FastLoad: Speeding Up Data Loading of Both Sparse Matrix and Vector for SpMV on GPUs

Jinyu Hu, Huizhang Luo, *Member, IEEE*, Hong Jiang, *Fellow, IEEE*, Guoqing Xiao, *Member, IEEE*, and Kenli Li, *Senior Member, IEEE*

***Abstract*—Sparse Matrix-Vector Multiplication (SpMV) on GPUs has gained significant attention because of SpMV's importance in modern applications and the increasing computing power of GPUs in the last decade. Previous studies have emphasized the importance of data loading for the overall performance of SpMV and demonstrated the efficacy of coalesced memory access in enhancing data loading efficiency. However, existing approaches fall far short of reaching the full potential of data loading on modern GPUs. In this paper, we propose an efficient algorithm called FastLoad, that speeds up the loading of both sparse matrices and input vectors of SpMV on modern GPUs. Leveraging coalesced memory access, FastLoad achieves high loading efficiency and load balance by sorting both the columns of the sparse matrix and elements of the input vector based on the number of non-zero elements while organizing non-zero elements in blocks to avoid thread divergence. FastLoad takes the Compressed Sparse Column (CSC) format as an implementation case to prove the concept and gain insights. We conduct a comprehensive comparison of FastLoad with the CSC-based SpMV, cuSPARSE, CSR5, and TileSpMV, using the full SuiteSparse Matrix Collection as workload. The experimental results on RTX 3090 Ti demonstrate that our method outperforms the others in most matrices, with geometric speedup means over CSC-based, cuSPARSE, CSR5, and TileSpMV being 2.12×, 2.98×, 2.88×, and 1.22×, respectively.**

***Index Terms*—Coalesced memory access, GPU, sparse matrix-vector multiplication.**

I. Introduction

THE Graphs Processing Unit (GPU) is a throughput-oriented architecture applicable across diverse application domains such as machine learning, high-performance computing (HPC), and computer graphics [1]. Its exceptional throughput highlights the significance of efficient memory data loading [2]. To achieve high throughput, GPUs are equipped with coalesced units that detect continuous memory access, which reduces the number of memory requests [3]. However, when accelerating irregular applications like sparse matrices, GPU performance is notably inferior to regular applications [4]. Regular applications benefit from nearly equal computation among threads, enabling continuous data loading. In contrast, irregular applications suffer from imbalanced computation across threads and encounter challenges in accessing data efficiently, leading to suboptimal performance [5]. As a result, researchers have extensively explored methods to enhance the speedup of irregular applications on GPUs in recent years.

Among the irregular applications, SpMV holds significant importance in various domains, such as deep learning [6], sparse iterative solvers [7], and graph processing problems [8], etc., but it often becomes a performance bottleneck. This bottleneck comes from the inherent characteristics of sparse matrices, e.g., sparsity and irregular distribution of non-zero elements. To address the sparsity issue, various basic storage formats have been devised to reduce storage requirements and enhance implementation efficiency. These basic formats include Compress Sparse Row (CSR), Compress Sparse Column (CSC), Coordinate (COO), Diagonal (DIA), and Ell pack (ELL). However, these basic storage formats alone are insufficient in dealing with the irregular distribution of non-zero elements which causes load imbalance and low data loading efficiency during SpMV operations. For example, in the case of the CSR format, conventional parallel strategies involving row splitting and non-zero splitting can lead to load imbalance among threads and poor data loading efficiency.

To address these challenges, numerous SpMV algorithms and sparse matrix formats have been proposed. The Merge-based SpMV method, proposed by Merrill et al. [9], CSR5 introduced by Liu et al. [10], and TileSpMV presented by Niu et al. [11], all demonstrate excellent load balance. VCSR, as proposed by Karimi et al. [3], has shown that coalesced memory access of the sparse matrix can enhance overall performance by improving data loading efficiency. However, the aforementioned designs share an inherent drawback: they cannot achieve coalesced memory access for the input vector.

In this paper, we propose a novel approach called FastLoad to improve the data loading efficiency by achieving coalesced memory access for both the sparse matrix and input vector of SpMV on GPUs. We use CSC as an implementation case of FastLoad to prove the concept and gain insights. By sorting the columns of the sparse matrix based on the number of non-zero

Received 23 December 2023; revised 25 September 2024; accepted 28 September 2024. Date of publication 9 October 2024; date of current version 21 October 2024. This work was supported in part by the National Key R&D Program of China under Grant 2023YFB3001705, in part by the National Natural Science Foundation of China under Grant 62321003, Grant 62172157, and Grant 62102141, and in part by the Programs of the Hunan Province under Grant 2024JJ2026 and Grant 2023GK2002. Recommended for acceptance by A. Sussman. *(Corresponding author: Huizhang Luo.)*

Jinyu Hu, Huizhang Luo, Guoqing Xiao, and Kenli Li are with the College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China (e-mail: hujinyu@hnu.edu.cn; luohuizhang@hnu.edu.cn; xiaoguoqing@hnu.edu.cn; lkl@hnu.edu.cn).

Hong Jiang is with the Computer Science and Engineering Department, University of Texas at Arlington, Arlington, TX 76019 USA (e-mail: hong.jiang@uta.edu).

Digital Object Identifier 10.1109/TPDS.2024.3477431

Group Name (Matrix Number)	FastLoad(μs)	cusparse(μs)	TileSpMV(μs)	CSR5(μs)	Basic CSC(μs)
HB(292)	6.6311	32.5455	9.0088	23.8919	9.4296
Sandia(192)	6.9679	36.0062	9.9633	24.8761	24.9389
Meszaros(166)	11.8361	44.3485	13.4281	33.6107	25.8784
DIMACS10(151)	86.7853	118.1814	146.7377	257.3278	130.0859
Lpnetlib(138)	6.8547	35.4989	9.0276	25.9212	9.8286
JGD_Homology(128)	9.6768	37.1191	11.7809	37.8256	10.3381
VDOL(91)	7.8624	39.8879	10.4464	28.2259	73.0954
Bai(78)	6.3013	31.7247	8.8241	23.4947	7.4589
Pajek(76)	7.4146	37.2196	9.2464	28.8736	13.2644
Gset(67)	7.8285	33.8816	9.7546	27.0006	15.5135

Fig. 1. The heat map of geometry means over different methods on different groups.

elements in each column and the corresponding input vector elements, FastLoad can keep coalesced memory access to both the sparse matrix and the input vector. To keep the load balanced among threads and warps, FastLoad splits the non-zero elements into different blocks based on the number of elements in the column.

Although the CSC format has the potential to achieve coalesced memory access for FastLoad, it lacks sufficient efficiency and optimization compared to other formats. In particular, when computing the output elements of the output vector in its addition operations, the CSC format performs poorly due to irregular accesses to intermediate multiplication results [12]. To overcome this drawback, we utilize segment sum and prefix sum techniques for each block, further enhancing the overall performance of the FastLoad algorithm.

This work makes the following contribution:

- We show that existing SpMV algorithms have exhibited a low efficiency in loading data due to insufficient use of the coalesced memory access.
- We present a novel solution to address the data loading problem of SpMV, called FastLoad, an efficient algorithm implemented based on the CSC format. FastLoad achieves coalesced memory access for both sparse matrices and input vectors, as well as a well-balanced load for SpMV operations through a two-step process: *Step 1:* Columns of the sparse matrix and the corresponding input vector elements are sorted based on the number of non-zero elements of each column in the sparse matrix. *Step 2:* The non-zero elements are split into blocks to prevent thread divergence, which can keep the load well balanced.
- In our experiments, we conducted a comprehensive comparison between the FastLoad algorithm, CSC-based, cuSPARSE (v11.1), CSR5 [10], and TileSpMV [11]. Fig. 1 displays the geometric mean of execution times for the different methods across 10 groups of matrices. In all 10 matrix groups, our proposed FastLoad kernel exhibited higher performance than other state-of-the-art methods. Overall, across the entire set of 2757 matrices, FastLoad achieved an average speedup of 2.12× over basic CSC SpMV, 2.98× over cuSPARSE, 2.88× over CSR5, and 1.22× over TileSpMV, respectively. These results demonstrate the significant performance improvements offered by our approach in comparison to existing methods.

Algorithm 1: A Pseudocode of Parallel CSC SpMV.

Input: $colPtr[]; rowIdx[]; value[]; inputX[]$
Output: $y[]$
1 **for** i *from* 0 *to* $sizeof(colPtr[]) - 2$ ***in parallel*** **do**
2 **for** $j \leftarrow colPtr[i]; j < colPtr[i+1]; j++$ **do**
3 $y[rowIdx[j]] += value[j] \times inputX[i];$
4 **end**
5 **end**

The rest of the paper is organized as followed: Section II presents relevant background information and related work. Section III presents the motivation for the FastLoad research. Section IV presents the FastLoad algorithm with its process and implementation details. Section V presents the results of the performance evaluation. Section VI concludes this work and discusses future work.

II. Background and Related Work

A. Sparse Matrix-Vector Multiplication

SpMV concerns the multiplication of a sparse matrix and a dense input vector. Sparse matrices of modern SpMV-based problems exhibit two distinct characteristics. First, they contain significantly fewer non-zero elements than the total number of elements of the matrix, as determined by the number of rows multiplied by the number of columns. Second, the distribution of these non-zero elements within the matrix is stochastic, adding to the complexity of the structure. Algorithm 1 provides a pseudocode of CSC-based SpMV. In this algorithm, a column of the sparse matrix is allocated to a warp (Line 1-5). The corresponding value to the input vector of warp is fixed. The thread accesses the value of the matrix through *value* array.

B. GPU and Coalesced Memory Accessing

GPUs are highly data-parallel many-core processors. A typical GPU architecture comprises multiple stream multiprocessors (SMs), and various memory hierarchy layers. Each SM has multiple CUDA cores and shared memory(L1 cache). The Memory hierarchy layers include L2 cache, texture memory, and global memory. From a programming perspective, Nvidia CUDA provides a programming model with three levels: thread, thread block, and grid. The programmer can define the number of threads in a thread block and the number of blocks in a grid. In CUDA, a warp consists of 32 threads, and all the threads within a warp must reside within the same SM. This characteristic enables efficient parallel operations within a warp. When we launch a kernel in GPU, the grid dimension and the block dimension can be defined by the user. Consequently, each block in a grid has a *blockId*. Each thread in a block has a *threadId*. Each thread in a grid also has a unique id called *globalId*. Each thread in a warp has a *laneId*.

Coalesced memory access is an optimization technique used in parallel computing systems, especially in GPUs, to enhance

performance. In GPUs, data is processed in parallel by multiple threads. When neighboring threads within a thread block or a warp access consecutive memory locations, the GPU can optimize memory transactions. By fetching larger data chunks from memory with fewer memory transactions, performance is improved. Take load/store data from GPU for example, when an array of data is stored in GPU memory, where each thread in a warp needs to load/store from/to a specific element in the array. Due to the consecutive thread access to consecutive elements in the array, the GPU can combine the memory requests into a single memory transaction. By maximizing memory access coalesced, GPUs can effectively leverage parallelism and optimize data transfer, resulting in improved overall performance.

C. Related Work

SpMV has been extensively studied across various platforms over the past decades, including multicore, many-core, and GPU architectures. Williams et al. investigated the application of SpMV on multicore platforms, considering both homogeneous (e.g., AMD dual-core, Intel quad-core) and heterogeneous platforms (e.g., STI Cell, Sun Niagara2) [13]. Liu et al. [14] focused on x86-based many-core processors and addressed challenges related to sparsity, irregular memory accesses, and load imbalance on Intel Xeon Phi Coprocessor. It aimed to understand the impact of sparsity, memory accesses, and load balancing on the overall performance of SpMV [14]. Bell et al. [15] explored SpMV on GPUs without introducing new storage formats for sparse matrices. They evaluated different basic formats, including COO, CSR, CSC, ELL, DIA, and hybrid formats, to harness the potential performance improvements offered by GPUs. The results highlighted the significant potential of applying SpMV on GPUs to enhance performance.

Extensive research on SpMV has led to increasingly finer-grained implementations. The objective is to mitigate the impact of matrix sparsity [9], memory access patterns [16], and load imbalance on overall performance [10]. Consequently, significant research efforts have been devoted to format design, which plays a crucial role in addressing these challenges. Many of the proposed format designs are variants or optimizations of certain basic format. These advanced formats are tailored to specific characteristics and requirements of sparse matrices and computing platforms. The aim is to optimize memory access patterns, minimize storage requirements, and improve load balancing. The focus on format design highlights the importance of considering the nuances of SpMV and its influence on performance. By refining and customizing formats to suit the unique characteristics of matrices and target platforms, researchers can further enhance optimization and efficiency in SpMV computations.

Improvements to COO format: The COO format is a simple storage format consisting of three arrays: row indices, column indices, and values of the non-zero elements. Several researchers, such as Dang [17], BRO-COO [18], ALIGNED-COO [19], and BCCOO [20], have developed new formats based on COO. For instance, Yan designed BCCOO-based SpMV algorithm called yaSpMV, which divides the matrix into blocks and uses bit flags to store indices, addressing bandwidth limitations. Additionally, yaSpMV incorporates techniques like segmented sum to improve performance.

Improvements based on CSR format: The CSR format is widely used in sparse matrix operations, serving as the foundation for numerous format designs. CSR-Adaptive [21], PCSR [22], BCSR [23], and BIN-CSR [24] improved SpMV performance by optimizing memory access, data reuse, and load mapping to threads. Yesil et al. accelerated SpMV for Power-Law graphs by leveraging locality [25]. Chu et al. improve the performances of SpMV by three methods which are flat, line enhancement, and adaptive selection. The optimization method is based on the CSR format [26]. The introduction section has presented the mainstream SpMVs based on the CSR format.

Improvements based on CSC format: Similar to the CSR, the CSC format consists of three arrays which are *colPtr*, *rowIdx*, and *value*. *value* and *rowIdx* store the value and corresponding row index respectively. *colPtr* array records the number of elements before the corresponding column. CSC format is also one of the most popular sparse matrix storage formats.

From the platform perspective: the CSC format can be the basic or part of the sparse algorithm based on distributed or multi-GPU systems. For example: MSREP [27] designs a sparse matrix representation framework that applies in multi-GPU systems and uses CSC format as one of the implementations. TCSC [28] designs a new distributed graph analytics system based on Triply CSC.

From an application perspective, the CSC format is also used in sparse matrix multiplication (SpGEMM), deep neural networks, and graph algorithms like breadth-first search. Doubly compressed sparse column (DCSC), designed by Aydin et al. is used for accelerating SpGEMM [29]. Mohammad et al. apply in-memory CSC data structure for the traversal and storage of the neural network layer [30]. Li et al. combine the CSC format and merge-based algorithm to accelerate sparse matrix sparse vector multiplication (SpMSpV) [31]. Skyler et al. design value compressed sparse column (VCSC) which takes advantage of the redundancy of the sparse matrix [32]. CSCV, proposed by Ye et al. can reduce the memory bandwidth used in SpMV and they implement this CSCV algorithm for Computed Tomography imaging [33].

Improvements based on ELL format: The ELL format consists of two parts: column indices and values. The non-zero elements retain the same row indices. ELL format is well-suited for vector architectures. Ad-ELL [34], ELL-WARP [35], SELL-C [36], BiELL [37], slice-ELL [38], BELLPACK [39], and SELL-P [40] combine the ELL format with specific architectures to achieve better load balance and improved performance.

Improvements based on DIA format: DIA is suitable for matrices with non-zero elements restricted to the diagonal. DIA consists of two arrays: one stores the non-zero elements, while the other stores the offset of the diagonal from the longest diagonal. There is some research like HDI [41] and DDD-SPLIT [42] that use the DIA format to improve performance. Gao et al. developed a new strategy called TaiChi to enhance the performance of Binary SpMV [43]. Binary SpMV is primarily used in weblink analysis, integer factorization, and compressed sensing. TaiChi divides the matrix into a sparse part and a dense

		CSR5[10]	VCSR[3]	Merge based[9]	yaSpMV [20]	Slice ELL [32]	Taichi [37]	TileSpMV [11]	FastLoad
		CSR	CSR	CSR	COO	ELL	DIA	HYB	CSC
Pre process	Blocking/Tile	●	●	●	●	●		●	●
	Partition						●		
Data loading	Input matrix	●	●	●	●	●		●	●
	Input vector								●
	Share memory	●	●	●			●	●	
Multiplication	Load balance	●	●	●	●	●	●	●	●
	Merge algorithm			●					
Addition	Atomic add							●	●
	Segment sum	●		●	●				●
Data storage	Share memory	●				●	●	●	
	Atomic add	●						●	●

Fig. 2. Summary of the related work: We select several algorithms based on different formats and compare them with FastLoad. The comparison is divided into different parts which include pre-processing, data loading, mathematic operation, and data storage.

part. The dense part is primarily stored in the DIA format, while the sparse part is stored in the CSR format. By fully utilizing the characteristics of binary SpMV and combining it with the DIA format, TaiChi demonstrates excellent performance.

Improvements based on hybrid formats: The distribution of non-zero elements in a matrix is often irregular, making it challenging to find a single format that suits every matrix. To address this, the HYB format combines two different formats, typically ELL and COO [44]. The denser part of the matrix can be stored in the ELL format, while the sparser part is stored in the COO format. Besides, ELL can also combine with CSR [45], DIA [46], and CSR can combine with COO [47]. TileSpMV, developed by Niu, takes a different approach by using six different formats to store the sparse matrix [11].

Adopting machine learning technologies for SpMV: In recent years, with the advancement of machine learning, researchers have realized its potential in selecting suitable formats for sparse matrices. Sedaghati et al. utilize machine learning techniques to determine the most appropriate representation for a matrix across different platforms [48]. Du et al. developed AlphaSparse, which can generate a format that suits the matrix, along with the corresponding SpMV kernel [49].

III. Motivation

In this section, we first analyze FastLoad against existing work to gain a better understanding of what motivated the FastLoad research. Then, we demonstrate the drawbacks of CSR-based and CSC-based SpMV with a detailed example. Finally, we design an experiment to mimic an ideal scenario to demonstrate the potential of coalesced memory access, which motivates the FastLoad algorithm.

A. Comprehensive Analysis of SpMV and its Representative Implementations

SpMV can be divided into five main steps: *Step 1:* pre-processing input matrix and vector for the SpMV. *Step 2:* loading data of the sparse matrix and vector from global memory to threads. *Step 3:* performing multiplication operation. *Step 4:* performing addition operation. *Step 5:* storing the results back from threads to global memory. In Fig. 2, we select the works mentioned above in each storage format and indicate if a step is implemented (along with a summary of the implementation). It is noted that FastLoad improves on all 5 steps, where the performance improvement mainly stems from 1) efficient data loading of sparse matrix and input vector; 2) well balanced load; and 3) optimized CSC format with segment sum and prefix sum.

B. A Motivational Example

We have also provided visualizations to illustrate Steps 2-4 via CSR-based and CSC-based SpMV in Fig. 3. The matrix A is based on a real matrix called jgl009 from SuiteSparse Matrix Collection [50]. It is found that CSR-based SpMV can reach good coalesced memory access during the loading of data from the sparse matrix. However, the data loading of the input vector proves to be irregular and non-continuous, as evidenced by the red arrows in the multiplication part of CSR-based SpMV. On the other hand, in the multiplication part of CSC-based SpMV, access to the input vector is better compared to the CSR format. However, when it comes to loading the temporary results and adding them to the corresponding result vector y, the process is non-continuous. In the next section, we show that FastLoad can address the above problems.

C. Two-Phase SpMV

To visualize the impact of data loading on the performance of SpMV and the rooms for potential improvement of the existing methods, we have designed an ideal experiment called "two-phased SpMV," where the data loading of multiplication and storing of addition are both accomplished via coalesced memory access. This is made possible by assuming the ideal possibility of combining the advantages of CSC and CSR formats with a coalesced memory access design. The CSC-based SpMV performs efficiently in loading the data of the input vector for the multiplication phase. On the other hand, the CSR-based SpMV proves to be advantageous for adding the temporary results from the same row in the addition phase.

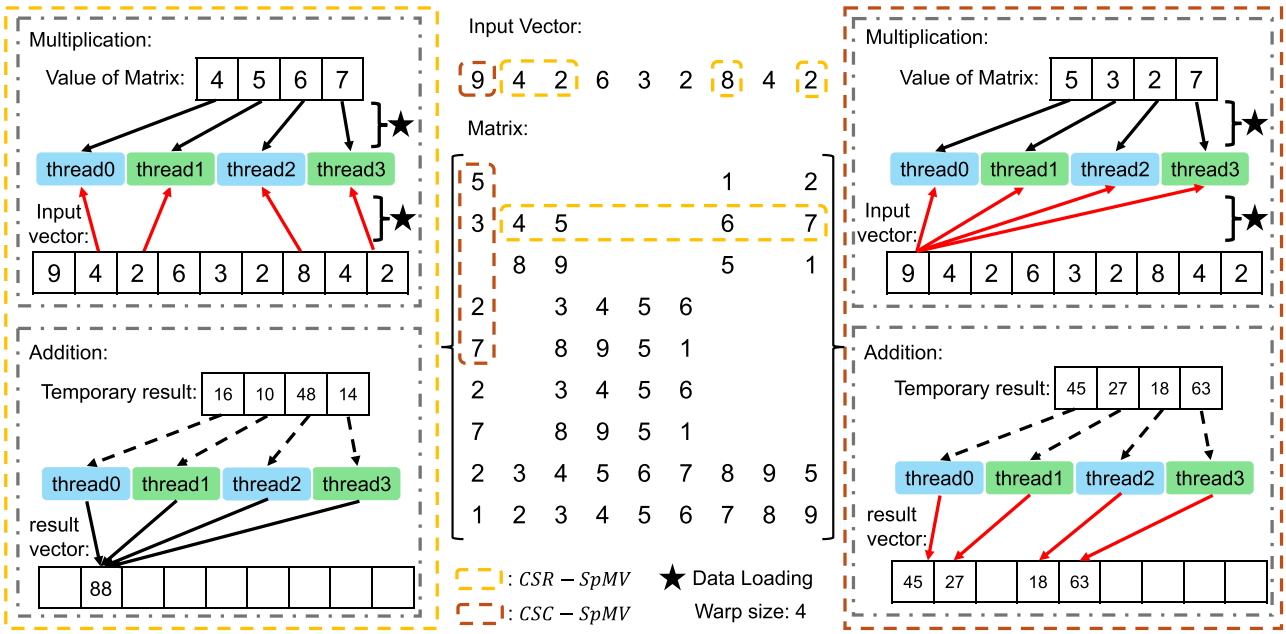


Fig. 3. The brief flow chart of SpMV based on CSR and CSC format. The arrows in the multiplication and addition sections signify the data loading or storing back of SpMV. The dotted arrows indicate the temporary data in the threads. The red arrows highlight the motivation of this work, where the non-continuous access of the input vector in CSR-based multiplication cannot achieve coalesced memory access. In CSC-based SpMV, the red arrows indices that the access of the input vector and storing temporary results also do not achieve coalesced memory access.

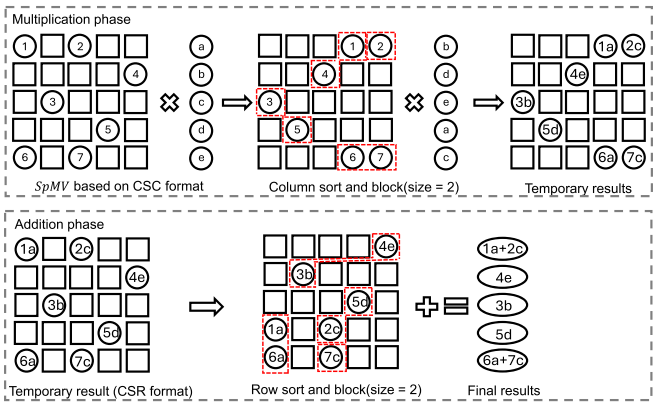


Fig. 4. The illustration for two-phase SpMV.

As illustrated in Fig. 4, the experiment is conducted in the following manner. In the multiplication phase, we utilize the CSC format for the sparse matrix. We first sort the columns of the sparse matrix and the input vector according to the number of non-zero elements of each column. Then, we show how the elements are allocated to a warp. We assign elements of each column to warps in this way: Starting with the first non-empty column, a continuous 32 columns group as the length, and the number of non-zero elements of the first non-empty column is the height. In this way, the threads in a warp can access the continuous locations of both the sparse matrix and input vector. Moreover, blocking is adopted to splitting the non-zero elements to keep load balance. Lastly, the temporary results of the multiplication are stored in temporary arrays, using the CSC format.

In the addition phase, we re-arrange the format of the multiplication phase. The conversion process involves transforming the format from multiplication to CSR. Notably, the matrix values now represent the results of multiplication, rather than the original values of the matrix. Similar to the multiplication phase, sorting on rows is executed such that the addition operations are with coalesced memory access. Additionally, we utilize blocking to compute the local sum within a block, thereby enhancing the performance of the addition operations. It is important to note that the two-phase SpMV is designed specifically to demonstrate potential improvements in existing methods. Our focus is solely on calculating the time taken for the multiplication and addition phases, excluding the time spent on the format transformation from the multiplication phase to the addition phase.

We conducted a comparison of the two-phase SpMV with the other three methods: cuSPARSE, TileSpMV, and CSR5 using over 200 matrices. As shown in Fig. 5 the two-phase SpMV outperforms all other methods in terms of GFLOPS, which highlight that there is still untapped potential for improvement. This motivates us to propose our FastLoad.

IV. The FastLoad Algorithm

A. The FastLoad SpMV

FastLoad is based on the "two-phase" SpMV, the difference is that it uses optimization strategies based on the CSC format in the whole process (including the multiplication phase and addition phase). Unlike CSC-based SpMV, FastLoad accesses the input

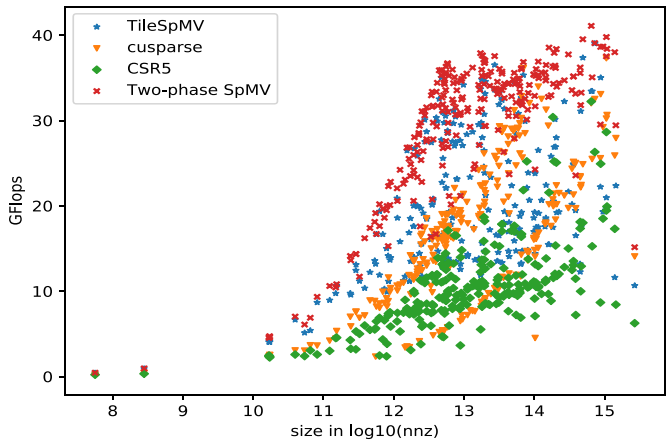


Fig. 5. The result of two-phase SpMV.

vector by *blkColIdx* and *blkLength* arrays. Algorithm 2 presents the pseudocode of the parallel FastLoad SpMV algorithm. The matrix A is partitioned into multiple blocks (Section IV-B). A block of matrix A is allocated to a warp (Line 1-10). Consequently, a warp corresponds to an element in *blkPtr*. The *blkHeight* denotes the number of loops that the block needs to iterate (Line 2-9). In an iteration, the *blkColIdx* array contains the starting column index of each block. The elements in *blkPtr* add the corresponding elements in *blkLength* multiplied with the corresponding iteration round is the corresponding index in *value* and *rowIdx* (Line 4). The column index in *blkColIdx* adds the *laneId* of the thread as the index of the value in *sortedX* (Line 5). Consequently, the threads within a warp can achieve coalesced memory access on both the sparse matrix and the input vector. Following the multiplication step, the *rowIdx* indicates the result to which the temporary results are added (Line 8).

Fig. 6 shows how FastLoad reaches the coalesced memory access for loading the sparse matrix and input vector. It consists of two steps. The first step involves sorting the columns of the sparse matrix based on the number of non-zero elements in each column (matrix $A \rightarrow$ sorted matrix A). In the second step, the input sparse matrix is partitioned into multiple blocks. Each block is with a fixed length of 32 and a variable height. This choice aligns with the warp size of 32 threads in GPUs, ensuring efficient utilization of computing resources and minimizing thread divergence. The height of the blocks is determined by the variation in non-zero elements across columns, which helps in grouping together dense parts of the matrix within a block or a sequence of blocks. The details are shown in Section IV-B.

Because FastLoad is optimized from CSC format, it will cause the conflict of addition of the multiplication result. To address this issue, we introduce two improvement methods of FastLoad. FastLoad categorizes the blocks into different scenarios, enabling the use of different addition techniques, such as segment sum and prefix sum, to calculate the final result. By adopting the addition method based on the specific block characteristics, FastLoad maximizes computational efficiency and optimizes the final output vector generation. The details are shown in Section IV-C.

B. The Storage Structure and Process Detail of FastLoad

The storage structure of FastLoad closely resembles the CSC format and involves three auxiliary arrays, including *blkColidx*,*blkLength*, and *blkHeight*. The *blkColIdx* array stores the column index of the first element in each block, which helps achieve continuous access to both the sparse matrix and input vector. The *blkLength* array stores the length of each block. The *blkHeight* array represents the height of each block and can be seen as the number of loops that threads in a warp need to iterate.

Except for three auxiliary arrays, there are three main arrays based on the CSC format, which are *blkPtr*, *rowIdx*, and *value*. The *value* array and the *rowIdx* array record the matrix's values and the corresponding row indices respectively. The *blkPtr* array maintains the index of the first element of each block in the *value* array.

Let's consider the example in Fig. 6 to illustrate the detail of the process. We assume a warp consists of 4 threads. In block 0, the first non-empty column is column 0, which contains 2 elements. The value assigned to block 0 is the first 2 values of a continuous 4-column which starts from column 0. Therefore, the height of the first block is 2. The *blkPtr*, *blkColIdx*, *blkHeight*, and *blkLength* arrays record 0, 0, 2, and 4 for block 0, respectively. Similar to block 0, the first non-empty column of block 5 is column 2, and the number of elements in column 2 remains 1. The value assigned to block 5 is the first 1 value of the remained of a continuous 4-column which starts from column 2. Similarly, the *blkPtr*, *blkColIdx*, *blkHeight*, and *blkLength* arrays record 34, 2, 1, and 4 for block 5, respectively. The arrangement of *rowIdx* and *value* arrays follows a zigzag pattern.

C. Two Improvement Methods of FastLoad

As mentioned in Section IV-A, FastLoad is implemented based on the CSC format. We use *atomicadd* to add the temporary result of multiplication to the corresponding result vector y. However, as the size of the input matrix increases, the *atomicadd* operation becomes less efficient.

To address this issue, we propose two improvement methods based on a key observation, aiming to enhance the efficiency of the addition operation. In Fig. 6, the *rowIdx* array shows that the adjacent elements have the same row index, which means they can add together inside the warp by using the shuffle operation. Consequently, we cite a method called *segment sum* [51]. Segment sum is SIMD-friendly method [10]. It leverages the *shuffle* operation to efficiently add elements from the same row within a warp. Besides, another situation not shown in Fig. 6 is that it is possible for all the values in a warp from the same row. In this situation, we can use *prefix sum*.

By employing these methods, FastLoad optimizes the addition operation in a manner that minimizes the use of *atomicadd* operations, resulting in improved overall performance

V. Experimental Results

A. Experimental Setup

Our experimental platform is NVIDIA GeForce RTX 3090 Ti with Ubuntu Linux v22.04 installed. The GPU driver version is

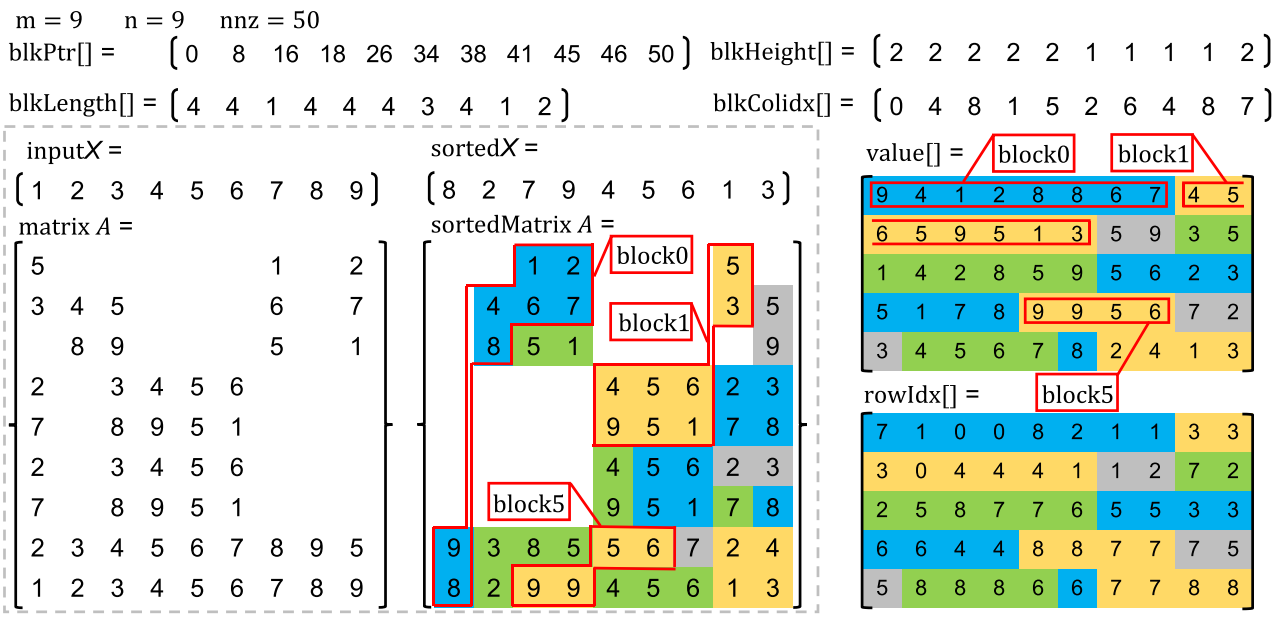


Fig. 6. An example of FastLoad: The original matrix is a real matrix called *jgl009*. The matrix is sorted based on the number of non-zero elements in each column. The block length is set as the warp size (In this example, it is set as 4 due to space issue). In addition to the basic arrays, FastLoad includes three auxiliary arrays. The threads within a warp can multiply the matrix elements with the corresponding vector elements. For instance, if element 4in block 0 needs to be multiplied with the corresponding vector elements, the *laneId* of the corresponding thread can determine the index of the vector element by adding the corresponding value in the *blkColIdx* array.

Algorithm 2: A Pseudocode of Parallel FastLoad SpMV.

Input: $blkPtr[]; blkLength[]; blkHeight[];$
$blkColIdx[]; value[]; rowIdx[]; sortedX[];$
Output: $y[]$

```
for i from 0 to sizeof(blkPtr[]) − 2 in parallel do
    for j ← 0; j < blkHeight[i]; j++ do
        if laneId < blkLength[i] then
            pointer ← blkPtr[i] + laneId + j × blkLength[i];
            pointerX ← laneId + blkcolIdx[i];
            value = value[pointer];
            rowIdx = rowIdx[pointer];
            atomicAdd(y[rowIdx], value × sortedX[pointerX]);
        end
    end
end
```

525.105.17, and the CUDA runtime version is 11.1. We compare our work with cuSPARSE v11.1 kernel *cuSPARSE-SpMV* [52], TileSpMV [11], CSR5 [10] and CSC-based SpMV. cuSPARSE SpMV kernel is designed by NVIDIA. TileSpMV is a method that divides the sparse matrix into tiles and uses different storage formats to store the tiles. CSR5 compresses the sparse matrix into regular size. The CSC-based SpMV is the basic SpMV algorithm on GPU. We use one warp to calculate one column of the sparse matrix. We use all 2757 sparse matrices in the SuiteSparse Matrix Collection as our test workload [50]. The source code of this work is downloadable at https://github.com/MinttHu/FastLoad.git.

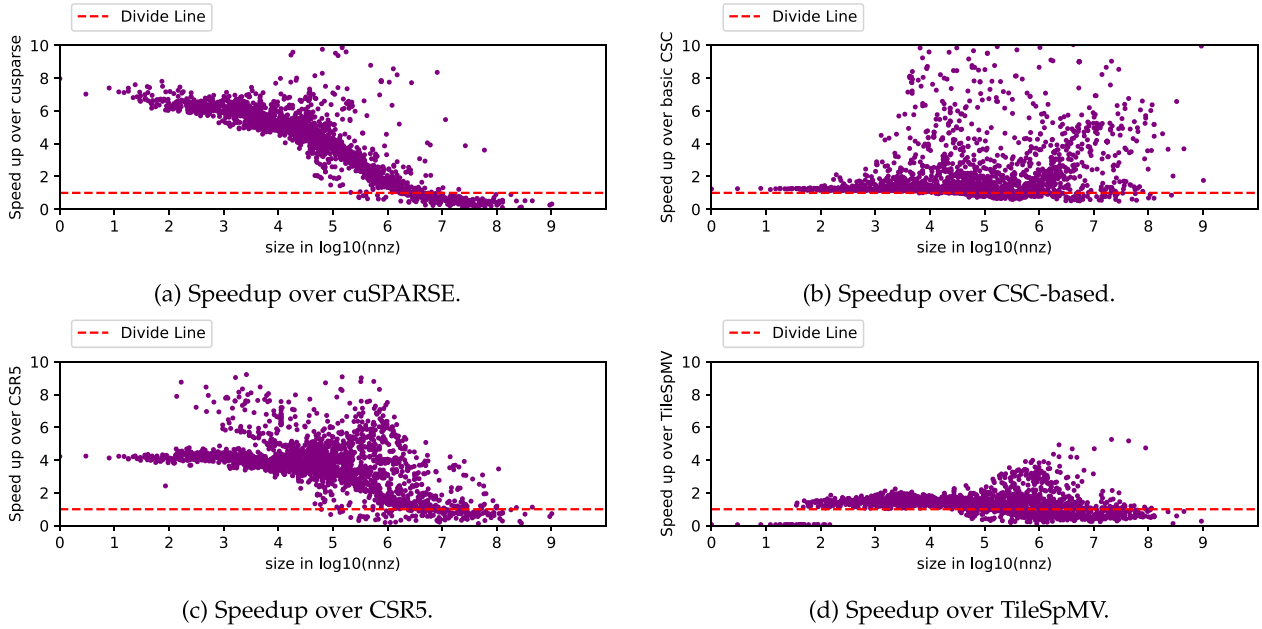
Our experiment set is adapted to the GPU architecture (thread block, warp, and thread). A thread block contains 2 warps in order to avoid calculation overhead. One warp corresponds to one block of the matrix. To attach a better load balance, we set 1 as the height of the block and set the length of the block as 32. The detail of the distribution of tasks is in Section IV-A.

B. Performance Comparison Over Existing SpMVs

In Fig. 1, we select 10 groups of matrices in the dataset and test the performance in each group. The result shows that FastLoad exhibits a good performance over all 10 different matrix groups.

The performance results are presented in Fig. 7, showcasing the speedup achieved over the different workloads. In the figure, the dot positioned above the dividing line represents FastLoad being faster than the existing SpMVs, while the dot below the dividing line represents FastLoad being slower than the existing SpMVs. In Fig. 7(a) and (c), the speedup over cuSPARSE and CSR5 becomes slower as the number of non-zero elements in the matrix increases. This suggests that when the matrix size becomes large, the impact of *atomicadd* operations may affect overall performance. In Fig. 7(b) and (d), FastLoad exhibits relatively stable speedup over CSC-based SpMV and TileSpMV.

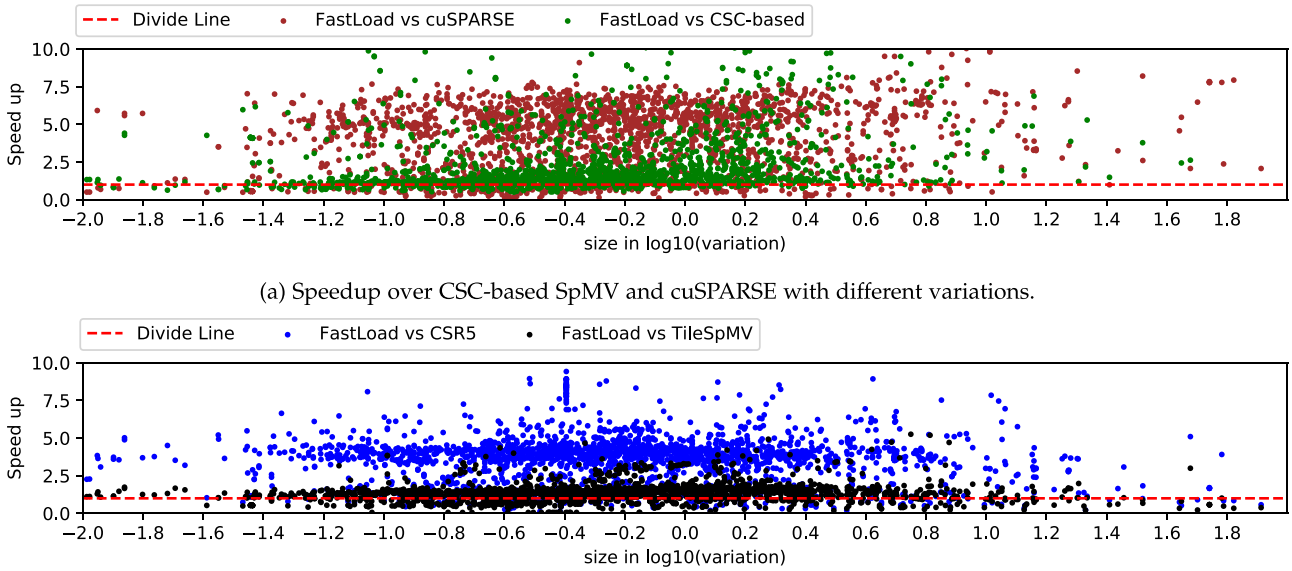
It is evident that our method demonstrates favorable performance across most matrices. Specifically, it outperforms CSC-based SpMV in 2342 matrices, cuSPARSE in 2326 matrices, CSR5 in 2402 matrices, and TileSpMV in 1978 matrices.



(a) Speedup over cuSPARSE. (b) Speedup over CSC-based.

(c) Speedup over CSR5. (d) Speedup over TileSpMV.

Fig. 7. Speedup over the different workloads. FastLoad significantly outperforms the cuSPARSE, CSC-based, and CSR5 (slightly outperforms TileSpMV).



(a) Speedup over CSC-based SpMV and cuSPARSE with different variations.

(b) Speedup over CSR5 and TileSpMV with different variations.

Fig. 8. Performance analysis of Fastload on different irregular matrices. We calculate the variation of each matrix and use it as the x-axis. The variation is increased with the irregular of the matrix. It is found that FastLoad can handle different irregular matrices.

The speedup achieved over these benchmarks is 2.12×, 2.98×, 2.88×, and 1.22× , respectively.

C. Performance Analysis of Fastload on Different Irregular Matrices

In Fig. 8, we analyze the speedup of FastLoad across various existing SpMVs based on the variation of each matrix. The variation of each matrix is calculated as the standard deviation of row length divided by the average mean of row length [9]. It is observed that the speedup is primarily achieved on matrices with smaller variations. The speedup of FastLoad is obviously better compared to CSC-based SpMV, cuSPARSE, and CSR5 with different irregular matrices.

FastLoad does not outperform significantly on TileSpMV as other existing SpMVs. The main reason for this discrepancy is that some matrices exhibit significant variations, and FastLoad is not good at handling these specific cases

TABLE I
COMPARISON OF DIFFERENT HEIGHTS OF FASTLOAD AND ITS VARIANTS

Variant	TileSpMV		CSR5		cuSPARSE	
	Ratio	Speed up	Ratio	Speed up	Ratio	Speed up
non_control	81%	0.79	90%	1.79	89%	2.01
Block_height=1	81%	1.26	90%	2.84	89%	3.02
Block_height=2	81%	1.26	92%	2.83	90%	3.03
Block_height=4	81%	1.24	92%	2.79	90%	2.99
Block_height=8	80%	1.21	92%	2.72	90%	2.94
Block_height=16	74%	1.16	92%	2.61	89%	2.81
Block_height=32	67%	1.05	90%	2.35	85%	2.53

effectively. In contrast, TileSpMV performs better in such scenarios, as it can intelligently select the appropriate format for small blocks, allowing for improved performance. Noted that FastLoad can be added to alternative formats for TileSpMV.

D. Effectiveness of Different Block Height

In order to assess the impact of different block heights on the efficiency of FastLoad. We introduced different limitations of the block height and compared them against existing SpMVs. Table I presents the results of the comparison. We evaluated FastLoad with no height limit against FastLoad with different height limits, and compare them with existing SpMVs. It is evident that FastLoad without a height limit did not perform optimally. The speedup over different SpMVs is not as good as others.

The height limit servers to control the divergence between warps, thereby imposing a restriction on maximum block height. From the table, it is evident that when the height limit is set to 8 or lower, the performance in terms of faster ratio and speedup is relatively good. However, when the height limit is increased to 16 or 32, the impact of warp divergence becomes more pronounced, leading to a decrease in performance.

E. Effectiveness of the Two Improvement Methods

We conducted a total of eight different versions of FastLoad, including one without a height limit, six versions with different height limits, and one version with optimization. For a more detailed analysis, we selected four different types of matrices. In Fig. 9, the label 'optimized' refers to FastLoad with the proposed improvement, while the accompanying number indicates the specific height limit used. FastLoad with the proposed improvement outperforms the other variants. Particularly, in Fig. 9(a), the performance improvement is significant. There are two main reasons for this improvement. First, the control of divergence between warps helps enhance performance. Second, the additional steps introduced in the improvement, such as segment sum and prefix sum, effectively reduce the burden on *atomicadd* operations. These results validate the effectiveness of the strategy employed in FastLoad.

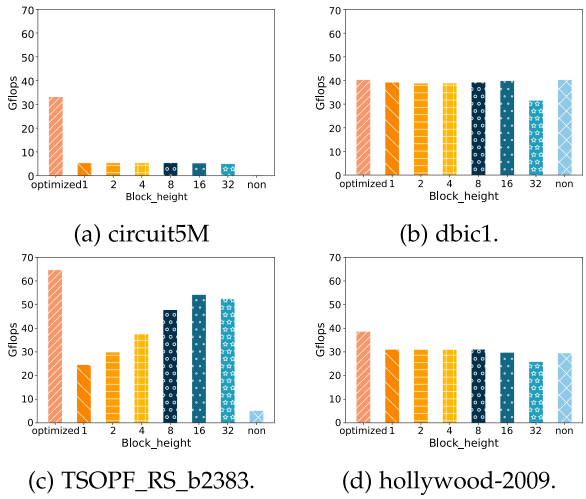


Fig. 9. Comparison of improvement over different block heights.

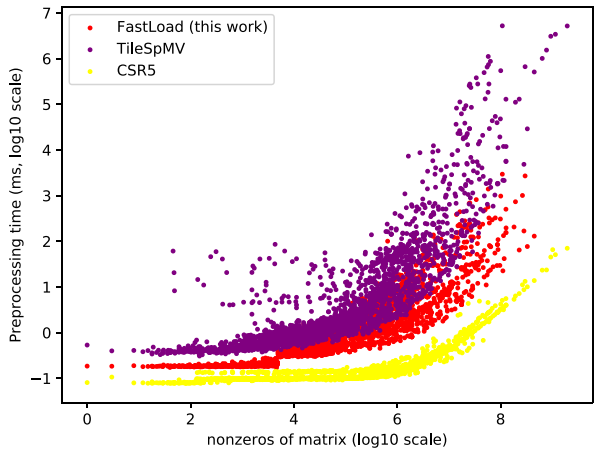


Fig. 10. Comparison of preprocessing cost of SpMV methods.

F. Preprocessing Overhead Analysis

We conducted measurements to assess the preprocessing overhead involved in converting a basic CSR matrix to our proprietary format. The preprocess of FastLoad and CSR5 is performed on the GPU. The preprocess of TileSpMV uses MPI on the CPU. Fig. 10 shows the preprocess time of CSR5, TileSpMV, and our method. The preprocess time of our method is faster than TileSpMV in most matrices and is close to CSR5 when the number of nonzero is less than 10^4. Despite the preprocessing time being longer than CSR5 when the matrix size is large, it is acceptable when executing the SpMV kernel in an iterative solver.

G. Peak Memory Consumption Analysis

We compare the peak memory consumption of FastLoad with the other four methods using the four matrices in Fig. 9. As shown in Fig. 11, the peak memory consumption of FastLoad is higher than the other four methods, as it includes three auxiliary arrays. With the matrix size increase, the size of auxiliary

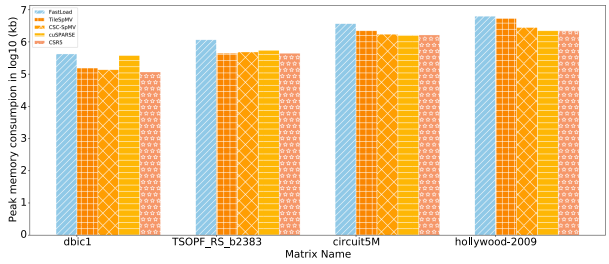


Fig. 11. Comparison of peak memory consumption between FastLoad and the other four methods.

arrays also increased. Consequently, the memory consumption of FastLoad becomes larger.

VI. Conclusion and Future Work

In this paper, we propose a novel SpMV algorithm called FastLoad. FastLoad improves the efficient data loading of the sparse matrix and input vector by achieving coalesced memory access and keeps well-balanced load distribution among threads and warps. Furthermore, FastLoad uses two improvement methods (segment sum and prefix sum) to reduce the use of *atomicadd* operations and improve the efficiency of the addition part compared to CSC-based SpMV.

For future improvements, to further evaluate FastLoad's performance, we propose combining it with a selective algorithm. As an illustration, we could integrate FastLoad with TileSpMV and compare the performance of TileSpMV with and without FastLoad. This comparison will provide valuable insights into FastLoad's capabilities and potential enhancements.

Acknowledgment

The authors are grateful to the anonymous reviewers for their valuable and helpful comments on improving the manuscript.

References

[1] I. Gelado and M. Garland, "Throughput-oriented GPU memory allocation," in *Proc. 24th Symp. Princ. Pract. Parallel Program.*, New York, NY, USA: ACM, 2019, pp. 27–37.

[2] Z. Yan, Y. Lin, L. Peng, and W. Zhang, "Harmonia: A high throughput B tree for GPUs," in *Proc. 24th Symp. Princ. Pract. Parallel Program.*, New York, NY, USA: ACM, 2019, pp. 133–144.

[3] E. Karimi, N. B. Agostini, S. Dong, and D. Kaeli, "VCSR: An efficient GPU memory-aware sparse format," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 12, pp. 3977–3989, Dec. 2022.

[4] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí, "Solving dense linear systems on graphics processors," in *Proc. Eur. Conf. Parallel Process.*, Springer, 2008, pp. 739–748.

[5] V. Volkov and J. W. Demmel, "Benchmarking gpus to tune dense linear algebra," in *Proc. 2008 ACM/IEEE Conf. Supercomput.*, IEEE, 2008, pp. 1–11.

[6] M. Wang et al., "Deep graph library: A graph-centric, highly-performant package for graph neural networks," 2019, *arXiv:1909.01315*.

[7] Y. Zhao, W. Zhou, X. Shen, and G. Yiu, "Overhead-conscious format selection for spmv-based applications," in *Proc. 2018 IEEE Int. Parallel Distrib. Process. Symp.*, IEEE, 2018, pp. 950–959.

[8] J. Kepner et al., "Mathematical foundations of the graphblas," in *Proc. 2016 IEEE High Perform. Extreme Comput. Conf.*, IEEE, 2016, pp. 1–9.

[9] D. Merrill and M. Garland, "Merge-based parallel sparse matrix-vector multiplication," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, IEEE, 2016, pp. 678–689.

[10] W. Liu and B. Vinter, "CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *Proc. 29th ACM Int. Conf. Supercomput.*, New York, NY, USA: ACM, 2015, pp. 339–350.

[11] Y. Niu, Z. Lu, M. Dong, Z. Jin, W. Liu, and G. Tan, "TileSpMV: A tiled algorithm for sparse matrix-vector multiplication on GPUs," in *Proc. 2021 IEEE Int. Parallel Distrib. Process. Symp.*, IEEE, 2021, pp. 68–78.

[12] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo, "Sparse matrix-vector multiplication on GPGPUs," *ACM Trans. Math. Softw.*, vol. 43, no. 4, pp. 1–49, 2017.

[13] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *Proc. 2007 ACM/IEEE Conf. Supercomput.*, New York, NY, USA: ACM, 2007, pp. 1–12.

[14] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *Proc. 27th Int. ACM Conf. Int. Conf. Supercomput.*, New York, NY, USA: ACM, 2013, pp. 273–282.

[15] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proc. Conf. High Perform. Comput. Netw. Storage Anal.*, New York, NY, USA: ACM, 2009, pp. 1–11.

[16] D. Fujiki, N. Chatterjee, D. Lee, and M. O'Connor, "Near-memory data transformation for efficient sparse matrix multi-vector multiplication," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, New York, NY, USA: ACM, 2019, pp. 1–17.

[17] H.-V. Dang and B. Schmidt, "Cuda-enabled sparse matrix–vector multiplication on GPUs using atomic operations," *Parallel Comput.*, vol. 39, no. 11, pp. 737–750, 2013.

[18] W. T. Tang et al., "Accelerating sparse matrix-vector multiplication on GPUs using bit-representation-optimized schemes," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, New York, NY, USA: ACM, 2013, pp. 1–12.

[19] M. Shah and V. Patel, "An efficient sparse matrix multiplication for skewed matrix on GPU," in *Proc. 2012 IEEE 14th Int. Conf. High Perform. Comput. Commun. IEEE 9th Int. Conf. Embedded Softw. Syst.*, IEEE, 2012, pp. 1301–1306.

[20] S. Yan, C. Li, Y. Zhang, and H. Zhou, "yaSpMV: Yet another SpMV framework on GPUs," *ACM Sigplan Notices*, vol. 49, no. 8, pp. 107–118, 2014.

[21] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, IEEE, 2014, pp. 769–780.

[22] M. M. Dehnavi, D. M. Fernández, and D. Giannacopoulos, "Finite-element sparse matrix vector multiplication on graphic processing units," *IEEE Trans. Magn.*, vol. 46, no. 8, pp. 2982–2985, Aug. 2010.

[23] W. Xu, H. Zhang, S. Jiao, D. Wang, F. Song, and Z. Liu, "Optimizing sparse matrix vector multiplication using cache blocking method on fermi GPU," in *Proc. 2012 13th ACIS Int. Conf. Softw. Eng., Artif. Intell., Netw. Parallel/Distrib. Comput.*, IEEE, 2012, pp. 231–235.

[24] D. Weber, J. Bender, M. Schnoes, A. Stork, and D. Fellner, "Efficient GPU data structures and methods to solve sparse linear systems in dynamics applications," in *Computer Graphics Forum*. New York, NY, USA: Wiley, 2013, pp. 16–26.

[25] S. Yesil, A. Heidarshenas, A. Morrison, and J. Torrellas, "Speeding up SpMV for power-law graph analytics by enhancing locality & vectorization," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, IEEE, 2020, pp. 1–15.

[26] G. Chu et al., "Efficient algorithm design of optimizing SPMV on GPU," in *Proc. 32nd Int. Symp. High- Perform. Parallel Distrib. Comput.*, 2023, pp. 115–128.

[27] J. Chen et al., "MSREP: A fast yet light sparse matrix framework for multi-GPU systems," 2022, *arXiv:2209.07552*.

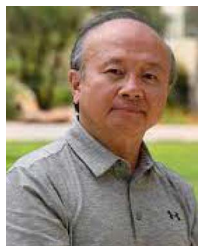
[28] M. H. Mofrad, R. Melhem, Y. Ahmad, and M. Hammoud, "Efficient distributed graph analytics using triply compressed sparse format," in *Proc. 2019 IEEE Int. Conf. Cluster Comput.*, IEEE, 2019, pp. 1–11.

[29] A. Buluc and J. R. Gilbert, "On the representation and multiplication of hypersparse matrices," in *Proc. 2008 IEEE Int. Symp. Parallel Distrib. Process.*, IEEE, 2008, pp. 1–11.

[30] M. H. Mofrad, R. Melhem, Y. Ahmad, and M. Hammoud, "Multithreaded layer-wise training of sparse deep neural networks using compressed sparse column," in *Proc. 2019 IEEE High Perform. Extreme Comput. Conf.*, IEEE, 2019, pp. 1–6.
[31] H. Li, H. Yokoyama, and T. Araki, "Merge-based parallel sparse matrix-sparse vector multiplication with a vector architecture," in *Proc. 2018 IEEE 20th Int. Conf. High Perform. Comput. Commun.; IEEE 16th Int. Conf. Smart City; IEEE 4th Int. Conf. Data Sci. Syst. (HPCC/SmartCity/DSS)*, IEEE, 2018, pp. 43–50.
[32] S. Ruiter, S. Wolfgang, M. Tunnell, T. Triche Jr., E. Carrier, and Z. DeBruine, "Value-compressed sparse column (VCSC): Sparse matrix storage for redundant data," 2023, *arXiv:2309.04355*.
[33] W. Ye, C. Huang, J. Huang, J. Li, Y. Lu, and Y. Jiang, "An integral-equation-oriented vectorized spmv algorithm and its application on CT imaging reconstruction," in *Proc. 2022 IEEE Int. Parallel Distrib. Process. Symp.*, IEEE, 2022, pp. 773–783.
[34] M. Maggioni and T. Berger-Wolf, "Optimization techniques for sparse matrix–vector multiplication on GPUs," *J. Parallel Distrib. Comput.*, vol. 93, pp. 66–86, 2016.
[35] J. Wong, E. Kuhl, and E. Darve, "A new sparse matrix vector multiplication graphics processing unit algorithm designed for finite element problems," *Int. J. Numer. Methods Eng.*, vol. 102, no. 12, pp. 1784–1814, 2015.
[36] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide simd units," *SIAM J. Sci. Comput.*, vol. 36, no. 5, pp. C401–C423, 2014.
[37] J. Zhang et al., "Efficient sparse matrix–vector multiplication using cache oblivious extension quadtree storage format," *Future Gener. Comput. Syst.*, vol. 54, pp. 490–500, 2016.
[38] A. Dziekonski, A. Lamecki, and M. Mrozowski, "A memory efficient and fast sparse matrix vector product on a GPU," *Prog. Electromagn. Res.*, vol. 116, pp. 49–63, 2011.
[39] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on GPUs," *ACM Sigplan Notices*, vol. 45, no. 5, pp. 115–126, 2010.
[40] H. Anzt et al., "Load-balancing sparse matrix vector product kernels on GPUs," *ACM Trans. Parallel Comput.*, vol. 7, no. 1, pp. 1–26, 2020.
[41] J. Godwin, J. Holewinski, and P. Sadayappan, "High-performance sparse matrix-vector multiplication on GPUs for structured grid computations," in *Proc. 5th Annu. Workshop Gen. Purpose Process. Graph. Process. Units*, New York, NY, USA: ACM, 2012, pp. 47–56.
[42] L. Yuan, Y. Zhang, X. Sun, and T. Wang, "Optimizing sparse matrix vector multiplication using diagonal storage matrix format," in *Proc. 2010 IEEE 12th Int. Conf. High Perform. Comput. Commun.*, IEEE, 2010, pp. 585–590.
[43] J. Gao, W. Ji, Z. Tan, Y. Wang, and F. Shi, "TaiChi: A hybrid compression format for binary sparse matrix-vector multiplication on GPU," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 12, pp. 3732–3745, Dec. 2022.
[44] W. T. Tang, W. J. Tan, R. S. M. Goh, S. J. Turner, and W.-F. Wong, "A family of bit-representation-optimized formats for fast sparse matrix-vector multiplication on the GPU," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 9, pp. 2373–2385, Sep. 2015.
[45] A. Maringanti, V. Athavale, and S. B. Patkar, "Acceleration of conjugate gradient method for circuit simulation using cuda," in *Proc. 2009 Int. Conf. High Perform. Comput.*, IEEE, 2009, pp. 438–444.
[46] M. Maggioni, T. Berger-Wolf, and J. Liang, "GPU-based steady-state solution of the chemical master equation," in *Proc. 2013 IEEE Int. Symp. Parallel Distrib. Process. Workshops Phd Forum.*, IEEE, 2013, pp. 579–588.
[47] A. Monakov and A. Avetisyan, "Implementing blocked sparse matrix-vector multiplication on nvidia GPUs," in *Proc. Int Workshop on Embedded Comput. Syst.*, Springer, 2009, pp. 289–297.
[48] N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan, "Automatic selection of sparse matrix representation on GPUs," in *Proc. 29th ACM Int. Conf. Supercomput.*, New York, NY, USA: ACM, 2015, pp. 99–108.
[49] Z. Du, J. Li, Y. Wang, X. Li, G. Tan, and N. Sun, "Alphasparse: Generating high performance SPMV codes directly from sparse matrices," in *Proc. Int. Conf. High Perform. Comput. Netw., Storage Anal.*, IEEE, 2022, pp. 1–15.
[50] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1–25, 2011.
[51] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd, and J. Manferdelli, "Fast scan algorithms on graphics processors," in *Proc. 22nd Annu. Int. Conf. Supercomput.*, New York, NY, USA: ACM, 2008, pp. 205–213.
[52] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, "Cusparse library," in *Proc. GPU Technol. Conf.*, 2010, pp. 1–310.

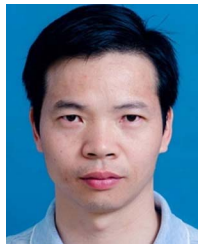
Jinyu Hu received the BS degree from Central South University, China, in 2020, and the MS degree from University College London, U.K., in 2021. He is currently working toward the PhD degree with the Hunan University, China. His research interests include high-performance computing, scientific data management, and parallel and distributed processing.

Huizhang Luo (Member, IEEE) received the BS and PhD degrees in computer science from Chongqing University, China, in 2012 and 2017, respectively. He is currently an associate professor in Hunan University. Before that, he was a postdoctoral researcher with the Department of Electrical and Computer Engineering at NJIT. His research interests include memory systems, high-performance computing, and non-volatile memory.



Hong Jiang (Fellow, IEEE) received the BSc degree in computer engineering from the Huazhong University of Science and Technology, Wuhan, China, the MASc degree in computer engineering from the University of Toronto, Toronto, Canada, and the PhD degree in computer science from the Texas A&M University, College Station, Texas, USA. He is currently chair and Wendell H. Nedderman Endowed professor of Computer Science and Engineering Department with the University of Texas at Arlington. Prior to joining UTA, he served as a program director with National Science Foundation (2013–2015) and he was at University of Nebraska-Lincoln since 1991, where he was Willa Cather professor of computer science and engineering. He has graduated 17 PhD students and supervised 20 post-doctoral fellows and visiting scholars. He is currently supervising/co-supervising more than 10 PhD students and post-doc fellows. His present research interests include computer architecture, computer storage systems and parallel I/O, high-performance computing, Big Data computing, cloud computing, performance evaluation. He is an associate editor of the IEEE Transactions on Computers and recently served as an associate editor of the IEEE Transactions on Parallel and Distributed Systems. He has more than 300 publications in major journals and international Conferences in these areas, including IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, IEEE TRANSACTIONS ON COMPUTERS, PROCEEDINGS OF IEEE, ACM-TACO, ACM-ToS, USENIX ATC, FAST, EUROSYS, ISCA, MICRO, SOCC, LISA, SIGMETRICS, ICDE, DATE, ICDCS, IPDPS, MIDDLEWARE, OOPLAS, ECOOP, SC, ICS, HPDC, INFOCOM, ICPP, etc., and his research has been supported by NSF and industry.

Guoqing Xiao (Member, IEEE) is currently an associate professor in computer science and technology with the Hunan University (HNU), China. Before joining HNU, he worked as a postdoctoral fellow with the Data Systems Group of the David R. Cheriton School of Computer Science, University of Waterloo, Canada. His current research interests mainly include parallel and distributed processing, high-performance computing and supercomputing, artificial intelligence and Big Data computing. He have published more than 30 papers in peer-reviewed international journals and conferences. He is a member of the CCF.



Kenli Li (Senior Member, IEEE) received the PhD degree in computer science from the Huazhong University of Science and Technology, China, in 2003. He is currently a Cheung Kong professor of computer science and technology with Hunan University, the dean of the College of Computer Science and Electronic Engineering, Hunan University. His major research interests include parallel and distributed processing, high-performance computing, and Big Data management. He has published more than 250 research papers in international conferences and journals such as IEEE TRANSACTIONS ON COMPUTERS, IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, IEEE TRANSACTIONS ON CLOUD COMPUTING, ICPP, ICDCS, etc. He serves on the editorial board of the IEEE TRANSACTIONS ON COMPUTERS. He is an outstanding member of the CCF.