Optimization of Large-Scale Sparse Matrix-Vector Multiplication on Multi-GPU Systems

JIANHUA GAO, School of Artificial Intelligence, Beijing Normal University, Beijing, China
WEIXING JI, School of Artificial Intelligence, Beijing Normal University, Beijing, China
YIZHUO WANG, School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China

Sparse matrix-vector multiplication (SpMV) is one of the important kernels of many iterative algorithms for solving sparse linear systems. The limited storage and computational resources of individual GPUs restrict both the scale and speed of SpMV computing in problem-solving. As real-world engineering problems continue to increase in complexity, the imperative for collaborative execution of iterative solving algorithms across multiple GPUs is increasingly apparent. Although the multi-GPU-based SpMV takes less kernel execution time, it also introduces additional data transmission overhead, which diminishes the performance gains derived from parallelization across multi-GPUs. Based on the non-zero elements distribution characteristics of sparse matrices and the tradeoff between redundant computations and data transfer overhead, this article introduces a series of SpMV optimization techniques tailored for multi-GPU environments and effectively enhances the execution efficiency of iterative algorithms on multiple GPUs. First, we propose a two-level non-zero elements-based matrix partitioning method to increase the overlap of kernel execution and data transmission. Then, considering the irregular non-zero elements distribution in sparse matrices, a long-row-aware matrix partitioning method is proposed to hide more data transmissions. Finally, an optimization using redundant and inexpensive short-row execution to exchange costly data transmission is proposed. Our experimental evaluation demonstrates that, compared with the SpMV on a single GPU, the proposed method achieves an average speedup of 2.00× and 1.85× on platforms equipped with two RTX 3090 and two Tesla V100-SXM2, respectively. The average speedup of 2.65× is achieved on a platform equipped with four Tesla V100-SXM2.

CCS Concepts: • **Computing methodologies** → **Massively parallel algorithms**; • **Mathematics of computing** → **Solvers;**

Additional Key Words and Phrases: Multi-GPU system, sparse matrix-vector multiplication, data transmission hiding, sparse matrix partitioning

ACM Reference Format:
Jianhua Gao, Weixing Ji, and Yizhuo Wang. 2024. Optimization of Large-Scale Sparse Matrix-Vector Multiplication on Multi-GPU Systems. *ACM Trans. Arch. Code Optim.* 21, 4, Article 69 (November 2024), 24 pages. https://doi.org/10.1145/3676847

This work was supported by the Postdoctoral Fellowship Program of China Postdoctoral Science Foundation under Grant No. GZC20230261 and the National Natural Science Foundation of China under Grant No. 61972033.
Authors' Contact Information: Jianhua Gao, School of Artificial Intelligence, Beijing Normal University, Beijing, Beijing, China; e-mail: gaojh@bnu.edu.cn; Weixing Ji (Corresponding author), School of Artificial Intelligence, Beijing Normal University, Beijing, Beijing, China; e-mail: jwx@bnu.edu.cn; Yizhuo Wang, School of Computer Science and Technology, Beijing Institute of Technology, Beijing, Beijing, China; e-mail: frankwyz@bit.edu.cn.

ACM 1544-3973/2024/11-ART69
https://doi.org/10.1145/3676847

1 Introduction

Sparse Matrix-Vector Multiplication (SpMV) is one of the most time-consuming kernels in many iterative solving algorithms and is widely used in scientific and engineering computations. For example, in numerical simulation, partial differential equations modeled from real-world problems are discretized using methods such as finite element and finite difference, resulting in sparse linear systems that can be solved by computers. Iterative algorithms such as the **Generalized Minimal Residual (GMRES)** and **Conjugate Gradient (CG)** are commonly employed to solve these problems, and SpMV serves as the primary performance bottleneck among these algorithms. Figure 1 presents the time proportion of SpMV in the CG algorithm for more than 2,600 sparse matrices. It can be observed that as the number of non-zeros in sparse matrices increases, SpMV computations become the significant performance bottleneck. Therefore, optimizing the performance of SpMV is of great importance.

The acceleration and optimization of SpMV on multi-core or many-core processors has received increasing attention from researchers in the past decades, and significant research progress has been made in accelerating SpMV on a single GPU [17, 41]. However, due to the limited memory capacity and computing capability, large-scale SpMV computation still suffers from high latency on a single GPU. Large-scale SpMV [9, 12, 44] has a wide range of applications similar to general SpMV, such as numerical simulations and graph analysis. In these applications, when the scale of the target problem is large, that is, when the coefficient matrix of the linear system in numerical simulations is large-scale, or the number of vertices in graph analysis is significant, the involved SpMV is a large-scale SpMV calculation. When the size of sparse matrices exceeds the size of GPU memory, it results in frequent data transmission between CPU and GPU during each iteration. This significantly hampers the efficiency of problem-solving.

An increasing number of supercomputers are including GPUs as their core acceleration units and equipping multiple GPU devices within a single compute node. In the latest TOP500 list (November 2022),[1] the top-10 systems include Leonardo from Italy, Perlmutter and Selene from the United States, all of which have four NVIDIA A100 GPUs per node. Similarly, Summit and Sierra, also from the United States, have six and four NVIDIA Volta GV100 GPUs per compute node, respectively. Therefore, maximizing the collaborative computing capabilities of multiple GPUs to accelerate large-scale SpMV is important and prospective.

Despite their advantage in providing more computing and storage resources for SpMV computations, multi-GPU systems introduce significant transmission overhead among GPUs. Our extensive research on 24 large-scale sparse matrices reveals that the time spent on data transmission takes a significant portion of the overall SpMV calculation time: up to 53% on a two-GPU platform and 63% on a four-GPU platform. This observation indicates that reducing data transmission overhead in multi-GPU SpMV computing is extremely important. Currently, researchers have proposed some optimization techniques for SpMV on multi-GPU systems [1, 10, 11, 23–25, 28, 30, 33, 43, 45, 47]. On one hand, these optimizations mainly focus on specific iterative algorithms for multi-GPU optimization. However, due to the diversity of SpMV applications and iterative algorithms, the applicability of these methods is limited. On the other hand, the mainstream approach in these works is to use graph partitioning to transform the sparse matrix into a band matrix, aiming to minimize the data transmission overhead. However, this partitioning algorithm has two drawbacks. First, it incurs a high preprocessing overhead. Compared with relatively lightweight SpMV computations, graph partitioning requires row-column permutation, and the result vector needs to be rearranged to its initial order after the iterations. The time overhead of the partitioning can even surpass the

[1] https://www.top500.org/lists/top500/2022/11/

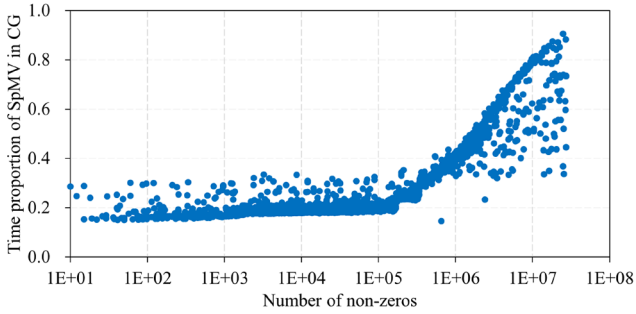


Fig. 1. Time proportion of SpMV in the CG algorithm.

performance benefits of multi-GPU parallel computing. Second, graph partitioning demands significant memory requirements. It requires two copies of the sparse matrix to be stored in memory simultaneously for the conversion from the input matrix to the banded matrix. These drawbacks are highly detrimental to the computation of large-scale sparse matrices.

This article introduces a series of SpMV optimization techniques tailored for multi-GPU and effectively enhances the execution efficiency of iterative algorithms on multiple GPUs. First, we show the overall framework of iterative algorithms on a multi-GPU system and explain the necessity of synchronizing the result vector of SpMV across multiple GPUs. To mitigate the data transmission overhead, this article proposes a two-level non-zero elements-based matrix partitioning and a long-row-aware matrix partitioning method. These methods enable to hiding data transmission process as much as possible. Additionally, we introduce an optimization strategy that uses redundant and inexpensive computation in exchange for costly data transmission. Finally, experimental evaluations on two different platforms demonstrate that compared with the SpMV performance on a single GPU, the proposed optimization method achieves an average speedup of 2.00× and 1.85× on platforms equipped with two RTX 3090 and two Tesla V100-SXM2, respectively. On a platform equipped with four Tesla V100-SXM2, our method achieves an average speedup of 2.65×. We also apply our multi-GPU SpMV to the CG solver, and a higher average speedup is achieved.

The structure of this article is organized as follows: Section 2 presents an overview of related work for optimizing SpMV on multi-GPU systems. Section 3 introduces several optimization methods proposed in this article. Section 4 provides experimental evaluations of the proposed algorithms on two different multi-GPU platforms. Finally, Section 5 concludes the article.

2 Related Work

2.1 SpMV Optimization on Single GPU

GPUs are equipped with a large number of lightweight computing cores, allowing a large number of computing threads to be executed simultaneously. The irregular distribution of non-zeros in sparse matrices makes SpMV computation on GPU more susceptible to load imbalance. To solve this problem, Garland et al. [40] reorganize the irregular SpMV calculation into regular *map*, *scan,* and *reduce* operations to improve the performance of SpMV. Ashari et al. [3] combine rows containing similar non-zeros into the same bin and launch different computing kernels for different bins. Daga and Greathouse [14] and Greathouse and Daga [26] propose CSR-Stream and CSR-VectorL to solve the load imbalance problem in SpMV calculation. The CSR-Stream fixes the number of non-zeros processed by each warp. CSR-VectorL allocates multiple workgroups to handle extremely long rows. Subsequently, the CSR5-based SpMV [38] and merge-based SpMV [39] allocate strictly the same number of non-zeros to each threads block, which greatly improves the load imbalance

on irregular sparse matrices. Anzt et al. [2] and Flegar and Anzt [19] partition the sparse matrix into blocks with a similar number of non-zeros and assign a warp to each block. Gao et al. [20] propose adaptive multi-row folding and non-zero-based blocking to alleviate load imbalance in CMRS [32].

In addition, the powerful parallel computing capabilities of GPU and the inherent discrete memory access pattern of SpMV make improving memory access efficiency a key direction for optimizing SpMV on GPU. CSR-Scalar [5] is a basic parallel implementation of CSR-based SpMV on GPU, where each matrix row is assigned to a single thread. However, its performance degrades as the number of non-zeros per row increases due to the uncoalesced memory accesses to column indexes and non-zeros. To address these limitations, CSR-Vector [4–6] allocates a threads group per row to achieve coalesced memory access. The setting of group size has been discussed in several works [4–6, 21, 42, 48]. At the expense of a larger memory footprint, the extended vector algorithm [16, 29] allocates a new vector of the same size as the value array, and the multiplied element in the multiplication vector for each non-zero is stored into the allocated vector to maximize coalesced memory accesses on the GPU.

Furthermore, considering that sparse matrices from diverse applications exhibit different patterns in the distribution of non-zeros, it becomes the fact that no single sparse format or SpMV algorithm can consistently achieve the highest performance across all matrices or hardware platforms. Consequently, several SpMV optimization techniques have emerged, using auto-tuning technology [13, 27, 35, 36, 46] or machine learning approaches [7, 8, 18, 34, 49] to find the most suitable sparse format, SpMV algorithm, or parameter configuration. The search space of these methods contains sparse compression formats or SpMV algorithms that have been proposed by experts. However, for a certain problem, there may be better sparse formats or SpMV algorithms that experts have not designed. Therefore, recent research efforts [17] are focused on facilitating the automatic generation of novel formats or SpMV algorithms. Our recent SpMV review [22] provides a more detailed and systematic introduction to SpMV optimization efforts on a single GPU.

2.2 SpMV Optimization on Multi-GPU Systems

SpMV computation on the multi-GPU system introduces additional data transmission between GPUs, and its overhead is closely associated with matrix partitioning. Therefore, a critical approach to optimizing SpMV on the multi-GPU system is to design appropriate matrix partitioning algorithms to reduce data transmission overhead across GPUs.

Cevahir et al. [10] propose a multi-GPU optimization method for CG solver. They use the JDS format (which requires sorting the matrix) to encode the sparse matrix and employ an SpMV implementation similar to CSR-based SpMV for merging access to index and value data. Load-based partitioning is used to partition the sparse matrix across multiple GPUs.

Guo et al. [28] and Karwacki et al. [30] both use the HYB compression format. In Reference [28], two threads are launched using OpenMP to distribute the ELL and COO encoded parts to two GPUs for computation, followed by results reduction on the host side. Additionally, this work achieves the overlap of **host-to-device (H2D)** transmission, kernel execution, and **device-to-host (D2H)** transmission using CUDA streams on each GPU. However, the method is only applicable to the system with two GPUs and lacks generality. Karwacki et al. [30] propose a multi-GPU implementation of a uniformization method for solving Markov models, in which SpMV is the most important kernel. In this work, the ELL and COO encoded parts are evenly partitioned first and then assigned to two GPUs for computation.

Verschoor et al. [45] and Abdelfattah et al. [1] both use the BCSR compression format. Verschoor et al. [45] employ a load-aware partitioning method to divide the BCSR-encoded sparse matrix among multiple GPUs. After the partitioning is done, the sub-matrix handled by each GPU is

sorted based on the number of blocks per row. Abdelfattah et al. [1] design three kernels suitable for different BCSR block sizes.

Yang et al. [47] focus on the optimization of the GMRES algorithm. They use the quasi-optimal partitioning method provided in METIS [31] to distribute the non-zero elements (non-zeros) near the main diagonal, aiming to reduce the transmission overhead between GPUs. Lin et al. [37] focus on a simple and fast reordering method to reduce the amount of data transmission.

Schaa et al. [43] propose a performance prediction model for multi-GPU systems, considering different types of multi-GPU systems such as distributed memory and shared memory configurations. This model enables GPU developers to infer the performance acceleration of their applications on any number of GPUs. Gao et al. [23–25] propose a profiling-based performance model for modeling the main components of the PCG algorithm.

Li et al. [33] use a blocked ELL format, and the encoded matrix is first evenly partitioned row-wise into one-dimensional blocks, with the number of blocked rows equal to the number of GPU devices. Then, on each GPU, the corresponding rows are evenly partitioned column-wise to achieve the overlap of the current blocks' SpMV computation and the transmission required for the next block. Compared with one-dimensional partitioning, two-dimensional partitioning often incurs significant preprocessing overhead, as it requires changing the order of non-zeros in the sparse matrix. Additionally, the irregular distribution of non-zeros in the sparse matrix is not considered in this work. Chen et al. [11] propose the MSREP framework for sparse matrix representation on multi-GPU systems. It uses strict load-balanced matrix partitioning to ensure that each GPU handles a similar number of non-zeros. However, this also introduces additional overhead for reducing the results from multiple GPUs, which is not suitable for the prevalent iterative scenarios in SpMV applications. We conduct a performance comparison with the work in Section 4, demonstrating the superiority of our proposed method.

In summary, there are still several challenges in SpMV computations on multi-GPU systems. These challenges include high preprocessing overhead and significant memory requirements for rearranging matrices, as well as the underutilization of non-zeros distribution. Differently, our proposed method just requires scanning the row offset array of CSR format, which is lightweight and friendly to large-scale sparse matrices. Moreover, the non-zeros distribution is used to guide row-wise cost estimation for result transmission and vector inner product.

3 Methodology

SpMV is primarily used in iterative algorithms, thus most optimization methods are proposed based on multiple invocations to SpMV during the iterative process. Algorithm 1 presents the outline of the CG algorithm. It can be observed that, in each iteration, the computation result of SpMV (line 4) is used in subsequent dot product (line 5) and scalar-vector multiplication (line 6), generating the multiplication vector $\boldsymbol{p}_i$ of SpMV for the next iteration (line 8).

This article focuses on the acceleration and optimization of SpMV on multi-GPU systems while considering its application in iterative scenarios. We consider an iterative framework as shown in Figure 2. After each GPU completes its SpMV kernel execution, the all-to-all synchronization is performed among multiple GPUs to ensure that each GPU has the complete result vector. Subsequently, multiple GPUs collectively execute other calculations in the iteration. Before executing the next SpMV, each GPU already has the complete multiplication vector, eliminating the need for further data synchronization. Starting from the naive non-zeros-based matrix partitioning, we first propose two-level non-zeros-based matrix partitioning to overlap the kernel computation and data transmission. Considering the irregular distribution in sparse matrices, we then design a long-row-aware matrix partitioning method. Finally, an optimization method that uses a redundant and lightweight kernel execution in exchange for expensive data transmission is proposed.

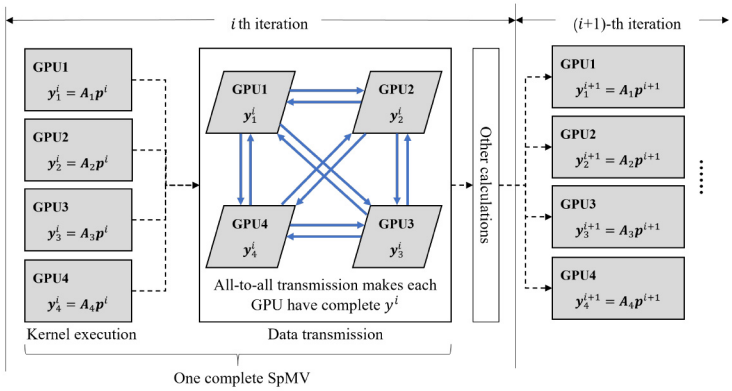


Fig. 2. Workflow of iterative algorithms on multi-GPU platforms (taking four GPUs for example).

ALGORITHM 1: Conjugate gradient algorithm

Input: coefficient matrix $\boldsymbol{A}$, right-hand-side vector $\boldsymbol{b}$, initial guess $\boldsymbol{x}_0$, tolerance threshold ϵ, and maximum iterations MAX

Output: approximate solution $\hat{\boldsymbol{x}}$

1 $\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$; /* SpMV */

2 $rOld = \boldsymbol{r}_0^T \boldsymbol{r}_0$; $\boldsymbol{p}_0 = \boldsymbol{r}_0$;

3 **for** $(i = 1; (\sqrt{rOld} \geq \|\boldsymbol{b}\|\epsilon)$ && $(i \leq MAX)\,; i++)$ **do**

4 $\quad \boldsymbol{y}_i = \boldsymbol{A}\boldsymbol{p}_{i-1}$; /* SpMV */

5 $\quad \alpha = \frac{rOld}{(\boldsymbol{p}_{i-1}^T \boldsymbol{y}_i)}$;

6 $\quad \boldsymbol{x}_i = \boldsymbol{x}_{i-1} + \alpha \boldsymbol{p}_{i-1}$; $\boldsymbol{r}_i = \boldsymbol{r}_{i-1} - \alpha \boldsymbol{y}_i$;

7 $\quad rNew = \boldsymbol{r}_i^T \boldsymbol{r}_i$; $\beta = \frac{rNew}{rOld}$; $rOld = rNew$;

8 $\quad \boldsymbol{p}_i = \boldsymbol{r}_i + \beta \boldsymbol{p}_{i-1}$;

9 **end**

10 $\hat{\boldsymbol{x}} = \boldsymbol{x}_{i-1}$;

3.1 Non-zeros-based Matrix Partitioning

In SpMV, the calculations for different rows are independent. Therefore, an intuitive partitioning is to evenly divide the sparse matrix by rows into multiple sub-blocks, with each GPU handling one sub-block. However, considering the irregular distribution of non-zeros across different matrix rows, this partitioning method may cause load unbalance among multiple GPUs. One more reasonable partitioning approach is based on the **number of non-zeros (NNZ)**, because it largely determines the workload of SpMV. We refer to this partitioning method as non-zeros-based partitioning in this article. Non-zeros-based partitioning divides the sparse matrix into multiple sub-blocks, where the number of sub-blocks equals the number of GPUs in the system. Each sub-block consists of consecutive matrix rows and has a similar number of non-zeros to others, approximately equal to $nnz/nGPU$, where nnz is the total number of non-zeros in a sparse matrix and $nGPU$ is the number of GPU devices in a system. Figure 3(a) illustrates an example of non-zeros-based partitioning. In the example, the matrix has a size of 7×7 and contains 17 non-zeros. When the number of GPUs is two, the results of non-zeros-based partitioning are $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$, where the first sub-block $\boldsymbol{A}_1$ consists of the first two rows, containing 8 non-zeros, and the second sub-block $\boldsymbol{A}_2$ consists of the last five rows, containing 9 non-zeros.

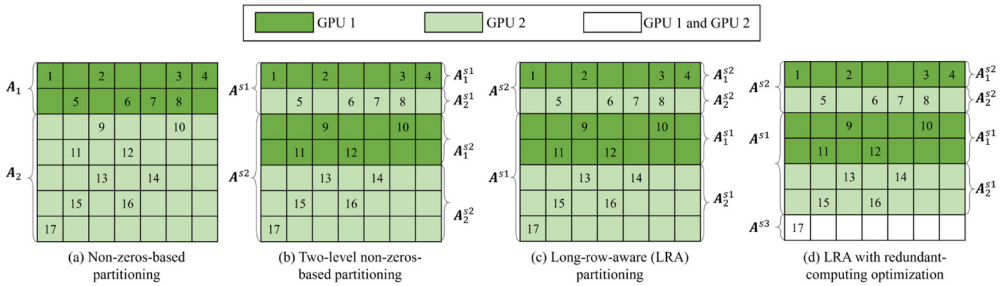


Fig. 3. Example of sparse matrix partitioning (taking two GPUs for example). The subscript 1 indicates that the first GPU is responsible for handling that sub-block, and the superscript $s1$ indicates that the corresponding SpMV kernel is executed in the first CUDA stream.

In the SpMV algorithm using non-zeros-based partitioning, two threads are allocated to control each GPU device and call the SpMV kernel for the assigned sub-block. Finally, each GPU transmits its calculated results to other GPUs, ensuring that each GPU has the complete result vector. Due to the inherent parallelism in multi-GPU systems, when a GPU completes the kernel execution ahead of others, it can transmit its calculated results to another GPU, and another GPU can proceed with its independent kernel execution while concurrently receiving transmitted data. This enables an initial overlap between kernel execution and data transmission across multiple GPUs. Figure 4(a) illustrates the computation and transmission process in SpMV using non-zeros-based partitioning. The second GPU takes more time for kernel execution and data transmission, because the sub-block it deals with contains more non-zeros and more matrix rows than the first GPU.

One of the advantages of non-zeros-based partitioning is its low partitioning overhead. Taking the most popular sparse compression format CSR as an example, given the number of GPU devices in the system, it only requires traversing the array that stores the offsets of non-zeros for each row. Additionally, compared with rows-based partitioning, the SpMV computation using the non-zeros-based partitioning exhibits better load balance. However, the transmission overhead introduced by multi-GPU computations can significantly reduce the overall performance benefits. In some sparse matrices, the introduced transmission overhead can even exceed the performance gains achieved in the kernel execution. In those cases, the performance of SpMV on multi-GPU systems is inferior to that on a single GPU. Taking the sparse matrix *europe_osm* as an example, the kernel execution using two GPUs achieves a speedup of 1.48× over one GPU. However, when considering the overhead of data transmission, the overall speedup is less than 1.

3.2 Two-level Non-zeros-based Matrix Partitioning

In non-zeros-based matrix partitioning, each GPU needs to complete the kernel execution for the assigned sub-block before transmitting its calculated results. It means that, on each GPU, the kernel execution and data transmission always run sequentially. Therefore, we propose a two-level non-zeros-based partitioning algorithm to achieve overlap between kernel execution and data transmission on each GPU. Two-level non-zeros-based matrix partitioning divides the sparse matrix into $2 \times nGPU$ sub-blocks, where each sub-block consists of multiple consecutive matrix rows, and different sub-blocks have a similar number of non-zeros, approximately $nnz/(2 \times nGPU)$.

Figure 3(b) illustrates an example of two-level non-zeros-based matrix partitioning on two GPUs. First, the non-zeros-based matrix partitioning is used to divide the matrix into two sub-matrix blocks, denoted as $\boldsymbol{A}^{s1}$ and $\boldsymbol{A}^{s2}$. Then, the non-zeros-based partitioning is used again to further

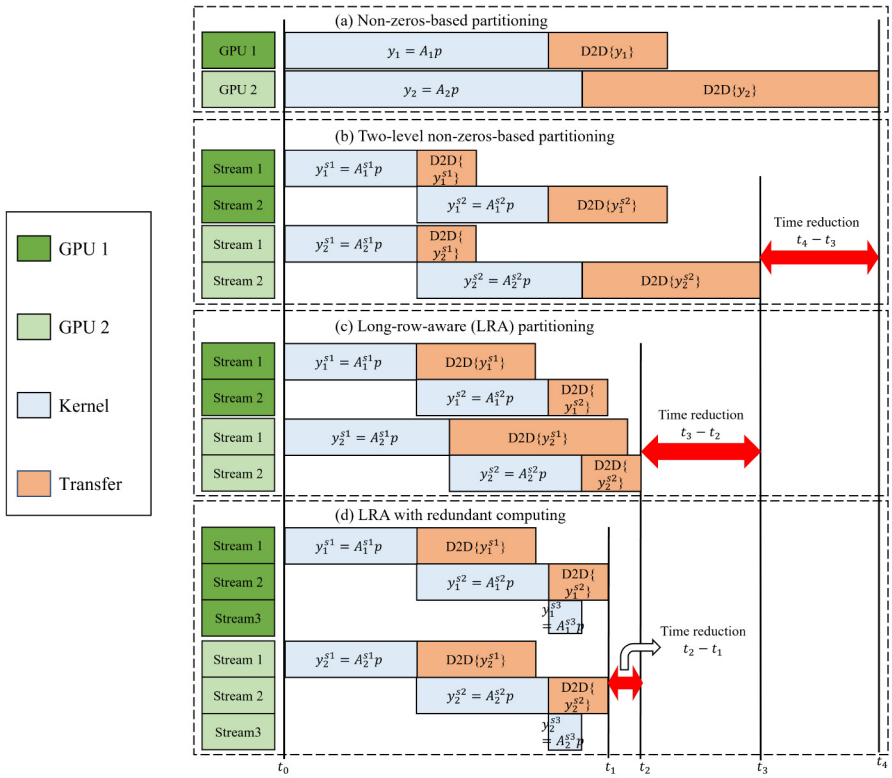


Fig. 4. Comparison of kernel execution and data transmission processes in SpMV using different sparse matrix partitioning methods. Taking (a) as an example, D2D{$\boldsymbol{y}_1$} represents transmission of result $\boldsymbol{y}_1$ from the first GPU to the second GPU.

divide each sub-block into two sub-blocks, resulting in four sub-blocks. In the example, the sub-block $\boldsymbol{A}^{s1}$ is divided into $\boldsymbol{A}_1^{s1}$ and $\boldsymbol{A}_2^{s1}$, and $\boldsymbol{A}^{s2}$ is further divided into $\boldsymbol{A}_1^{s2}$ and $\boldsymbol{A}_2^{s2}$.

In the computation process, the first GPU is responsible for processing sub-blocks $\boldsymbol{A}_1^{s1}$ and $\boldsymbol{A}_1^{s2}$, while the second GPU handles $\boldsymbol{A}_2^{s1}$ and $\boldsymbol{A}_2^{s2}$. Each GPU launches two CUDA streams to separately run the SpMV kernels for two sub-blocks, enabling the overlap of data transmission for the first sub-block and kernel execution for the second sub-block. As shown in Figure 4(b), taking the first GPU as an example, its first stream (Stream 1) handles the sub-block $\boldsymbol{A}_1^{s1}$, while Stream 2 handles the sub-block $\boldsymbol{A}_1^{s2}$. After Stream 1 completes the assigned SpMV kernel, it starts sending the calculated results to the second GPU. Meanwhile, Stream 2 starts executing the assigned SpMV kernel, thereby hiding the transmission of $\boldsymbol{y}_1^{s1}$, represented as D2D{$\boldsymbol{y}_1^{s1}$} in Figure 4(b). Compared with the non-zeros-based partitioning in Figure 4(a), the proposed two-level non-zeros-based matrix partitioning algorithm reduces the time overhead by $t_4 - t_3$.

3.3 Long-row-aware Sparse Matrix Partitioning

Due to the irregular distribution of non-zeros, different sub-blocks often take different data transmission overheads, further affecting the performance benefits of the two-level non-zeros-based partitioning. Observing Figures 3(b) and 4(b) and taking the second GPU as an example, sub-blocks $\boldsymbol{A}_2^{s1}$ and $\boldsymbol{A}_2^{s2}$ contain a similar number of non-zeros, thus taking similar kernel execution time.

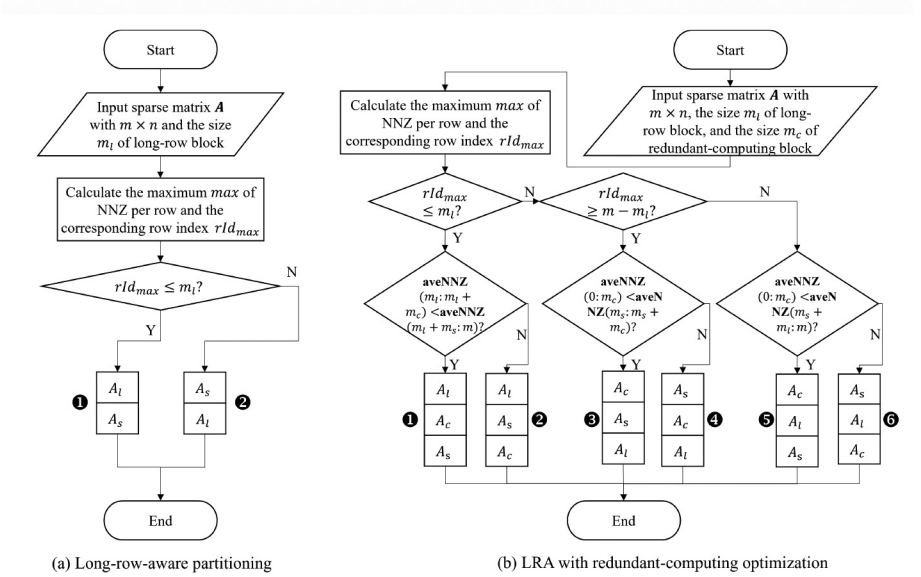


Fig. 5. Two matrix partitioning algorithms: (a) Long-row-aware partitioning, (b) long-row-aware partitioning with redundant-computing optimization.

However, in terms of transmission, given a fixed transmission bandwidth, the time overhead of the transmission process depends on the data transmission amount, i.e., the number of matrix rows contained in each sub-block. Sub-block $\boldsymbol{A}_2^{s1}$ only contains one matrix row, while sub-block $\boldsymbol{A}_2^{s2}$ contains three matrix rows. Therefore, in Figure 4(b), the transmission process D2D$\{\boldsymbol{y}_2^{s1}\}$ for the calculated result $\boldsymbol{y}_2^{s1}$ of sub-block $\boldsymbol{A}_2^{s1}$ takes nearly two-thirds less time overhead compared with D2D$\{\boldsymbol{y}_2^{s2}\}$. The large transmission overhead of the second sub-block on both GPUs limits the overall performance benefits of the two-level non-zeros-based partitioning. Therefore, we further propose a long-row-aware matrix partitioning algorithm. It first identifies the locations of long-row blocks in the matrix and then uses the non-zeros-based partitioning method to assign the short-row blocks to multiple GPUs, where the short-row blocks are the sub-matrix excluding the long-row blocks. Afterward, the non-zeros-based partitioning method is used again to assign the long-row blocks to multiple GPUs. Now, each GPU is assigned one long-row block and one short-row block.

Figure 5(a) illustrates the workflow of the long-row-aware partitioning. Given an $m \times n$ sparse matrix $\boldsymbol{A}$ as input, $\boldsymbol{A}_l$ and $\boldsymbol{A}_s$ represent its long-row block and short-row block, respectively. The objective of the algorithm is to find the positions of $\boldsymbol{A}_l$ and $\boldsymbol{A}_s$ in $\boldsymbol{A}$. Let d_l represent the ratio of rows in the long-row block to the total matrix rows. The number of matrix rows contained in the long-row block is calculated as $m_l = \lfloor d_l \times m \rfloor$. Letting max denote the maximum NNZ per row in $\boldsymbol{A}$, we can calculate its corresponding row index rId_max. If $rId\_max \leq m_l$, i.e., the longest row is located within the first m_l rows, then the sub-block composed of the first m_l rows is considered as the long-row block $\boldsymbol{A}_l$. Otherwise, the sub-block composed of the last m_l rows is considered as the long-row block.

For an irregular sparse matrix, when the long-row block and short-row block contain a similar number of non-zeros, the short-row block contains more matrix rows, resulting in higher transmission overhead for the corresponding calculated result. Therefore, the SpMV kernel for the short-row block should be executed first, followed by the kernel for the long-row block. This allows for

the overlap of the result transmission for the short-row block and kernel execution for the long-row block, thereby hiding more of the transmission process. Figure 3(c) illustrates an example of long-row-aware matrix partitioning. $\boldsymbol{A}_1^{s1}$ and $\boldsymbol{A}_1^{s2}$ are the short-row block and long-row block, respectively, assigned to the first GPU, while $\boldsymbol{A}_2^{s1}$ and $\boldsymbol{A}_2^{s2}$ are the short-row block and long-row block, respectively, assigned to the second GPU. Figure 4(c) shows the execution process for this example. Each GPU first executes the SpMV kernel for the short-row block and then for the long-row block. Taking the first GPU as an example, the long-row block $\boldsymbol{A}_1^{s2}$ and short-row block $\boldsymbol{A}_1^{s1}$ contain the same NNZ, so we assume that their kernel execution time is largely the same. However, the number of rows in the short-row block is twice that of the long-row block, resulting in double the transmission overhead over the long-row block. In Figure 4(c), when the kernel execution for the short-row block is completed, the kernel execution for the long-row block $\boldsymbol{y}_1^{s1} = \boldsymbol{A}_1^{s1} \times \boldsymbol{p}$ and the transmission D2D$\{\boldsymbol{y}_1^{s1}\}$ for the short-row block start simultaneously. As D2D$\{\boldsymbol{y}_1^{s1}\}$ in Figure 4(c) is longer than D2D$\{\boldsymbol{y}_1^{s1}\}$ in Figure 4(b), the long-row-aware partitioning method can hide more of the transmission process. It can be observed that, compared with the two-level non-zeros-based partitioning, using the long-row-aware matrix partitioning can reduce the time overhead of $t_3 - t_2$.

3.4 Redundant-computing Optimization

From Figure 4(c), it can be observed that the efficiency of kernel execution and data transmission corresponding to long-row blocks affects the overall efficiency of SpMV on multi-GPU systems. The parallel computation on multiple GPUs accelerates the execution of the kernel, but the introduced transmission overhead becomes a critical factor limiting performance improvement. Based on the fact that SpMV on a single GPU does not require data transmission, we further propose redundant-computing optimization. The main idea is to identify the most cost-effective sub-block from the short-row block. Then, every GPU executes the SpMV kernel for this sub-block, without the requirement for transmitting the calculated results to other GPUs. Here, the most cost-effective block refers to the block that contains the maximum number of rows but takes the shortest kernel execution time.

Figure 5(b) illustrates the long-row-aware partitioning with the redundant-computing optimization. Let $\boldsymbol{A}_c$ represent the redundant-computing sub-block. The objective of the algorithm is to find the locations of $\boldsymbol{A}_l$, $\boldsymbol{A}_s$, and $\boldsymbol{A}_c$ within the matrix $\boldsymbol{A}$. Let d_l and d_c denote the ratios of the matrix rows contained in the long-row block and redundant-computing block to the total number of matrix rows, respectively. We can calculate the matrix rows m_l, m_s, and m_c contained in the long row block, short row block, and redundant-computing block, respectively. Similarly, the row index rId_max of the longest row is calculated, and it determines the location of $\boldsymbol{A}_l$. Assuming $rId\_max \leq m_l$, the first m_l rows are composed of the long-row block $\boldsymbol{A}_l$. Conversely, if $rId\_max \geq m - m_l$, then the last m_l rows are composed of the long-row block. Otherwise, some m_l rows in the middle of $\boldsymbol{A}$ are considered as the long-row block. Taking the first case for example, where the first m_l rows are considered as the long row block $\boldsymbol{A}_l$, the algorithm further determines whether the subsequent m_c rows or the last m_c rows of the matrix have a smaller average NNZ per row. The $aveNNZ(m_l : m_l + m_c)$ in Figure 5(b) represents the average NNZ per row in the sub-matrix block from the $(m_l + 1)$-th row to the $(m_l + m_c)$-th row. When the number of matrix rows is the same, the sub-block with a smaller average NNZ per row contains less NNZ, thus taking less kernel execution cost, making it a more suitable candidate for designation as the redundant-computing block $\boldsymbol{A}_c$. Then, the remaining m_s rows are considered as the short row block $\boldsymbol{A}_s$. Therefore, the corresponding partitioning results in the discussed example are ❶ or ❷ presented in Figure 5(b). The remaining cases follow a similar pattern. Once $\boldsymbol{A}_l$, $\boldsymbol{A}_s$, and $\boldsymbol{A}_c$ are located, the non-zeros-based partitioning is used to divide $\boldsymbol{A}_l$ and $\boldsymbol{A}_s$ into $nGPU$ sub-blocks, respectively.

Table 1. Hardware and Software

		Platform 1	Platform 2
Hardware	Host	CPU: Intel(R) Core(TM) i9-10920X, 3.5 GHz, 12 cores Memory: 192 GB	CPU: Intel(R) Xeon(R) Gold 5218 v5, 2.3 GHz, 16 cores Memory: 192 GB
	Device	GPU: GeForce RTX 3090, 1.70 GHz, 10,496 cores Memory: 24 GB	GPU: Tesla V100-SXM2, 1.53 GHz, 5,120 cores Memory: 32 GB
Software	OS	64-bit Ubuntu 18.04	64-bit CentOS 7.9.2009
	Compiler	nvcc:11.1.74; gcc/g++: 9.2.0	nvcc:11.6.55; gcc/g++: 7.5.0
	Library	cuSPARSE: 11.1	cuSPARSE: 11.6

In the example presented in Figure 3(d), let us take $d_l = 0.3$ and $d_c = 0.1$. This yields $m_l = 2$, $m_s = 4$, and $m_c = 1$. The longest row of the matrix is the first row, satisfying the condition $rId\_max \leq m_l$ as depicted in Figure 5(b). Therefore, the first two rows of the matrix are considered as the long-row block $\boldsymbol{A}_l$. Next, we compare $aveNNZ(3:3) = 2$ (i.e., the NNZ in the third row of the matrix) with $aveNNZ(7:7) = 1$ (i.e., the NNZ in the last row of the matrix). Consequently, the last row is composed of the redundant-computing block $\boldsymbol{A}_c$, and the middle four rows are considered as the short-row block $\boldsymbol{A}_s$, corresponding to the partitioning result ❷ in Figure 5(b).

Building upon the long-row-aware partitioning, the third CUDA stream is launched on each GPU to handle the redundant-computing sub-block. On one hand, the kernel execution for the sub-block can overlap with the transmission process for the calculated result of the long-row blocks on each GPU. On the other hand, its calculated results of SpMV do not require further data transmission.

Figure 4(d) illustrates the execution process of SpMV using redundant-computing optimization. The short-row block assigned to the second GPU contains fewer non-zeros and matrix rows compared with Figure 4(c). As a result, both the kernel execution time and transmission time are reduced. Each GPU launches the third CUDA stream (Stream 3) to execute the redundant-computing SpMV kernel, allowing it to overlap with the data transmission process of the long-row blocks, D2D$\{\boldsymbol{y}_1^{s2}\}$ or D2D$\{\boldsymbol{y}_2^{s2}\}$. Therefore, the redundant-computing optimization enables hiding a portion of the kernel execution while reducing the transmission overhead.

4 Experimental Evaluation

4.1 Setup

Table 1 presents the hardware and software configurations of two platforms used in experimental evaluation. The first platform is equipped with two GeForce RTX 3090 interconnected via NVLink, and the second platform is equipped with four Tesla V100-SXM2 interconnected pairwise via NVLink.

Table 2 presents the tested sparse matrix set. It includes 24 large-scale sparse matrices sourced from the SuiteSparse Matrix Collection [15]. The number of rows, NNZ, average NNZ per row (Ave.), as well as the minimum (Min.) and maximum (Max.) NNZ per row, are presented for each sparse matrix. It can be observed that all matrices in the set contain non-zeros over 20 million.

We evaluate the performance of the proposed algorithms on the first platform with two GPUs (referred to as P1-2GPU) and on the second platform with two GPUs and four GPUs (referred to as P2-2GPU and P2-4GPU, respectively). In the next section, we first discuss the parameter selection for the long-row-aware partitioning and redundant-computing optimization. Subsequently, the

Table 2. Tested Sparse Matrix Set

ID	Matrix	Rows	NNZ	Ave.	Min.	Max.
1	kmer_U1a	67,716,231	138,778,562	2	1	35
2	kmer_A2a	170,728,175	360,585,172	2	1	40
3	kmer_V2a	55,042,369	117,217,600	2	1	39
4	kmer_P1a	139,353,211	297,829,984	2	1	40
5	kmer_V1r	214,005,017	465,410,904	2	1	8
6	webbase-2001	118,142,155	1,019,903,190	9	0	3,841
7	FullChip	2,987,012	26,621,983	9	1	2,312,481
8	ljournal-2008	5,363,260	79,023,142	15	0	2,469
9	rgg_n_2_23_s0	8,388,608	127,002,786	15	0	40
10	rgg_n_2_24_s0	16,777,216	265,114,400	16	0	40
11	uk-2002	18,520,486	298,113,762	16	0	2,450
12	com-LiveJournal	3,997,962	69,362,378	17	1	14,815
13	uk-2005	39,459,925	936,364,282	24	0	5,213
14	GAP-twitter	61,578,415	1,468,364,884	24	0	2,997,469
15	indochina-2004	7,414,866	194,109,311	26	0	6,985
16	nlpkkt160	8,345,600	225,422,112	28	5	28
17	nlpkkt200	16,240,000	440,225,632	28	5	28
18	nlpkkt240	27,993,600	760,648,352	28	5	28
19	it-2004	41,291,594	1,150,725,436	28	0	9,964
20	arabic-2005	22,744,080	639,999,458	28	0	9,905
21	stokes	11,449,533	349,321,980	31	1	720
22	twitter7	41,652,230	1,468,365,182	35	0	2,997,469
23	GAP-web	50,636,151	1,930,292,948	38	0	12,869
24	sk-2005	50,636,154	1,949,412,601	38	0	12,870

performance of the recently proposed multi-GPU SpMV algorithm, MSREP [11], is compared with the proposed algorithm. Finally, we compare the performance of the CG solver using different SpMV algorithms. We use "NZ" to represent the non-zeros-based partitioning, "2NZ" to represent the two-level non-zeros-based partitioning, "LRA" to represent the long-row-aware partitioning, and "LRA+RC" to represent long-row-aware partitioning with redundant-computing optimization. All floating-point numbers involved in SpMV and CG are stored and computed using the double-precision format. Besides, we implement the aforementioned SpMV algorithms based on the CSR-Vector [4–6] implementation from the open-source library CUSP[2] and complete other operators in CG solver by calling the APIs provided in cuSPARSE.

4.2 Experimental Results

4.2.1 Parameter Selection. In this section, we evaluate the impact of two parameters, d_l (the proportion of rows in the long-row block) and d_c (the proportion of rows in the redundant-computing block), on the performance of SpMV. The objective of redundant-computing optimization is to reduce the transmission overhead. However, too large d_c contradicts the intention of accelerating computation using multiple GPUs. Considering the extreme case where $d_c = 1$, i.e., all GPUs need to perform SpMV for the entire sparse matrix. The time of SpMV on multiple GPUs is equivalent

[2] https://github.com/cusplibrary/cusplibrary

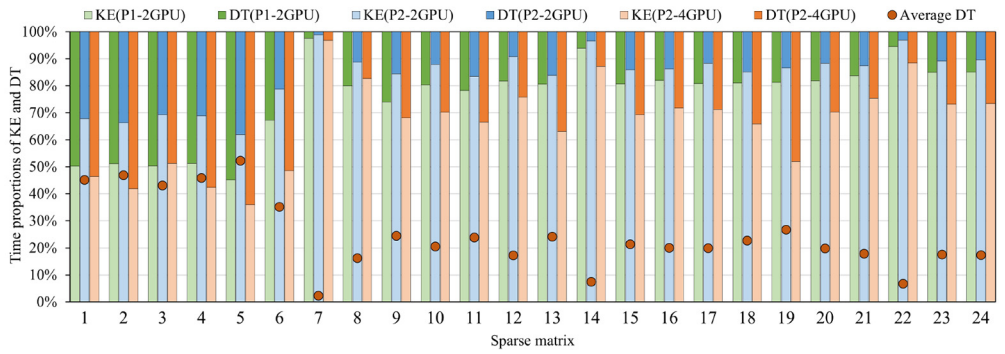


Fig. 6. Time proportions of kernel execution (KE) and data transmission (DT) in SpMV using the non-zeros-based partitioning on different platforms. The "average DT" represents the average transmission time proportion for each matrix across the three platforms.

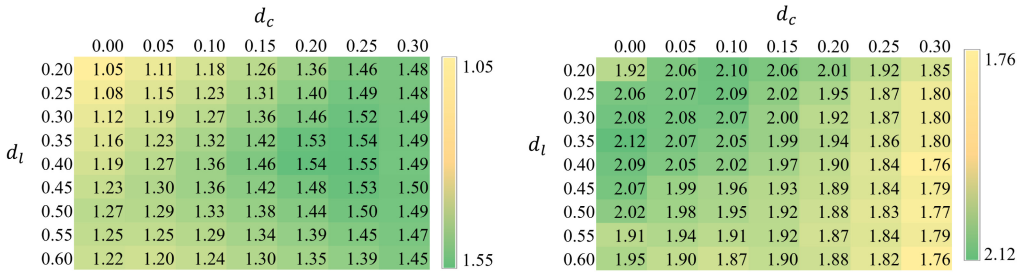
to that on a single GPU, and using multiple GPUs becomes meaningless. Therefore, we set its variation range from 0 to 0.3 with a step size of 0.05. This allows for exploring different levels of redundant computation. Additionally, a long-row block usually has fewer rows than a short-row block, so we set the variation range of d_l from 0.2 to 0.6 with a step size of 0.05.

Figure 6 illustrates the time proportions of **kernel execution (KE)** and **data transmission (DT)** in SpMV using the non-zeros-based partitioning. It can be observed that the average transmission proportion for the first five matrices is above 40%, while the average transmission proportion for the remaining matrices is below 40%. The kernel execution time is closely related to the number of nonzeros in the sparse matrix, while the number of rows directly influences the data transmission overhead. It can be observed from Table 2 that the first five matrices have a smaller average NNZ per row and do not contain long rows. Consequently, these matrices take lower kernel execution overhead and higher data transmission overhead compared with the other matrices. Our subsequent experimental results reveal that the performance of the first five matrices and the remaining matrices varies differently with changes in d_c and d_l. Therefore, we present the experimental results separately for these two subsets of matrices in this subsection to analyze their performance changes.

Figures 7–9 illustrate the SpMV performance comparison under different parameter settings on three experimental platforms. We can make the following observations from the figures:

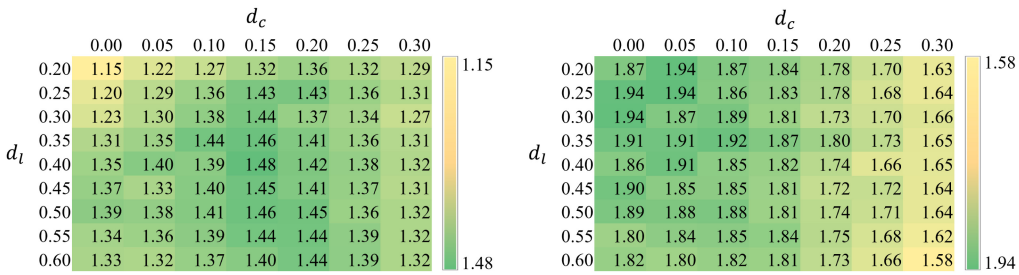
(1) Across three platforms, when d_c equals 0, i.e., the redundant-computing optimization is not used, the overall performance of the long-row-aware partitioning exhibits an initially increasing and then decreasing trend as d_l increases. As presented in Table 2, the first five sparse matrices have a relatively regular data distribution, with small variations in the NNZ per row. Therefore, the best overall performance is achieved with $d_l = 0.50$ for all three platforms. However, for the remaining matrices with irregular data distributions and containing a few long rows, the optimal d_l for the best overall performance is less than 0.50 for all three platforms. Specifically, on P1-2GPU, the long-row-aware partitioning achieves the highest average speedup of 2.12× at $d_l = 0.35$. On P2-2GPU, the highest average speedup of 1.94× is obtained at $d_l = 0.25$ or $d_l = 0.30$. On P2-4GPU, the highest average speedup of 2.86× is achieved at $d_l = 0.35$.

(2) When d_c is greater than 0, the performance variations differ for the first five matrices and the remaining matrices. For the first five matrices, the parameter d_c that yields the highest



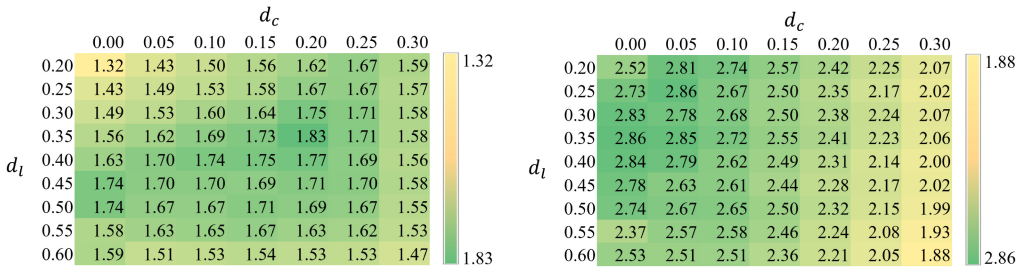
(a) The first five matrices (b) The remaining matrices

Fig. 7. SpMV performance comparison under different parameter settings (P1-2GPU). Each value represents the average speedup of SpMV performance on the multi-GPU platform to that on a single GPU across all tested matrices.



(a) The first five matrices (b) The remaining matrices

Fig. 8. SpMV performance comparison under different parameter settings (P2-2GPU).



(a) The first five matrices (b) The remaining matrices

Fig. 9. SpMV performance comparison under different parameter settings (P2-4GPU).

average speedup is typically larger, specifically 0.25, 0.15, and 0.20 for three platforms, respectively. For the remaining matrices, the optimal d_c is generally smaller, specifically 0, 0 or 0.05, and 0 or 0.05 for three platforms, respectively. Figure 10 illustrates the time variation of different parts in SpMV with increasing d_c. When d_l is appropriately chosen, the total runtime is determined by the kernel execution of the first and second sub-blocks on each GPU, as well as the transmission overhead for the second sub-block (❶). For the first five matrices, increasing d_c, i.e., increasing the proportion of redundant computation, the

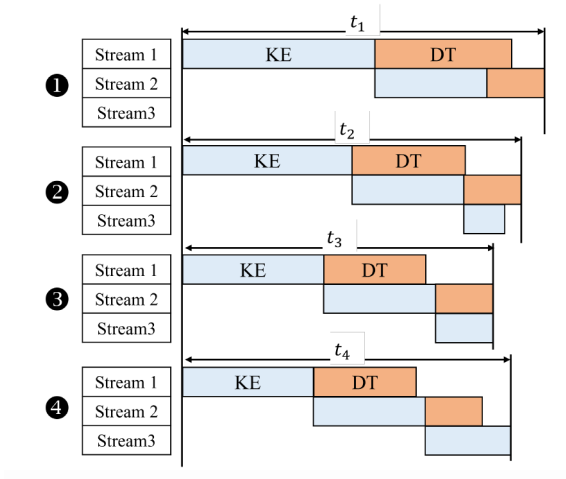


Fig. 10. Runtime change of different parts of SpMV with increasing d_c.

Table 3. The Rule for Setting Parameters

Platform	$aveNNZ$<8			$aveNNZ \geq 8$		
	LRA	LRA+RC		LRA	LRA+RC	
	d_l	d_l	d_c	d_l	d_l	d_c
P1-2GPU	0.50	0.40	0.25	0.35	0.35	0.00
P2-2GPU	0.50	0.40	0.15	0.30	0.25	0.05
P2-4GPU	0.50	0.35	0.20	0.35	0.25	0.05

kernel execution for the redundant-computing block is hidden behind the transmission process for the second sub-block, resulting in a reduction of the kernel execution time for the first sub-block (❷). As a result, the overall runtime decreases ($t_1 > t_2$). Continuing to increase d_c, the kernel execution time for the redundant-computing block becomes comparable to the transmission time of the second sub-block, leading to the minimum overall runtime (❸), i.e., $t_1 > t_2 > t_3$. However, increasing d_c beyond this point causes the reduced kernel execution time for the first sub-block to be smaller than the increased kernel execution time for the redundant-computing block, resulting in a deterioration of performance (❹), i.e., $t_3 < t_4$. As for the remaining matrices, their transmission overhead is relatively low. Increasing d_c directly transitions the SpMV execution from ❶ to ❸ or ❹, so their optimal d_c tends to be smaller.

Based on the above experimental results and observations, we conclude a parameter setting rule based on the average NNZ per row, as presented in Table 3. Typically, sparse matrices with smaller average NNZ per row have larger dimensions and relatively fewer non-zeros, resulting in lower kernel execution overhead and higher data transmission overhead in multi-GPU computations. Therefore, it is more suitable to set a relatively larger d_c.

We use PLUB to denote the **P**erformance **L**oss of the parameter setting **U**nder the **B**est **(PLUB)** parameter setting. The average PLUB across all tested matrices is calculated as Equation (1):

$$Average\ PLUB = \sum_{i=1}^{n} \frac{t_p^i - t_s^i}{t_s^i}, \tag{1}$$

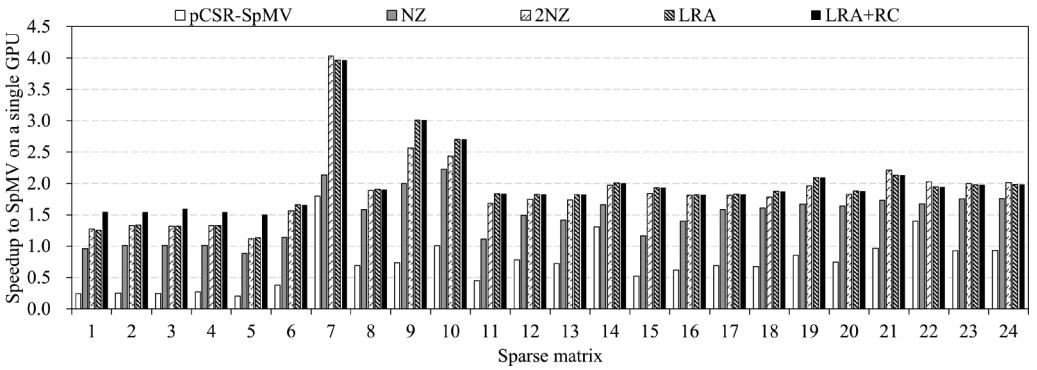


Fig. 11. Performance comparison of different SpMV algorithms (P1-2GPU). NZ, 2NZ, LRA, and LRA+RC, respectively, denote the SpMV using non-zeros-based partitioning, two-level non-zeros-based partitioning, long-row-aware partitioning, and long-row-aware partitioning with redundant-computing optimization.

where n is the number of tested sparse matrices, t_p^i is the runtime of SpMV using the experimental parameter setting for the ith sparse matrix, and t_s^i is the shortest runtime of SpMV across all parameter variations. We calculate that the average PLUB for the LRA algorithm on P1-2GPU, P2-2GPU, and P2-4GPU is approximately 1%, 2%, and 2%, respectively. For the LRA+RC algorithm, the PLUB on all three platforms is around 3%.

From the above discussion, it can be observed that the settings of d_l and d_c are not only related to the sparse matrices but also to the used platform. Our experimental rule for parameter setting results in a relatively small performance loss on the tested dataset and platforms. For new datasets or hardware platforms, it is possible to identify the optimal parameter configuration by using different parameter configurations in the initial few iterations of the iterative algorithm and then using the best configuration for subsequent iterations.

4.2.2 Performance Comparison. The SpMV performance on a single GPU serves as the baseline. We consider pCSR-SpMV (a CSR-based SpMV algorithm proposed in MSREP [11]) and non-zeros-based matrix partitioning as two existing algorithms and compare them with the algorithms proposed in this article. The pCSR-SpMV divides a sparse matrix into sub-matrix blocks with an equal NNZ, at the cost of breaking the integrity of partial matrix rows. Each GPU needs to transmit its calculated results back to the CPU to reduce and calculate the final result. Therefore, the time of pCSR-SpMV presented in this article includes both the kernel execution time and the time to transmit the results back to the CPU. Additionally, the non-zeros-based matrix partitioning is similar to the work in Reference [30] but with a different implementation based on the more general CSR instead of HYB.

Figures 11–13, respectively, illustrate the performance speedups of different SpMV algorithms against SpMV on a single GPU on three different platforms, with the sparse matrix indices on the x-axis corresponding to the matrix **indices (ID)** in Table 2. Figure 14 summarizes the average speedup.

The following observations can be made:

(1) Across three platforms, the proposed methods in this article have achieved varying degrees of performance improvement compared with the existing methods.
(2) For the first five sparse matrices, the performance improvement from the non-zeros-based partitioning compared with SpMV on a single GPU is relatively small, with the speedup close

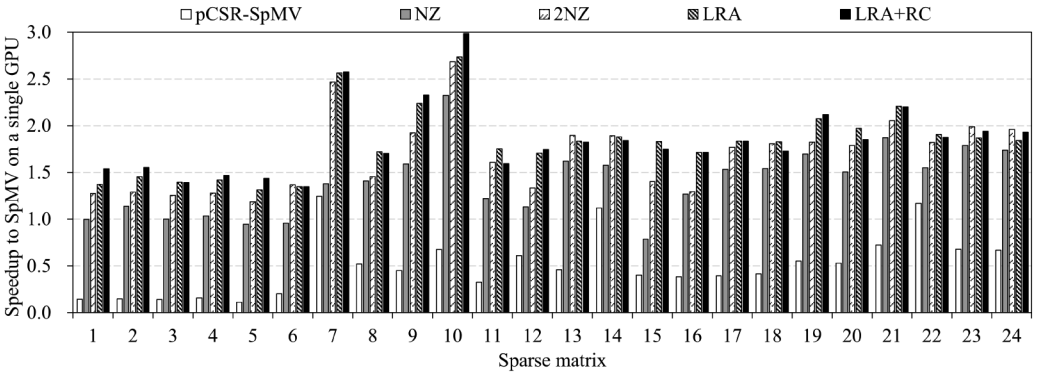


Fig. 12. Performance comparison of different SpMV algorithms (P2-2GPU).

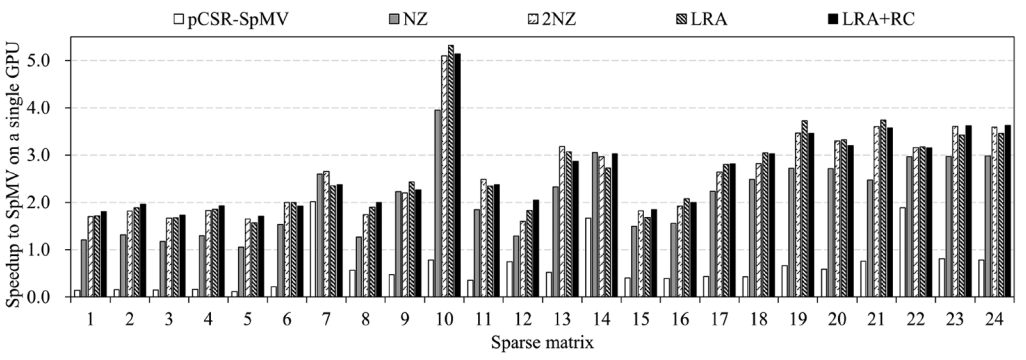


Fig. 13. Performance comparison of different SpMV algorithms (P2-4GPU).

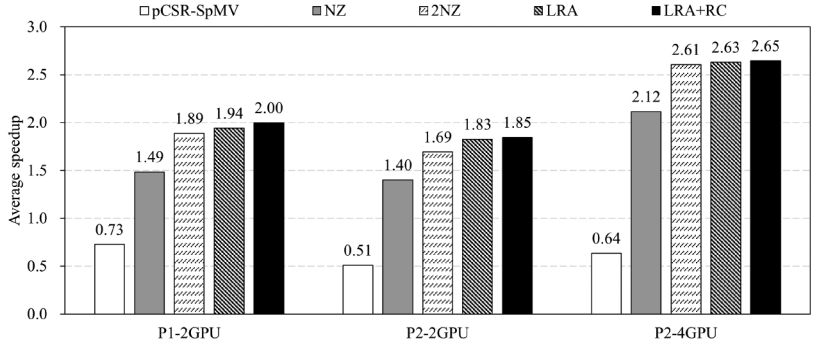


Fig. 14. Average performance comparison of different SpMV algorithms.

to 1. From Table 2 and Figure 6, it can be observed that these five matrices have a relatively small average NNZ per row. The transmission time in the SpMV using non-zeros-based partitioning accounts for a significant proportion, offsetting the performance gain from multi-GPU computation. However, our proposed optimization methods achieve higher speedup on these matrices. Specifically, for these five matrices, the non-zeros-based partitioning

achieves average speedups of 0.98×, 1.02×, and 1.21× on P1-2GPU, P2-2GPU, and P2-4GPU, respectively, while our proposed long-row-aware partitioning with redundant-computing optimization achieves average speedups of 1.55×, 1.48×, and 1.83× across three platforms.

(3) For matrix *FullChip* with index of 7, the naive non-zeros-based partitioning achieves a speedup over 2× on P1-2GPU. From Table 2, it can be observed that the longest row of the matrix contains over two million non-zeros, while the average NNZ per row is only 9. In the CSR-Vector algorithm, a group of cooperative threads is allocated for each matrix row, and the number of threads in the group depends on the average NNZ per row in the sparse matrix. Therefore, on a single GPU, the long rows of this matrix severely limit the overall performance of SpMV. When using the non-zeros-based matrix partitioning, the sub-block containing long rows has a higher average NNZ per row, resulting in an increased number of threads for handling long rows and thus a significant performance improvement. When using the two-level non-zeros-based or long-row-aware partitioning, the CSR-Vector algorithm selects the optimal thread configuration for the long-row block, leading to further significant performance improvements. Compared with the two-level non-zeros-based partitioning, the long-row-aware partitioning and redundant-computing optimization do not provide additional performance gains. The reason is that the kernel execution of the long-row block is the performance bottleneck, and the computation and transmission of the short-row block are hidden in the two-level non-zeros-based partitioning. Besides, on P2-4GPU, the CSR-Vector algorithm selects the optimal thread configuration for the long-row block when using non-zeros-based partitioning, so further performance improvement is not achieved by using two-level non-zeros-based partitioning and other optimization methods.

(4) Matrices *rgg_n_23_s0* and *rgg_n_24_s0* with indexes 9 and 10 are both regular diagonal matrices, and the thread configurations for these matrices remain unchanged after partitioning. However, their speedups on two GPUs exceed 2×. Analysis using NVIDIA Nsight Compute tool reveals that compared with the kernel execution on a single GPU, each kernel on P1-2GPU not only has less computation and memory access transactions but also achieves a higher L2 cache hit rate (increased from 19.98% to 30.82%), resulting in significant performance improvement.

(5) For most matrices, the performance of pCSR-SpMV on multiple GPUs is inferior to its performance on a single GPU. This is because it requires transmitting the calculated results from GPUs back to the CPU for result merging. Although it achieves significant performance improvement in the kernel execution part by using strict load-balanced partitioning, the data transmission overhead introduced in the result merging stage offsets the performance gains obtained in the kernel stage. Considering that, in GPU-based iterative applications, other vector operations besides SpMV are also performed on the GPU, the merged results obtained on the CPU side need to be segmented and transmitted back to the corresponding GPUs, introducing more overhead. Therefore, pCSR-SpMV is not suitable for iterative scenarios, and we do not compare it with our methods in the next section of application evaluation.

Overall, compared with the SpMV on a single GPU, utilizing multiple GPUs with non-zeros-based partitioning brings limited performance gains due to the impact of transmission overhead. On P1-2GPU, P2-2GPU, and P2-4GPU, the achieved average speedups are 1.49×, 1.40×, and 2.12×, respectively. Our proposed two-level non-zeros-based partitioning achieves significant performance improvements by hiding the transmission overhead of the first sub-block. The average speedups on the three platforms are 1.89×, 1.69×, and 2.61×, respectively. The

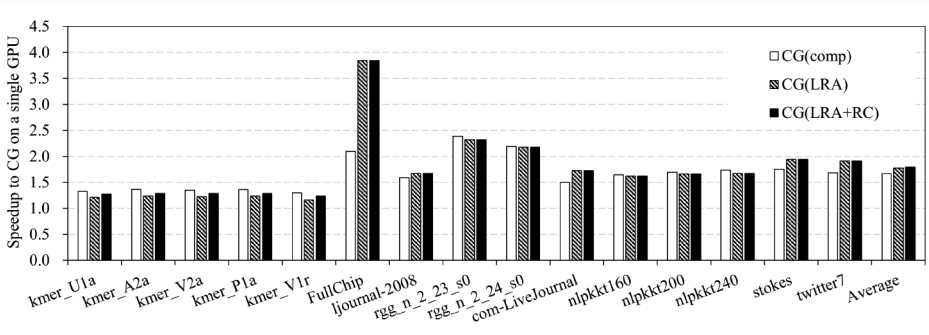


Fig. 15. Performance comparison of different CG implementations (P1-2GPU).

long-row-aware partitioning further improves overall performance by adjusting the execution order of long-row blocks and short-row blocks. It achieves average speedups of 1.94×, 1.83×, and 2.63×. The redundant-computing optimization uses inexpensive computation in exchange for costly transmission, particularly benefiting matrices with a higher proportion of transmission overhead. For the first five sparse matrices, using redundant computation on top of long-row-aware partitioning increases the average speedups from 1.27×, 1.39×, and 1.74× to 1.55×, 1.48×, and 1.83×, respectively. Across all tested matrices, the average speedups are 2.00×, 1.85×, and 2.65×.

Comparing the average speedups between P2-2GPU and P2-4GPU, it can be observed that the parallel efficiency on four GPUs is lower than that on two GPUs. There are three main reasons. First, as the number of GPUs increases, the transmission process becomes more complex, leading to higher transmission overhead introduced. From Figure 6, it can be observed that, in most matrices, the proportion of transmission time on P2-4GPU is higher than that on P2-2GPU. The experimental results show that the average transmission time on P2-4GPU is approximately 1.67× that on P2-2GPU. Second, unlike dense matrix computations, the irregular non-zeros distribution in sparse matrices causes the execution time of the SpMV kernel to not decrease proportionally with the increasing number of GPUs. Taking the non-zeros-based partitioning as an example, we calculate the average parallel efficiency considering only the kernel execution time on P2-2GPU and P2-4GPU, resulting in 82% and 75%, respectively. Third, using multi-threading with OpenMP to manage multiple GPUs brings additional overhead, such as creating, synchronizing, and destroying threads. The experimental results reveal that the multi-threading overhead on P2-4GPU is approximately twice that on P2-2GPU.

4.2.3 Application Evaluation. In this section, we evaluate the performance benefits of the proposed SpMV optimization methods in the CG iterative algorithm. The maximum iteration is set to 1,000, and the error tolerance is set to 10^{-5}. The performance of the CG solver on a single GPU serves as the baseline. Following the work in Reference [30], which uses non-zeros-based partitioning and HYB compression, we implement a similar CG algorithm based on CSR compression as a comparative algorithm. In this algorithm, data transmission for the multiplication vector of SpMV is performed first, followed by SpMV kernel execution. We use CG(comp) to represent the algorithm, and CG(LRA) and CG(LRA+RC) represent the CG algorithms using the proposed LRA and LRA+RC optimization, respectively. Figures 15–17 present the speedups of the three CG algorithms to the CG algorithm on a single GPU for different platforms. The rightmost group of bars (Average) represents the average speedup.

We can observe that both of the proposed optimization methods in this article achieve higher average speedups on three platforms. The first five sparse matrices have more rows, which means

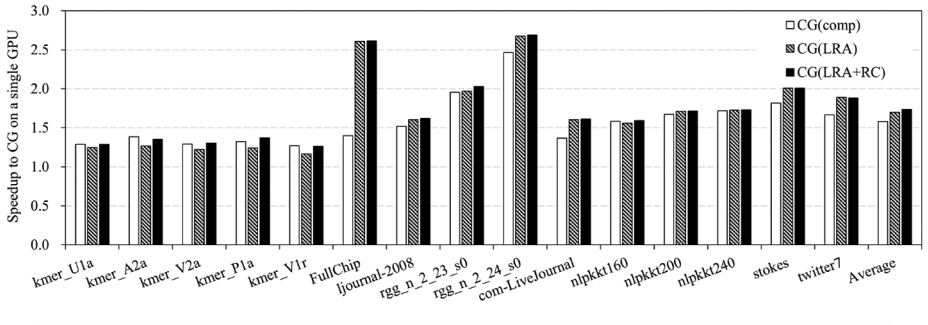


Fig. 16. Performance comparison of different CG implementations (P2-2GPU).

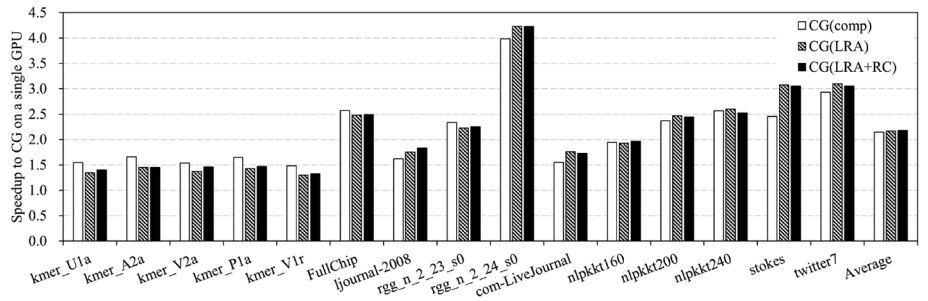


Fig. 17. Performance comparison of different CG implementations (P2-4GPU).

a higher overhead of vector calculations. On a single GPU, the vector calculations account for approximately 37% to 44% of the overall CG algorithm for these five matrices. Compared with CG(comp), CG(LRA) and CG(LRA+RC) require redundant vector calculations, resulting in lower speedups for these five matrices. However, on most other sparse matrices, CG(LRA) and CG(LRA+RC) outperform CG(comp). CG(comp) requires data transmission for the multiplication vector before each kernel execution. It results in limited overall performance gains. Although it is possible to overlap the vector transmission and kernel execution in CG(comp) by further partitioning the sparse matrix along the column dimension [33], it often requires changing the data arrangement of the sparse matrix, which incurs significant preprocessing overhead and is not desirable for large-scale sparse matrices. Our proposed optimization algorithms, although requiring redundant computations for other vector calculations in the iterative algorithm, provide more opportunities to hide and reduce transmission overhead by executing kernels before transmission. Moreover, they only require scanning the row pointer array of CSR and take minimal preprocessing overhead.

4.2.4 Preprocessing Overhead. The preprocessing of the proposed optimization method is presented in Figure 5(b) and includes finding the locations of long-row blocks, short-row blocks, and redundant-computing blocks based on the distribution of non-zeros in sparse matrices. We analyzed the preprocessing overhead on three different hardware configurations by calculating the minimum, arithmetic mean, geometric mean, and maximum of the ratio of preprocessing overhead to the execution time of one single-GPU-based SpMV kernel for each test matrix. The results are summarized in Table 4. It can be observed that the preprocessing overhead introduced

Table 4. Preprocessing Overhead, Normalized by the Execution Time of One Single-GPU-based SpMV Kernel

Metric	P1-2GPU	P2-2GPU	P2-4GPU
Minimum	0.01	0.01	0.01
Arithmetic mean	0.32	0.18	0.20
Geometric mean	0.21	0.11	0.12
Maximum	0.83	0.57	0.87

by our optimization method is less than one SpMV kernel execution for all tested matrices. Compared to the thousands of SpMV invocations in iterative algorithms, the preprocessing overhead is negligible.

5 Conclusion

Due to the limited memory capacity and computational capability of a single GPU, efficient implementation of large-scale SpMV on multi-GPU systems is crucial for the fast solving of large-scale problems. To reduce the data transmission overhead introduced by multi-GPU computation, this article proposes a two-level non-zeros-based matrix partitioning to hide the data transmission process between multiple GPUs. Considering the irregular distribution of non-zeros in sparse matrices, a long-row-aware partitioning strategy is further proposed to hide more transmissions. Furthermore, inspired by the fact that SpMV on a single GPU does not require data transmission, we perform redundant SpMV computing for the most cost-efficient sub-block to reduce the total transmission amounts. Finally, performance evaluations on platforms equipped with two RTX 3090, two Tesla V100-SXM2, and four Tesla V100-SXM2, respectively, demonstrate that the proposed methods achieve average speedups of 2.00×, 1.85×, and 2.65× to the SpMV on a single GPU. These performance improvements also result in a faster CG solver.

Acknowledgments

We thank all reviewers for their insightful comments.

References

[1] Ahmad Abdelfattah, Hatem Ltaief, and David E. Keyes. 2015. High performance multi-GPU SpMV for multi-component PDE-Based applications. In *21st International Conference on Parallel and Distributed Computing: Parallel Processing (Euro-Par'15) (Lecture Notes in Computer Science,* Vol. 9233*)*, Jesper Larsson Träff, Sascha Hunold, and Francesco Versaci (Eds.). Springer, 601–612. DOI: https://doi.org/10.1007/978-3-662-48096-0_46

[2] Hartwig Anzt, Terry Cojean, Chen Yen-Chen, Jack J. Dongarra, Goran Flegar, Pratik Nayak, Stanimire Tomov, Yuhsiang M. Tsai, and Weichung Wang. 2020. Load-balancing sparse matrix vector product kernels on GPUs. *ACM Trans. Parallel Comput.* 7, 1 (2020), 2:1–2:26. DOI: https://doi.org/10.1145/3380930

[3] Arash Ashari, Naser Sedaghati, John Eisenlohr, Srinivasan Parthasarathy, and P. Sadayappan. 2014. Fast sparse matrix-vector multiplication on GPUs for graph applications. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*, Trish Damkroger and Jack J. Dongarra (Eds.). IEEE Computer Society, 781–792. DOI: https://doi.org/10.1109/SC.2014.69

[4] Muthu Manikandan Baskaran and Rajesh Bordawekar. 2009. Optimizing sparse matrix-vector multiplication on GPUs. *IBM Research Report RC24704* W0812–047 (2009).

[5] Nathan Bell and Michael Garland. 2008. *Efficient Sparse Matrix-vector Multiplication on CUDA*. NVIDIA Technical Report NVR-2008-004. NVIDIA Corporation.

[6] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *ACM/IEEE Conference on High Performance Computing (SC'09)*. ACM, 1–11. DOI: https://doi.org/10.1145/1654059.1654078

[7] Akrem Benatia, Weixing Ji, Yizhuo Wang, and Feng Shi. 2016. Sparse matrix format selection with multiclass SVM for SpMV on GPU. In *45th International Conference on Parallel Processing (ICPP'16)*. 496–505. DOI: https://doi.org/10.1109/ICPP.2016.64

[8] Akrem Benatia, Weixing Ji, Yizhuo Wang, and Feng Shi. 2018. BestSF: A sparse meta-format for optimizing SpMV on GPU. *ACM Trans. Archit. Code Optim.* 15, 3, Article 29 (Sept. 2018), 27 pages. DOI: https://doi.org/10.1145/3226228

[9] Daniele Buono, Fabrizio Petrini, Fabio Checconi, Xing Liu, Xinyu Que, Chris Long, and Tai-Ching Tuan. 2016. Optimizing sparse matrix-vector multiplication for large-scale data analytics. In *International Conference on Supercomputing (ICS'16)*, Ozcan Ozturk, Kemal Ebcioglu, Mahmut T. Kandemir, and Onur Mutlu (Eds.). ACM, 37:1–37:12. DOI: https://doi.org/10.1145/2925426.2926278

[10] Ali Cevahir, Akira Nukada, and Satoshi Matsuoka. 2009. Fast conjugate gradients with multiple GPUs. In *9th International Conference on Computational Science (ICCS'09) (Lecture Notes in Computer Science,* Vol. 5544*)*, Gabrielle Allen, Jaroslaw Nabrzyski, Edward Seidel, G. Dick van Albada, Jack J. Dongarra, and Peter M. A. Sloot (Eds.). Springer, 893–903. DOI: https://doi.org/10.1007/978-3-642-01970-8_90

[11] Jieyang Chen, Chenhao Xie, Jesun Sahariar Firoz, Jiajia Li, Shuaiwen Leon Song, Kevin J. Barker, Mark Raugas, and Ang Li. 2022. MSREP: A fast yet light sparse matrix framework for multi-GPU systems. *CoRR* abs/2209.07552 (2022).

[12] Yuedan Chen, Guoqing Xiao, Fan Wu, Zhuo Tang, and Keqin Li. 2020. tpSpMV: A two-phase large-scale sparse matrix-vector multiplication kernel for manycore architectures. *Inf. Sci.* 523 (2020), 279–295. DOI: https://doi.org/10.1016/J.INS.2020.03.020

[13] Yuedan Chen, Guoqing Xiao, Zheng Xiao, and Wangdong Yang. 2019. hpSpMV: A heterogeneous parallel computing scheme for SpMV on the Sunway TaihuLight supercomputer. In *21st IEEE International Conference on High Performance Computing and Communications, the 17th IEEE International Conference on Smart City, the 5th IEEE International Conference on Data Science and Systems (HPCC/SmartCity/DSS'19)*, Zheng Xiao, Laurence T. Yang, Pavan Balaji, Tao Li, Keqin Li, and Albert Y. Zomaya (Eds.). IEEE, 989–995. DOI: https://doi.org/10.1109/HPCC/SMARTCITY/DSS.2019.00142

[14] M. Daga and J. L. Greathouse. 2015. Structural agnostic SpMV: Adapting CSR-adaptive for irregular matrices. In *IEEE 22nd International Conference on High Performance Computing (HiPC'15)*. 64–74. DOI: https://doi.org/10.1109/HiPC.2015.55

[15] Timothy A. Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.* 38, 1 (2011), 1:1–1:25. DOI: https://doi.org/10.1145/2049662.2049663

[16] Yangdong Deng, Bo D. Wang, and Shuai Mu. 2009. Taming irregular EDA applications on GPUs. In *International Conference on Computer-Aided Design (ICCAD'09)*, Jaijeet S. Roychowdhury (Ed.). ACM, 539–546. DOI: https://doi.org/10.1145/1687399.1687501

[17] Zhen Du, Jiajia Li, Yinshan Wang, Xueqi Li, Guangming Tan, and Ninghui Sun. 2022. AlphaSparse: Generating high performance SpMV codes directly from sparse matrices. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'22)*. IEEE, 1–15. DOI: https://doi.org/10.1109/SC41404.2022.00071

[18] Ernesto Dufrechou, Pablo Ezzatti, and Enrique S. Quintana-Ortí. 2021. Selecting optimal SpMV realizations for GPUs via machine learning. *Int. J. High Perf. Comput. Applic.* 35, 3 (2021). DOI: https://doi.org/10.1177/1094342021990738

[19] Goran Flegar and Hartwig Anzt. 2017. Overcoming load imbalance for irregular sparse matrices. In *7th Workshop on Irregular Applications: Architectures and Algorithms (IA3@SC'17)*. ACM, 2:1–2:8. DOI: https://doi.org/10.1145/3149704.3149767

[20] Jianhua Gao, Weixing Ji, Jie Liu, Senhao Shao, Yizhuo Wang, and Feng Shi. 2021. AMF-CSR: Adaptive multi-row folding of CSR for SpMV on GPU. In *27th IEEE International Conference on Parallel and Distributed Systems (ICPADS'21)*. IEEE, 418–425. DOI: https://doi.org/10.1109/ICPADS53394.2021.00058

[21] Jianhua Gao, Weixing Ji, Jie Liu, Yizhuo Wang, and Feng Shi. 2024. Revisiting thread configuration of SpMV kernels on GPU: A machine learning based approach. *J. Parallel Distrib. Comput.* 185 (2024), 104799. DOI: https://doi.org/10.1016/j.jpdc.2023.104799

[22] Jianhua Gao, Bingjie Liu, Weixing Ji, and Hua Huang. 2024. A systematic literature survey of sparse matrix-vector multiplication. *arXiv preprint arXiv:2404.06047* (2024).

[23] Jiaquan Gao, Yu Wang, and Jun Wang. 2017. A novel multi-graphics processing unit parallel optimization framework for the sparse matrix-vector multiplication. *Concurr. Comput. Pract. Exp.* 29, 5 (2017). DOI: https://doi.org/10.1002/CPE.3936

[24] Jiaquan Gao, Yu Wang, Jun Wang, and Ronghua Liang. 2016. Adaptive optimization modeling of preconditioned conjugate gradient on multi-GPUs. *ACM Trans. Parallel Comput.* 3, 3 (2016), 16:1–16:33. DOI: https://doi.org/10.1145/2990849

[25] Jiaquan Gao, Yuanshen Zhou, Guixia He, and Yifei Xia. 2017. A multi-GPU parallel optimization model for the preconditioned conjugate gradient algorithm. *Parallel Comput.* 63 (2017), 1–16. DOI: https://doi.org/10.1016/J.PARCO.2017.04.003

[26] Joseph L. Greathouse and Mayank Daga. 2014. Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*. IEEE, 769–780.

[27] Ping Guo, Liqiang Wang, and Po Chen. 2014. A performance modeling and optimization analysis tool for sparse matrix-vector multiplication on GPUs. *IEEE Trans. Parallel Distrib. Syst.* 25, 5 (2014), 1112–1123. DOI: https://doi.org/10.1109/TPDS.2013.123

[28] Ping Guo and Changjiang Zhang. 2016. Performance optimization for SpMV on Multi-GPU systems using threads and multiple streams. In *International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PAD Workshops'16)*. IEEE Computer Society, 67–72. DOI: https://doi.org/10.1109/SBAC-PADW.2016.20

[29] Kai He, Sheldon X.-D. Tan, Hengyang Zhao, Xuexin Liu, Hai Wang, and Guoyong Shi. 2016. Parallel GMRES solver for fast analysis of large linear dynamic systems on GPU platforms. *Integration* 52 (2016), 10–22. DOI: https://doi.org/10.1016/j.vlsi.2015.07.005

[30] Marek Karwacki, Beata Bylina, and Jaroslaw Bylina. 2012. Multi-GPU implementation of the uniformization method for solving Markov models. In *Federated Conference on Computer Science and Information Systems (FedCSIS'12)*, Maria Ganzha, Leszek A. Maciaszek, and Marcin Paprzycki (Eds.). 533–537. https://fedcsis.org/proceedings/2012/pliks/377.pdf

[31] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.* 20, 1 (1998), 359–392. DOI: https://doi.org/10.1137/S1064827595287997

[32] Zbigniew Koza, Maciej Matyka, Sebastian Szkoda, and Łukasz Mirosław. 2014. Compressed multirow storage format for sparse matrices on graphics processing units. *SIAM J. Scient. Comput.* 36, 2 (2014), C219–C239. DOI: https://doi.org/10.1137/120900216

[33] Cheng Li, Min Tang, Ruofeng Tong, Ming Cai, Jieyi Zhao, and Dinesh Manocha. 2020. P-cloth: Interactive complex cloth simulation on multi-GPU systems using dynamic matrix assembly and pipelined implicit integrators. *ACM Trans. Graph.* 39, 6 (2020), 180:1–180:15. DOI: https://doi.org/10.1145/3414685.3417763

[34] Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. 2013. SMAT: An input adaptive auto-tuner for sparse matrix-vector multiplication. *ACM SIGPLAN Not.* 48, 6 (June 2013), 117–126. DOI: https://doi.org/10.1145/2499370.2462181

[35] K. Li, W. Yang, and K. Li. 2015. Performance analysis and optimization for SpMV on GPU using probabilistic modeling. *IEEE Trans. Parallel Distrib. Syst.* 26, 1 (2015), 196–205. DOI: https://doi.org/10.1109/TPDS.2014.2308221

[36] Shigang Li, Changjun Hu, Junchao Zhang, and Yunquan Zhang. 2015. Automatic tuning of sparse matrix-vector multiplication on multicore clusters. *Sci. China Inf. Sci.* 58, 9 (2015), 1–14. DOI: https://doi.org/10.1007/S11432-014-5254-X

[37] Shaozhong Lin and Zhiqiang Xie. 2017. A Jacobi_PCG solver for sparse linear systems on multi-GPU cluster. *J. Supercomput.* 73, 1 (2017), 433–454. DOI: https://doi.org/10.1007/S11227-016-1887-4

[38] Weifeng Liu and Brian Vinter. 2015. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *29th ACM on International Conference on Supercomputing (ICS'15)*, Laxmi N. Bhuyan, Fred Chong, and Vivek Sarkar (Eds.). ACM, 339–350. DOI: https://doi.org/10.1145/2751205.2751209

[39] Duane Merrill and Michael Garland. 2016. Merge-based parallel sparse matrix-vector multiplication. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'16)*, John West and Cherri M. Pancake (Eds.). IEEE Computer Society, 678–689. DOI: https://doi.org/10.1109/SC.2016.57

[40] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable parallel programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for? *Queue* 6, 2 (Mar. 2008), 40–53. DOI: https://doi.org/10.1145/1365490.1365500

[41] Yuyao Niu, Zhengyang Lu, Meichen Dong, Zhou Jin, Weifeng Liu, and Guangming Tan. 2021. TileSpMV: A tiled algorithm for sparse matrix-vector multiplication on GPUs. In *35th IEEE International Parallel and Distributed Processing Symposium (IPDPS'21)*. IEEE, 68–78. DOI: https://doi.org/10.1109/IPDPS49936.2021.00016

[42] István Reguly and Mike Giles. 2012. Efficient sparse matrix-vector multiplication on cache-based GPUs. In *Innovative Parallel Computing (InPar'12)*. 1–12. DOI: https://doi.org/10.1109/InPar.2012.6339602

[43] Dana Schaa and David R. Kaeli. 2009. Exploring the multiple-GPU design space. In *23rd IEEE International Symposium on Parallel and Distributed Processing (IPDPS'09)*. IEEE, 1–12. DOI: https://doi.org/10.1109/IPDPS.2009.5161068

[44] Yi Shan, Tianji Wu, Yu Wang, Bo Wang, Zilong Wang, Ningyi Xu, and Huazhong Yang. 2010. FPGA and GPU implementation of large scale SpMV. In *IEEE 8th Symposium on Application Specific Processors (SASP'10)*. IEEE Computer Society, 64–70. DOI: https://doi.org/10.1109/SASP.2010.5521144

[45] Mickeal Verschoor and Andrei C. Jalba. 2012. Analysis and performance estimation of the Conjugate Gradient method on multiple GPUs. *Parallel Comput.* 38, 10-11 (2012), 552–575. DOI: https://doi.org/10.1016/J.PARCO.2012.07.002

[46] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. 2014. yaSpMV: Yet another SpMV framework on GPUs. In *19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'14)*. ACM, New York, NY, USA, 107–118. DOI: https://doi.org/10.1145/2555243.2555255

[47] Bo Yang, Hui Liu, and Zhangxin Chen. 2016. Preconditioned GMRES solver on multiple-GPU architecture. *Comput. Math. Appl.* 72, 4 (2016), 1076–1095. DOI: https://doi.org/10.1016/J.CAMWA.2016.06.027

[48] Hiroki Yoshizawa and Daisuke Takahashi. 2012. Automatic tuning of sparse matrix-vector multiplication for CRS format on GPUs. In *15th IEEE International Conference on Computational Science and Engineering (CSE'12)*. IEEE Computer Society, 130–136. DOI: https://doi.org/10.1109/ICCSE.2012.28

[49] Yue Zhao, Jiajia Li, Chunhua Liao, and Xipeng Shen. 2018. Bridging the gap between deep learning and sparse matrix format selection. In *23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'18)*. Association for Computing Machinery, New York, NY, USA, 94–108. DOI: https://doi.org/10.1145/3178487.3178495

Received 29 January 2024; revised 4 June 2024; accepted 22 June 2024