ApSpGEMM: Accelerating Large-scale SpGEMM with Heterogeneous Collaboration and Adaptive Panel

DEZHONG YAO, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
SIFAN ZHAO, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
TONGTONG LIU, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
GANG WU, National Super Computing Center in Zhengzhou, Zhengzhou University, Zhengzhou, China
HAI JIN, School of computer science and technology, Huazhong University of Science and Technology, Wuhan, China

The *Sparse General Matrix-Matrix multiplication* (SpGEMM) is a fundamental component for many applications, such as *algebraic multigrid methods* (AMG), graphic processing, and deep learning. However, the unbearable latency of computing high-dimensional, large-scale sparse matrix multiplication on GPUs hinders the development of these applications. An effective approach is heterogeneous cores collaborative computing, but this method must address three aspects: (1) irregular non-zero elements lead to load imbalance and irregular memory access, (2) different core computing latency differences reduce computational parallelism, and (3) temporary data transfer between different cores introduces additional latency overhead. In this work, we propose an innovative framework for collaborative large-scale sparse matrix multiplication on CPU-GPU heterogeneous cores, named ApSpGEMM. ApSpGEMM is based on sparsity rules and proposes reordering and splitting algorithms to eliminate the impact of non-zero element distribution features on load and memory access. Then adaptive panels allocation with affinity constraints among cores improves computational parallelism. Finally, carefully arranged asynchronous data transmission and computation balance communication overhead. Compared with state-of-the-art SpGEMM methods, our approach provides excellent absolute performance on matrices with different sparse structures. On heterogeneous cores, the GFlops of large-scale sparse matrix multiplication is improved by 2.25 to 7.21 times.

Dezhong Yao, Sifan Zhao, Tongtong Liu, and Hai Jin are affiliated with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology.
This work is supported by the National Natural Science Foundation of China under Grant No.62072204, and the National Key Research and Development Program of China under Grant No.2021YFB1714600. The computation is completed in the HPC Platform of Huazhong University of Science and Technology and supported by the National Supercomputing Center in Zhengzhou.
Authors' Contact Information: Dezhong Yao, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China; e-mail: dyao@hust.edu.cn; Sifan Zhao (Corresponding author), School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, Hubei, China; e-mail: sifan@hust.edu.cn; Tongtong Liu, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, Hubei, China; e-mail: tliu@hust.edu.cn; Gang Wu, National Super Computing Center in Zhengzhou, Zhengzhou University, Zhengzhou, Henan, China; e-mail: gangwu@zzu.edu.cn; Hai Jin, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, Hubei, China; e-mail: hjin@hust.edu.cn.

ACM 1544-3973/2025/03-ART20
https://doi.org/10.1145/3703352

CCS Concepts: • **Computer systems organization → Multicore architectures**; • **Theory of computation → Massively parallel algorithms**; • **Software and its engineering → Scheduling;**

Additional Key Words and Phrases: SpGEMM, adaptive panels, GPU, heterogeneous cores

ACM Reference Format:
Dezhong Yao, Sifan Zhao, Tongtong Liu, Gang Wu, and Hai Jin. 2025. ApSpGEMM: Accelerating Large-scale SpGEMM with Heterogeneous Collaboration and Adaptive Panel. *ACM Trans. Arch. Code Optim.* 22, 1, Article 20 (March 2025), 23 pages. https://doi.org/10.1145/3703352

1 Introduction

SpGEMM is widely applied in various fields, including AMG [7], graphic processing [18], deep learning [21] and geometric transformations [16] in CAD/CAE. SpGEMM operates two sparse matrices A and B to compute a result $C = A \times B$. To enhance SpGEMM performance on GPUs, the widely used method is Gustavson's row-row algorithm [13], which computes the result matrix C in parallel by aligning the ***non-zero elements*** (**NZs**) of A in the corresponding rows of B. Due to the sparse structural features of the input and output matrices, accelerating SpGEMM on GPUs [10, 14] has long been a noteworthy area of focus. As a result, a number of recent efforts have been made to develop GPU implementations for SpGEMM [1, 34].

Existing approaches assume that the entire SpGEMM computation can be completed within the available memory of a single GPU. However, the sizes of sparse matrices can be quite large in real world. Koichi Shirahata et al. [36] found that graphs with over 100 million vertices and over 1 billion edges (requiring over a terabyte of uncompressed storage) are very common in social networks and scientific computing. Due to memory constraints [4, 35], even the most advanced and memory-efficient SpGEMM implementations [26, 29] are unable to handle these large-scale matrices on GPUs. Recently, the development of unified memory [25] has provided a new approach to address memory limitations. Therefore, the collaborative computation of large-scale SpGEMM by heterogeneous CPUs and GPUs is worth attention.

Previous work HPMaX [17] has attempted to address *dense matrix multiplication* (DGEMM) through collaborative CPU and GPU. However, the random distribution of NZs makes computing SpGEMM on heterogeneous cores more challenging compared with DGEMM. First, in the kernel, the regular distribution of NZs in DGEMM allows for predictable memory access patterns. This means that threads can read and write multiple consecutive NZs while ensuring load balancing. However, in SpGEMM, the distribution of NZs is random and irregular. The ***number of NZs*** (**NNZs**) each thread is responsible for is unpredictable, and reading NZs one by one incurs significant memory access overhead. Secondly, due to differences in computational resources and the irregular distribution of NZs, the CPU and GPU incur different latencies when computing SpGEMM. This implies that it is hard to guarantee computational parallelism if we allocate different parts of the matrix to the CPU and GPU separately without constraints. Finally, regardless of whether it's computing input matrices or storing output matrices, the temporary data transfer between CPU and GPU are inevitable. However, on PCIe-based heterogeneous devices, these temporary data transfers significantly increase the computation's waiting latency.

To address the aforementioned challenges, we propose ApSpGEMM,[1] an adaptive-panel-based heterogeneous collaborative approach for large-scale SpGEMM. We first designed a lightweight analyzer to extract the distribution features of NZs. Next, we define an efficient sparsity ordering rule to quickly reorder and split the matrix based on the NZs' features. To achieve load balancing

[1]The source code is available at https://github.com/CGCL-codes/ApSpGEMM

and reduce memory access overhead, ApSpGEMM uses an adaptive thread allocation algorithm for the three types of panels from the matrix splitting. Considering the computational differences between the CPU and GPU, we introduce the concept of core affinity to adaptively allocate panels across heterogeneous cores. Lastly, we proposed an asynchronous multiplication approach and carefully designed scheduling strategies to overlap the multiplication and transmission of panels across different cores.

We evaluated the computational performance of matrix multiplication for a set of sparse matrices selected from the matrix collection. Our experimental results demonstrate that, in the majority of matrices, ApSpGEMM improves performance by 1.12 to 2.31 times compared with the state-of-the-art in-core GPU method. Through the implementation of asynchronous overlap scheduling and adaptive panel allocation, heterogeneous collaboration further enhances the multiplication of large-scale matrices by 2.25 to 7.21 times.

Contribution In summary, we make the following contributions.

- We introduce ApSpGEMM, an adaptive-panel-based heterogeneous collaborative approach, aimed at addressing the challenges posed by the irregular distribution of NZs and memory limitations encountered in large-scale SpGEMM computations.
- We present a comprehensive four-step design, including Matrix Pre-analysis, Matrix Splitting, In-core Computation, and Heterogeneous Collaboration to optimize the SpGEMM.
- We introduce the novel concept of core affinity analysis, which carefully considers the impact of sparsity on panel multiplication performance across different cores, thereby enabling the development of an adaptive allocation scheme for panels.
- We propose an asynchronous multiplication approach and scheduling strategies to effectively overlap the multiplication and transmission of panels across different cores, mitigating the impact of temporary data transfer latencies.
- Through extensive experiments, we demonstrate that our method achieves significantly higher GFlops compared with existing methods for the $C = AA^T$ operation, and large-scale matrices perform better on heterogeneous cores than on single devices.

2 Background and Motivation

2.1 SpGEMM and Row-Row Algorithm

This study concentrates on the ***compressed sparse rows*** (**CSR**) format, which is the most commonly employed data structure for representing sparse matrices [14]. As shown in Figure 1, the CSR format comprises two position index lists: "rowptr" and "colptr", along with a list of data. Specifically, the "colptr" contains the column indices of the NZs, the "data" stores all of the NZs values, and the "rowptr" indicates the starting position of each row's NZs in the "data".

In SpGEMM, the computation of the sparse matrices A and B, and the resultant matrix C is represented by $C_{ij} = \sum_k A_{ik} \cdot B_{kj}$. Currently, a widely adopted algorithm for sparse matrix multiplication is Gustavson's row-row formulation [13], as shown in Algorithm 1.

2.2 Motivation of This Work

In the field of scientific computing, there are large-scale sparse matrix computations. For example, "nlpkt200", "uk-2002", and "stokes", originating from deep learning, graph processing, and semiconductor technology, respectively, have NNZs reaching billions. The NNZs in the resulting matrix $C = A \times A$ is 5 to 10 times greater than that of matrix A itself. Although many efforts have been devoted to optimizing SpGEMM computation [38, 43], these methods assume sufficient computation core memory [4, 10]. A natural approach is to evenly split large matrices into tiles for iterative computation, but this wastes CPU computing resources and leads to load balancing and memory access issues on GPUs, resulting in low computational performance.

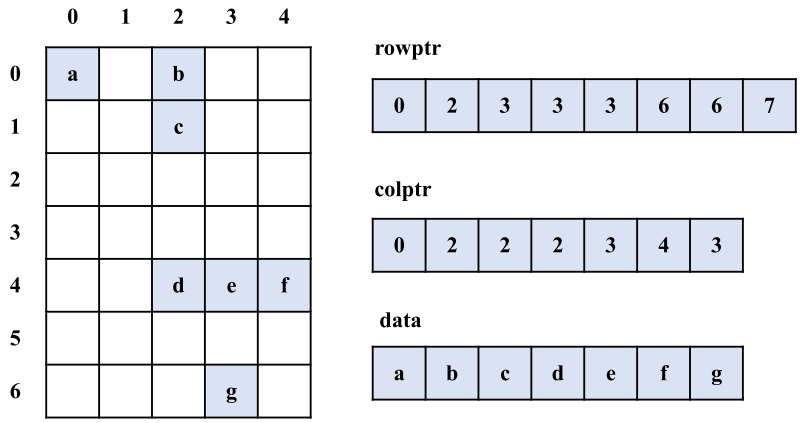


Fig. 1. Matrix CSR storage format.

ALGORITHM 1: Gustavson's Row-Row SpGEMM Algorithm

```
Input: sparse matrix A and B
Output: sparse matrix C
Initialize all elements in matrix C as zeroes
for all A_{i*} in matrix A do
    for all A_{ik} in row A_{i*} do
        for all B_{kj} in row B do
            value = A_{ik} × B_{kj}
            if C_{ij} ∉ C_{i*} then
                insert(C_{ij}, C_{i*})
                C_{ij} ← value
            else
                C_{ij} ← C_{ij} + value
            end if
        end for
    end for
end for
```

This work attempts to address the above issues through CPU and GPU collaboration. However, heterogeneous collaborative SpGEMM faces the following challenges: (1) imbalance of sparse matrix loads and irregular memory access on the GPU due to the distribution features of NZs, (2) different matrix computation latency on the CPU and GPU due to variations in sparsity, and (3) additional latency overhead from temporary data transfers between heterogeneous cores.

We have designed a four-step approach ApSpGEMM to address the aforementioned challenges. Based on the features of NZs distribution, ApSpGEMM split the matrix into sub-matrix named panels, where the size of the panels depends on the splitting algorithm and the matrix itself. These panels serve as the fundamental units for computation and transmission on both GPU and CPU.

3 Overview

As depicted in Figure 2, the flow of ApSpGEMM includes two stages. The first stage involves preprocessing the sparse matrices, consisting of the Matrix Pre-analysis and Matrix Splitting steps.

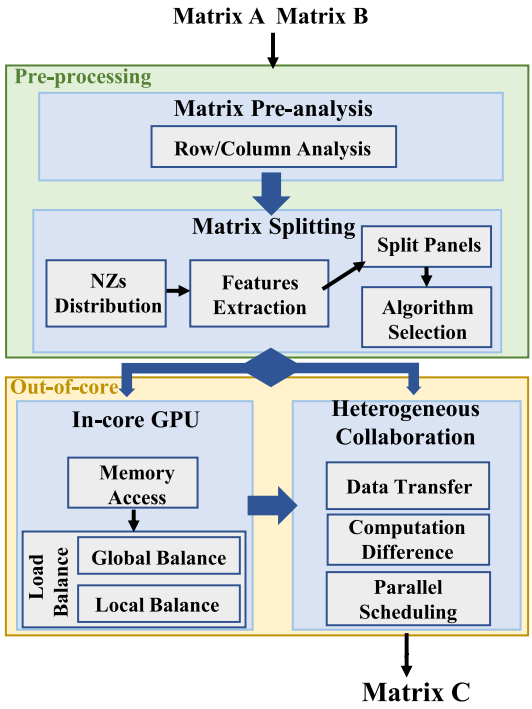


Fig. 2. The workflow of ApSpGEMM. Matrix pre-analysis to obtain matrix features, matrix splitting to generate panels, in-core computation to accomplish specific panels multiplication, and heterogeneous collaboration implements parallel computing between CPU and GPU.

The second stage is out-of-core computing, which comprises the In-core and Heterogeneous Collaborative computing steps.

Matrix Pre-analysis. Matrix Pre-analysis is the foundational stage, providing crucial insights into the distribution of NZs within the matrix. This information guides subsequent steps such as Matrix Splitting and In-core GPU computation. However, it is noteworthy that the analytical cost of this stage should be balanced against potential performance gains. For example, in the case of nsparse [28], allocating approximately 30% of the execution time for pre-analysis to analyze the features of NZs distribution underscores the need for a fast, efficient, and lightweight analyzer capable of extracting essential feature information swiftly.

Matrix Splitting. As illustrated in Figure 3, these matrices exhibit various distributional features for NZs, such as band distribution, centralized distribution, and diagonal distribution. These features significantly impact computational efficiency in the core. For example, in the "cite-Patents" matrix of Figure 3(a), loading various rows into thread blocks directly can lead to some blocks being underutilized, while others become densely occupied. Hence, it's crucial to devise a strategic algorithm based on the distinct distributional features of NZs. This algorithm optimally splits the matrix to make the most of the computational and storage resources on the target core.

In-core GPU. This step mainly involves multiplying panels from matrix A with panels from matrix B on GPU. Several crucial considerations apply here. Firstly, the unique distribution features of the panels require implementing different multiplication operations tailored to panels with different features. Secondly, determining optimal panel sizes for thread blocks is a challenging task. Lastly, carefully allocating thread blocks and threads is crucial for memory access and load balance.

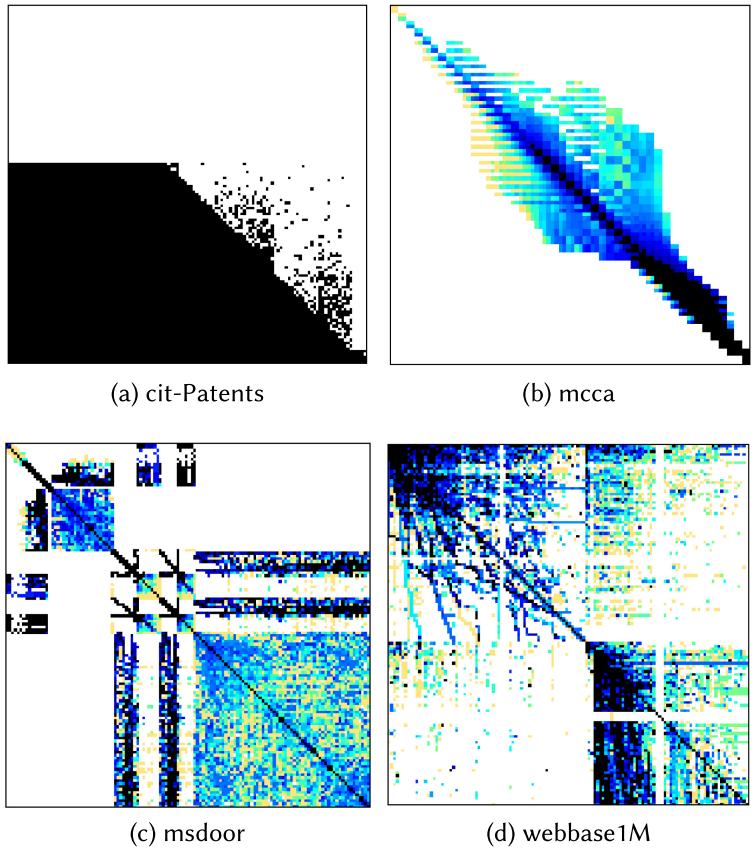


Fig. 3. NZs distribution patterns of common matrices.

Heterogeneous Collaboration. When dealing with large-scale matrices that exceed the available GPU graphics memory capacity, the concurrent use of multiple heterogeneous cores proves effective. However, this approach introduces challenges like transfer latency, differing computational capabilities, and the complex coordination of scheduling. Therefore, strategies are needed to allocate panels of varying sizes and features between the GPU and CPU. It's also crucial to minimize communication and computation latency in this heterogeneous computing paradigm.

Considering the general evaluation metrics for matrix computations, the objective of Ap-SpGEMM centers on addressing large-scale SpGEMM computations across two stages while minimizing the total computational time, denoted as T. Mathematically, this optimization problem can be formally expressed as Equation (1):

$$\begin{aligned} Argmin(T) =& t_1(pre\text{-}analysis) + t_2(splitting) + t_3(In\text{-}core\ computation) \\ &+ t_4(heterogeneous\ collaboration), \end{aligned} \tag{1}$$

where each component t_i signifies the time associated with its respective step.

It should be noted that the panels generated during the matrix splitting stage will impact the computational performance in the second stage. While we analyze the distribution features of NZs in Figure 3, we do not adopt band or localized centralized splitting methods because these approaches cannot be extended to general matrices. Instead, we opt for the versatile approach of

sparsity splitting. Its inherent adaptability allows us to efficiently handle matrices with different features, playing a crucial role in achieving the seamless integration of in-core computation and heterogeneous collaboration steps.

4 Matrix Pre-Processing

In this section, we describe the matrix pre-analysis and matrix splitting steps. We propose lightweight analysis algorithms and sparsity splitting rules to generate panels associated with the distribution features of NZs.

4.1 Matrix Pre-Analysis

Many works design multiplication algorithms for matrices with different NZs features. The result is that they incur high latency to analyze the distribution of NZs. For example, Rasouli et al. [32] not only count the NNZs in rows and columns for diagonal matrices, but also calculate the NNZs and positions on the diagonal. While the overhead of computing diagonal NNZs and positions is tolerable for small matrices, it becomes quite time-consuming for larger matrices and loses scalability on general matrices. Therefore, we need to extract as little information as possible without considering the NZs distribution features.

For each input matrix, we methodically extract the following pertinent information: (a) the NNZs in each row of matrix A, (b) the NNZs in each row of matrix B, and (c) the NNZs within each column of matrix A.

By utilizing the prefix sum algorithm for the inherent rowptr index in matrix A, we can quickly determine the number of NNZs in each row. Similarly, we can obtain the NNZs for each row in matrix B.

Furthermore, owing to the inherent characteristic of the CSR format wherein NZs within the same column share identical indexes in the "colptr", we can expediently determine the NNZs present in each column.

The collected information will guide the optimization of the next three steps. Further elucidation of the utilization of this critical information is expounded upon in subsequent sections of this academic exposition.

4.2 Matrix Splitting

To categorize the distribution of NZs features effectively, we use a sparsity criterion denoted as S. It splits the matrix into two distinct categories—sparse panels and dense panels—based on the level of sparsity. Sparsity S is defined as the ratio of NNZs to the total number of elements in the matrix. Specifically, for a given row i in the matrix with a total number of elements denoted as E_i and NNZs as NNZ_i, S is formally expressed as the quotient NNZ_i/E_i.

During the splitting of rows within matrix A, a crucial step involves reordering these matrices, as an example in Figure 4. Specifically, first, the sparsity S for each row is computed using information obtained from the matrix pre-analysis step. Subsequently, a descending order reorder of all the rows in matrix A based on their respective S is carried out using a simple and efficient Quick Sort algorithm, while ensuring the retention of "colptr" and "data" lists in matrix A. Although Quick Sort is unstable, it only alters the relative positions of indexes with the same S, which does not affect the splitting of the matrix. Similarly, matrix B is also reordered based on the S. It's worth noting that our reordering operations pertain exclusively to the matrix indices, optimizing reordering efficiency. Following the reordering, rows with S exceeding the row sparsity threshold P are classified as dense panels, otherwise, they are classified as sparse panels. In most applications, a matrix is considered sparse if the proportion of NZs is less than 5%. Therefore, P is typically valued at 0.05.

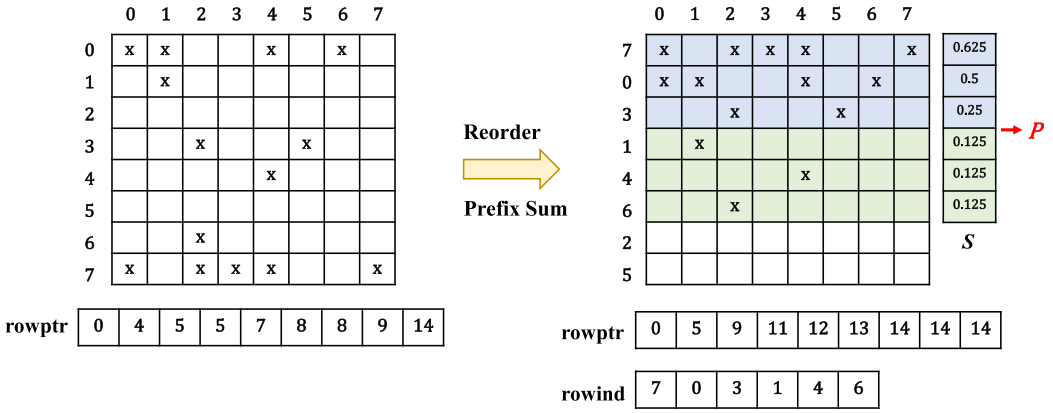


Fig. 4. The S and P are the sparsity and row sparsity thresholds, respectively. The S value for each row is calculated through its NNZs, which are obtained from the prefix sum algorithm in the matrix pre-analysis step. The matrix is reordered in descending order based on sparsity. The "rowptr" is reordered to get the new matrix "rowptr", and the "rowind" stores the row index of the old matrix corresponding to the new matrix.

After the reordering, the matrices A and B are now organized into distinct components known as dense panels and sparse panels. During computation, three types may arise: multiplication of sparse panels by sparse panels (SpGEMM), multiplication of sparse panels by dense panels (SpMM), and multiplication of dense panels by dense panels (DGEMM). It's important to note that while we refer to these components as panels, the operations within them are essentially matrix multiplications. We use symbols A^p and B^p to refer to panels from matrices A and B. Subsequent sections of this article will detail optimization techniques relevant to these three distinct types of panel multiplication.

5 Out-of-Core Computation

5.1 In-Core Computation

In Section 2.1, to parallelize Gustavson's algorithm on the GPU, the critical task is parallelizing the two for-loops at lines 2 and 3 within Algorithm 1. This means that in the GPU kernel, each thread block is responsible for computing panel multiplication involving distinct rows. Each thread within a block computes the corresponding NZs within these rows. Efficient parallelization depends on meticulous consideration of two key aspects for blocks and threads: load balancing and memory access optimization.

(a) Effective management of rows and NZs is central to load balancing. Allocating an optimal workload to each thread block is fundamental, as excessive workloads may lead to frequent global memory accesses to transfer rows to shared memory, causing a performance bottleneck. Conversely, an overly conservative allocation may result in inefficient use of shared memory resources. Within a single block, maintaining a balanced distribution of workload among threads is also vital. Imbalances can lead to some threads bearing excessive computational loads while others are underutilized. Thus, achieving global and local load balance is paramount in GPU parallelization.

(b) Memory access involves threads retrieving rows from panel B^p (as referenced by the NZs of panel A^p) from global memory and loading these rows into the shared memory. This means that each NZ in the A^p requires one global memory access. This can result in a significant overhead in memory access operations, especially when compared with the computation itself.

To address load balancing and memory access challenges and lay the groundwork for Section 5.2, we propose the following solutions for SpGEMM, SpMM, and DGEMM:

SpGEMM. We use a systematic approach to achieve global load balance. This involves categorizing contiguous rows from A^p into discrete bins. We leverage the knowledge from the matrix pre-processing step, generating multiple bins, which are subsequently allocated to distinct thread blocks. Each thread block is responsible for executing the multiplication operation for its designated bin.

Previous methods used atomic operations to allocate a fixed number of adjacent rows into the corresponding bins [23, 28]. Although this method is simple, it is inefficient. Firstly, it performs unnecessary binning operations for entirely empty rows. Secondly, it focuses on the number of rows rather than the memory size occupied by the rows, which leads to wasted shared memory space. We attempt to address the aforementioned issues through a dynamic binning approach, where each bin is allocated a different number of rows based on the size of the shared memory. However, dynamic binning requires sufficient prior knowledge to find suitable rows. Fortunately, the matrix pre-analysis and reordering steps solve this problem. Given a panel, not only are all rows reordered according to a descending order rule, but the NNZs of each row are also known. Within the limits of shared memory size, deciding the number of rows for each bin is straightforward.

The goal of local load balancing is to allocate threads within a given block so that each thread is responsible for processing certain NZs in a row of A^p. This means that the threads need to call CUDA cores to access global memory to retrieve the relevant NZs of B^p and perform element-wise multiplication on the NZs they are responsible for. Therefore, allocating the optimal number of threads for a row is crucial to ensuring efficient memory access and fast element-wise multiplication operations.

On the one hand, allocating an insufficient number of threads for one row in A^p would require multiple iterations per thread, leading to suboptimal memory access and frequent CUDA core launches. Conversely, allocating an excessive number of threads could result in a notable surplus of idle threads. On the other hand, although matrix reordering and dynamic binning reduce the risk of thread load imbalance to some extent, the differences in NNZs between rows still exist. Therefore, if a fixed thread allocation scheme is used for each row, the long and short rows within the panel will still affect the balance of thread workloads. However, using a dynamic thread allocation method, such as adaptively assigning the number of threads for each row, would introduce significant decision-making overhead.

After weighing the pros and cons of the two thread allocation strategies, our goal is to ensure that each thread's iteration count closely matches the number of rows that the respective thread group must handle. In a thread block with T threads, we divide it into G groups, each containing M threads, where $M = T/G$. As shown in Figure 5, the local load balancing strategy encompasses various thread group configurations. For example, when $M = 8$, it requires 4 iterations; when $M = 4$, it requires 3 iterations, and when $M = 2$, it involves 4 iterations to complete the processing tasks, as depicted.

Using insights from the pre-analysis of the matrix, we start by initializing the value of M based on the average row length across all rows in A^p. We then proceed to systematically assign these thread groups to A^p. However, given the potential existence of long rows, it's crucial to note that the maximum iteration count a thread in such a row would need to perform is defined as $iter_{\text{thread}} = NNZ_{\max}/M$. To ensure that this maximum iteration counts closely aligns with the number of rows, denoted as n_{rows}, that each group needs to process, we make an adjustment to the value of M using a heuristic approach, as shown in Algorithm 2.

SpMM. The operation involves multiplying a sparse panel A^p by a dense panel B^p, resulting the output panel denoted as Y, as depicted in Figure 6(a). Unlike sparse B^p, the NZs of dense B^p

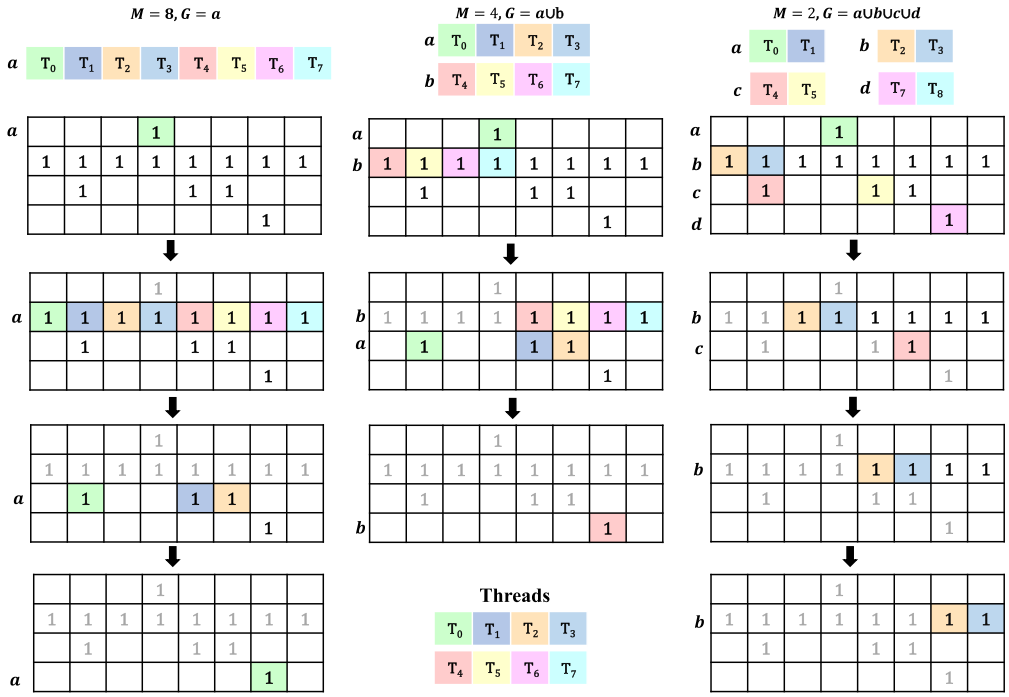


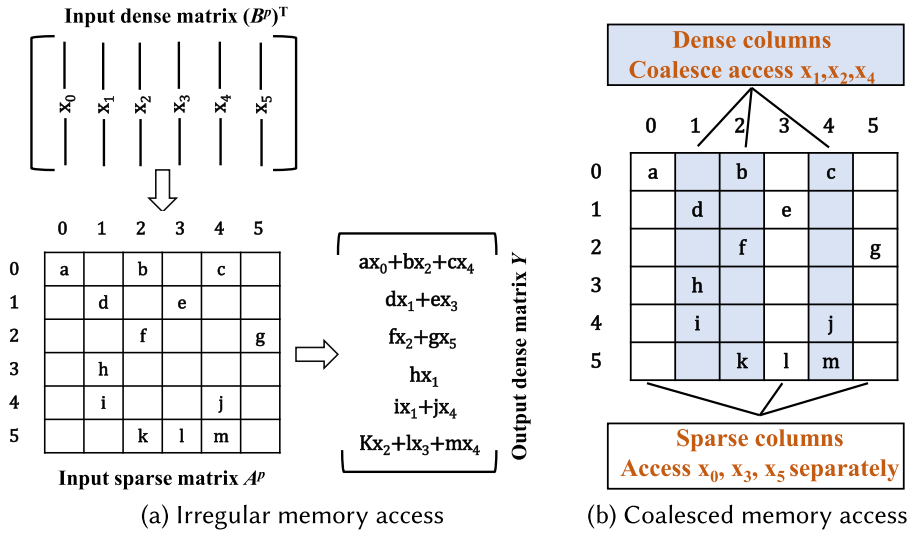
Fig. 5. Local load balancing using different group sizes M. A thread block has 8 threads. The three columns represent the three grouping schemes. $M = 8$, it requires 4 iterations, the maximum number of iterations. $M = 4$, it requires 3 iterations. $M = 2$, it requires 4 iterations, but coalesced memory access is poor.

ALGORITHM 2: Adjusting Thread Allocation

1: Random initialize M
2: **Input:** $NNZ_{max}M$
3: calculate $iter_{thread} \leftarrow \frac{NNZ_{max}}{M}$
4: **while** $iter_{thread} \gg 2 \times n_{rows}$ or $iter_{thread} \ll 2 \times n_{rows}$ **do**
5: **if** $iter_{thread} > 2 \times n_{rows}$ **then**
6: $M \leftarrow M \times (\frac{n_{rows}}{2 \times iter_{thread}})$
7: **else if** $iter_{thread} < 2 \times n_{rows}$ **then**
8: $M \leftarrow M \times (\frac{iter_{thread}}{2 \times n_{rows}})$
9: **end if**
10: **if** $G > NNZ_A$ **then**
11: $G \leftarrow NNZ_A$
12: **end if**
13: **end while**

are contiguous. This means that in Algorithm 1, the middle loop (line 3) reads a small amount of NZs from A^p, while the inner loop (line 4) reads a large amount of NZs from B^p. As a result, the NZs from A^p occupy only a small amount of space in shared memory, while most of the space is occupied by the NZs of B^p. Therefore, the main memory access overhead comes from the frequent exchange of B^p's NZs between global memory and shared memory.

To address this issue, our approach focuses on improving the reutilization of NZs within A^p. We observed that NZs sharing the same column in A^p can simultaneously access elements of the same row in B^p from global memory. Consequently, the number of times NZs in the same column



(a) Irregular memory access (b) Coalesced memory access

Fig. 6. The (a) represents the process of a sparse panel A^p multiplied by dense panel B^p to obtain the resulting panel Y. In this case, the NZs of A^p are scattered, and the thread accesses memory irregularly. The (b) represents the dense columns 1, 2, and 4 that are accessed by coalesced memory.

of A^p access global memory can be reduced to once. Additionally, the more NZs there are in the same column of A^p, the greater the benefit derived from this process. Based on this observation, we define a dense column as one with more than one NZ, and otherwise, it is considered a sparse column. The upper limit of NZs in dense columns depends on the thread group size and the size of shared memory. We assume the shared memory size is S, the number of threads in the group is M, the number of columns in B^p is K, and the average NZs in dense columns is Q, with each NZ represented as 8 bytes of storage. Therefore, the NZs read by the threads must satisfy $4(MK + MQ) < S$. Thus, the size of a dense column is $2 < Q < \frac{S}{4M} - K$.

In dense columns, threads access global memory to load the corresponding rows from B^p into shared memory, and NZs in these dense columns share the data from the referenced rows. As exemplified in Figure 6(a), A^p contains 13 NZs, requiring a total of 13 global memory accesses to load the corresponding rows in B^p. In contrast, in Figure 6(b), dense columns (columns 1, 2, and 4) only require three threads to access global memory, totaling three times, to load the vectors x_1, x_2, and x_4, respectively. For sparse columns, it takes four global memory accesses to load the vectors x_0, x_3, and x_5. The number of global memory accesses is reduced from 13 to 7.

DGEMM. Dense panels multiplication efficiency is significantly higher than that of SpGEMM and SpMM in the kernel. Dense panels inherently possess a high degree of data reuse and follow a sequential element access pattern. As a result, in the kernel, there is no need for specific discussions regarding memory access and thread allocation to multiply dense panels with every row/column. Currently, there is ongoing work aimed at reducing the computational complexity of DGEMM by decreasing the number of multiplication operations, such as AlphaZero [9] and Strassen. ApSpGEMM focuses on kernel-level optimization, while AlphaZero and Strassen focus on operator-level optimization, and the two are not in conflict. Therefore, while we do not specify which DGEMM algorithm is optimal, these works are compatible with ApSpGEMM. Incorporating Strassen into the kernel configuration of the ApSpGEMM to accelerate dense panels multiplication is a natural choice.

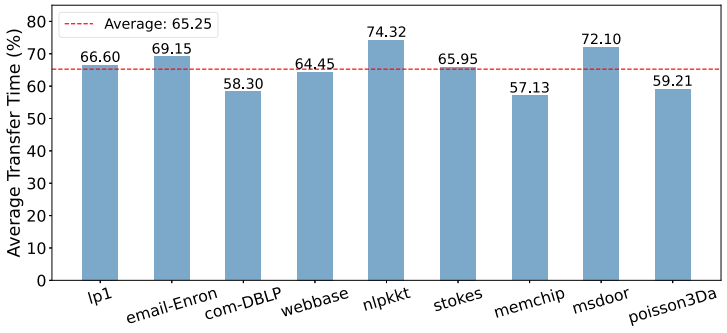


Fig. 7. The percentage of data transfer time to the total time. The red dashed line represents the mean value of 76.13%.

Configuration. ApSpGEMM uses the maximum available shared memory (128 KB on RTX 3080) and the largest kernel size (1024 threads) to guarantee full hardware utilization. It is important to note that the shared memory size rated for the RTX 3080 is 64 KB, and the L1 cache size is also 64 KB. ApSpGEMM combines shared memory and L1 cache, which are shared among 1024 threads. During In-core Computation, some data is read-only, such as the NNZs information corresponding to rows or columns, the sparsity list S, and the rowind indices. These data are characterized by being frequently accessed and read-only during kernel execution. To maximize the storage of NNZs data in shared memory, ApSpGEMM defines the above read-only data in "__*constant*__" and stores them in the constant memory (64 KB).

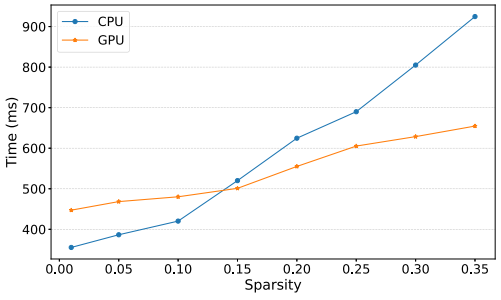
ApSpGEMM loads the target bin into shared memory, while adjacent bins are preloaded into the L2 cache, thereby reducing global memory access overhead. Additionally, a group may consist of one or more warps. On the RTX 3080, each warp is fixed at 32 threads by default. Therefore, in ApSpGEMM, the number of threads in a group is usually set as a multiple of 32.

5.2 Heterogeneous Collaboration

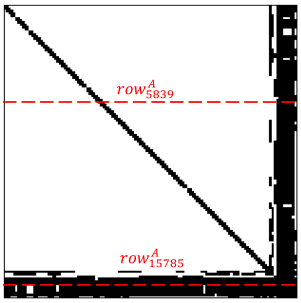
As described in Section 2.2, there are two key challenges in the heterogeneous collaboration step:

(1) The computational latency differs between CPU and GPU for panels with different distribution features. Synchronizing the computation time between the CPU and GPU to ensure they complete almost simultaneously maximizes parallel processing performance. Therefore, it requires establishing a discerning criterion for adaptively allocating panels to the appropriate cores.
(2) Empirical investigations using matrices from the SuiteSparse Matrix Collection, executed on an NVIDIA GeForce RTX 3080 GPU and an Intel(R) Xeon(R) Gold 5117 CPU, have revealed a significant presence of data transfer overhead between these heterogeneous cores, as illustrated in Figure 7. In light of this observation, optimizing the data transfer process becomes a pivotal consideration.

Panels with varying sparsity levels from Section 4.2 are allocated for computation on either the CPU or GPU. To empirically evaluate the performance implications of sparsity levels on computation, we conducted a series of tests measuring computation times for panels with varying sparsity levels, utilizing both the GPU NVIDIA GeForce RTX 3080 and the CPU Intel(R) Xeon(R) Gold 5117, as elucidated in Figure 8(a). The outcomes of these experiments reveal a discernible trend, wherein panels featuring higher sparsity levels demonstrate superior suitability for GPU-based computations, whereas those with lower sparsity levels exhibit enhanced compatibility with CPU-based



(a) The affinity trend



gupta3	row^A_{5839}	row^A_{15785}
Row number	5839	15785
Sparsity	0.0095	0.1897
Affinity	0.0095	0.1897
CPU(ms)	6.7	114.6
GPU(ms)	9.1	76.8

(b) Gupta3 affinity example

Fig. 8. The (a) represents the difference in computation time for matrices of the same size but different sparsity levels on CPU and GPU. The (b) represents the computation time on the CPU and GPU when multiplying matrix A (gupta3) by the transpose of matrix B for different affinity rows. In the table, "row^A_{5839}" refers to row 5839 of matrix A being multiplied by row 5839 of matrix B, and "row^A_{15785}" follows the same logic.

computations. We refer to this trend as panels affinity, where $Affinity = \frac{Sparsity(A^p)+Sparsity(B^p)}{2}$. Figure 8(b) demonstrates the impact of affinity on SpGEMM in heterogeneous processors in real-world applications. We illustrate this process by multiplying the matrix "gupta3" from an optimization problem domain by its transpose. The "gupta3" matrix has 16,783 rows and columns with 9,323,427 NZs. The table in Figure 8(b) shows that the computation time for the multiplication of row^A_{5839} is smaller on the CPU, whereas for row^A_{15785}, the computation time is smaller on the GPU.

To synchronize the computation times on heterogeneous cores, determining an allocating ratio for panels becomes crucial. This ratio defines how panels are allocated for processing on the CPU versus the GPU, aiming to minimize the discrepancy in their computation times. We define computational coefficient as $K = CPU_{exe}/GPU_{exe}$, where CPU_{exe} and GPU_{exe} represent the respective computation times for the CPU and GPU to perform a single multiplication operation. Consequently, we derive the GPU-to-CPU allocation ratio as $R = K/(K + 1)$. Based on our experimental observations, we have determined that setting R within the range of 60% to 65% yields optimal performance for the input matrix on our specific experimental platform. It is important to note, however, that the optimal value of R may be influenced by changes in the configuration of the CPU or GPU. Therefore, there may be a need to adjust R to match different hardware setups. Nonetheless, the fundamental principle of selecting an appropriate R to distribute computational tasks between the CPU and GPU remains an important consideration.

As shown in Figure 7, it is clear that the time needed for data transfer between the CPU and GPU exceeds the time allocated for computational tasks. This disparity can be attributed to the inherent limitations of the PCI-e architecture, which serves as the interconnect between these heterogeneous cores. In this architecture, each data transfer direction is handled by a single engine, restricting the CPU and GPU to perform data transfers in only one direction at a time. This inherent constraint introduces the potential for data transfer bottlenecks during concurrent operations.

To improve efficiency and mitigate this issue, we leverage the concept of computation and transfer overlap. Each computation on the heterogeneous cores involves two separate data exchanges: first, the transfer of CPU panels to GPU memory, and then the transmission of computed results from the GPU back to the CPU memory. By carefully coordinating these operations, we establish a synergistic relationship between computation and data transfers. This helps minimize waiting overhead and maximizes the potential for asynchronous concurrency.

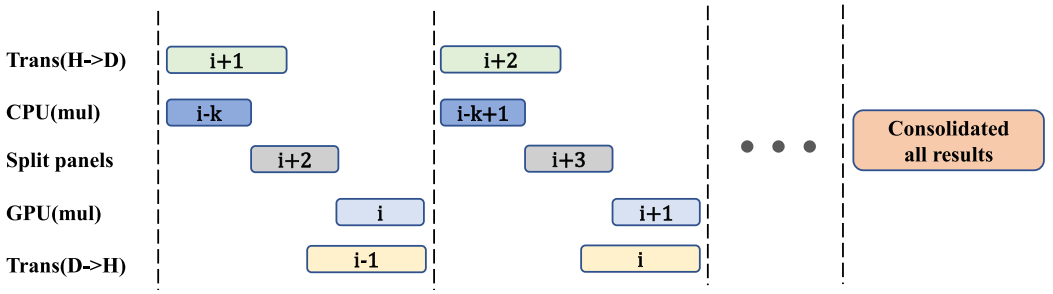


Fig. 9. Each interval between dashed lines represents a round of overlapping time. The time for transferring the (i+1)th panels from the CPU to the GPU and the (i-1)th results from the GPU to the CPU overlaps with the time taken for the CPU and GPU to compute the panels, as well as the time required for the step-wise matrix splitting. After multiple rounds, all intermediate results are consolidated.

As depicted in Figure 9, our approach to achieving overlap involves meticulous synchronization of the i-th computation with both the (i-1)th GPU-to-CPU transfer and the (i+1)th CPU-to-GPU transfer. This synchronized orchestration ensures that while the GPU is actively engaged in the computation of the current iteration (i-th), the CPU concurrently executes two essential tasks: receiving the results of the subsequent iteration (i-1)th and preparing to transfer data from the preceding iteration (i+1)th. This simultaneous handling of data transfers and computational tasks optimizes the temporal efficiency of the entire process, as exemplified in the figure.

Furthermore, incremental matrix splitting offers unique advantages over the notion of splitting the entire matrix in a single step followed by panel transfer. This step-by-step approach maintains the effectiveness of the splitting process while strategically allowing overlap between the splitting and data transfer operations. This concurrency facilitates the simultaneous execution of panel splitting and data transfer, leading to improved overall efficiency in the matrix multiplication process.

Algorithm 3 describes the panels' allocation process. Line 6 represents the result of averaging the sparsity of one row from matrix A with one row from matrix B, followed by sorting. This means that each pair $\{row_i^A, row_j^B\}$ in Row_Pair[ ] is ordered by their affinity for the GPU, from high to low. Lines 8–14 represent the allocation of panels for GPU and CPU based on affinity and the allocation ratio. The index corresponding to R serves as the dividing line. The row pairs before this index are allocated as panels for the GPU to perform in-core computation, while the row pairs after this index are calculated by the CPU. Notably, the method of overlapping computation and transfer is applied in lines 10 and 11.

6 Evaluation

6.1 Experimental Setup

We conducted experiments using the NVIDIA GeForce RTX 3080 and the Intel(R) Xeon(R) Gold 5117 platforms. Table 1 provides detailed specifications for both setups. The host operating system was Ubuntu Linux 18.04. We utilized driver version 470.57.02 and the CUDA 11.6 toolkit for GPU implementations. Compilation was done with NVCC compiler version 11.6.124. In our comparative analysis, we benchmarked ApSpGEMM against cuSPARSE,[2] AC-SpGEMM [40], spECK [31], and TileSpGEMM [29].

[2]Nvidia CuSPARSE. https://docs.nvidia.com/cuda/cusparse/

Table 1. Experimental Platform Specifications

Component	GPU	CPU
Device	NVIDIA GeForce RTX 3080	Intel Xeon Gold 5117
CUDA Cores	8960	N/A
Core Clock Speed	1.71 GHz	2.00 GHz
VRAM	12 GB GDDR6	N/A
Memory Bandwidth	760.0 GB/s	115.2 GB/s
CPU Cores	N/A	14
CPU Clock Speed	N/A	2.00 GHz
RAM	N/A	128 GB DDR4-2400
Max Threads	1024/Block	28

ALGORITHM 3: Adaptive Panel Allocation Algorithm

Require: Sparse matrices A, B, computational coefficient K
Ensure: Result matrix $C = A \times B$

Step 1: Preprocessing
Analyze NNZs and Reorder
Calculate GPU-to-CPU allocation ratio $R = \frac{K}{K+1}$
Step 2: Panel Allocation and Computation
for row_i^A in matrix A and row_j^B in matrix B **do**
    Row_Pair[ ]=Sparsity_Mean_Sort(row_i^A, row_j^B)
end for
for $G = 0$ to $R \times$ length(Row_Pair)-1 **do**
    overlapping computation and transfer\
    Assign Row_Pair[G] to GPU
    In-core Computation
end for
for $C = R \times$ length(Row_Pair) to length(Row_pair) - 1 **do**
    Assign Row_Pair[C] to CPU
end for
Combine final result matrix C

6.2 Matrices Datasets

Our computational efforts focused on the SpGEMM operation denoted as $C = AA^T$, aligning with established conventions in prior SpGEMM research [29, 31, 40]. We selected experimental matrices from the SuiteSparse Matrix collection[3] and the Network Repository.[4] Specifically, we selected matrices with NNZs ranging from 3 million to 150 million for matrix C. The matrices designated for the execution of heterogeneous cores were excluded. Approximately 400 matrices met our criteria in both repositories. However, many were duplicates, resulting in 273 distinct matrices.

For precision, we used double data type in our experiments. We meticulously chose nineteen matrices for detailed analysis. The first nine underwent SpGEMM exclusively on the GPU, while the remaining nine, due to their size, underwent SpGEMM on both the CPU and GPU. Table 2

[3]SuiteSparse Matrix Collection. https://sparse.tamu.edu/
[4]Network Repository. https://networkrepository.com/

Table 2. Here are Nineteen Representative Matrices

Type	Matrix	Abbr.	n	NNZs(A)	flops(AA^T)	NNZs(AA^T)	Compression Ratio
Small Matrices	msdoor	msd	0.42	19.17	2084.38	62.82	16.59
	memchip	mem	2.70	13.34	138.96	29.27	2.37
	webbase-1M	web	1.00	3.10	139.12	51.01	1.36
	af_shell10	af	1.50	52.70	3681.66	142.70	12.90
	poisson3Da	poi	0.01	0.35	23.61	3.00	3.90
	human-gene1	human	0.02	24.67	143246.27	224.32	319.29
	pdb1HYS	pdb	0.04	4.30	1111.49	19.61	28.34
	delaunay_n19	dela	0.52	3.15	39.53	10.86	1.82
	hood	hood	0.22	9.89	1124.06	34.24	16.41
	nd3k	nd3k	0.009	3.28	2551.94	18.49	69.00
Large-scale Matrices	nlpkkt	nlp	16.24	440.23	24932.82	2425.94	10.28
	stokes	stokes	11.45	349.32	9424.18	2115.15	4.46
	ljournal-2008	lj	5.36	79.02	7828.66	4245.41	1.84
	soc-LiveJournal1	soc-lj	4.85	68.99	5915.63	3366.05	1.76
	com-LiveJournal	com-lj	4.00	69.36	8580.90	4859.09	1.77
	cage15	cage	5.15	99.20	9564.11	4177.25	2.29
	wikipedia-20070206	wiki0206	3.57	45.03	12796.04	4802.94	2.66
	wikipedia-20060925	wiki0925	2.98	37.27	10030.09	3750.38	2.67
	uk-2002	uk	18.52	298.11	29206.61	3194.99	9.14

For each matrix, the values of "n", "NNZs(A)", "flops(AA^T)", and "NNZs(AA^T)" are all expressed in millions (M).

provides a comprehensive overview of these matrices. "Abbr" stands for matrix name abbreviation. "n" represents the number of rows (and columns) in matrix A, while "NNZs(A)" indicates the total non-zero elements. "flops(AA^T)" measures the floating-point operations needed for AA^T multiplication, considering the multiply-add operation as two flops. "NNZs(AA^T)" is the total NZs in the resulting matrix C. Finally, the "Compression Ratio" parameter shows the ratio of half the flops to the NNZs(AA^T) ratio, indicating the average number of floating-point operations required to generate a NZ in the resulting matrix.

6.3 In-Core GPU Performance

Performance Comparison. Figure 10 provides a performance comparison between our proposed method and four prominent SpGEMM techniques using 273 matrices where C is computed as $C = AA^T$ on RTX 3080. Notably, cuSPARSE, a vendor-provided library, shows limitations in handling a subset of matrices in the dataset. In contrast, AC-SpGEMM, spECK, TileSpGEMM, and our method demonstrate a broader capability, successfully performing computations on nearly all matrices.

Analyzing the data, we observe distinct trends in the performance curves. cuSPARSE starts with the lowest performance point and exhibits a gradual increase. AC-SpGEMM slightly surpasses cuSPARSE initially, while spECK consistently achieves over 40 GFlops for most matrices. TileSpGEMM and our method both approach the 40 GFlops threshold initially. Upon closer examination, our method consistently outperforms others, especially for matrices with higher compression ratios. We identify the peak performance point (dark blue point) furthest from the fitted line. The peak GFlops for cuSPARSE, AC-SpGEMM, spECK, TileSpGEMM, and our method are 65.23, 83.07, 107.07, 164.31, and 197.54 GFlops, respectively. Compared with cuSPARSE, AC-SpGEMM, spECK, and TileSpGEMM, our method achieves 3.03x, 2.38x, 1.84x, and 1.20x higher peak GFlops, respectively. Assessing average GFlops across the five methods, we find values of 25.42, 33.20, 45.90, 52.55, and 58.62 GFlops, respectively. Our method outperforms its counterparts with average GFlops that are

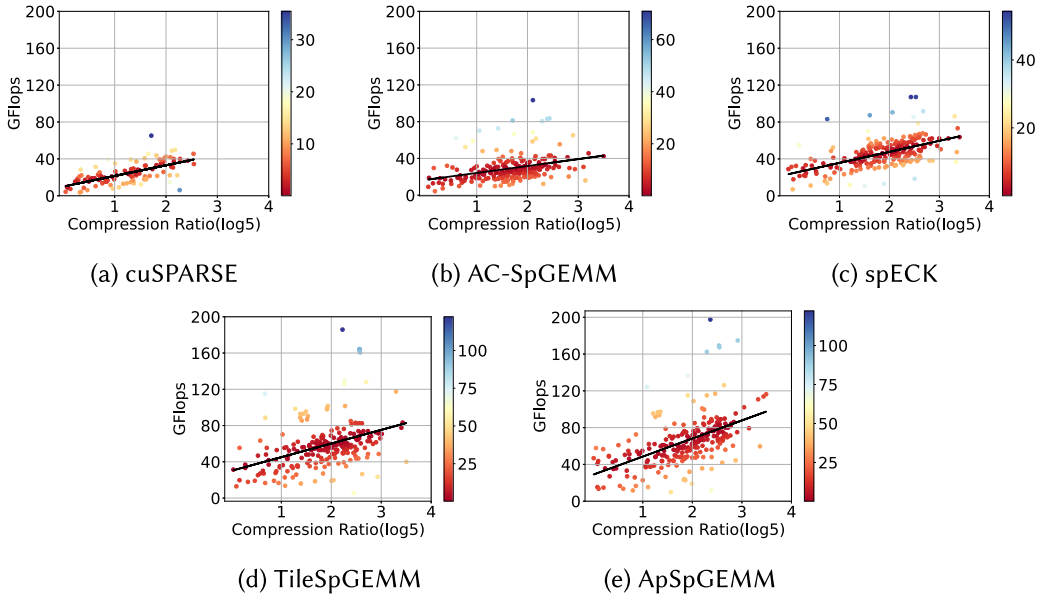


Fig. 10. Performance comparison of our method with four state-of-the-art SpGEMM methods on a 3080 GPU for $C = AA^T$ with double precision. The horizontal coordinate represents the compression ratio (log5 scale), the vertical coordinate represents performance (GFlops), and the scatterplot color represents the distance of the data points from the fitted curve.

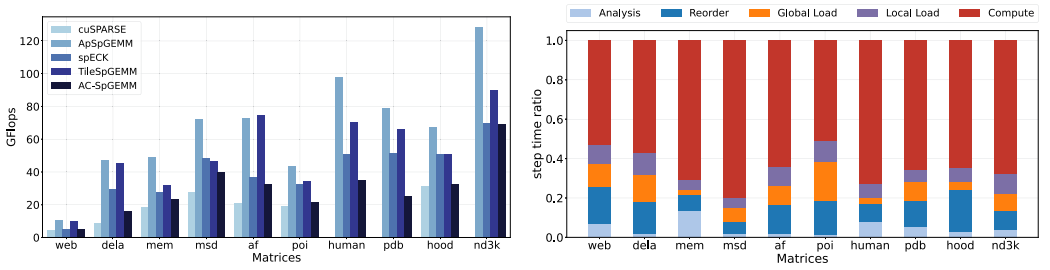
Table 3. Performance Comparison of SpGEMM and GEMM

Matrix	n	NNZs	SpGEMM (ms)	DGEMM (s)
poi	0.01	0.35	0.23	0.26
human	0.02	24.67	5.35	1.14
pdb	0.04	4.30	0.43	4.95
nd3k	0.009	3.28	0.51	0.08
pwt	0.03	0.33	0.22	3.75

2.31x, 1.77x, 1.28x, and 1.12x greater, respectively. In summary, our method consistently delivers superior computational performance for SpGEMM compared with the other four state-of-the-art methods.

In addition, Table 3 reports the comparison of SpGEMM's wall clock time cost against dense GEMM on various tasks. In these comparisons, SpGEMM uses ApSpGEMM and DGEMM uses CUBLAS. Due to GPU memory size limitations, the size of DGEMM is restricted to below 50000×50000. As shown in the results, the time cost of ApSpGEMM remains at the millisecond level, whereas GEMM is generally at the second level. The computation time of SpGEMM is positively correlated with the compression ratio, while GEMM's time is related to the number of rows and is independent of NNZs.

Single-step Time. Table 2 and Figure 11(a) offer detailed insights into the performance of the 10 small matrices at in-core GPU. A thorough analysis of Figure 11(a) highlights our method's outstanding performance, surpassing others in 8 of the 10 matrices assessed. While TileSpGEMM excels in "dela" and "af" matrices, it's worth noting our approach remains competitive even in these cases. Our method's performance boost is most noticeable in matrices with higher



(a) Thirteen GFlops of matrices. (b) Single-step time

Fig. 11. (a) Detail performance comparison of the 10 matrices at in-core GPU. (b) The percentages of execution time for the main steps in ApSpGEMM framework. Reorder belongs to the matrix splitting step; and Global load, Local load, and Compute belong to the in-core GPU step.

compression ratios. In contrast, aside from cuSPARSE, the four other methods show relatively similar performance in matrices with lower compression ratios.

Significantly, our method notably outperforms in four matrices: "mem", "msd", "human", and "nd3k". This superiority stems from two main factors. First, the remarkably high compression ratios in "human" and "nd3k" matrices provide a broader scope for our method to excel in computational efficiency. Second, the distribution of non-zeros in the "mem" and "msd" matrices naturally aligns with the reordering rule based on sparsity. This means that matrix reordering and load assignment take up a smaller portion of the overall computation time.

We conducted a series of precise experiments to comprehensively analyze the time distribution of the different steps. This study involved the execution of 10 carefully selected matrices, with a detailed recording of the time taken by each step. Figure 11(b) visually represents the percentage of time attributed to each step in relation to the overall computation for each matrix. It's important to note that Global load, Local load, and Compute are integral components of the in-core GPU computation phase. Figure 11(b) clearly shows that matrices "mem" and "msd" allocate the least time to the reordering step, while dedicating a significant portion to the computation step. On average, across all thirteen matrices, the time distribution among the steps is approximately as follows: Analysis accounts for about 5%, Reorder constitutes roughly 18%, Load Balance occupies around 16%, and Computation predominates with an approximate share of 61%.

6.4 Heterogeneous Collaboration Performance

Allocation Ratio. To verify the best CPU/GPU panels scheduling ratio R, we conducted a series of performance evaluations with nine large-scale matrices on two types of heterogeneous cores. The outcomes, shown in Figure 12, demonstrate progressive performance improvement with increasing R, from 55% to 65%. The peak performance is notably observed between 60% and 65%. However, once R exceeds this range, performance starts to decline, and there's a significant drop after surpassing the 70% threshold. It's worth noting that the optimal allocation ratio (R) varies for two heterogeneous platforms. For instance, in "lj", "soc-lj" and "com-lj", the maximum GFlops are achieved at R values of 55% and 60%, respectively.

Performance Comparison. We systematically performed a comprehensive performance analysis on three matrices, each with three distinct configurations: single-CPU, single-GPU, and *heterogeneous collaboration* (Hete) setups. For single-CPU SpGEMM, we utilized Gustavson's row-row algorithm. In single-GPU processing, we employed a straightforward chunking approach for matrix loading and subsequent SpGEMM computations. In the heterogeneous setup, we followed the approach outlined in Section 5.4 to coordinate the execution of SpGEMM operations.

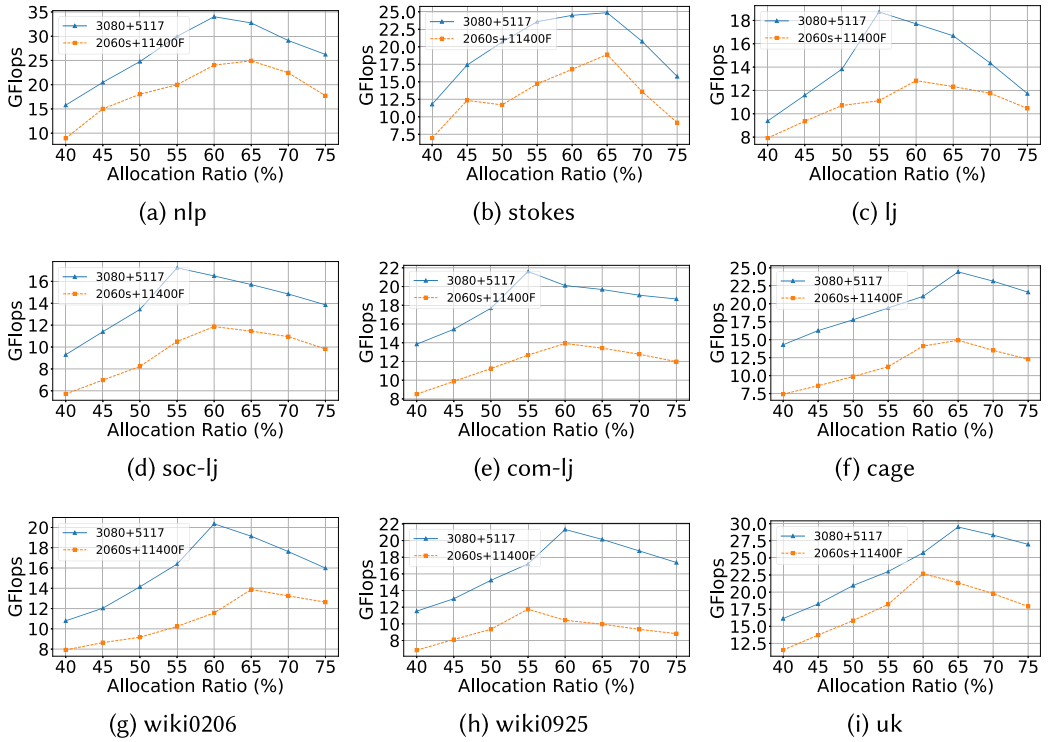


Fig. 12. Comparing the computational performance of nine large-scale matrices by adjusting the panels' allocation ratio R. The two heterogeneous cores are respectively RTX 3080 and Gold 5117, and RTX 2060s and i5-11400F.

The findings presented in Figure 13 strongly emphasize the superior performance effectiveness inherent in the heterogeneous collaboration, surpassing both GPU and CPU settings. Heterogeneous collaboration improves GFlops by an average of 7.21 times and 2.25 times compared with single CPU and single GPU, respectively. It's noteworthy that the performance enhancement is more pronounced for "stokes", "j", and "uk", attributed to their relatively higher compression ratios.

7 Related Work

SpGEMM in parallel is more complex than other sparse kernels like *sparse matrix-vector multiplication* (SpMV) [3, 42, 44] and *sparse-dense matrix multiplication* (SpMM) [37] due to the impact of the NZs distribution features on data loading and memory access. Since the pursuit of optimizing SpGEMM has garnered significant scholarly attention in contemporary discourse. Over the years, SpGEMM has garnered significant attention on various modern parallel platforms [10, 14], such as GPUs [1, 34], FPGAs [19, 22], specific devices [30, 39], and distributed clusters [5, 15]. This section commences by scrutinizing scholarly investigations that have concentrated on the GPU acceleration of SpGEMM. Subsequently, we delve into an exposition of optimization endeavors pertaining to matrix multiplication within heterogeneous computational environments.

A substantial body of research has been dedicated to the optimization of SpGEMM on GPUs. Notably, Bellet et al. [7] introduced the ***Expansion, Sorting, and Compression*** (**ESC**) approach, which dissects the computation process into three principal stages: ESC. In the initial phase, this

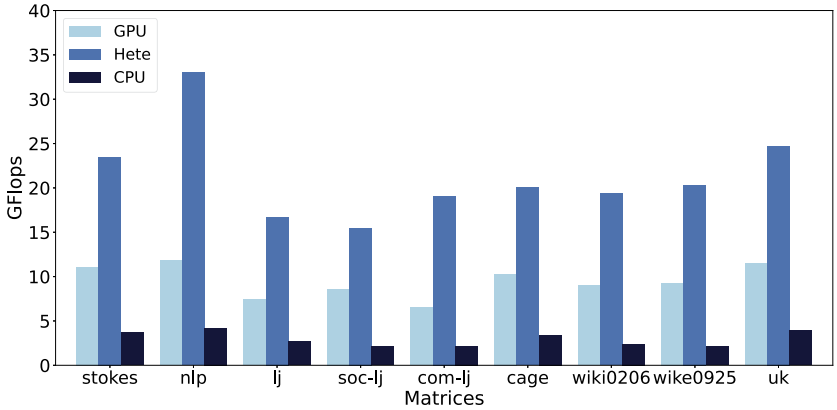


Fig. 13. Comparison of Heterogeneous Collaborative GFlops with CPU and Single-GPU.

method generates intermediate products by expanding sparse matrices (Expand). Subsequently, these intermediate results are subjected to sorting based on their respective row and column indices (Sort). Finally, the approach amalgamates values with colliding indices to derive the ultimate result (Compress). Rivera et al. [33] have directed their attention toward optimizing GPU resource utilization, particularly when the input matrix exhibits specific structural features. To this end, they have proposed two distinct algorithms, denoted as TSM2R and TSM2L, tailored for computing two categories of "tall-and-skinny" matrix-matrix multiplications on GPU architectures. Furthermore, various studies have explored techniques aimed at refining load balancing in the context of SpGEMM optimization. For example, Nagasaka et al. [27] have employed a two-step binning process, one for symbolic and another for numeric stages. Lee et al. [20] have devised a block reorganizer facilitating parallel splitting and gathering of computations for individual blocks. Merging strategies play a pivotal role in SpGEMM optimization and involve the utilization of sorted lists of intermediate results, typically through merge-sort-like algorithms [11, 12]. RMerge [11], for instance, decomposes input matrices into sub-matrices, which are efficiently merged using specialized algorithms. Similarly, bhSPARSE [23] dynamically selects among different merging solutions and optionally integrates elements of the ESC approach.

Effective matrix splitting assumes paramount importance within the domain of SpGEMM computations. This significance arises not solely from its role in enhancing load distribution but also from its substantive contribution to the optimization of data locality [37]. Illustratively, the work [24] has split rows into 38 bins based on computational workload and has assigned distinct optimization methods to each bin. Akbudak et al. [1] introduce a meticulously crafted *hypergraph partitioning* (HP) model, strategically designed to mitigate communication costs while simultaneously achieving a well-balanced workload distribution. In a complementary vein, Ballard et al. [6] have introduced a comprehensive framework that delineates the minimal communication prerequisites for both parallel and sequential SpGEMM computations. Furthermore, they have formulated a methodology for the identification of communication-optimal algorithms tailored to the specific features of input matrices, employing the hypergraph partitioning paradigm as the foundational approach. Building upon these endeavors, Selvitopi et al. [2] have extended the applicability of the HP model to encompass diverse parallel SpGEMM algorithms, including variants such as outer-product, inner-product, and row-by-row-product computations. Collectively, this concerted research effort underscores the paramount importance of proficient matrix partitioning in the continual advancement of SpGEMM optimization.

Efforts to address the computational challenges posed by large-scale matrix multiplication have led to investigations into parallelization strategies that harness both CPUs and GPUs. Benatia et al. [8] introduced a methodology that involves the horizontal splitting of the input matrix into multiple block rows, coupled with the application of a machine-learning-based performance model for the predictive determination of optimal sparse formats. Subsequently, a mapping algorithm is employed to judiciously allocate these block rows among the available CPUs and GPUs within the computational system. In a related vein, Xia et al. [41] proposed a splitting approach that distinguishes between out-of-core and in-core computations, with the aim of achieving the synchronization of computation and communication between CPUs and GPUs. This strategy is thoughtfully designed to minimize reliance on dynamic memory allocation, thus enhancing efficiency.

8 Limitation and Future Works

Our research focuses on accelerating SpGEMM multiplication for high-dimensional, large-scale matrices, but there are still some limitations and room for future improvement. On one hand, in GPU memory, ApSpGEMM requires additional space to maintain the "rowind" list until panel computation is complete. On the other hand, while optimizing memory access, we reduce the overhead of threads reading NZs from global memory to shared memory, but we do not reduce the overhead of writing the temporary product results from shared memory back to global memory. In the future, we need to design a compressed storage format that can map new matrix and original matrix information to replace CSR. Additionally, by predicting the NZs in the temporary product, we can preallocate storage space in global memory to write temporary product results in batches.

9 Conclusion

In this research, we focus on optimizing SpGEMM on GPU architecture. We analyze various potential issues that Gustavson's row-row algorithm may encounter. To address these challenges, we propose a comprehensive framework named ApSpGEMM with four phases: matrix pre-analysis, matrix splitting, device computation, and optional heterogeneous device scheduling. Furthermore, under ApSpGEMM guidance, we propose a specific and feasible solution. First, we perform a lightweight analysis to gather essential information about NZs in the input matrix. Using this information, we strategically rearrange matrices A and B to enable efficient computations (SpGEMM, SpMM, DGEMM). We also optimize load balancing and memory access for these operations using techniques like bin methods, thread grouping, and merging dense columns. For matrices exceeding GPU memory capacity, we employ a hybrid CPU+GPU approach. This involves scheduling matrices across different devices based on affinity considerations to overlap transmission, computation, and splitting operations and reduce execution time.

References

[1] Kadir Akbudak and Cevdet Aykanat. 2014. Simultaneous input and output matrix partitioning for outer-product-parallel sparse matrix-matrix multiplication. *SIAM Journal on Scientific Computing* 36, 5 (2014), C568–C590.

[2] Kadir Akbudak, Oguz Selvitopi, and Cevdet Aykanat. 2018. Partitioning models for scaling parallel sparse matrix-matrix multiplication. *ACM Transactions on Parallel Computing* 4, 3 (2018), 13:1–13:34.

[3] Christie L. Alappat, Georg Hager, Olaf Schenk, and Gerhard Wellein. 2023. Level-based blocking for sparse matrices: Sparse matrix-power-vector multiplication. *IEEE Transactions on Parallel and Distributed Systems* 34, 2 (2023), 581–597.

[4] Rahman Ghasempour Balagafshe, Alireza Akoushideh, and Asadollah Shahbahrami. 2022. Matrix-matrix multiplication on graphics processing unit platform using tiling technique. *IAES Indonesian Journal of Electrical Engineering and Computer Science* 28, 2 (2022), 1012–1019.

[5] Grey Ballard, Aydin Buluç, James Demmel, Laura Grigori, Benjamin Lipshitz, Oded Schwartz, and Sivan Toledo. 2013. Communication optimal parallel multiplication of sparse random matrices. In *Proc. of the 2013 Symposium on Parallelism in Algorithms and Architectures (SPAA'13)*. 222–231.

[6] Grey Ballard, Alex Druinsky, Nicholas Knight, and Oded Schwartz. 2016. Hypergraph partitioning for sparse matrix-matrix multiplication. *ACM Transactions on Parallel Computing* 3, 3 (2016), 18:1–18:34.
[7] Nathan Bell, Steven Dalton, and Luke N. Olson. 2012. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM Journal on Scientific Computing* 34, 4 (2012), C123–C152.
[8] Akrem Benatia, Weixing Ji, Yizhuo Wang, and Feng Shi. 2020. Sparse matrix partitioning for optimizing SpMV on CPU-GPU heterogeneous platforms. *SAGE International Journal of High Performance Computing Applications* 34, 1 (2020), 66–80.
[9] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J. R. Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis, and Pushmeet Kohli. 2022. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature* 610, 7930 (2022), 47–53.
[10] Valentin Le Fèvre and Marc Casas. 2023. Efficient execution of SpGEMM on long vector architectures. In *Proc. of the 2023 International Symposium on High-Performance Parallel and Distributed Computing (HPDC'23)*. 101–113.
[11] Felix Gremse, Andreas Höfter, Lars Ole Schwen, Fabian Kiessling, and Uwe Naumann. 2015. GPU-accelerated sparse matrix-matrix multiplication by iterative row merging. *SIAM Journal on Scientific Computing* 37, 1 (2015), C54–C71.
[12] Felix Gremse, Kerstin Küpper, and Uwe Naumann. 2018. Memory-efficient sparse matrix-matrix multiplication by row merging on many-core architectures. *SIAM Journal on Scientific Computing* 40, 4 (2018), C429–C449.
[13] Fred G. Gustavson. 1978. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Software* 4, 3 (1978), 250–269.
[14] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. 2019. Adaptive sparse tiling for sparse matrix multiplication. In *Proc. of the 2019 SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'19)*. 300–314.
[15] Sangwoo Hong, Heecheol Yang, Youngseok Yoon, and Jungwoo Lee. 2023. Straggler-exploiting fully private distributed matrix multiplication with chebyshev polynomials. *IEEE Transactions on Communications* 71, 3 (2023), 1579–1594.
[16] Vladislav Ishimtsev, Alexey Bokhovkin, Alexey Artemov, Savva Ignatyev, Matthias Nießner, Denis Zorin, and Evgeny Burnaev. 2020. CAD-Deform: Deformable fitting of CAD models to 3D scans. In *Proc. of the Computer Vision 2020 European Conference (ECCV'20)*. 599–628.
[17] Homin Kang, Hyuck-Chan Kwon, and Duksu Kim. 2020. HPMaX: Heterogeneous parallel matrix multiplication using CPUs and GPUs. *Springer Computing* 102, 12 (2020), 2607–2631.
[18] Jeremy Kepner, Peter Aaltonen, David A. Bader, Aydin Buluç, Franz Franchetti, John R. Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, Carl Yang, John D. Owens, Marcin Zalewski, Timothy G. Mattson, and José E. Moreira. 2016. Mathematical foundations of the GraphBLAS. In *Proc. of the 2016 High Performance Extreme Computing Conference (HPEC'16)*. 1–9.
[19] Fumiya Kono, Naohito Nakasato, and Maho Nakata. 2023. Accelerating 128-bit floating-point matrix multiplication on FPGAs. In *Proc. of the 2023 Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'23)*. 204.
[20] Jeongmyung Lee, Seokwon Kang, Yongseung Yu, Yong-Yeon Jo, Sang-Wook Kim, and Yongjun Park. 2020. Optimization of GPU-based sparse matrix multiplication for large sparse networks. In *Proc. of the 2020 International Conference on Data Engineering (ICDE'20)*. 925–936.
[21] Jiancong Li, Houji Zhou, Yi Li, and Xiangshui Miao. 2023. A memristive neural network based matrix equation solver with high versatility and high energy efficiency. *Science China Information Sciences* 66, 2 (2023), 122402.
[22] Fabiano Libano, Paolo Rech, and John Brunhaver. 2023. Efficient error detection for matrix multiplication with systolic arrays on FPGAs. *IEEE Trans. Comput.* 72, 8 (2023), 2390–2403.
[23] Weifeng Liu and Brian Vinter. 2014. An efficient GPU general sparse matrix-matrix multiplication for irregular data. In *Proc. of the 2014 International Parallel and Distributed Processing Symposium (IPDPS'14)*. 370–381.
[24] Weifeng Liu and Brian Vinter. 2015. A framework for general sparse matrix-matrix multiplication on GPUs and heterogeneous processors. *J. Parallel and Distrib. Comput.* 85, C (2015), 47–61.
[25] Xinjian Long, Xiangyang Gong, Bo Zhang, and Huiyang Zhou. 2023. An intelligent framework for oversubscription management in CPU-GPU unified memory. *Springer Journal of Grid Computin* 21, 1 (2023), 11.
[26] Zhengyang Lu and Weifeng Liu. 2023. TileSpTRSV: A tiled algorithm for parallel sparse triangular solve on GPUs. *CCF Transactions on High Performance Computing* 5, 2 (2023), 129–143.
[27] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydın Buluç. 2019. Performance optimization, modeling and analysis of sparse matrix-matrix products on multi-core and many-core processors. *Parallel Comput.* 90, C (2019), 102545.
[28] Yusuke Nagasaka, Akira Nukada, and Satoshi Matsuoka. 2017. High-performance and memory-saving sparse general matrix-matrix multiplication for NVIDIA pascal GPU. In *Proc. of the 2017 International Conference on Parallel Processing (ICPP'17)*. 101–110.

[29] Yuyao Niu, Zhengyang Lu, Haonan Ji, Shuhui Song, Zhou Jin, and Weifeng Liu. 2022. TileSpGEMM: A tiled algorithm for parallel sparse general matrix-matrix multiplication on GPUs. In *Proc. of the 2022 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'22)*. 90–106.

[30] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David T. Blaauw, Trevor N. Mudge, and Ronald G. Dreslinski. 2018. OuterSPACE: An outer product based sparse matrix multiplication accelerator. In *Proc. of the 2018 International Symposium on High Performance Computer Architecture (HPCA'18)*. 724–736.

[31] Mathias Parger, Martin Winter, Daniel Mlakar, and Markus Steinberger. 2020. spECK: Accelerating GPU sparse matrix-matrix multiplication through lightweight analysis. In *Proc. of the 2020 SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'20)*. 362–375.

[32] Majid Rasouli, Robert M. Kirby, and Hari Sundar. 2021. A compressed, divide and conquer algorithm for scalable distributed matrix-matrix multiplication. In *Proc. of the 2021 International Conference on High Performance Computing in Asia-Pacific Region (HPC-Asia'21)*. 110–119.

[33] Cody Rivera, Jieyang Chen, Nan Xiong, Jing Zhang, Shuaiwen Leon Song, and Dingwen Tao. 2021. TSM2X: High-performance tall-and-skinny matrix-matrix multiplication on GPUs. *Journal of Parallel Distributed Computing* 151, C (2021), 70–85.

[34] Hesam Shabani, Abhishek Singh, Bishoy Youhana, and Xiaochen Guo. 2023. HIRAC: A hierarchical accelerator with sorting-based packing for SpGEMMs in DNN applications. In *Proc. of the 2023 International Symposium on High-Performance Computer Architecture (HPCA'23)*. 247–258.

[35] Banseok Shin, Sehun Park, and Jaeha Kung. 2023. Improving hardware efficiency of a sparse training accelerator by restructuring a reduction network. In *In Proc. of the 2023 Interregional NEWCAS Conference (NEWCAS'23)*. 1–5.

[36] Koichi Shirahata, Hitoshi Sato, and Satoshi Matsuoka. 2014. Out-of-core GPU memory management for MapReduce-based large-scale graph processing. In *Proc. of the 2014 International Conference on Cluster Computing (CLUSTER'14)*. 221–229.

[37] Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young-kyu Choi, Jason Lau, and Jason Cong. 2022. Sextans: A streaming accelerator for general-purpose sparse-matrix dense-matrix multiplication. In *Proc. of the 2022 International Symposium on Field-Programmable Gate Arrays (FPGA'22)*. 65–77.

[38] Alexander van der Grinten, Geert Custers, Duy Le Thanh, and Henning Meyerhenke. 2022. Fast dynamic updates and dynamic SpGEMM on MPI-Distributed graphs. In *Proc. of the 2022 International Conference on Cluster Computing (CLUSTER'22)*. 429–439.

[39] Yang Wang, Chen Zhang, Zhiqiang Xie, Cong Guo, Yunxin Liu, and Jingwen Leng. 2021. Dual-side sparse tensor core. In *Proc. of the 2021 ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'21)*. 1083–1095.

[40] Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. 2019. Adaptive sparse matrix-matrix multiplication on the GPU. In *Proc. of the 2019 SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'19)*. 68–81.

[41] Yang Xia, Peng Jiang, Gagan Agrawal, and Rajiv Ramnath. 2021. Scaling sparse matrix multiplication on CPU-GPU nodes. In *Proc. of the 2021 International Parallel and Distributed Processing Symposium (IPDPS'21)*. 392–401.

[42] Serif Yesil, Azin Heidarshenas, Adam Morrison, and Josep Torrellas. 2023. WISE: Predicting the performance of sparse matrix vector multiplication with machine learning. In *Proc. of the 2023 SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP'23)*. 329–341.

[43] Chao Zhang, Maximilian H. Bremer, Cy P. Chan, John Shalf, and Xiaochen Guo. 2022. ASA: Accelerating sparse accumulation in column-wise SpGEMM. *ACM Transactions on Architecture and Code Optimization* 19, 4 (2022), 49:1–49:24.

[44] Yichen Zhang, Shengguo Li, Fan Yuan, Dezun Dong, Xiaojian Yang, Tiejun Li, and Zheng Wang. 2023. Memory-aware optimization for sequences of sparse matrix-vector multiplications. In *Proc. of the 2023 International Parallel and Distributed Processing Symposium (IPDPS'23)*. 379–389.

Received 3 March 2024; revised 27 September 2024; accepted 17 October 2024