$ rounds of parallel computing for SpMV on GPU. The computing mode of ELL is the same as that of CSR. So, the access time of two arrays of values and columns is $\left(\left\lceil \frac{S_i \times K}{BW} \right\rceil + \left\lceil \frac{S_s \times K}{BW} \right\rceil \right) \times \frac{1}{CR}$ according to Eq. (23). The vector X cannot be continuously accessible in a thread and RW is the size of one element. However, the data set with the length of a block can be read into the shared memory from the global memory to reduce the time to access the global memory. Therefore, the access time is $\frac{S_s \times (K - BS)}{CR \times S_s}$ according to Eq. (23). An element is read only from the array of row indices in a thread and the access time is $\frac{S_i}{CR \times S_i}$. The write time of the vector is $\frac{S_s}{CR \times S_s}$. Hence, AM is expressed by

$$\begin{aligned} AM = &\frac{P(X > 0) \times N}{C} \times \\ &\Bigg(\left(\left\lceil \frac{S_i \times K}{BW} \right\rceil + \left\lceil \frac{S_s \times K}{BW} \right\rceil \right) \times \frac{1}{CR} \\ &+ \frac{S_s \times (K - BS)}{CR \times S_s} + \frac{S_i}{CR \times S_i} + \frac{S_i}{CR \times S_i} \Bigg). \end{aligned} \quad (39)$$

The mean $E(Q)$ is used as the length of a row in the same way as that of CSR. Thus, the computing time of multiplication in a thread is $E(Q)/F_s$ (for single-precision), and the total computing time of multiplication is expressed as

$$CTC_m = \frac{P(X > 0) \times N}{C} \times \frac{E(Q)}{F_s}. \quad (40)$$

The results of multiplication must be accumulated to a value in each thread. However, the position of values must not be calculated in a thread, because two arrays of columns and values are one-to-one. Hence, CTC_a for SpMV using ELL is expressed as

$$CTC_a = \frac{P(X>0)\times N}{C} \times \frac{E(Q)}{F_s}. \tag{41}$$

Finally, CT for SpMV using ELL is given by

$$\begin{aligned} CT &= AM + CTC_m + CTC_a \\ &= \frac{P(X>0)\times N}{C} \times \\ &\quad \Bigg(\left(\left\lceil \frac{S_i \times K}{BW} \right\rceil + \left\lceil \frac{S_s \times K}{BW} \right\rceil \right) \times \frac{1}{CR} \\ &\quad + \frac{S_s \times (K - BS)}{CR \times S_s} + \frac{2}{CR} + \frac{2 \times E(Q)}{F_s} \Bigg). \end{aligned} \tag{42}$$

4.3.4 *Performance Analysis for HYB*

The DTT for SpMV using HYB is expressed by

$$DTT = \frac{S_{hyb} + 2 \times S_s \times N}{B}. \tag{43}$$

Since HYB contains COO and ELL, the above DTT can be expressed as

$$DTT = \frac{S_{coo} + S_{ell} + 2 \times S_s \times N}{B}. \tag{44}$$

According to Eq. (20), we have

$$\begin{aligned} &DTT = \\ &\frac{(NNZ(X>K) - K \times P(X>K) \times N) \times (S_s + 2S_i)}{B} \\ &+ \frac{(S_s + S_i) \times P(X>0) \times N \times K}{B} \\ &+ \frac{S_i \times P(X>0) \times N + 2 \times S_s \times N}{B}. \end{aligned} \tag{45}$$

The CT of HYB is made up of two parts: CT of COO and CT of ELL. Notice that N, NNZ, and $E(X)$ in Eq. (29) are replaced by $P(X > K) \times N$, $NNZ(X > K) - K \times P(X > K) \times N$, and $E(X|X > K) - K$. CT_{coo} can be calculated according to the Eq. (29) as follows:

$$\begin{aligned} &CT_{coo} = \\ &\frac{NNZ(X>K) - K \times P(X>K) \times N}{C} \times \left(\frac{4}{CR} + \frac{1}{F_s} \right) \\ &+ \frac{P(X>K) \times N \times W}{C} \times \left\lceil \frac{E(X|X>K) - K}{W} \right\rceil \times \frac{1}{F_s}. \end{aligned} \tag{46}$$

The $E(Q)$ in the Eq. (42) is replaced by $E(X|X \le K)$, and CT_{ell} can be calculated as follows:

$$\begin{aligned} &CT_{ell} = \\ &+ \frac{P(X>0) \times N}{C} \Bigg(\left(\left\lceil \frac{S_i \times K}{BW} \right\rceil + \left\lceil \frac{S_s \times K}{BW} \right\rceil \right) \times \frac{1}{CR} \\ &+ \left\lceil \frac{S_s \times (K - BS)}{BW} \right\rceil \times \frac{1}{CR} + \frac{2}{CR} + \frac{2E(X|X \le K)}{F_s} \Bigg). \end{aligned} \tag{47}$$

CT of HYB can be expressed as

$$CT = CT_{coo} + CT_{ell}. \tag{48}$$

According to the Eqs. (46) and (47), CT is

$$\begin{aligned} &CT = \\ &\frac{NNZ(X>K) - K \times P(X>K) \times N}{C} \times \left(\frac{4}{CR} + \frac{1}{F_s} \right) \\ &+ \frac{P(X>K) \times N \times W}{C} \times \left\lceil \frac{E(X|X>K) - K}{W} \right\rceil \times \frac{1}{F_s} \\ &+ \frac{P(X>0) \times N}{C} \Bigg(\left(\left\lceil \frac{S_i \times K}{BW} \right\rceil + \left\lceil \frac{S_s \times K}{BW} \right\rceil \right) \times \frac{1}{CR} \\ &+ \left\lceil \frac{S_s \times (K - BS)}{BW} \right\rceil \times \frac{1}{CR} + \frac{2}{CR} + \frac{2E(X|X \le K)}{F_s} \Bigg). \end{aligned} \tag{49}$$

Finally, T can be calculated according to the Eq. (21):

$$T = DDT + CT_{coo} + CT_{ell}. \tag{50}$$

5 Performance Optimizing for SpMV

The appropriate storage format can be selected according to the distribution pattern of a sparse matrix through the above performance analysis. The performance optimization workflow for SpMV using CSR, ELL, and COO consists of two steps, i.e., establishment of a probability model and estimation of performance. The optimization method is to choose the format with the smallest estimated execution time for SpMV. The optimization workflow for SpMV using COO, CSR, and ELL is shown in Fig. 1. The performance estimation based on single-precision or double-precision for SpMV can be calculated by entering different parameters. The appropriate storage format can be selected by comparing the different estimates. The format with the minimum estimation value is the best suited storage format for the sparse matrix. The actual computing time and the estimated value are consistent to be shown in Figs. 3, 4, 5, and 6. Generally, the original sparse matrices are stored in the COO format. The performance estimation for COO, CSR, and ELL, which are described by Eqs. (25), (29), (31), (36), (38), and (42), can be obtained through Algorithm 1.

The performance optimizing workflow for SpMV using HYB consists of three steps, i.e., establishment of a probability model, split of the sparse matrix into COO and ELL by a threshold K, and estimation of the performance of SpMV under different thresholds K. The optimization workflow for SpMV using HYB is shown in Fig. 2. If a sparse matrix A is stored in HYB, A should be split into COO and ELL formats. A parameter K should be provided when splitting. The choice of parameter K can affect the performance of SpMV using HYB. The optimal value of K can minimize the value of Eq. (50). Hence, the optimal solution of K can be found by solving Eq. (50). The method for finding the optimal value of K is presented in Algorithm 2. The optimization method is to choose the threshold K with the smallest estimate to split the sparse matrix for HYB.

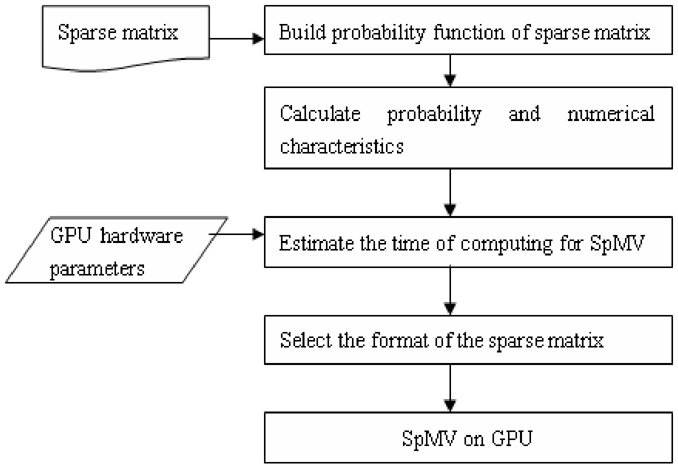
Sparse matrix
Build probability function of sparse matrix
Calculate probability and numerical characteristics
GPU hardware parameters
Estimate the time of computing for SpMV
Select the format of the sparse matrix
SpMV on GPU

Fig. 1. The optimization workflow for SpMV using COO, CSR, and ELL.

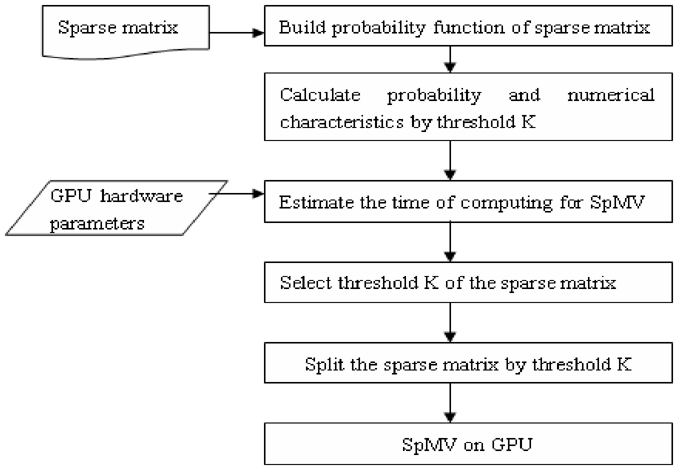
Sparse matrix
Build probability function of sparse matrix
Calculate probability and numerical characteristics by threshold K
GPU hardware parameters
Estimate the time of computing for SpMV
Select threshold K of the sparse matrix
Split the sparse matrix by threshold K
SpMV on GPU

Fig. 2. The optimization workflow for SpMV using HYB.

Algorithm 1 Performance Estimation for SpMV.

```
Require: Three arrays of COO: $\mathbf{row}, \mathbf{column}, \mathbf{value}$; the numbers of rows and columns: $N$ and $M$; the number of non-zeros: $NNZ$; the parameters of GPU: $B$, $F$, $BW$, $CR$, $C$, $W$, $BS$, $S_i$, $S_s$, $S_d$.
Ensure: The performance estimation values of COO, CSR, and ELL: $T_{coo}$, $T_{csr}$, $T_{ell}$.
1: int * $num\_nonzeros\_row$; //Store the number of non-zeros per row.
2: int * $sum\_num\_row$; //Store the number of rows with the same number of non-zeros.
3: Get_num_nonzeros($rows$, $num\_nonzeros\_row$, $N$, $NNZ$); //Get the number of non-zeros per row and stored in $num\_nonzeros\_row$.
4: Sum_num_row($num\_nonzeros\_row$, $sum\_num\_row$, $M$); //Sum the number of rows with the same number of non-zeros and stored in $sum\_num\_row$.
5: $T_{coo}$ = Get_estimate_coo($sum\_num\_row$, $B$, $F$, $BW$, $CR$, $C$, $W$, $BS$, $S_i$, $S_s$, $S_d$);
6: $T_{csr}$ = Get_estimate_csr($sum\_num\_row$, $B$, $F$, $BW$, $CR$, $C$, $W$, $BS$, $S_i$, $S_s$, $S_d$);
7: $T_{ell}$ = Get_estimate_ell($sum\_num\_row$, $B$, $F$, $BW$, $CR$, $C$, $W$, $BS$, $S_i$, $S_s$, $S_d$);
8: return $T_{coo}$, $T_{csr}$, $T_{ell}$.
```

Algorithm 2 Performance Optimization for HYB.

```
Require: Three arrays of COO: $\mathbf{row}, \mathbf{column}, \mathbf{value}$; the numbers of rows and columns: $N$ and $M$; the number of non-zeros: $NNZ$; the parameters of GPU: $B$, $F$, $BW$, $CR$, $C$, $W$, $BS$, $S_i$, $S_s$, $S_d$.
Ensure: The optimal solution of the threshold $K$.
1: int $T = MAX$; //$T$ takes a larger number.
2: int * $num\_non-zeros\_row$;
3: int * $sum\_num\_row$;
4: Get_num_nonzeros($rows$,$num\_nonzeros\_row$,$N$,$NNZ$);
5: Sum_num_row($num\_nonzeros\_row$,$sum\_num\_row$,$M$);
6: for each $i$ in $[E(X)..\max\{i \mid P(X=i) > 0\}]$ do
7:     $T_{hyb}$ = Get_estimate_hyb($sum\_num\_row$, $B$, $F$, $BW$, $CR$, $C$, $W$, $BS$, $S_i$, $S_s$, $S_d$, $i$);
8:     if $T_{hyb} < T$ then
9:         $K = i$;
10:        $T = T_{hyb}$;
11:    end if
12: end for;
13: return $K$.
```

6 Experimental Evaluation

6.1 Experiment Settings

The following test environment has been used for all benchmarks. A personal computer is equipped with one Intel Core i5-3230M running at 2.6 GHz and a NVIDIA Geforce GT 645M GPU. The CPU has 2 cores with 4 threads. The GPU has 384 CUDA processor cores, working on 0.78 GHz clock rate and 2 GB global memory with 128 bits bandwidth and 0.9 GHz clock rate, with CUDA compute capacity 3. As for software, the test machine ran the 64-bit Windows 8 and NVIDIA CUDA toolkit 5.0. All the evaluation results are averaged after running 100 times.

All benchmarks are chosen from the UF Sparse Matrix Collection [35], whose features are shown in Table 1 of Section 5.1 of the supplementary material. Most of these matrices are derived from scientific computing and real engineering applications. The probability distributions of these sparse matrices are shown in Table 2 of Section 5.1 of the supplementary material.

NVIDIA Corporation provides two libraries (CUBLAS and CUSPARSE) to support matrix computation. Both 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 also provides three levels of functions for sparse matrices, with the first level for ADD operation, second for MUL operation of SpMV [2], and the third

TABLE 2
Parameter Table

Parameter	Description	Value
S_i	The size of an integer	4 bytes
S_s	The size of single-precision number	4 bytes
S_d	The size of double-precision number	8 bytes
B	The bandwidth of PCIe	16 GB/s
F	The clock rate of SP	0.76 Gflop/s
BW	The bus width of the global memory	128 bits
CR	The clock rate of the global memory	900 MHz
C	The number of stream processors	384
W	The number of threads per warp	32

TABLE 3
Percentage of Relative Difference between Estimated Value and Tested Value

Sparse	COO(%)		CSR(%)		ELL(%)	
matrix	Single	Double	Single	Double	Single	Double
bbmat	-6.93	-12.40	24.59	18.71	-3.95	-11.34
Cheby_ shev4	-8.32	-12.50	18.00	-2.80	-15.78	-6.02
twotone	-25.35	-35.83	-21.62	-10.38	-10.04	-11.52
scircuit	-38.22	-21.06	-39.05	-27.88	-37.28	-18.82
PR02R	-9.75	-12.56	11.29	14.85	-7.25	-16.62
pwtk	-2.93	-5.50	-2.74	9.78	-11.22	-17.73
raefsky3	-9.40	-14.07	-5.92	-2.53	-14.49	-14.10
raefsky5	28.46	15.88	-3.40	34.29	-5.22	-8.40
rma10	-6.52	-10.10	13.78	14.33	-11.99	-10.46
TSOPF_ RS_ b300_c3	-10.69	-13.81	-11.44	-11.34	NA	NA

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. A HYB function for SpMV has a parameter, which has tree values: *AUTO*, *USER*, *MAX*. A function automatically selects a threshold segmentation 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*, the threshold will be $\max\{i \,|\, P(X = i) > 0\}$ and HYB will become ELL. Due to the official and high-performance features of the CUSPARSE, the library is widely used in linear system solving. We test SpMV based on COO, CSR, ELL, and HYB by CUSPARSE functions.

6.2 Performance Estimation and Test

We adopt the following steps in our experiments. (1) Build a probability model for each sparse matrix. (2) Estimate the performance of SpMV using COO, CSR, ELL, and HYB formats for each sparse matrix. (3) Test the actual performance of SpMV using COO, CSR, ELL, and HYB functions of CUSPARSE for each sparse matrix. (4) Assess the consistency of estimates and actual tested values.

The sparse matrix computation time varies greatly because of scale disparities. The computation time is proportional to the scale of the computation. We define *speedup* as the ratio of computing scale to computing time. The scale of the computation for SpMV is *NNZ*. We adopt *speedup* to describe the performance, because the computing time of SpMV based on various sparse matrices has a relatively wide range and is not suitable for comparison in the charts. The *speedup* is calculated by $NNZ/T \times 10^{-6}$. The estimates of *speedup* using the COO, CSR, and ELL formats can be calculated according to the parameters in Table 2. The results are shown in Fig. 3 for single-precision floating point numbers and Fig. 5 for double-precision. Fig. 4 gives the results of tested data for single-precision floating point numbers, and Fig. 6 gives the results for double-precision.

The percentage of relative difference between an estimated value and its tested value is calculated by $(EV - TV)/TV \times 100$, which EV is the estimated value and TV is the tested value. A positive percentage of relative difference indicates that the estimated value is greater than the tested value. On the contrary, it means that the estimated value is less than the tested value.

We have the following important observations from our experimental data.

(1) It is observed from Table 3 that the estimated values and tested values have good consistency, with the absolute value of the percentage of relative difference between an estimated value and its tested value to be less than 20% for 47/58=81% cases. The order of magnitude of estimated speedup and that of tested speedup for each sparse matrix based on the three formats (COO, CSR, and ELL) are the same according to Figs. 3, 4, 5, and 6.

(2) As can be seen from Table 3, in most (48 out of 58) cases, the tested speedup is higher than the estimation, because the data access for SpMV on GPU can be optimized by the shared memory and the texture memory. However, in some (10 out of 58) cases, the estimated speedup is higher than the tested speedup due to the uneven distribution of non-zero elements, in particular for the CSR format.

(3) The performance of parallel computing on GPU is closely related to the utilization of the threads. The load balance between threads can improve the utilization of GPU. The performance of SpMV using the ELL format is usually (for 7 out of 10 matrices) better than that of SpMV using COO and CSR, because the size of the data set assigned to each thread in the ELL format is the same. However, the estimation for the sparse matrix TSOPF/TSOPF_RS_b300_c3 is far worse than that of COO and CSR, because $\max\{i \,|\, P(X = i) > 0\}$ of TSOPF/TSOPF_RS_b300_c3 is far greater than $E(X)$. So, the dense matrix of ELL must be filled with a large num-

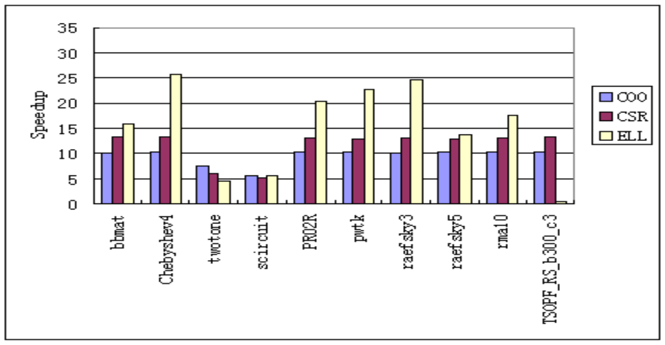

Fig. 3. Estimated speedup (single-precision).

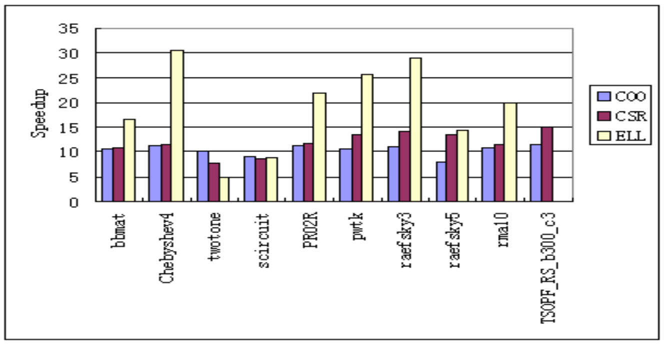

Fig. 4. Tested speedup (single-precision).

ber of zeros, leading to the computed data size increasing sharply. The SpMV for TSOPF/TSOPF_RS_b300_c3 using the ELL format cannot be computed on GPU in the actual test, illustrating that it is inappropriate to adopt the ELL format.

(4) The performance of SpMV using the CSR format is usually better than that of SpMV using COO (except ATandT/twotone and Hamm/scircuit) for single-precision data, as shown in Figs. 3–4. The performance bottleneck of SpMV is data access for single-precision data, because the computing speed of single-precision data on GPU is very fast. Each thread of SpMV using the COO format reads respectively an element from three arrays (**row**, **column**, **value**), whose length is *NNZ*. The SpMV using the COO format must read more data from memory than that using the CSR format, and the SpMV using the CSR format can read continuous multiple data by bus bandwidth, because a row of CSR format is read sequentially in one thread. However, for ATandT/twotone and Hamm/scircuit, the skewness of probability distribution is too large relative to the mean, leading to non-zeros to be extremely uneven between rows. Such extremely imbalanced loads between threads make the performance of SpMV using the CSR format degrade.

(5) The performance of SpMV using the CSR format is usually worse than that of SpMV using COO (except TSOPF/TSOPF_RS_b300_c3) for double-precision data, as shown in Figs. 5–6. The performance bottleneck of SpMV is computing for double-precision data, because the computing speed of double-precision data on GPU is slow. The imbalance of computing between threads has great influence on performance of SpMV using the CSR format. However, for TSOPF/TSOPF_RS_b300_c3, the performance of the accumulative reduction algorithm for SpMV using the COO format will be affected, because the skewness of probability distribution is large.

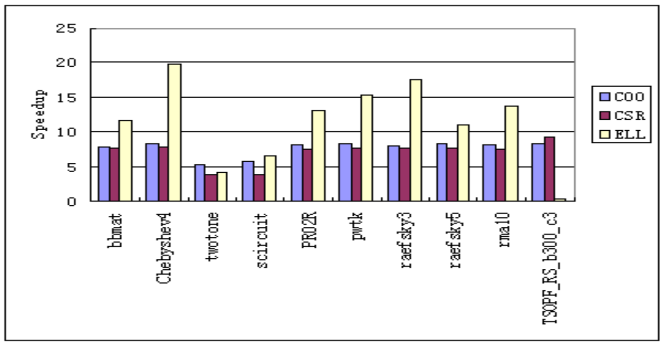

Fig. 5. Estimated speedup (double-precision).

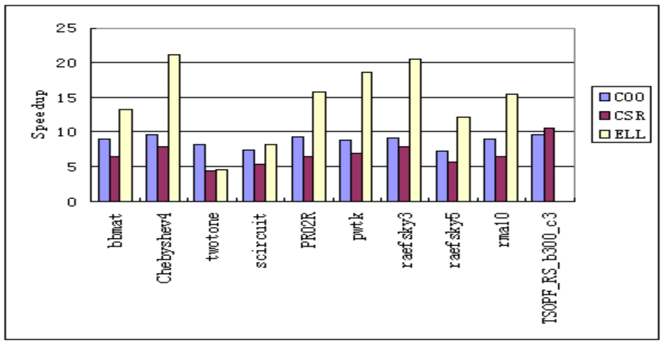

Fig. 6. Tested speedup (double-precision).

Additional performance data are demonstrated and analyzed in 5.2 of the supplementary material.

7 Conclusions

In this paper, we use a probabilistic model to analyze and optimize the performance of SpMV. 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 method does not need additional benchmarks to get optimized parameters. Our performance modeling is based on the probability mass function, which fully reflects the distribution characteristics of non-zeros in a sparse matrix, and does not need benchmark matrices to get the properties and parameters for SpMV. Our performance modeling consists of three steps, i.e., probability analysis, performance estimation, and performance optimization. The proposed approach in this paper is general, and is neither limited by any GPU programming language nor restricted to any specific GPU architecture, because it is based on a mathematical and analytical model. In future work, we will extend the current SpMV performance modeling to handle SpMV kernels on multi-GPUs and CPU-GPU clusters.

ACKNOWLEDGMENTS

The authors would like to express their sincere gratitude to all the reviewers for their valuable and constructive comments. The research was partially funded by the Key Program of National Natural Science Foundation of China (Grant No. 61133005), and the National Natural Science Foundation of China (Grant Nos. 90715029, 61070057, 60603053, 61370095).

REFERENCES

[1] N. Bell and M. Garland, Implementing sparse matrix-vector multiplication on throughput-oriented processors. Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. ACM, 2009, pp. 1-11.

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

[3] W. Yang, K. Li, Y. Liu, L. Shi, and C. Wang, Optimization of Quasi Diagonal Matrix-Vector Multiplication on GPU, International Journal of High Performance Computing Applications. 2013(In press, doi:10.1177/1094342013501126).

[4] 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, 2003, pp. 917-924.

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

[6] F. Vazquez, G. Ortega, J. J. Fernandez, and E. M. Garzon, Improving the performance of the sparse matrix vector product with GPUs, in Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology, ser. CIT 10. IEEE Computer Society, 2010, pp. 1146-1151.

[7] D. Grewe and A. Lokhmotov, Automatically generating and tuning gpu code for sparse matrix-vector multiplication from a high-level representation, in Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units, ser. GPGPU-4. ACM, 2011, pp. 12:1-12:8.

[8] J. C. Pichel, F. F. Rivera, M. Fernandez, and A. Rodriguez, Optimization of sparse matrix-vector multiplication using reordering techniques on GPUs, Microprocessors and Microsystems, vol. 36, no. 2, 2012, pp. 65-77.

[9] 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).

[10] A. N. Yzelman and D. Roose, High-Level Strategies for Parallel Shared-Memory Sparse Matrix-Vector Multiplication. IEEE Transactions on Parallel and Distributed Systems (2013),(In press, doi: http://doi.ieeecomputersociety.org/10.1109/TPDS.2013.31).

[11] J. W. Choi, A. Singh, and R. W. Vuduc, Model-driven autotuning of sparse matrix-vector multiply on GPUs, in PPoPP 10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming, 2010, pp. 115-126.

[12] V. Karakasis, T. Gkountouvas, K. Kourtis, G. Goumas, and N. Koziris. An Extended Compression Format for the Optimization of Sparse Matrix-Vector Multiplication. IEEE Transactions on Parallel and Distributed Systems Sept. vol. 24, no. 10, 2013, pp. 1930-1940.

[13] J. Kurzak, W. Alvaro, and J. Dongarra, Optimizing matrix multiplication for a short-vector simd architecture-cell processor, Parallel Comput., vol. 35, no. 3, 2009, pp. 138-150.

[14] E.-J. Im, K. Yelick, and R. Vuduc, Sparsity: Optimization framework for sparse matrix kernels, Int. J. High Perform. Comput. Appl., vol. 18, no. 1, 2004, pp. 135-158.

[15] M. M. Baskaran and R. Bordawekar, Optimizing sparse matrixvector multiplication on GPUs, Research Report RC24704, IBM TJ Watson Research Center, Tech. Rep., Dec. 2008.

[16] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. C.W. R. Vuduc, and K. Yelick, Self-adapting linear algebra algorithms and software, Proceedings of the IEEE, vol. 93, no. 2, 2005, pp. 293-312.

[17] A. Monakov, A. Lokhmotov, and A. Avetisyan, Automatically tuning sparse matrix-vector multiplication for GPU architectures. High Performance Embedded Architectures and Compilers. Springer Berlin Heidelberg, 2010, p.111-125.

[18] Z. Wang, X. Xu, W. Zhao, Y. Zhang, and S. He, Optimizing sparse matrix-vector multiplication on CUDA, in Education Technology and Computer (ICETC), 2010 2nd International Conference on, vol. 4, June 2010, pp. V4-109 -V4-113.

[19] X. Yang, S. Parthasarathy, and P. Sadayappan, Fast sparse matrixvector multiplication on GPUs: implications for graph mining, Proc. VLDB Endow., vol. 4, no. 4, Jan. 2011, pp. 231-242.

[20] A. H. El Zein and A. P. Rendell, Generating optimal CUDA sparse matrix-vector product implementations for evolving GPU hardware. Concurrency and Computation: Practice and Experience 24.1 (2012): 3-13.

[21] Y. Kubota and D. Takahashi, Optimization of sparse matrix-vector multiplication by auto selecting storage schemes on GPU. Computational Science and Its Applications-ICCSA 2011. Springer Berlin Heidelberg, 2011. 547-561.

[22] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu, Program optimization space pruning for a multithreaded GPU, in Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization, ser. CGO 08. ACM, 2008, pp. 195-204.

[23] Y. Zhang and J. Owens, A quantitative performance analysis model for GPU architectures, in Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture, ser. HPCA 11, Feb. 2011, pp. 382-393.

[24] H. Pabst, B. Bachmayer, and M. Klemm, Performance of a Structure-detecting SpMV using the CSR Matrix Representation. Parallel and Distributed Computing (ISPDC), 2012 11th International Symposium on. IEEE, 2012, pp. 3-10.

[25] M. M. Dehnavi, D. Fernandez, and J. L. Gaudiot, Parallel Sparse Approximate Inverse Preconditioning on Graphic Processing Units. IEEE Transactions on Parallel and Distributed Systems Sept. vol. 24, no. 9, 2013, pp. 1852-1862.

[26] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, An adaptive performance modeling tool for GPU architectures, in Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ser. PPoPP 10. ACM, 2010, pp. 105-114.

[27] S. Hong and H. Kim, An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness, in Proceedings of the 36th annual international symposium on Computer architecture, ser. ISCA 09. ACM, 2009, pp. 152-163.

[28] K. Kothapalli, R. Mukherjee, M. Rehman, S. Patidar, P. Narayanan, and K. Srinathan, A performance prediction model for the CUDA GPGPU platform, in 2009 International Conference on High Performance Computing (HiPC), Dec. 2009, pp. 463-472.

[29] J. Li, G. Tan, M. Chen, and N. Sun, SMAT: an input adaptive auto-tuner for sparse matrix-vector multiplication. Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation. ACM, 2013, pp. 117-126.

[30] D. Schaa and D. Kaeli, Exploring the multiple-GPU design space, in Proceedings of the 2009 IEEE International Parallel & Distributed Processing Symposium, ser. IPDPS 09, May 2009, pp. 1-12.

[31] P. Guo, L. Wang, and P. Chen. A Performance Modeling and Optimization Analysis Tool for Sparse Matrix-Vector Multiplication on GPUs. IEEE Transactions on Parallel and Distributed Systems 99.1 (2013)(In press, doi: http://doi.ieeecomputersociety.org/10.1109/TPDS.2013.123).

[32] J. C. Pichel and F. F. Rivera. Sparse matrix vector multiplication on the Single-Chip Cloud Computer many-core processor. Journal of Parallel and Distributed Computing, 2013, 73(12): 1539-1550.

[33] S. B. Indarapu, M. Maramreddy, and K. Kothapalli. Architecture- and Workload-Aware Heterogeneous Algorithms for Sparse Matrix Vector Multiplication. in Proc. ICPADS 13, Dec. 2013 (In press).

[34] N.L. Johnson, S. Kotz, A. Kemp, (1993) Univariate Discrete Distributions (2nd Edition). Wiley. ISBN 0-471-54897-9 (p. 36)

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

Kenli Li received the Ph.D. degree in computer science from Huazhong University of Science and Technology, China, in 2003. He was a visiting scholar at University of Illinois at Urbana-Champaign from 2004 to 2005. He is currently a full professor of computer science and technology at Hunan University and associate director of National Supercomputing Center in Changsha. His major research includes parallel computing, grid and cloud computing, and DNA computing. He has published more than 90 papers in international conferences and journals, such as IEEE-TC, IEEE-TPDS, JPDC, ICPP, CCGrid. He is an outstanding member of CCF.

Wangdong Yang received the M.S. degree from Central South University, China, in 2006. He is currently working toward the Ph.D. degree at Hunan University, China. He is a professor of computer science and technology at Hunan City University, China. His research interests include modeling and programming for distributed computing systems, parallel algorithms, grid and cloud computing.

Keqin Li is a SUNY Distinguished Professor of computer science. He was an Intellectual Ventures endowed visiting chair professor (2011–2013) at Tsinghua University, China. His research interests are mainly in design and analysis of algorithms, parallel and distributed computing, and computer networking. He has over 280 refereed research publications. He is currently or has served on the editorial board of *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, *Journal of Parallel and Distributed Computing*, *International Journal of Parallel, Emergent and Distributed Systems*, *International Journal of High Performance Computing and Networking*, *International Journal of Big Data Intelligence*, and *Optimization Letters*.