J. Parallel Distrib. Comput. 104 (2017) 49–60

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

A hybrid computing method of SpMV on CPU–GPU heterogeneous computing systems

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

[a] College of Computer Science and Electronic Engineering, Hunan University, Changsha, Hunan 410008, China
[b] College of Information Science and Engineering, Hunan City University, Yiyang, Hunan 413000, China
[c] The National Supercomputing Center in Changsha, Hunan University, Hunan 410008, China
[d] Department of Computer Science, State University of New York, New Paltz, NY 12561, USA

HIGHLIGHTS

- Adopt a CPU–GPU hybrid parallel programming model for SpMV performance enhancement.
- We develop a sparse matrix partitioning algorithm based on a distribution function, so that SpMV can be computed by both CPU and GPU.
- We find a strategy which optimizes the overall parallel computing time of SpMV on a CPU–GPU heterogeneous computing system.

ARTICLE INFO

Article history:
Received 3 November 2014
Received in revised form
19 December 2016
Accepted 24 December 2016
Available online 6 January 2017

Keywords:
Heterogeneous computing
Hybrid storage format
Partition
Sparse matrix–vector multiplication

ABSTRACT

Sparse matrix–vector multiplication (SpMV) is an important issue in scientific computing and engineering applications. The performance of SpMV can be improved using parallel computing. The implementation and optimization of SpMV on GPU are research hotspots. Due to some irregularities of sparse matrices, the use of a single compression format is not satisfactory. The hybrid storage format can expand the range of adaptation of the compression algorithms. However, because of the imbalance of non-zero elements, the parallel computing capability of a GPU cannot be fully utilized. The parallel computing capability of a CPU is also rising due to increased number of cores in CPU. However, when a GPU is computing, the CPU controls the process instead of contributing to the computational work. It leads to under-utilization of the computing power of CPU. Due to the characteristics of the sparse matrices, the data can be split into two parts using the hybrid storage format to be allocated to CPU and GPU for simultaneous computing. In order to take full advantage of computing resources of CPU and GPU, the CPU–GPU heterogeneous computing model is adopted in this paper to improve the performance of SpMV. With analysis of the characteristics of CPU and GPU, an optimization strategy of sparse matrix partitioning using a distribution function is proposed to improve the computing performance of SpMV on the heterogeneous computing platform. The experimental results on two test machines demonstrate noticeable performance improvement.

1. Motivation

Sparse matrix–vector multiplication (SpMV) is an essential operation in solving linear systems. For many scientific and engineering applications, the matrices can be very large and sparse. Furthermore, these sparse matrices may have various sparse characteristics. It is a challenging issue to adopt an appropriate algorithm to implement and optimize SpMV for a given parallel computing environment. This paper addresses this challenge by presenting a hybrid parallel programming model, a novel sparse matrix partitioning algorithm, and performance analysis and optimization of SpMV on a CPU–GPU heterogeneous computing platform.

1.1. Suitable compressed format of sparse matrix

Design of a suitable storage format for the sparse matrix can improve the computing performance of SpMV. There already exist a lot of storage formats to match the various features of sparse matrices. DIA (diagonal format) stores elements in every

* Corresponding author at: College of Computer Science and Electronic Engineering, Hunan University, Changsha, Hunan 410008, China.
E-mail addresses: yangwangdong@163.com (W. Yang), lkl@hnu.edu.cn (K. Li), lik@newpaltz.edu (K. Li).

http://dx.doi.org/10.1016/j.jpdc.2016.12.023
0743-7315/

diagonal, which is suitable for the storage of a diagonal matrix [1]. ELL (ELLPACK format) uses an $n \times k$ matrix to store the data, where k denotes the maximum number of non-zero elements in a single row, and it is appropriate for a matrix with non-zero elements evenly distributed between rows [1]. If the number of nonzero elements varies widely among rows, the computing scale of data will be increased as a large amount of data was filled. COO (coordinate format) uses a list of (row, column, value) tuples to store a matrix. CSR (compressed sparse row format) is a widely used format exploiting the row compression technology, which can achieve parallel computing through data partitioning by rows [1]. However, there is load imbalance for parallel computing because of length difference between rows. PKT (packet format) is a new sparse matrix representation to exploit the locality of non-zero elements. The packet format decomposes a matrix into a given number of fixed-size partitions which are stored in a specialized packet data structure. If the data distribution is not concentrated, the number of block packet will increase, resulting in a large amount of data filled in to increase the computing scale of data. Li et al. [17] provided a strategy to choose the appropriate storage format according to the distribution features of the non-zero elements in a sparse matrix. However, due to irregular sparse characteristic of a sparse matrix, it is difficult to achieve better compression effect using single compressed storage format. Thus, some hybrid compressed storage formats are used, such as HYB (hybrid format), where some discrete non-zero elements are stored in the COO storage in hybrid compressed format, and more non-zero elements are stored by ELL. At present, HYB has been implemented on GPU, such as the HYB function developed by NVIDIA [24] for SpMV, but it is not implemented on any CPU–GPU heterogeneous computing system.

1.2. New computing platform for SpMV

Modern GPU (graphics processing units) programming has been extensively used in the last several years for resolving a broad range of computationally demanding and complex problems. Brodtkorb et al. [4] provided an overview of hardware and traditional optimization techniques for the GPU and gave a step-by-step guide to profile-driven development. The introduction of some vendor specific technologies, such as NVIDIA's Compute Unified Device Architecture (CUDA), further accelerates the adoption of high-performance parallel computing to commodity computers. Using CUDA, not only SpMV performs well, but also the programming is relatively easy. However, for the imbalances of non-zero elements of rows, the parallel computing power of GPU cannot be fully utilized. The parallel computing power of CPU is also rising due to increase in the number of cores in CPU. However, when GPU is computing, CPU controls the process instead of contributing to the computational work, and leads to under-utilization of CPU. The CPU–GPU heterogeneous computing model can take full advantage of the CPU and GPU resources to improve computing performance. Furthermore, due to different computing models between CPU and GPU, computational efficiency cannot be fully aroused using the same storage format for CPU and GPU, and a hybrid format can solve this problem.

1.3. Our contribution

The contribution and content of the paper are summarized as follows. We adopt a CPU–GPU hybrid parallel programming model (Section 3.2) for SpMV performance enhancement. Based on a distribution function of sparse matrices (Section 4.1) and by using a hybrid storage format of COO and ELL (or DIA) (Section 4.2), we develop a sparse matrix partitioning algorithm (Section 4.3), so that SpMV can be computed by both CPU and GPU (Section 4.4). Furthermore, using analytical results on the CPU and GPU computing times (Section 5.1), we are able to find a strategy which optimizes the overall parallel computing time of SpMV on a CPU–GPU heterogeneous computing system (Section 5.2).

In this paper, we use SpMV CUDA kernels developed by NVIDIA [24] on NVIDIA GPU and MKL BLAS functions developed by Intel [13] on Intel CPU for our experiments. Our experimental results on two test machines demonstrate noticeable performance improvement, with *flop-rate* improvement 11.07% compared to computing with GPU-only, and *speedup* 11.89 compared to computing with CPU-only, for the 40 tested cases of 10 sparse matrices, 3 test machines, and 2 precision levels (Section 6).

2. Related research

In order to improve the performance of SpMV, some new storage models using the blocked strategy are provided, such as blocked compressed sparse row (BCSR) format [5], row-grouped CSR (RGCSR) format [25], blocked ELLPACK (BELLPACK) format [2], sliced coordinate (SCOO) format [28], sliced ELLPACK (SELLPACK) format [20], fixed scale blocked format [9], doubly separated block diagonal (DSBD) format [31]. However the effect of these storage models using the blocked strategy is not good for different sparse matrices with various sparsity features. Some storage models use a reordering technique to expand the scope of a sparse matrix [20,26]. But the process of reordering is very costly, leading to certain impact on computing performance of SpMV. In addition, some hybrid storage models are provided to improve the efficiency of compression of a sparse matrix, such as hybrid ELL and COO formats (HYB), hybrid DIA and CSR formats [30], hybrid JDS and CSR formats [7], and hybrid COO and CSR [3]. But these hybrid formats are suitable for specific types of sparse matrices and implemented on a single processor, and collaborative computing on CPU–GPU heterogeneous processors is relatively rare. Furthermore, how to determine the proportion of the division of two formats is a challenge. Some strategies partition sparse matrices according to the configuration of a computing environment, such as the cache scale of processors [11,28], the computing power of processors [3,12], the data transmission bandwidth [6,15,27], the computational methodology of CPU and GPU [14] and the sparse characteristics of the sparse matrix [12,15,16]. But due to the complexity of a heterogeneous system and the diversity of sparse matrices, sometimes the models do not work effectively.

A heterogeneous computing platform is now a universal concern by everyone. How to make full use of the computing power of heterogeneous computing platforms is a great challenge. Researchers have conducted extensive research on the issue. Our approach differs from the others, which focuses on the optimization of architecture properties and data sharing. We mainly analyze the sparsity feature by a distribution function to optimize the division of tasks in order to balance the workload between CPU and GPU.

3. CPU–GPU heterogeneous computing

3.1. GPU computing architecture

The modern 3D graphics processing unit (GPU) has evolved from a fixed-function graphics pipeline to a programmable parallel processor with computing power exceeding that of multicore CPUs. In November 2006, NVIDIA corporation introduced Tesla architecture which unifies the vertex and pixel processors and extends them to enable high-performance parallel computing applications. The basic computing unit of GPGPU is called streaming multiprocessors (SM). As a GPU at the bottom of the independent hardware structure, SM can be seen as an SIMT

processing unit. In March 2010, NVIDIA corporation introduced Fermi architecture, and GF100 with Fermi architecture has 4 graphics processing clusters (GPC), 16 SMs and 512 cores. For Fermi architecture, each SM has 32 cores, 12KB L1 cache and 2-warps scheduling. In May 2012, NVIDIA corporation introduced Kepler architecture. For Kepler architecture, each SM contains 192 scalar processors (SP) and 32 special function units (SFU). In addition, each SM contains 64K shared memory for threads to share data or communicate in the block. Using the model explicitly to access memory, the access speed of the shared memory is close to that of register without bank conflict. Each SM contains some registers, which are allocated by each thread in the execution. A graphics processing cluster (GPC) is composed of 2 SMs. Two SMs share one GPC and L1 and texture cache. Only four GPCs share the L2 cache. All SMs share the global memory [21]. NVIDIA launched Maxwell architecture in 2014. This architecture provides substantial application performance improvements over prior architectures by featuring large dedicated shared memory, shared memory atomics, and more active thread blocks per SM. NVIDIA launched Pascal architecture in 2016. NVIDIA's new NVIDIA Tesla P100 accelerator using the groundbreaking new NVIDIA Pascal GP100 GPU takes GPU computing to the next level. GP100 is composed of an array of Graphics Processing Clusters (GPCs). Each GPC inside GP100 has ten SMs. Each SM has 64 CUDA Cores and four texture units. With 60 SMs, GP100 has a total of 3840 single precision CUDA Cores and 240 texture units. Tesla P100 features NVIDIA's new high-speed interface, NVLink, that provides GPU-to-GPU data transfers at up to 160 Gigabytes/second of bidirectional bandwidth which is 5 times the bandwidth of PCIe Gen 3 x16.

3.2. CPU–GPU hybrid parallel programming

With the rapid development of multicore technology, the number of cores in CPU has been increasing. The CPUs with 4-cores, 6-cores, 8-cores, and more cores enter the general computing environment to improve rapidly the parallel processing power. A heterogeneous computing environment can be built up with GPU and multicore CPU.

The GPU does not have a process control capability as a device in CUDA, which is controlled by CPU. The data are transported from host memory to the global memory of GPU. Then CPU invokes the calculation process of GPU by calling the kernel function [23].

OpenMP provides a simple and easy-to-use parallel computing capability of multi-threading on multicore CPUs [8]. A heterogeneous programming model can be established by combining OpenMP and CUDA for a CPU–GPU heterogeneous computing environment. OpenMP dedicates one thread for controlling the GPU, while the other threads are used to share the workload among the remaining CPU cores. Fig. 1 shows the CPU–GPU heterogeneous parallel computing model.

Initially, the data must be divided into two sets which are assigned to CPU and GPU respectively. Then, two groups of threads are created in the parallel section of OpenMP, where a single thread is dedicated to controlling the GPU while other threads undertake the CPU workload by utilizing the remaining CPU cores [29].

4. Sparse matrix partitioning for CPU–GPU parallel computing

4.1. The distribution function of sparse matrices

A is a sparse matrix. N is the number of rows in A and M is the number of columns in A. Define a distribution function (DF) $f : \Omega_A \to B$, where $\Omega_A = \{R_1, R_2, \ldots, R_M\}$ is domain and R_i represents row vector set (RVS) in which each row has i non-zero elements. $B = \{b_1, b_2, \ldots, b_M\}$ is range, where b_i represents the number of rows with i non-zero elements in A. So Ω_A and B meet the following properties.

$$\begin{aligned} &f(R_i) = b_i, \quad R_i \in \Omega_A, b_i \in B, \\ &A = \bigcup_{i=1}^{M} R_i, \quad R_i \cap R_j = \phi,\ i \neq j, \\ &\sum_{i=1}^{M} b_i = N. \end{aligned} \tag{1}$$

4.2. The hybrid format for sparse matrices

HYB has better performance when a matrix has a small number of non-zero elements per row, and most rows have nearly the same number of non-zero elements but there may be a few irregular rows with much more non-zero elements. The matrix is split into two parts, i.e., ELL (or DIA) and COO, such that the most rows which are nearly equal are stored by ELL (stored by DIA for the quasi diagonal matrix) and the other few irregular rows with much more non-zero elements are stored by COO. The coordinate (COO) format is a particularly simple storage scheme with tuples of (*row*, *column*, *value*). The arrays *row*, *column*, and *value* store the row indices, column indices, and values of the non-zero elements in a matrix respectively. 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 *data* array, where rows with less than K non-zeros are zero-padded. Similarly, the corresponding column indices are stored in a dense N-by-K *index* array, again with a sentinel value used for padding. The DIA format is formed by two arrays, i.e., *data* stores the non-zero values with N-by-K matrix and *offset* array stores the offset of each diagonal with respect to the main diagonal.

HYB is a hybrid format of COO and ELL (or DIA). Given a threshold K, the part exceeding K non-zeros in a row is extracted to be stored by COO and the other part is stored by ELL (or DIA) in order to minimize zero-padding. A sparse matrix can be divided into two parts, i.e., COO and ELL (or DIA), by threshold K. Let us consider the following example:

$$A = \begin{pmatrix} 3 & 0 & 0 & 0 \\ 0 & 1 & 4 & 0 \\ 6 & 0 & 2 & 8 \\ 0 & 5 & 0 & 7 \end{pmatrix}.$$

Assume that $K = 2$. Then, we have

$$\text{COO}: \begin{cases} row = \begin{pmatrix} 3 \end{pmatrix}, \\ column = \begin{pmatrix} 4 \end{pmatrix}, \\ value = \begin{pmatrix} 8 \end{pmatrix}. \end{cases} \qquad \text{ELL}: \begin{cases} data = \begin{pmatrix} 3 & 0 \\ 1 & 4 \\ 6 & 2 \\ 5 & 7 \end{pmatrix}, \\ index = \begin{pmatrix} 1 & * \\ 2 & 3 \\ 1 & 3 \\ 2 & 4 \end{pmatrix}. \end{cases}$$

or

$$\text{COO}: \begin{cases} row = \begin{pmatrix} 3 & 4 \end{pmatrix}, \\ column = \begin{pmatrix} 1 & 2 \end{pmatrix}, \\ value = \begin{pmatrix} 6 & 5 \end{pmatrix}. \end{cases} \qquad \text{DIA}: \begin{cases} data = \begin{pmatrix} 3 & 0 \\ 1 & 4 \\ 2 & 8 \\ 7 & 0 \end{pmatrix}, \\ offset = \begin{pmatrix} 0 & 1 \end{pmatrix}. \end{cases}$$

While the ELL format is well-suited for vector architectures, its efficiency rapidly degrades when the number of non-zeros per row varies. DIA is suitable for compression and storage of a diagonal matrix. If the data of a sparse matrix do not concentrate on the diagonal and have more dispersed distribution area, the more

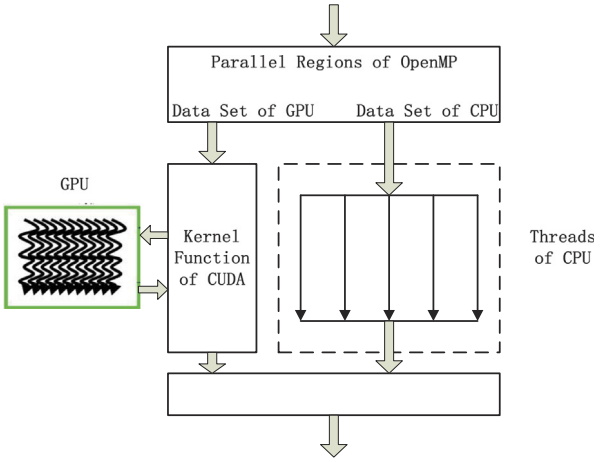


Fig. 1. The CPU–GPU heterogeneous parallel computing model.

diagonals should be converted into more columns in the data matrix of DIA, leading performance deterioration. 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 (or DIA) and the remaining entries in COO.

Define the number of non-zero elements in the RVS R_i in Ω_A to be $NNZ(R_i) = f(R_i) \times i = b_i \times i$. For a subset Ω' of Ω_A, $NNZ(\Omega')$ is the number of non-zero elements in Ω' and $NNZ(\Omega') = \sum_{R\in\Omega'_A} NNZ(R)$, where R is the RVS.

Define $\Omega'_{i\le K}$ to be a subset of Ω_A, in which the NNZ of the RVS's are no more than K. $NNZ(\Omega'_{i\le K})$ is calculated by Eq. (2):

$$NNZ(\Omega'_{i\le K}) = \sum_{i=1}^{K}(NNZ(R_i)). \tag{2}$$

Define $\Omega'_{i>K}$ to be a subset of Ω_A, in which the NNZ of the RVS's are more than K. $NNZ(\Omega'_{i>K})$ is calculated by Eq. (2):

$$NNZ(\Omega'_{i>K}) = \sum_{i=K+1}^{M}(NNZ(R_i)). \tag{3}$$

The number of non-zero entries in ELL (or DIA) of hybrid format is calculated by Eq. (4):

$$NNZ = NNZ(\Omega'_{i\le K}) + K \times \sum_{i=K}^{M} b_i. \tag{4}$$

The number of non-zeros of the remaining entries in COO of hybrid format is calculated by Eq. (5):

$$NNZ = NNZ(\Omega'_{i>K}) - K \times \sum_{i=K}^{M} b_i. \tag{5}$$

4.3. The partitioning algorithm

Given a threshold K, the part exceeding K non-zeros in a row is extracted to be stored by COO, and the other part is stored by ELL or DIA. The sparse matrix includes two parts. One contains the rows with more than K non-zeros and the other contains the rows with less than K non-zeros. The first part is classified into ELL part. For the second part, only K non-zeros in the front of the rows with more than K non-zeros are classified into ELL part, and the rest of the second part is classified into COO part. The partitioning algorithm for an N-by-M sparse matrix A into two parts is shown in Algorithm 1. The number of loops using Algorithm 1 is less than the number of rows in the sparse matrix and there are little additions and multiplications in Algorithm 1, so the execution time using Algorithm 1 has little impact on the computing of SpMV. If the sparse matrix is a quasi diagonal matrix, the non-zero elements on the diagonals should be split to form the block, which is computed using DIA format on GPU.

Algorithm 1 Partitioning algorithm for HYB.

Require: The domain Ω_A of DF, i.e., $R_1, R_2, \ldots, R_M$; The range B of DF, i.e., $b_1, b_2, \ldots, b_M$; the threshold K of the number of non-zero elements of one row.

Ensure: A hybrid representation of A with sub-matrices $SubMatrix_{GPU}$ and $SubMatrix_{CPU}$.

```
for j ← 1 to M do
  if j ≤ K then
    SubMatrix_GPU ← SubMatrix_GPU ∪ R_j;
  else
    for i ← 1 to b_j do
      if A is the quasi diagonal matrix then
        V ← the K non-zero elements around the main diagonal from the ith row in R_j;
      else
        V ← the first K non-zero elements from the ith row in R_j;
      end if
      SubMatrix_GPU ← SubMatrix_GPU ∪ V;
      V' ← the remaining non-zero elements of the ith row in R_j;
      SubMatrix_CPU ← SubMatrix_CPU ∪ V';
    end for
  end if
end for
return SubMatrix_GPU and SubMatrix_CPU.
```

4.4. Implementation of SpMV for CPU–GPU heterogeneous computing

The submatrix which is stored by ELL (or DIA) is more suitable for execution on the GPU's grid of threads, which can make full use of the parallelism of GPU. But due to more uneven distribution, the data set of COO is better suited for CPU. The execution process on CPU–GPU for SpMV includes three steps.

(1) The sparse matrix is split into two parts, i.e., $SubMatrix_{GPU}$ and $SubMatrix_{CPU}$, using the threshold K according to Algorithm 1.

(2) The $SubMatrix_{CPU}$ is stored by COO format and is executed on CPU using the SpMV function (mkl_cspblas_coogemv) in MKL blas lib. The $SubMatrix_{GPU}$ is stored by DIA format and is executed on GPU using the SpMV function (spmv_dia)in CUSP lib if the sparse matrix A is the quasi diagonal matrix; otherwise, it is stored by ELL format and is executed on GPU using the SpMV function (cusparsehybmv) in CUSPARSE lib.

(3) The result of SpMV on CPU and that of GPU are added on GPU using the vector addition function (cusparseAxpyi) in CUSPARSE lib.

The implementation for SpMV uses OpenMP to achieve parallelism on a CPU–GPU heterogeneous computing platform, as shown in Algorithm 2.

5. Performance analysis and optimization

Multicore CPU is a computing model of multiple instruction stream multiple data stream (MIMD) and GPU is a computing model of single instruction stream multiple thread stream (SIMT). The computing tasks should be assigned to the CPU and GPU according to the characteristics of each calculation

Algorithm 2 The process of SpMV on CPU–GPU heterogeneous computing system.

Require: A hybrid representation of A with sub-matrices using Algorithm 1: $SubMatrix_{GPU}$ and $SubMatrix_{CPU}$.
Ensure: The result vector: X.

```
#pragma omp parallel//Set parallel code area
{
  #pragma omp sections nowait
  {
    #pragma omp section //Build the parallel section for GPU.
    {
      x1 ← Call the DIA kernel function spmv_dia (SubMatrix_GPU) of CUSP lib;// A is the quasi diagonal matrix.
      or
      x1 ← Call the ELL kernel function cusparsehybmv (SubMatrix_GPU) of CUSPARSE lib;// A is not the quasi diagonal matrix.
    }
    #pragma omp section ////Build the parallel section for CPU.
    {
      x2 ← Call the COO function mkl_cspblas_coogemv (SubMatrix_CPU) of MKL blas lib.
    }
  }
}
x ← Call the vector addition function cusparseAxpyi(x1,x2) in CUSPARSE lib.
return x.
```

in the heterogeneous computing environment to improve the performance of SpMV. For hybrid format, the ratio of zero padding can be controlled by adjusting the threshold of partition in Algorithm 1. So the hybrid format can adapt to more types of sparse matrices. Moreover, data transmission is also reduced between CPU and GPU, because ELL or DIA part is put into GPU to be computed only. The sparse matrix stored by hybrid format is divided into two parts with only one needing to be transferred into GPU, such as ELL part of HYB format and DIA part of HDC format [30]. So data transmission is also reduced between CPU and GPU because another part does not need to be transferred into GPU.

5.1. Performance estimate of SpMV on CPU–GPU

The data assigned to GPU must be transported from host memory to global memory of GPU by PCIe. Data transmission between CPU and GPU does impact on performance of SpMV, but SpMV will be performed many times after the sparse matrix was transported into the GPU for a solving process using iterative solver. So computing performance improvement using GPU will offset the adverse effects of data transmission [18]. Global memory has greater access latency than shared memory, which is shared by a thread warp. Multicore CPU can reduce the delay of memory access by a multi-level cache mechanism, but the use of the cache is controlled by CPU itself and the program cannot control by coding.

With large data bandwidth and large thread bundle, GPU is suitable for the calculation of a data set with a large volume of regular data. But the frequency of GPU is lower than that of CPU. The differences in data access and computing power of both CPU and GPU should be considered when a sparse matrix is partitioned.

It is very difficult to accurately estimate computing time of SpMV on CPU and GPU, because SpMV is a very irregular for numerical calculation. Some estimation methods for SpMV are provided on GPU and CPU, such as [12,16,20]. But the actual estimation results are not very accurate for different kinds of sparse matrices with various sparsity features. In addition, there are other tasks of the operating system to be executed on CPU, leading to the available computing resources of CPU difficult to determine, so the computing time of SpMV on CPU is very difficult to accurately estimate. So we adopt a kind of relatively simple estimation method to improve the efficiency of estimating. The optimal threshold K may be found by Algorithm 3. We find that these values around the optimal value have similar performance in testing, so the threshold K found by Algorithm 3 is effective for partitioning.

Assume that CPU $= (nc, fc)$, where nc is the number of cores in the CPU and fc is the frequency of the CPU.

There are $NNZ(CPU)$ non-zero elements in the COO format, which is divided from a sparse matrix and is assigned to multicore of CPU to be computed. So the CPU computing time TC of SpMV using COO can be approximately expressed by Eq. (6):

$$TC = \frac{2 \times NNZ(CPU)}{fc} \times \frac{1}{nc - 1}, \tag{6}$$

where we notice that there are $nc - 1$ cores that are assigned tasks, because one core is used to control the GPU.

The GPU computing time TG of SpMV depends on two parts, i.e., computing time TG and transmission time between CPU and GPU. But the sparse matrix is loaded into GPU only once in the whole process of solving a system of linear equations and SpMV is performed many times. The transmission time has little impact on the whole time of solving. So the transmission time is not considered in our model. We define the following variables. N is the number of rows in a sparse matrix and K is the partition threshold of ELL (or DIA) and COO. C is the number of streaming processors. The rate of multiplication and addition on SP can be considered to be the same, because SP can execute a multiplication and an addition operation with the same time. Define F to be the single-precision execution rate on SP. The computing time in a thread is $NNZ(GPU)/(N \times F)$. So the computing time is expressed as

$$TG = \frac{N}{C} \times \frac{NNZ(GPU)}{N \times F} = \frac{NNZ(GPU)}{C \times F}. \tag{7}$$

Finally, the CPU–GPU parallel computing time T of SpMV is expressed as

$$T = \max(TC, TG). \tag{8}$$

5.2. An optimizing strategy for SpMV on CPU–GPU

The performance optimizing workflow for SpMV on CPU–GPU consists of three steps, i.e., establishment of a DF (Section 4.1), split of a sparse matrix into COO and ELL (or DIA) by threshold K (Section 4.3), and estimate of the performance of SpMV under different threshold K (Section 5.1). A sparse matrix A can be split into COO and ELL (or DIA) formats by a threshold K. The choice of parameter K can affect the performance of SpMV on CPU–GPU. The optimal choice of K can minimize the value of Eq. (8). According to Eqs. (6) and (7), TG increases with the increasing of K and TC decreases. Hence, TG and TC have only one intersection point. If $K = x$ and $TG = TC$, then x is the optimal choice of K. If there is no value x such that $TG = TC$, the optimal choice of K can be found out by Algorithm 3, and the two computing tasks partitioned by the threshold K are balanced for the computing powers of CPU and GPU, so parallel computation efficiency can be improved to reduce the computing time of SpMV. If the NNZ of the row with most non-zero elements is M, the threshold K is in $[1 \ldots M]$. The sparse matrix is split into two parts, one is ELL (or DIA) part and another is COO part using each value in $[1 \ldots M]$. The ELL (or DIA)

part will be computed on GPU and the COO part will be computed on CPU. So the computing time on CPU–GPU can be estimated by Eqs. (6) and (7) (Lines 3–6 in Algorithm 3). The loads of CPU and GPU are balanced if the computing time on CPU and GPU is the same or closest. Lines 11–26 in Algorithm 3 seek the threshold K which makes the computing time on CPU and GPU is the same or closest.

Algorithm 3 Seeking the optimal threshold K for SpMV on CPU–GPU.

Require: The domain Ω_A of DF, i.e., $R_1, R_2, \ldots, R_M$; The range B of DF, i.e., $b_1, b_2, \ldots, b_M$;
Ensure: The optimal choice of the threshold K.

```
1:  for j ← 1 to M do
2:    SubMatrix_GPU and SubMatrix_CPU are obtained by Algorithm 1 using threshold j;
3:    NNZ(SubMatrix_GPU) ← Eq. (4);
4:    NNZ(SubMatrix_CPU) ← Eq. (5);
5:    TC ← Eq. (6);
6:    TG ← Eq. (7);
7:    if j = 1 then
8:      TC' ← TC;
9:      TG' ← TG;
10:   end if
11:   if TC = TG then
12:     K ← j;
13:     break;
14:   else
15:     if TG' < TC' and TG > TC then
16:       if TC' < TG then
17:         K ← j − 1;
18:       else
19:         K ← j;
20:       end if
21:       break;
22:     else
23:       TC' ← TC;
24:       TG' ← TG;
25:     end if
26:   end if
27: end for
28: return K.
```

In fact, for $f(R_i) = b_i$, if $b_i = 0$, R_i can be removed from the domain Ω_A for Algorithms 1 and 3. Assume the number of RVS's in $\Omega_A - \{R_i | f(R_i) = 0\}$ is q. Due to A is the sparse matrix, q is much less than the number of columns in A, and the number of loop seeking the optimal threshold are less than q. So the execution time of Algorithm 3 has little impact on the computing of SpMV.

6. Experimental evaluation

All benchmark are tested on three test machines. The first test machine (abbreviated TM1) 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 performance 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 second test machine (abbreviated TM2) is equipped with one Intel Core E5506 running at 2.13 GHz and a NVIDIA Geforce GTX 460 GPU. The CPU has 4 cores. The GPU has 336 CUDA processor cores working on 1.5 GHz clock and 1 GB global memory with 256-bit bus width and 1.9 GHz clock, with CUDA compute capability 2.1. As for software, TM2 ran the 64-bit Windows 7 and NVIDIA CUDA toolkit 5.0. The third test machine (abbreviated TM3) is equipped with an Intel i7-6700 CPU running at 3.40 GHz and a NVIDIA GTX1070 GPU with Pascal architecture. The CPU has 4 cores with hyper-threading technology. The GPU has 1092 CUDA processor cores, working on 1.683 GHz clock and 8 GB global memory with 256 bits bandwidth, with CUDA compute capacity 6.1, but the bus of the tested GPU only supports PCIe 3.0. The test machine runs the 64bit Windows 10 and NVIDIA CUDA toolkit 8.0.

All benchmarks are chosen from the UF Sparse Matrix Collection [10], whose main features are shown in Table 1. Most of these matrices are derived from scientific computing and real engineering applications. E(X) is the average number of non-zeros in rows in Table 1. $E(X) = NNZ/N$. $\max(x_i)$ is the number non-zeros of the row with the maximum number of non-zeros in Table 1.

All the evaluation results are averaged after running 100 times.

6.1. Test functions

NVIDIA corporation provides three libraries (CUBLAS, CUSPARSE and CUSP) to support matrix calculation. These libraries are provided as CUDA development tools and source codes. CUBLAS offers three levels of library functions, where the second level supports the SpMV of sparse matrices [22].

CUSPARSE also provides three levels of function for the sparse matrix, with the first level for ADD operation, the second level for MUL operation of SpMV [24], and the third level for MUL operation of sparse matrix. It uses both the CSR and HYB formats. HYB is a hybrid format of ELL and COO. The performance of HYB function is better than that of CSR function for most cases. HYB function for SpMV has a parameter, which has three values: *AUTO*, *USER*, *MAX*. The function automatically selects a threshold segmentation if the parameter is *AUTO*. The caller must provide a segmentation threshold if parameter is *USER*. If the threshold is 0, HYB will become COO. If the parameter is *MAX*, HYB will become ELL. Due to the official and high-performance feature of the CUSPARSE, the library is widely used for solving linear systems. We test SpMV on GPU-only based on HYB function, where the parameter is set to *AUTO* for TM1 and TM2. But the parameter is set to *USER* for TM3 and the threshold segmentation is obtained by our strategy [17]. NVIDIA provides another library, CUSP, to offer SpMV for the GPU platform. CUSP supports a variety of compression formats such as COO, DLA, CSR, ELL, and HYB. The COO, CSR, and HYB from CUSP show worse performance than CUSPARSE. We chose the DLA function in CUSP to test for DIA format.

Since Simon/bbmat, Muite/Chebyshev4, Boeing/pwtk, and Simon/raefsky3 have the obvious characteristics of quasi diagonal, the submatrices partitioned from the sparse matrices are performed SpMV using DIA format. Although ATandT/twotone and Fluorem/PR02R have some diagonals, they are scattered, resulting in decline in performance using DIA format. Simon/raefsky5 is a spindle and does not belong to the real quasi diagonal matrix. Bova/rma10 is not suitable for using DIA format because there are many non-zero elements missed on the diagonals. So ATandT/twotone, Hamm/scircuit, Fluorem/PR02R, Simon/raefsky5, Bova/rma10, and TSOPF/TSOPF_RS_b300_c3 are performed SpMV using ELL format.

The Intel Math Kernel Library provides developers of scientific and engineering software with a set of linear algebra, fast Fourier transforms and vector math functions optimized for the latest Intel processors. MKL contains LAPACK, the basic linear algebra subprograms (BLAS), and the extended BLAS (sparse) [13], which have high performance compared to the other libraries for most of the processors. We calculate the COO part of our SpMV on the CPU using the MKL.

Table 1
General information of the sparse matrices used in the experiments.

Sparse matrix	Dimension	*NNZ*	$E(X)$	$\max(x_i)$	Characteristic
Simon/bbmat	38744 ∗ 38744	1,771,722	45.729	132	
Muite/Chebyshev4	68121 ∗ 68121	5,377,761	78.944	81	
ATandT/twotone	120750 ∗ 120750	1,224,224	10.139	188	
Hamm/scircuit	170998 ∗ 170998	958,936	5.608	353	
Fluorem/PR02R	161070 ∗ 161070	8,185,136	50.817	88	
Boeing/pwtk	217918 ∗ 217918	5,926,171	27.194	90	
Simon/raefsky3	21200 ∗ 21200	1,488,768	70.225	80	
Simon/raefsky5	6316 ∗ 6316	168,658	26.703	54	
Bova/rma10	46835 ∗ 46835	2,374,001	50.689	145	
TSOPF/TSOPF_RS_b300_c3	42138 ∗ 42138	4,413,449	104.7	20,702	

Table 2
The relative difference of tested values over estimated values using Algorithm 3 for threshold K.

Sparse matrix	TM2			TM1		
	K_e	K_t	*RD* (%)	K_e	K_t	*RD* (%)
bbmat	120	116–122	0.0	122	120–122	0.0
Cheby_ shev4	72	72	0.0	72	64	11.1
scircuit	9	11	22.2	62	73	11.7
twotone	31	33	6.5	107	97	9.3
PR02R	62	58	6.5	56	47	16.1
pwtk	50	52–54	4.0	69	64–65	5.8
raefsky3	72	80	11.1	72	72,80	0.0
raefsky5	50	31–47	6.0	39	35–40	0.0
rma10	87	88–91	1.1	102	99–100	2.0
TSOPF_ RS_ b300_c3	140	140	0.0	212	140,212	0.0
Average relative difference			5.7			5.6

6.2. Comparison of estimated and tested values of the threshold K

Figs. 2 and 3 show the performance of SpMV using Algorithm 3 for various threshold K on CPU–GPU heterogeneous computing systems. Table 2 gives the relative difference (abbreviated *RD*) of tested values over estimated values using Algorithm 3 for threshold K. The relative difference is calculated by $|K_t - K_e|/K_e \times 100\%$, where K_t and K_e are the tested value and the estimated value respectively for threshold K. The average relative difference of the ten test matrices are 5.7% and 5.6% respectively for TM2 and TM1. We can find that the estimated values and tested values have good consistency.

6.3. Performance improvement of SpMV on CPU–GPU

Different sparse matrix computation times vary greatly due to the difference in matrix sizes. The computation time is proportional to the scale of a computation. We define *flop-rate* as the number of operations per second. The scale of the computation for SpMV is *NNZ*. We adopt *flop-rate* to describe the performance, because the computing times of SpMV for various sparse matrices are relatively wide apart, resulting in difficulty of comparison in the chart. Since each non-zero element should perform a multiplication and an addition operations for SpMV, the *flop-rate* is calculated by $((2 \times NNZ)/T) \times 10^{-9}$, where T is the computing time (in seconds) of SpMV.

The *flop-rate* of CPU-only, GPU-only, and CPU–GPU on TM1 for single-precision and double-precision are shown in Fig. 4. The *flop-rate* of CPU-only, GPU-only, and CPU–GPU on TM2 for single-precision and double-precision are shown in Fig. 5. The *flop-rate* of CPU-only, GPU-only, and CPU–GPU on TM3 for single-precision and double-precision are shown in Fig. 6.

The *flop-rate* improvement of using both CPU–GPU over GPU-only is calculated by $((flop - rate_1 - flop - rate_2)/flop - rate_2) \times 100\%$, where $flop - rate_1$ is the performance of SpMV on CPU–GPU and $flop - rate_2$ is that of GPU-only. The *flop-rate* improvement of using both CPU–GPU over GPU-only on TM1 and TM2 are shown in Fig. 7. The *speedup* of using both CPU–GPU over CPU-only is calculated by $flop - rate_1/flop - rate_3$, where $flop - rate_1$ is the performance of SpMV on CPU–GPU and $flop - rate_3$ is that of CPU-only. The *speedup* of using both CPU–GPU over CPU-only on TM1 and TM2 are shown in Fig. 8. The *flop-rate* improvement and

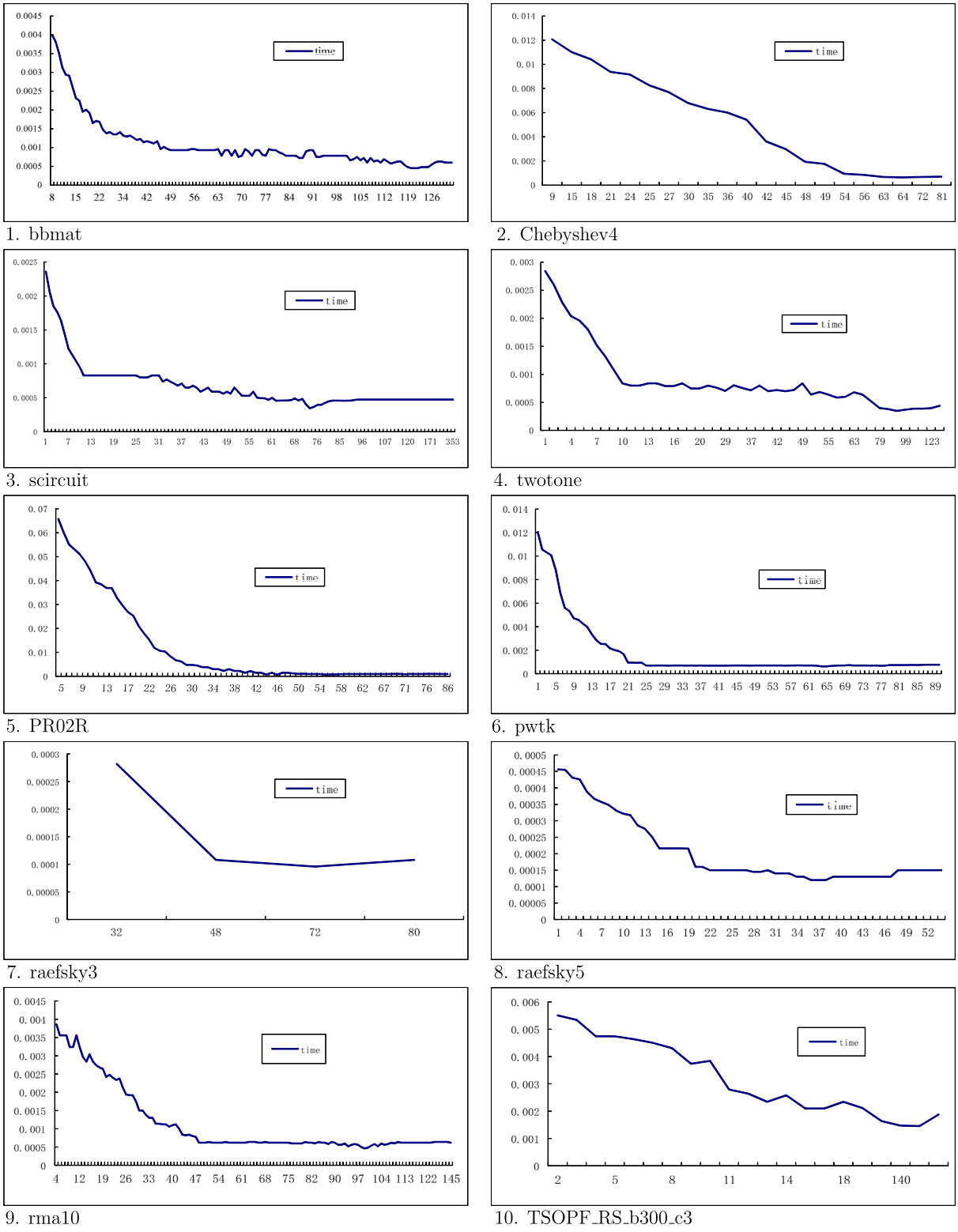


Fig. 2. The performance of SpMV using Algorithm 1 for various threshold K on the TM1 (unit:second).

speedup of using both CPU–GPU over GPU-only and CPU-only on TM3 are shown in Fig. 9.

We have the following important observations from our experimental data.

(1) The *flop-rate* of SpMV improves by 9.20%, 17.01%, and 8.17% for single-precision, and 9.33%, 16.29%, and 6.41% for double-precision, on the average by using both CPU–GPU compared with GPU-only on TM1, TM2, and TM3. Especially for the sparse matrix Simon/bbmat, the performance improves by more than 14% for TM1 and TM2, because the tasks divided using Algorithm 3 are relatively balanced for GPU and CPU.

(2) The *speedup* of SpMV is 23.80, 13.03, and 3.03 for single-precision, and 20.27, 10.64, and 2.56 for double-precision, on the average by using both CPU–GPU compared with CPU-only on TM1,

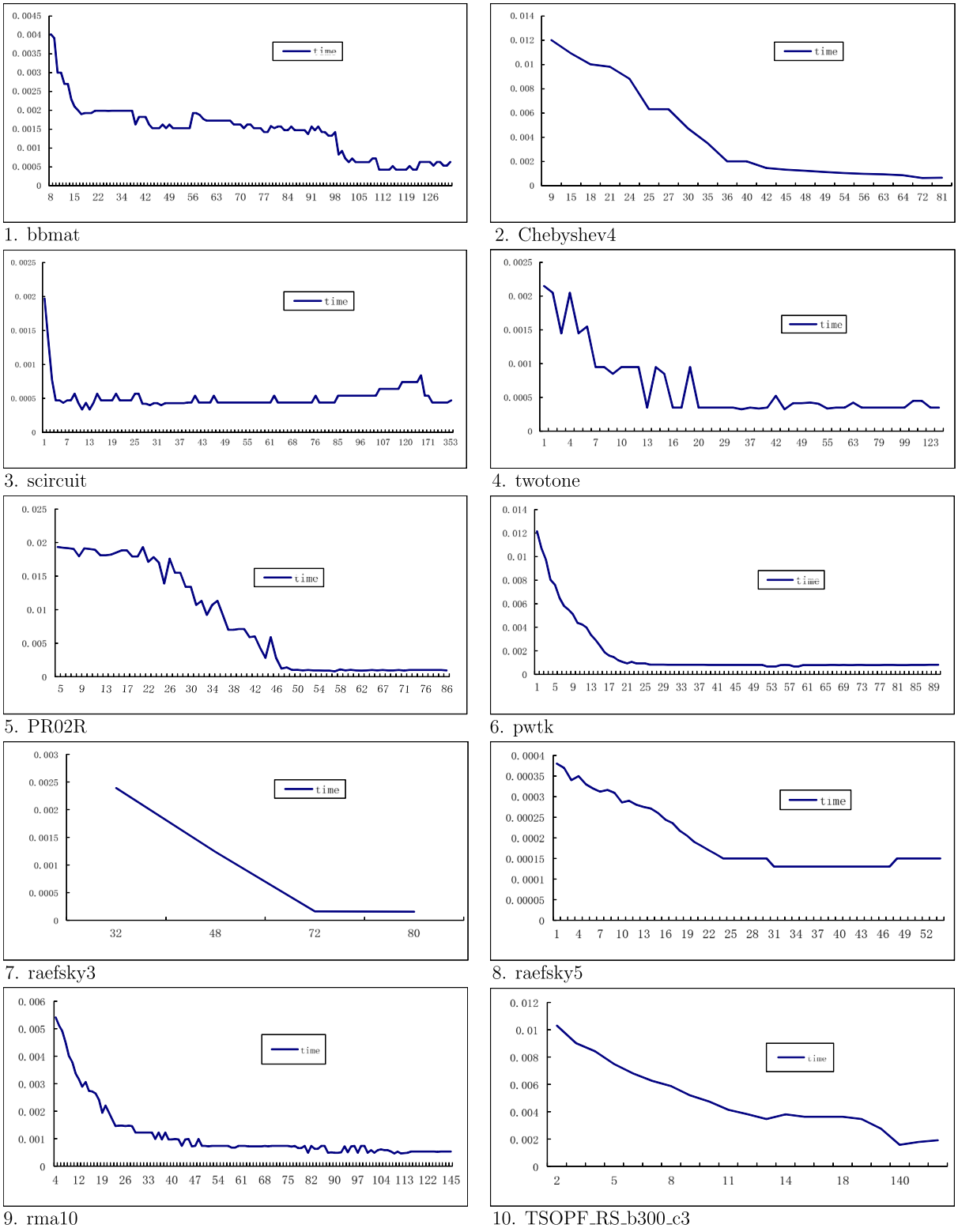
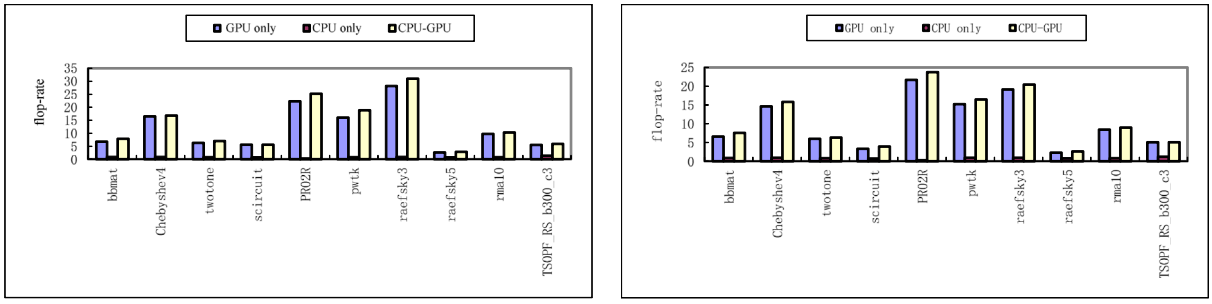


Fig. 3. The performance of SpMV using Algorithm 1 for various threshold K on the TM2 (unit:second).

TM2, and TM3. For Fluorem/PR02R, the performance improvement for SpMV on CPU–GPU is very significant, because the speedup of SpMV on GPU is high. The performance improvement for SpMV on CPU–GPU compared with CPU-only is not significant on TM3, because the performance of the CPU (i7 6700) is relatively close to that of GPU (GTX1070). Furthermore, The performance of SpMV on GPU is worse than that on CPU for Simon/raefsky5, because the scale of Simon/raefsky5 is too small to give full play to the parallel computing power of GPU. The effect of data transmission between CPU and GPU on the performance of SpMV is more obvious on TM3.

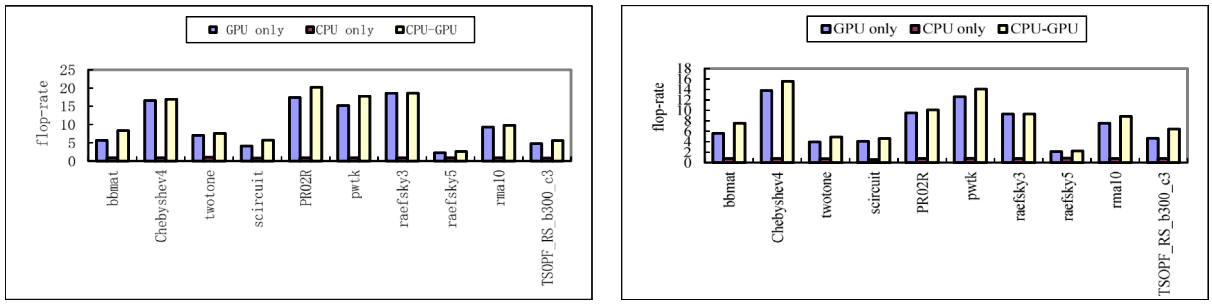
7. Concluding remarks

In this paper, we have developed a heterogeneous parallel computing method for SpMV based on a hybrid CPU–GPU



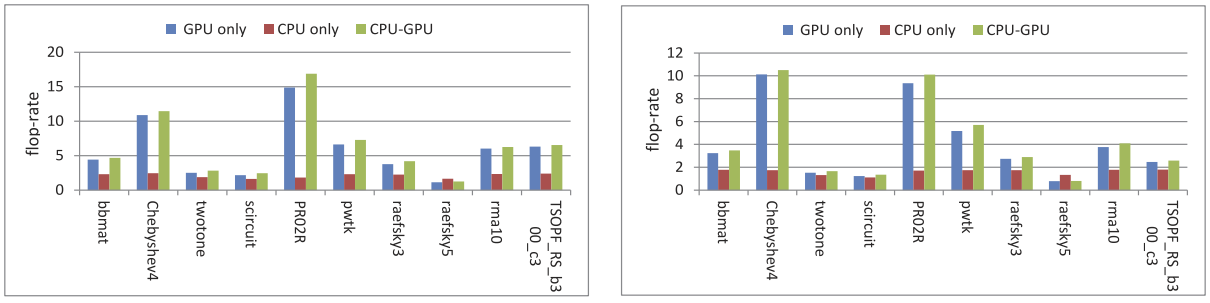
1. Single-precision 2. Double-precision

Fig. 4. Performance of CPU-only, GPU-only, and CPU–GPU on TM1 (unit:GFlop/s).



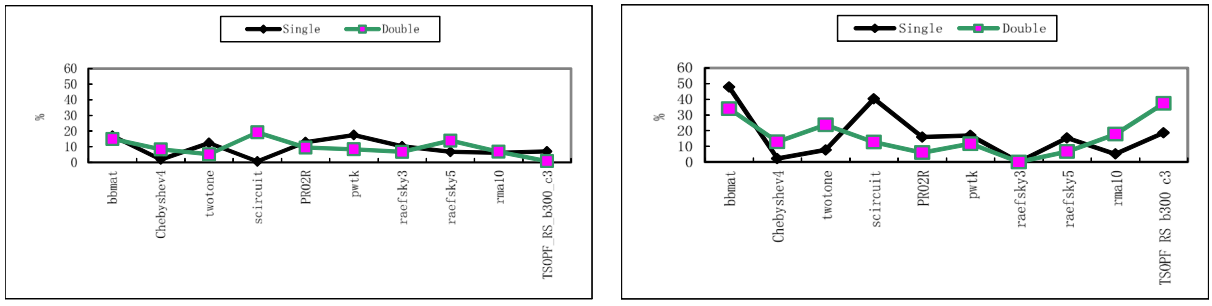
1. Single-precision 2. Double-precision

Fig. 5. Performance of CPU-only, GPU-only, and CPU–GPU on TM2 (unit:GFlop/s).



1. Single-precision 2. Double-precision

Fig. 6. Performance of CPU-only, GPU-only, and CPU–GPU on TM3 (unit:GFlop/s).

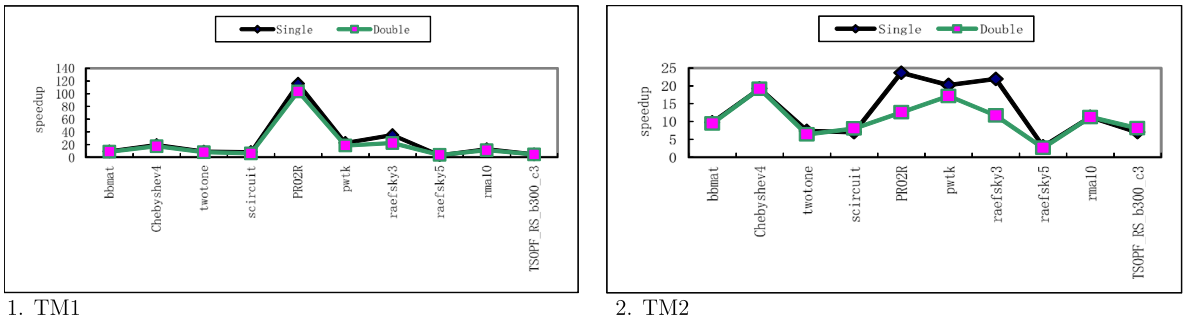


1. TM1 2. TM2

Fig. 7. The *flop-rate* improvement of using both CPU–GPU over GPU-only (unit:%).

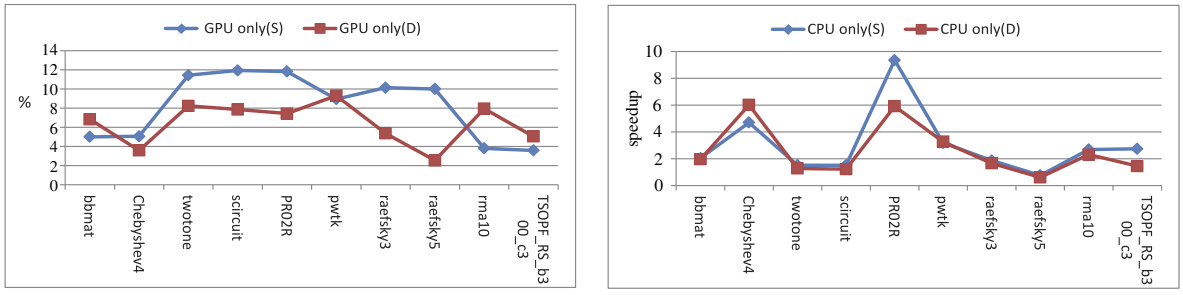
computing model. Our heterogeneous parallel computing model can make full use of the computing power of both CPU and GPU to improve the performance of SpMV. A sparse matrix can be split into two parts to be computed on CPU and GPU simultaneously. The partition of a sparse matrix can be optimized by performance analysis and workload balancing between CPU and GPU using a DF of sparse matrices.

The method is not only suitable for CPU–GPU cluster systems, but also applicable to other homogeneous and heterogeneous computing platforms. However, the performance of heterogeneous



1. TM1 2. TM2

Fig. 8. The *speedup* of using both CPU–GPU over CPU-only (unit:time).



1. The percentage of *flop-rate* improvement(unit:%) 2. The *speedup*(unit:time)

Fig. 9. The *flop-rate* improvement and *speedup* of using both CPU–GPU over GPU-only and CPU-only on TM3.

computing will be affected, because the data of CPU and GPU cannot be directly shared. AMD's APU Fusion chip can solve the problem of data sharing. In addition, the heterogeneous computing model with multi-GPUs and multi-CPUs is widely adopted in some supercomputers [19]. Our next step will be further investigation of heterogeneous computing strategies in these heterogeneous computing platforms and environments with further enhanced performance.

Acknowledgments

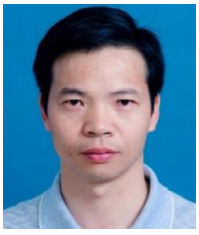
The authors deeply appreciate the anonymous reviewers for their comments on the manuscript. The research was partially funded by the National Natural Science Foundation of China (Grant Nos. 61572175, 61472124) and the Key Program of National Natural Science Foundation of China (Grant No. 61432005).

References

[1] M.M. Baskaran, R. Bordawekar, Optimizing sparse matrix vector multiplication on GPUs, Research Report RC24704, IBM TJ Watson Research Center, Tech. Rep., 2008, December.
[2] M. Belgin, G. Back, C.J. Ribbens, Pattern-based sparse matrix representation for memory-efficient SMVM kernels, in: Proceedings of the 23rd International Conference on Supercomputing, ACM, 2009, pp. 100–109.
[3] 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, ACM, 2010, pp. 80–88.
[4] A.R. Brodtkorb, T.R. Hagen, M.L. Sætra, Graphics processing unit (GPU) programming strategies and trends in GPU computing, J. Parallel Distrib. Comput. 73 (1) (2013) 4–13.
[5] L. Buatois, G. Caumon, B. Levy, Concurrent number cruncher: a GPU implementation of a general sparse linear solver, Int. J. Parallel Emerg. Distrib. Syst. 24 (3) (2009) 205–223.
[6] A. Buluc, S. Williams, L. Oliker, J. Demmel, Reduced-bandwidth multithreaded algorithms for sparse matrix–vector multiplication, in: 2011 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2011, pp. 721–733.
[7] A. Cevahir, A. Nukada, S. Matsuoka, Fast conjugate gradients with multiple GPUs. In: Proceedings of the International Conference on Computational Science (ICCS09), Baton Rouge, LA, May 2009, 2009, pp. 893–903.
[8] B. Chapman, G. Jost, R. Van Der Pas, Using OpenMP: Portable Shared Memory Parallel Programming, Vol. 10, The MIT Press, 2008.
[9] J.W. Choi, A. Singh, 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.
[10] T.A. Davis, Y. Hu, University of Florida sparse matrix collection, 2009.
[11] X. Feng, H. Jin, R. Zheng, K. Hu, J. Zeng, Z. Shao, Optimization of sparse matrix vector multiplication with variant CSR on GPUs, in: 2011 IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS), IEEE, 2011, pp. 165–172.
[12] 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.
[13] John L. Gustafson, Bruce S. Greer, Clearspeed whitepaper: Accelerating the intel math kernel library, 2007.
[14] S.B. Indarapu, M. Maramreddy, K. Kothapalli, Architecture-and workload-aware heterogeneous algorithms for sparse matrix vector multiplication, 2013. [Online]. http://cstar.iiit.ac.in/kkishore/spmv2.pdf.
[15] M. Kreutzer, G. Hager, G. Wellein, et al., Sparse matrix–vector multiplication on GPGPU clusters: A new storage format and a scalable implementation, in: Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & Ph.D. Forum, IPDPSW 12, IEEE Computer Society, Washington, DC, USA, 2012, pp. 1696–1702.
[16] J. Li, G. Tan, M. Chen, N. Sun, SMAT: an input adaptive auto-tuner for sparse matrix–vector multiplication, in: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, 2013, pp. 117–126.
[17] K. Li, W. Yang, K. Li, Performance analysis and optimization for SpMV on GPU using probabilistic modeling, IEEE Trans. Parallel Distrib. Syst. (2015) 196–205. http://doi.ieeecomputersociety.org/10.1109/TPDS.2014.2308221.
[18] K. Li, W. Yang, K. Li, A hybrid parallel solving algorithm on gpu for quasi-tridiagonal system of linear equations, IEEE Trans. Parallel Distrib. Syst. 27 (10) (2016) 2795–2808.
[19] F. Lu, J. Song, F. Yin, X. Zhu, Performance evaluation of hybrid programming patterns for large CPU/GPU heterogeneous clusters, Comput. Phys. Comm. 183 (6) (2012) 1172–1181.
[20] A. Monakov, et al., Automatically tunings parse matrix–vector multiplication for GPU architectures, in: YaleN. Patt, et al. (Eds.), High Performance Embedded Architectures and Compilers, in: Lecture Notes in Computer Science, vol. 5952, Springer, Berlin Heidelberg, 2010, pp. 111–125.
[21] NVIDIA, GPU. Computing Developer Home Page. Dostopno na: (2010). [Online]. http://developer.nvidia.com/object/gpucomputing.html.
[22] NVIDIA, CUDA. CUBLAS library programming guide. NVIDIA Corporation. edit 2 (2012). [Online]. http://docs.nvidia.com/cublas/index.html.
[23] NVIDIA CUDA C Programming Guide, Version 5.0, May 2012. [Online]. http://docs.nvidia.com/cuda-c-programming-guide/index.html.
[24] The NVIDIA CUDA Sparse Matrix Library (cuSPARSE), second ed., NVIDIA, 2012, [Online]. http://docs.nvidia.com/cusparse/index.html.

[25] T. Oberhuber, A. Suzuki, J. Vacata, New row-grouped CSR format for storing the sparse matrices on GPU with implementation in CUDA, Acta Tech. 56 (4) (2011) 447–466.
[26] 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.
[27] W.T. Tang, W.J. Tan, R. Ray, Y.W. Wong, et al. Accelerating sparse matrix–vector multiplication on GPUs using bit-representation-optimized schemes, in: Proceeding SC '13 Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2013.
[28] 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.
[29] W. Yang, K. Li, K. Li, Performance optimization using partitioned SpMV on GPUs and multicore CPUs, IEEE Trans. Comput. 64 (9) (2015) 2623–2636.
[30] 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.
[31] A.N. Yzelman, R.H. Bisseling, Two-dimensional cache-oblivious sparse matrix vector multiplication, Parallel Comput. 37 (12) (2011) 806–819.

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 heterogeneous computing systems, parallel algorithms, grid and cloud computing.



Kenli Li received the Ph.D. degree in computer science from Huazhong University of Science and Technology, China, in 2003 and the M.S. degree in mathematics from Central South University, China, in 2000. He was a visiting scholar at University of Illinois at Urbana–Champaign from 2004 to 2005. He is a full professor of computer science and technology at Hunan University and deputy director of National Supercomputing Center in Changsha. His major research includes parallel computing, grid and cloud computing, and DNA computing. He has over 100 research publications. He has published more than 100 papers in international conferences and journals such as IEEE-TC, IEEE-TPDS, JPDC, ICPP, CCGrid. He is an outstanding member of CCF.

Keqin Li is a SUNY distinguished professor of computer science. His current research interests include parallel computing and high-performance computing, distributed computing, energy-efficient computing and communication, heterogeneous computing systems, cloud computing, big data computing, CPU–GPU hybrid and cooperative computing, multicore computing, storage and file systems, wireless communication networks, sensor networks, peer-to-peer file sharing systems. He has over 300 research publications. He has published over 300 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. He is currently or has served on the editorial boards of *IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Computers, IEEE Transactions on Cloud Computing. 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*.