DeltaSPARSE: High-Performance Sparse General Matrix-Matrix Multiplication on Multi-GPU Systems

1st Shuai Yang, 2nd Changyou Zhang, 3rd Ji Ma
Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
yangshuai2022@iscas.ac.cn, changyou@iscas.ac.cn, maji21@mails.ucas.ac.cn

***Abstract*—Sparse General Matrix-Matrix Multiplication (SpGEMM) serves as a fundamental operation in the domains of sparse linear algebra and graph data processing. The majority of existing research predominantly concentrates on optimizing SpGEMM in the context of single GPU scenarios. Nevertheless, the growing prevalence of multi-GPU systems offers opportunities to harness the computational capabilities of multiple GPUs, thereby enhancing the performance of sparse general matrix-matrix multiplication. The efficacy of multi-GPU SpGEMM is chiefly constrained by two factors: (1) the irregular sparse pattern of sparse matrices, and (2) the load imbalance among multiple GPUs.**

To address these challenges, this paper presents DeltaSPARSE, the first algorithm to achieve significant speed-up for large-scale SpGEMM on multiple GPUs, to the best of our knowledge. Our algorithm incorporates hybrid accumulators, which dynamically choose the most suitable accumulator algorithm for rows exhibiting varying levels of sparsity. Moreover, we suggest a hierarchical task scheduling approach to partition and allocate tasks across diverse levels of parallel hardware, such as GPUs, blocks, warps, and threads.

Experimental outcomes utilizing the SuiteSparse matrix dataset reveal that DeltaSPARSE displays near-linear scalability in multi-GPU configurations. Furthermore, it attains substantial speed enhancements in comparison to the present state-of-the-art single GPU SpGEMM methods, including NSPARSE, spECK, bhSPARSE, and cuSPARSE, across matrices with various sparse characteristics.

***Index Terms*—SpGEMM, multiple GPUs, sparse matrix**

I. Introduction

Sparse General Matrix Multiplication (SpGEMM) plays a pivotal role as a fundamental sparse operator. For instance, in graph data processing [5], various operations such as loop detection [26], triangular counting [3], and multi-source breadth-first search [17] can be formulated using SpGEMM. Additionally, when accelerating the resolution of sparse linear algebraic equations through the utilization of Algebraic Multigrid Preconditioners [6], sparse matrix multiplication emerges as a significant time-consuming operation. Therefore, the development of an efficient Sparse General Matrix Multiplication algorithm has become essential for enhancing the performance of these applications. Furthermore, with the rapid advancement of the Multi-GPU platform, which offers both high computational power and cost-effectiveness, the implementation of Sparse General Matrix algorithms on multi-GPU systems holds unique and crucial significance.

Currently, Sparse General Matrix Multiplication can be categorized into row-row-based and tile-based methods. In the former approach, rows of matrix C are computed in parallel by merging the corresponding non-zero elements of matrices A and B. To accommodate different sparsity characteristics of rows, efficient sparse accumulators such as ESC [15], Hash [1], [10], [17], [18], [23], Merge [9], [11], [12], [14], [15], Dense [13], [23], and heap [4], [15] are designed. On the other hand, tile-based [19] methods accumulate results and store non-zero elements using tiles as the basic unit. However, to maintain satisfactory performance across a diverse range of sparse matrix patterns, these methods often incur considerable costs during the pattern analysis or the definition of compressed memory layouts.

In contrast to Multi-GPU Dense Matrix Multiplication, SpGEMM on a Multi-GPU platform remains an unresolved challenge for several reasons. **(1) Irregular sparse patterns.** The difference in distribution and amount of non-zero elements in each row of the input and output matrices poses a significant design challenge for efficient GPU data compression storage formats and accumulators. **(2) Multi-GPU platform load imbalance.** Load balancing of multiple GPUs is pivotal to the efficiency of the algorithm and the effective use of hardware. However, the complex and unpredictable computation, memory access, and communication costs of SpGEMM make it harder to establish scheduling strategies.

In view of the inherent problems in SpGEMM and the challenges encountered in designing a multi-GPU framework, we propose the first Multi-GPU Sparse General Matrix Multiplication method called *DeltaSPARSE*. The main contributions of this paper are as follows:

- A hybrid accumulator is designed, which transitions between hash and dense accumulator strategies through lightweight analysis, facilitating adjustment to irregular sparsity in sparse matrices.
- A hierarchical task scheduling strategy is developed, which accomplishes adaptive load balancing on multi-

ple GPUs through low-cost partitioning steps, thereby enabling optimal parallelism.
- DeltaSPARSE demonstrates near-linear scalability across diverse GPU configurations and shows significant speed enhancements relative to contemporary single-GPU SpGEMM techniques when addressing matrices with a range of sparseness characteristics.

The remainder of this paper is structured as follows: Section II enunciates relevant SpGEMM methods and provides an insight into the current state of research. Section III focuses on detailing the design and implementation of DeltaSPARSE for multi-GPU computing. Section IV compares DeltaSPARSE with existing state-of-the-art single-GPU SpGEMM methodologies, scrutinizes the scalability of DeltaSPARSE, and decomposes its runtime. Finally, Section V draws the research to a close through a summary and outlines prospective avenues of exploration.

II. Related work

A. *Sparse General Matrix Multiplication*

The process of discovering and designing Sparse Matrix Multiplication (SpGEMM) algorithms is significantly complex, with an extensive algorithm space. Establishing a more efficient algorithm from this wide space remains one of the fundamental open problems in computer science. Existing SpGEMM can generally be categorized into five types: ESC, Hash, Merge, Dense, and Tile-based methods.

The key steps in the ESC category involve storing intermediate results in temporary space (expansion), sorting intermediate results according to column indexes, and finally accumulating values per column index. Originating in CUSP [8], this algorithm was subsequently applied further in bhSPARSE [15] and AC-SpGEMM [24]. The sorting and accumulation process of this algorithm directly acts upon intermediate results, exhibiting commendable performance when the intermediate results are limited. However, this method requires substantial temporary space to store intermediate results. As the scale of intermediate results increases, sorting them comes with a high cost, leading to a dramatic decline in algorithm performance.

Hash accumulators can utilize hash features to substantially reduce random data access and effectively enhance hardware resource utilization [18]. Due to the irregular sparsity characteristics of sparse matrix multiplication, the number of non-zero elements in the output rows often exhibits significant variability. Naively allocating the same storage space inevitably leads to resource wastage. Usually, the two-stage strategy is employed to calculate the number of non-zero elements in output matrix rows and allocate just-enough storage [23]. The Hash method's primary time-consuming aspects involve the atomic accumulation operation of intermediate results and the subsequent sorting steps. As the number of non-zero elements in the result matrix increases, these operations become performance bottlenecks. Additionally, when the shared storage space of the GPU is insufficient, the hash table has to be stored in slower global storage [10], resulting in a more significant performance decrease.

Merging involves using sorted arrays to store intermediate results and implementing a merge algorithm for sorting the intermediate results. RMerge [11] divides the input matrix into submatrices and utilizes a highly efficient merge algorithm for sorting results. As merge arrays are usually of equal length, they are unable to effectively address irregular sparsity characteristics, leading to low resource utilization.

Dense accumulators [13], [23] are introduced for scenarios where matrix size is large, and the output matrix is dense. It applies for an array of the same length as the matrix column number, directly performing linear mapping and result accumulation based on the result column index. This method fundamentally eliminates atomic conflicts and high sorting costs in the Hash method, theoretically exhibiting superior performance for dense scenarios. However, high storage requirements also become the bottleneck of this method.

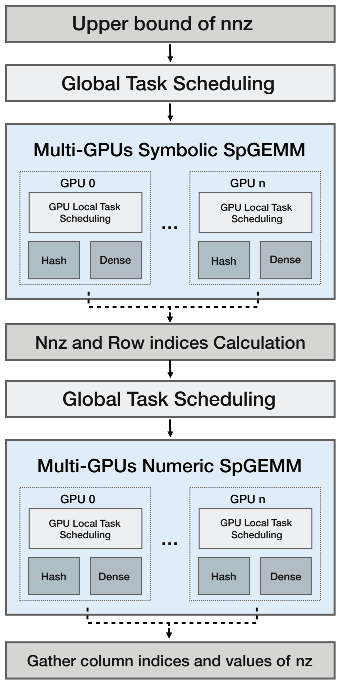
Tile-based methods extend from dense scenarios to sparse matrix scenarios. TileSpGEMM [19] partitions and schedules tasks based on the tile as the basic unit, effectively enhancing hardware utilization. It has superior performance in scenarios with higher sparsity. However, this technique introduces a notable format conversion time cost and performs poorly in extremely sparse scenarios.

B. *Multi-GPU Matirx Multiplication*

Current approaches for multi-GPU matrix multiplication primarily focus on dense matrices. SuperMatrix [7] decomposes the matrix into tiles and enables general matrix multiplication on SMP multicores. MAGMA [20] utilizes static scheduling to implement a multi-GPU linear algebra library. However, MAGMA faces limitations when dealing with heterogeneous systems. To overcome these limitations, StarPU [2] introduces dynamic scheduling algorithms, including work stealing and priority scheduling, resulting in consistent superlinear parallelism. NVIDIA's cuBLAS-XT [21] is a commercial multi-GPU L3 BLAS library that also adopts a tile strategy. However, its performance is hindered by frequent communication. In order to address this issue, BLASX [25] aims to overcome the insufficient communication and computation overlap in SuperMatrix and StarPU, as well as the frequent communication problem in cuBLAS-XT. BLASX achieves performance optimization through the implementation of optimization strategies such as algorithms-by-tiles, dynamic asynchronous runtime, and peer-to-peer (P2P) communication between GPUs. In the realm of sparse matrix multiplication, X. Liu [16] et al. have extended cuSpAMM to multi-GPU platforms, creating a multi-GPU sparse approximate matrix multiplication (SpAMM) method. However, their task partitioning scheme naively divides the tasks equally based on the number of rows, resulting in a significant load imbalance for sparse matrices with irregular sparsity characteristics.

III. DeltaSPARSE

DeltaSPARSE consists of seven distinct stages, as depicted in Figure1. Firstly, the computation of the upper bounds of non-zero elements per row in a lightweight output matrix



Upper bound of nnz
Global Task Scheduling
Multi-GPUs Symbolic SpGEMM
GPU 0
GPU Local Task Scheduling
Hash
Dense
...
GPU n
Nnz and Row indices Calculation
Global Task Scheduling
Multi-GPUs Numeric SpGEMM
Gather column indices and values of nz

Fig. 1. Stages involved in *DeltaSPARSE*. The stages of global and local task decomposition entail the meticulous partitioning of tasks by adopting a row-wise methodology based on the outcomes of statistical analysis.

C is performed. This serves as the basis for subsequent global task decomposition among GPUs. Following this, each GPU undergoes local task decomposition in the Multi-GPUs symbolic SpGEMM stage, employing a hybrid accumulation approach to determine the non-zero element counts per row in the partial result matrix C. The main GPU collects the individual computation results from each GPU, subsequently calculating the row offset array and the total number of non-zero elements in matrix C. This is followed by the execution of the global task decomposition stage in the numeric phase. In the Multi-GPUs symbolic SpGEMM stage, each GPU allocates storage space for the local matrix C based on the partitioning results, performs the computation of the column index array and the values of non-zero elements in C, and finally aggregates the resulting matrix C.

A. Adaptive Hybrid Accumulator

Hash Accumulator The hash accumulator has been widely utilized in previous research. One of the key factors in enhancing the efficiency of the hash accumulator is the reduction of collision rates during the hashing process. Sparse rows leverage a hashmap for linear probing and accumulation of results at specific positions, thereby facilitating rapid indexing when the hashmap possesses ample unoccupied space. Moreover, this approach exhibits lower storage requirements in comparison to alternative methods. However, as the hashmap becomes increasingly populated, the likelihood of hash collisions escalates substantially, giving rise to elevated linear probing expenses.

During the symbolic stage, it is solely necessary to document the column indices of non-zero elements within the output matrix, necessitating the allocation of a 32-bit integer array in shared memory. In contrast, throughout the numeric stage, the task of recording the column indices of non-zero elements and reducing intermediate results at corresponding indices demands the allocation of arrays comprising both 32-bit integer and 64-bit double types with lengths equivalent to the number of non-zero elements per row. Given the constraint of shared memory space on GPUs and the inherent inefficiency of global memory access, implementing an intermediate solution that allocates space in the global memory of the GPU would precipitate substantial performance degradation.

Algorithm 1: Hash with collision detection

```
input  : Row and column indices of $A$ and $B$
output: Rows with high collision rate, *failRows*;
        Count of failed rows, *failCount*;
        NNZ in each row of C, *nnzRowC*;
1  foreach nonzero entry $a_{ij}$ in $a_{i*}$ do
2      foreach nonzero entry $b_{jk}$ in $b_{j*}$ do
3          key ← (k*prime) & (mapLength - 1)
4          while collisionCount $< \epsilon$ and nnz $< \epsilon$ do
5              try to hit the hash entry with atomic
                 operation;
6              if collision then
7                  collisionCount ← collisionCount + 1;
8                  key ← (key + 1) & (mapLength - 1)
9              else
10                 insert k into hashmap;
11                 nnz ← nnz + 1;
12                 break;
13         if collisionCount $\geq \epsilon$ or nnz $\geq \epsilon$ then
14             break;
15     if collisionCount $\geq \epsilon$ or nnz $\geq \epsilon$ then
16         break;
17 if collisionCount $\geq \epsilon$ or nnz $\geq \epsilon$ then
       /* Record the indices and count
          of rows with excessively high
          collision rate                  */
18     failPerm[atomicAdd(failCount, 1)] = i;
19 else
20     nnzRowC[i] = nnz;
```

To mitigate the collision rate, we have adopted a twofold approach: the allocation of a hashmap with ample space exceeding the basic requirements, and the detection of hash collisions during the symbolic stage. The pseudo-code for the hash accumulator, inclusive of collision detection, is presented in Algorithm 1. For rows exhibiting excessively high collision rates, we document and subsequently reprocess them utilizing a Dense accumulator.

During the numeric stage, the accumulation, compression, and sorting steps are primarily time-consuming due to hash linear probing in the accumulation step, and sorting often occupies much of the numeric runtime. Approaches represented by NSPARSE [18] commonly utilize the count sort algorithm to orderly arrange the non-zero elements of the accumulator immediately following the accumulation step, all within the confines of the same kernel function. Although this algorithm obviates the need for atomic thread operations, its $O(NNZ_{c\_row}^2)$ time complexity causes the execution time to rise exponentially with the increasing number of non-zero elements per row in matrix C. *DeltaSPARSE* expedites the sorting step by introducing radix sort and selecting between count sort and radix sort based on the number of non-zero elements per row. Specifically, we obtain the threshold for the number of non-zero elements per row through experiments on the SuiteSparse dataset. For rows below the threshold, we continue using the count sort algorithm, i.e., counting the number of column indices smaller than the target element among the row's non-zero elements. For rows exceeding the threshold for non-zero elements, we write the disordered and compressed accumulated results into global memory and then employ radix sort for sorting.

Dense Accumulator For large and dense rows, the hash strategy suffers from severe performance degradation due to high memory requirements, high collision rates, and time-consuming sorting processes. We introduce a dense accumulator to handle these rows, with an allocation of a linear array with a length equivalent to the output matrix's column count. This approach fundamentally eliminates the need for hash collisions and sorting operations since the array index directly serves as the element's column index. As storing an element's column index becomes unnecessary, more intermediate results can be stored within hardware limitations. In extreme cases where an array with the size of the column index range cannot be instantiated in shared memory, the intermediate results for different column index ranges must be processed through iterative loops. To minimize the number of iterations, we analyze different platforms and hardware conditions to establish the maximum column index range.

We employ a bitmap to record the column indices of non-zero elements during the symbolic stage, relying primarily on atomic operations in shared memory. The introduction of a bitmap greatly reduces memory requirements. In the numeric stage, we allocate a linear array to store the intermediate results in addition to declaring a bitmap array to depict non-zero element distribution. After each iteration, we apply the prefix sum algorithm to obtain and write partial results.

Adaptive Strategy In the context of our study, the hashing strategy proves advantageous for matrices with small and sparse rows, whereas the dense strategy finds its utility in handling larger, denser matrices. The judicious selection of an appropriate accumulator, guided by the sparse characteristics of the rows, plays a pivotal role in optimizing performance, especially when dealing with irregularly sparse matrices. Through experimentation, we observed that among the myriad of influencing factors, the sparsity of matrix C's rows exerts the most significant influence on accumulator performance. Our empirical findings led us to establish a threshold of 7.60% sparsity as the criterion for strategy selection. Specifically, when the sparsity exceeds this threshold, opting for a dense accumulator is warranted, while a hash accumulator is the preferred choice for sparsity falling below this critical value.

B. Hierarchical Task Scheduling

In order to extend SpGEMM to a multi-GPU context while ensuring load balancing, maximization of hardware utilization, and minimization of algorithm execution time, it is essential to investigate the critical issue of designating appropriate computational tasks and allocating them across distinct levels of parallel hardware. However, devising task partitioning strategies is a complex endeavor owing to the irregular sparsity witnessed in sparse matrices and the inherent intricacies of the SpGEMM operator.

To address this challenge, we propose a hierarchical task partitioning strategy comprised of global and local task partitioning. The former manages the distribution of tasks among multiple GPUs, while the latter is responsible for the rescheduling of local tasks within an individual GPU.

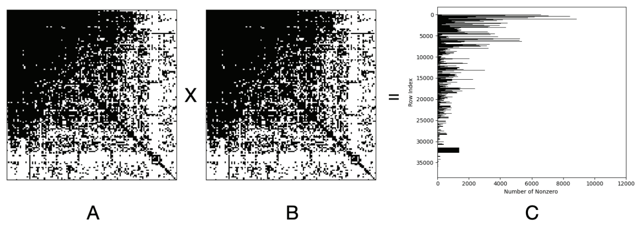


Fig. 2. Perform *C*=*A*×*A* operation of *email-Enron* matrix. The matrix A in the figure is equivalent to matrix B. The bar graph on the right side of the equation represents the non-zero elements per row in the resulting matrix C, where the x-axis signifies the number of non-zero elements and the y-axis denotes the row indices. In the computation *C*=*A*×*A*, the density of matrix *C* consistently surpasses that of *A*.

Global Task Scheduling Model In the context of dense matrix multiplication, the prevailing strategy entails an even distribution of rows among GPUs. This approach ensures optimal load balancing due to the uniform distribution of non-zero elements characteristic of dense matrices. However, sparse matrices exhibit substantial disparities in row lengths and positions, resulting in irregular sparsity. As depicted in Figure 2, the 'email-Enron' matrix demonstrates significant differences in row lengths—a trait inherent to irregular matrices frequently encountered within the *SuiteSparse* dataset. Additionally, the product matrix C displays analogous irregular sparsity, with a pronounced disparity between its maximum row length of 16,691 and minimum of a mere 510.

To establish a rational task partitioning methodology for matrices possessing these attributes, it is imperative to first devise a strategy for evaluating the costs linked to each subtask. Contrary to utilizing the sparsity of matrices A or B as a basis for partitioning, the sparsity of matrix C provides a more direct measure of the computational and memory access expenses. Consequently, the objective during the global task scheduling phase centers on evenly distributing the non-zero elements in matrix C among GPUs, thereby facilitating optimal partitioning and allocation of tasks.

Algorithm 2: Calculate the upper bound of the nnz in the i-th row of matrix C.

```
    input :  Row pointers and column indices of A,
             rpt_A, col_A;
             Row pointers of B, rpt_B;
    output:  upper bound of the nnz, upper;
1 upper ← 0;
2 foreach nonzero entry $a_{ij}$ in $a_{i*}$ do
3     upper ← upper + rpt_B[j + 1] - rpt_B[j];
```

In the symbolic phase, an upper limit on the quantity of non-zero elements per row is determined to estimate the row length and distribution of matrix C. This procedure is exemplified in Algorithm 2 and possesses a complexity of *O(nnz)*. Notably, this process eschews the use of atomic operations, thereby yielding low computational expenses. Subsequently, during the numeric phase, the row-wise tallies of non-zero elements acquired from the symbolic phase are employed for the purpose of global task partitioning.

Algorithm 3 delineates the GPU rendition of the task partitioning algorithm. Although the complexity of this algorithm, *O(np*m)*, is higher than that of algorithms premised on binary search, *O(np*log(m))*, where m indicates the number of rows in matrix C. However, it capitalizes on the thread resources provided by the GPU platform to yield a markedly superior execution efficiency when juxtaposed with the CPU variant of the binary search algorithm.

The symbolic and numeric phases receive as input the upper bound array of row non-zero element counts, denoted as *upper*, and the array of row non-zero element counts, denoted as *nnz_per_row*. Employing these inputs, the algorithm calculates the prefix sum in order to obtain the row pointer *rpt_c* of matrix C. The ultimate output consists of the starting row index for each GPU task.

To efficiently manage multiple GPUs simultaneously, we assign a CPU thread to each GPU. After the partitioning is completed, each GPU in the symbolic phase allocates GPU memory to store the local row non-zero element counts of matrix C based on the assigned number of rows. Similarly, in the numeric phase, each GPU allocates GPU memory to store the local row pointers, column indices, and non-zero element values of matrix C based on the assigned number of rows and non-zero element counts

Subsequent to the fulfillment of tasks by the individual GPUs, the results are consolidated onto a singular GPU. As no overlap occurs between the tasks carried out by distinct GPUs,

Algorithm 3: Global partitioning

```
    input :  Row pointers of C, rpt_c;
             Number of GPUs, np;
             Number of rows of C, row_num;
    output:  Row Index Offset per GPU,
             gpu_row_offset;
1  for i = 0 to np do
2      gpu_col_offset[i] ← i * nnz / np;
3  Launch: searchKernel(begin_col_idxs, rpt_c,
     row_num);
4  Function searchKernel(begin_col_idxs, rpt_c,
     row_num)
5      row_idx ← (blockIdx.x * blockDim.x) +
         threadIdx.x;
6      if row_idx ⩾ row_num then
7          return;
8      col_begin ← rpt_C[row_idx];
9      col_end ← rpt_C[row_idx + 1];
10     for i = 0 to np - 1 do
11         target_col ← gpu_col_offset[i];
12         if target_col ⩾ col_begin and target_col ¡
             col_end then
13             gpu_row_offset[i + 1] = row_idx;
```

the copying process can be executed entirely in parallel. To further diminish communication costs, we employ Nvidia's NVLink technology. Within our experimental system, the peer GPU-to-GPU communication bandwidth is established at 40GB/s. Theoretically, by leveraging NVLink 2.0 and NVSwitch technology, full connectivity among 16 GPUs can be achieved within a single server node, with a bidirectional communication bandwidth of 300GB/s for each pair of the 8 GPUs.

Local Task Scheduling Based on Decision Trees The greatest challenge faced by the SpGEMM algorithm is irregular sparsity, as illustrated in Figure 2. When this problem is divided into two subtasks, the first subtask displays a maximum row length of 16,691 and an average row length of 2,235, while the second subtask presents row lengths of 6,125 and 510. Such considerable disparities in row lengths and distributions create challenges for achieving efficient parallelism and memory access on GPUs. In order to address the aforementioned issue, inspired by the binning strategy utilized by NSPARSE [18], we have developed a local task scheduling model based on decision trees, which facilitates secondary partitioning of GPU-local tasks within both the symbolic and numeric phases, as well as the allocation of storage and computational hardware resources. The primary objective of this approach is to enhance resource utilization and ensure optimal load balancing.

During the symbolic phase, rows are reorganized and par-

TABLE I
PARAMETER SETTING FOR TASK SCHEDULING IN THE SYMBLIC PHASE

Range of Upper Bounds	Parameter Setting		
	Table Size	*TB Size*[a]	*Accumulator*
0-32	64	32	Hybrid
33-128	256	64	Hybrid
129-256	512	128	Hybrid
257-512	1024	256	Hybrid
513-1024	2048	512	Hybrid
1025-	/	1024	Dense

[a]Thread Block Size.

titioned based upon the range of upper bounds for non-zero element counts. For rows with varying upper bounds, a hybrid accumulator is employed, which adaptively selects between a hash accumulator and a dense accumulator. In an effort to decrease collision rates within the hash accumulator, the size of the hash table is set to twice the upper bound of non-zero elements per group. Furthermore, the thread block size is configured to be half the size of the hash table, thereby promoting increased concurrent execution of thread blocks per streaming multiprocessor (SM) and enhancing hardware resource utilization and occupancy. Table I delineates the partitioning range and parameter settings. It is important to highlight that, despite the fact that Dense accumulator necessitate the allocation of a larger quantity of shared storage space in comparison to hash accumulator, the storage cost during the symbolic phase remains considerably low due to the inherent lack of necessity to store the values of non-zero elements. Furthermore, Dense accumulator efficiently eradicates the significant expense associated with hash conflicts. Consequently, when encountering a scenario where the upper boundary of non-zero elements surpasses 1025, our preferred choice is to utilize Dense accumulator directly.

TABLE II
PARAMETER SETTING FOR TASK SCHEDULING IN THE NUMERIC PHASE

Range of Upper Bounds	Parameter Setting		
	Table Size	*TB Size*	*Accumulator*
0-128	256	64	Hybrid
129-256	512	128	Hybrid
257-512	1024	256	Hybrid
513-1024	2048	512	Hybrid
1025-2048	4096	1024	Hybrid
2049-	/	1024	Dense

During the numeric phase, we reorganize the partitions according to the range of non-zero elements derived from the symbolic phase, with the partition boundaries and parameter configurations presented in Table II. Upon surpassing a Hash table size of 4096, the shared storage demands of the Hash accumulator begin to exceed the hardware constraints imposed by specific GPUs, causing the atomic operations and sorting procedures in global memory to result in a considerable decrease in performance. Consequently, we opt to employ Dense accumulator in these situations.

Local task partitioning does not consistently yield positive results. In two particular scenarios, task partitioning might offer limited advantages or potentially result in diminished performance:

- **Low computational workload**: In the case of matrices characterized by a limited number of rows and non-zero elements, the duration dedicated to partitioning may approach or even exceed the required computational time.
- **Low Degree of Irregularity**: Low degree of irregularity signifies that there is a minimal difference in the lengths among various rows, indicating that they have similar resource requirements.

To attain optimal performance across diverse tasks, we introduce an innovative lightweight decision tree analysis strategy. This approach is employed to discern the necessity of task partitioning implementation. The employment of a solitary threshold for strategical selection proves insufficient when aiming to identify the ideal strategy for matrices displaying diverse characteristics. Taking into consideration the contributing factors to partitioning performance, as well as the potential computational costs of supplementary analyses, this study ultimately identifies the sparsity of matrix C (*sparsity_c*), the average number of non-zero elements per row in matrix C (*row_nnz_avg*), and the maximum number of non-zero elements per row in matrix C (*row_nnz_max*) as the pertinent features. It is important to note that, in the context of the symbolic phase, the upper limit of non-zero elements is utilized in lieu of the actual non-zero elements. In order to mitigate overfitting in the decision tree model, we imposed a maximum depth limit of 4 and allocated two-thirds of the dataset for training purposes, while the remaining portion was reserved for validation.

IV. EXPERIMENTAL RESULTS

A. *Experimental Setup*

The test system uses six NVIDIA Tesla V100-SXM2-16GB with CUDA 11.7, GPU driver version of 530.30.02 and an 80-core Intel(R) Xeon(R) Gold 6148 CPU with 572G RAM on Ubuntu 20.04. Direct peer-to-peer inter-GPU communication is enabled between the six GPUs. We are employing GCC version 9.4.0 and leveraging OpenMP to allocate a dedicated thread for each GPU. It is worth noting that our existing implementation is tailored specifically for single-node multi-GPU configurations. However, it is pertinent to highlight that this design can readily be extended to distributed environments through the integration of technologies such as MPI.

We compare DeltaSPARSE under a dual-GPU setup with state-of-the-art single GPU general sparse matrix algorithms, including cuSPARSE v11.7 [22], bhSPARSE [15], spECK [23], and NSPARSE [18]. Additionally, we perform experiments to examine the scalability of the algorithm, investigate the balance of multiple GPU loads to illustrate the effectiveness of task scheduling strategies, and ultimately breakdown the running time of the DeltaSPARSE algorithm. For the data set, we selected 16 representative sparse square matrices from the SuiteSparse Matrix Collection for in-depth comparison and

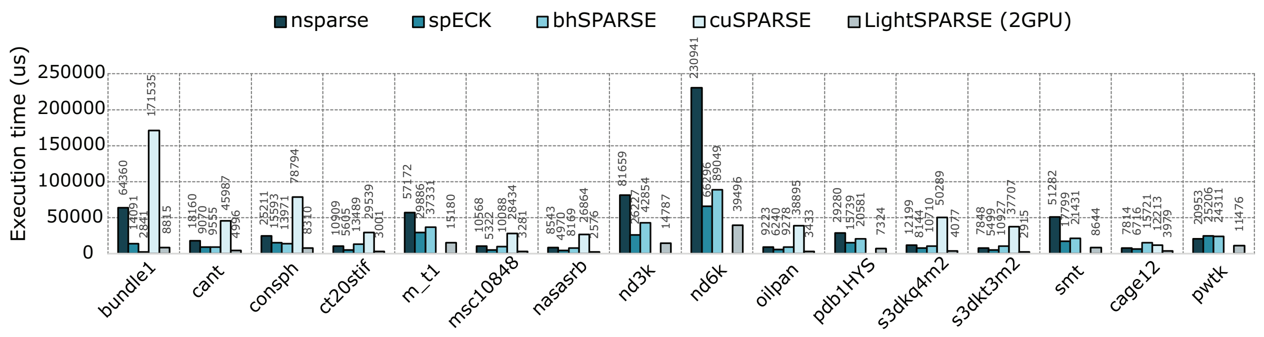


Fig. 3. Performance comparison of performing $C = A^2$ operation of the 16 representative matrices. An empty bar signifies the failure of the corresponding algorithm to execute SpGEMM on the matrix.

analysis. These matrices have diverse sparse features and are classic datasets in many sparse matrix studies. Table III shows detailed information. The compression ratio is defined as the ratio between the intermediate product of $C = A^2$ and the number of non-zero elements in matrix C.

TABLE III
INFORMATION OF THE 16 REPRESENTATIVE MATRICES. THE 'PROD.' IS THE NUMBER OF INTERMEDIATE PRODUCTS OF $C = A^2$.

Matrix A	***n(A)***	***nnz(A)***	***prod.***	***nnz(C)***	***compression rate***
bundle1	10K	770.9K	474.3M	24.0M	19.70
cant	62K	4.0M	269.5M	17.4M	15.45
consph	83K	6.0M	463.8M	26.5M	17.48
ct20stif	52K	2.7M	154.2M	10.0M	15.40
m_t1	97K	9.8M	1.1B	36.5M	28.86
msc10848	11K	1.1M	164.5M	6.2M	26.59
nasasrb	55K	2.7M	134.6M	8.3M	16.15
nd3k	9K	3.3M	1.3B	18.5M	69.00
nd6k	18K	6.9M	2.8B	42.2M	66.02
oilpan	74K	3.6M	188.9M	11.6M	16.27
pdb1HYS	36K	4.3M	555.3M	19.6M	28.34
s3dkq4m2	90K	4.8M	258.1M	13.3M	19.43
s3dkt3m2	90K	3.8M	156.4M	10.1M	15.46
smt	26K	3.8M	605.9M	19.4M	31.16
cage12	130K	2.0M	34.6M	15.2M	2.27
pwtk	218K	11.5M	626.1M	32.8M	19.10

B. Performance Comparison

In order to demonstrate the effectiveness of DeltaSPARSE, while considering the lack of comparable multi-GPU sparse general matrix multiplication methods, we performed the computation of $C = A^2$ using DeltaSPARSE with a 2-GPU configuration, as well as the state-of-the-art single-GPU sparse general matrix multiplication methods NSPARSE [18], spECK [23], bhSPARSE [15], and cuSPARSE v11.7 [22]. We then plotted their performance bar chart. As shown in Figure 3, our method DeltaSPARSE outperforms the other four methods in terms of performance on most representative matrices, especially for matrices with high computational workload and memory requirements, such as 'm_t1', 'nd3k', 'nd6k', 'pdb1HYS', 'smt' and 'pwtk'. While cuSPARSE v11.7 failed to execute due to memory constraints, DeltaSPARSE achieved acceleration factors of 1.97x, 1.77x, 1.68x, 2.15x, 2.06x and 2.20x compared to the second-best method, spECK, for these matrices. It is noteworthy that certain acceleration factors surpassed 2.0x, owing to the adaptability of the hybrid accumulators to rows with varying sparsity levels and the efficiency and cost-effectiveness of the task scheduling. With regards to the maximum acceleration factor, DeltaSPARSE exhibited speed improvements of up to 7.30x, 2.20x, 4.49x, and 19.46x in relation to NSPARSE, spECK, bhSPARSE, and cuSPARSE v11.7, respectively, while achieve on average 3.57x, 1.86x, 2.39x, and 9.83x speedups. We calculate the geometric mean to determine the average speedup. The subsequent average speedups mentioned in the following sections are calculated using the same method. It should be noted that these performance results are based on the 2-GPU configuration of DeltaSPARSE. As the number of GPUs increases, DeltaSPARSE will be able to effectively leverage the hardware to achieve performance gains. We will analyze the scalability of the method in the next section.

C. Scalability Analysis

Figure 4 illustrates the algorithm performance speedup of executing the $C = A^2$ operation on 16 representative matrices, comparing different GPU configurations to the single GPU setting. The examined matrices typically display near-linear scalability, represented by the average speedup factors of 1.64x, 2.02, 2.42x, 2.71x, and 3.02x for configurations ranging from 2 to 6 GPUs, respectively. Furthermore, peak speedup factors of 1.82x, 2.34x, 3.05x, 3.39x, and 3.91x have been realised. For matrices characterized by a higher computational complexity, embodied by 'm_t1', 'nd3k', 'nd6k' and 'pwtk', the computation of results constitutes the majority of the processing time. The proposed methods successfully leverage the performance-enhancing prospects offered by stacked multi-GPU hardware. It is notable that the irregular characteristics inherent to the matrices serve as critical determining factors in the scalability of the algorithm. They can result in an imbalance of GPU load distribution, which further diminishes the degree of algorithmic parallelism.

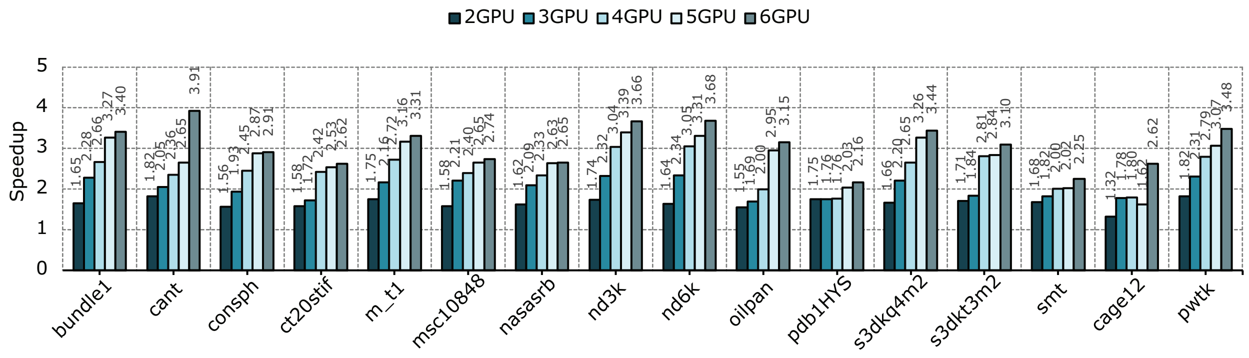


Fig. 4. The performance speedup ratio of multi-GPU SpGEMM compared to the single-GPU configuration on 16 representative matrices.

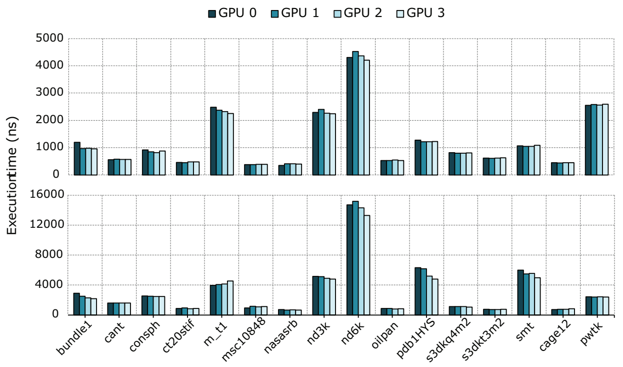


Fig. 5. Multi-GPU load balancing test under a 4-GPU configuration. The subplot above represents the execution time of the kernel during the symbolic stage for each GPU, and the subplot below represents the execution time of the kernel during the numeric stage for each GPU.

To assess the efficacy of the algorithm's load balancing, we consider a scenario involving a 4-GPU configuration, meticulously measuring the execution times of both symbolic and numeric phase kernels on each GPU. Since we concurrently launch kernels on all GPUs, minor differences in execution times imply minimal GPU idleness. As illustrated in Figure 5, the upper subfigure represents the execution times of symbolic phase kernels on each GPU, while the lower subfigure displays the execution times of numeric phase kernels on each GPU. It is evident that, for the majority of matrices, a commendable load balancing performance is achieved. Furthermore, we employ the coefficient of variation (CV) to precisely quantify the disparities in execution times among GPUs. In terms of the symbolic stage, the maximum coefficient of variation for GPU execution time is 0.096, while the minimum is as low as 0.006. Regarding the numeric stage, the maximum CV for GPU execution time is 0.114, with a minimum of 0.001. Notably, even for matrices with highly irregular sparsity, such as 'smt', the CV for the symbolic stage is a mere 0.014, and for the numeric stage, it is merely 0.064.

D. Runtime Breakdown of DeltaSPARSE

Figure 6 depicts the runtime breakdown of DeltaSPARSE in a configuration utilizing two GPUs. The bar chart sequentially presents the proportional runtime of various stages, including, in descending order: numeric phase kernel, numeric phase task scheduling, symbolic phase kernel, symbolic phase task scheduling, memory allocation, and memory copying. It is noted that the symbolic and numeric phase kernels, on average, account for 26.5% and 54.0% of the total runtime, respectively. Enhanced efficiency in memory allocation and copying is achieved through the application of asynchronous *cudaStream* launches. This enables the concurrent handling of memory allocation, computation, and communication workloads across multiple streams. Notably, the impact of scheduling overhead remains strikingly low. Despite the symbolic phase task scheduling requiring a marginally extended duration attributed to the calculation of the upper bound of nonzero elements per row, it constitutes merely 3.22% of the overall runtime. Similarly, the numeric phase task scheduling contributes a scant 0.46% to the total runtime.

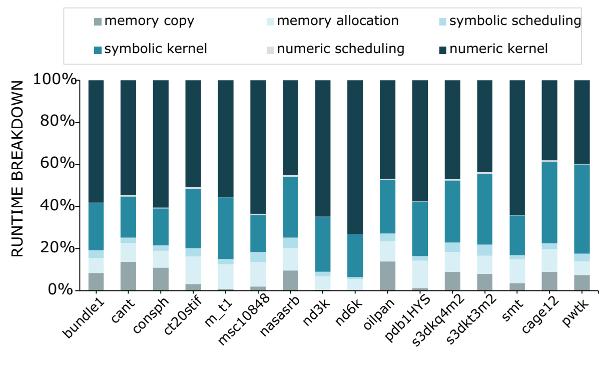


Fig. 6. The runtime breakdown of *DeltaSPARSE*. From top to bottom, the running time ratios of the numeric kernel, numeric stage scheduling, symbolic kernel, symbolic stage scheduling, memory allocation, and memory copy are presented.

V. Conclusion

This paper introduces DeltaSPARSE, a framework for multi-GPU Sparse Generalized Matrix Multiplication (SpGEMM). Our approach addresses the challenges faced by SpGEMM algorithms on multi-GPU platforms, including irregular sparse patterns in sparse matrices and load imbalance across multiple GPUs. To overcome these challenges, we propose an adaptive hybrid accumulator and a hierarchical scheduling strategy. The effectiveness of our method is demonstrated through near-linear scalability, reduced scheduling, storage allocation, and communication costs on representative matrices from the SuiteSparse Matrix Collection. In comparison to existing state-of-the-art single-GPU SpGEMM methods, DeltaSPARSE achieves significant acceleration. Our future work includes extending the application of SpGEMM to distributed clusters, enabling the method to be utilized in a wider range of scenarios.

Acknowledgment

We express our sincere gratitude for the invaluable feedback provided by all the reviewers. Changyou Zhang is the corresponding author of this paper. This research was supported by the National Key Research and Development Project of China: CAE Cloud Service Platform for Complex Equipment (2020YFB1709500).

References

[1] P. N. Q. Anh, R. Fan, and Y. Wen, “Balanced hashing and efficient GPU sparse general matrix-matrix multiplication,” in Proceedings of the International Conference on Supercomputing (ICS ’16), 2016.

[2] C. Augonnet, S. Thibault, R. Namyst, and P. A. Wacrenier, “StarPU: a unified platform for task scheduling on heterogeneous multicore architectures,” in Euro-Par 2009 Parallel Processing: 15th International Euro-Par Conference, Delft, The Netherlands, August 25-28, 2009, Proceedings, vol. 15, Springer Berlin Heidelberg, 2009, pp. 863-874.

[3] A. Azad, A. Buluç and J. Gilbert, “Parallel Triangle Counting and Enumeration Using Matrix Algebra,” 2015 IEEE International Parallel and Distributed Processing Symposium Workshop, Hyderabad, India, 2015, pp. 804-811.

[4] A. Azad, G. Ballard, A. Buluç, J. Demmel, L. Grigori, O. Schwartz, S. Toledo, and S. Williams, “Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication,” SIAM Journal on Scientific Computing, vol. 38, no. 6, pp. C624-C651, 2016.

[5] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, “Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations,” in Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP ’17), New York, NY, USA, 2017, pp. 235-248.

[6] N. Bell, S. Dalton, and L. N. Olson, “Exposing fine-grained parallelism in algebraic multigrid methods,” SIAM Journal on Scientific Computing, vol. 34, no. 4, pp. C123-C152, 2012.

[7] E. Chan, F. G. Van Zee, P. Bientinesi, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn, “SuperMatrix: a multithreaded runtime scheduling system for algorithms-by-blocks,” in Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP ’08), New York, NY, USA, Association for Computing Machinery, 2008, pp. 123-132.

[8] S. Dalton, N. Bell, L. Olson, and M. Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2014.

[9] S. Dalton, S. Baxter, D. Merrill, L. Olson, and M. Garland, “Optimizing sparse matrix operations on GPUs using merge path,” in Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS ’15), 2015, pp. 407-416.

[10] M. Deveci, C. Trott, and S. Rajamanickam, “Performance-portable sparse matrix-matrix multiplication for many-core architectures,” in Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW ’17), 2017, pp. 693-702.

[11] F. Gremse, A. Höfter, L. O. Schwen, F. Kiessling, and U. Naumann, “GPU-Accelerated Sparse Matrix-Matrix Multiplication by Iterative Row Merging,” SIAM Journal on Scientific Computing, vol. 37, no. 1, pp. C54-C71, 2015.

[12] F. Gremse, K. Küpper, and U. Naumann, “Memory-efficient sparse matrix-matrix multiplication by row merging on many-core architectures,” SIAM Journal on Scientific Computing, vol. 40, no. 4, pp. C429-C449, 2018.

[13] J. R. Gilbert, C. Moler, and R. Schreiber, “Sparse matrices in MATLAB: Design and implementation,” SIAM Journal on Matrix Analysis and Applications, vol. 13, no. 1, pp. 333-356, 1992.

[14] H. Ji, S. Lu, K. Hou, H. Wang, Z. Jin, W. Liu, and B. Vinter, “Segmented merge: A new primitive for parallel sparse matrix computations,” International Journal of Parallel Programming, pp. 1-13, 2021.

[15] W. Liu and B. Vinter, “An efficient GPU general sparse matrix-matrix multiplication for irregular data,” in Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS, 2014, pp. 370-381.

[16] X. Liu, Y. Liu, H. Yang, et al., “Accelerating approximate matrix multiplication for near-sparse matrices on GPUs,” Journal of Supercomputing, vol. 78, pp. 11464-11491, 2022.

[17] Y. Nagasaka, S. Matsuoka, A. Azad, and A. Buluç, “High-Performance Sparse Matrix-Matrix Products on Intel KNL and Multicore Architectures,” in Workshop Proceedings of the 47th International Conference on Parallel Processing (ICPP Workshops ’18), New York, NY, USA, Association for Computing Machinery, 2018, pp. 1-10, Art. 34.

[18] Y. Nagasaka, A. Nukada and S. Matsuoka, “High-Performance and Memory-Saving Sparse General Matrix-Matrix Multiplication for NVIDIA Pascal GPU,” 2017 46th International Conference on Parallel Processing (ICPP), Bristol, UK, 2017, pp. 101-110.

[19] Y. Niu, Z. Lu, H. Ji, S. Song, Z. Jin, and W. Liu, “TileSpGEMM: a tiled algorithm for parallel sparse general matrix-matrix multiplication on GPUs,” in Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP ’22), New York, NY, USA, Association for Computing Machinery, 2022, pp. 90-106.

[20] R. Nath, S. Tomov, T. Dong, and J. Dongarra, “Optimizing symmetric dense matrix-vector multiplication on GPUs,” in Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC ’11), New York, NY, USA, Association for Computing Machinery, 2011, Art. 6, pp. 1-10.

[21] NVIDIA, “cuBLAS” 2016. [Online]. Available: https://developer.nvidia.com/cublas.

[22] NVIDIA, “NVIDIA CUDA Sparse Matrix Library (cuSPARSE).” [Online]. Available: https://developer.nvidia. com/cusparse.

[23] M. Parger, M. Winter, D. Mlakar, and M. Steinberger, “SpECK: accelerating GPU sparse matrix-matrix multiplication through lightweight analysis,” in Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP ’20), New York, NY, USA, Association for Computing Machinery, 2020, pp. 362-375.

[24] M. Winter, D. Mlakar, R. Zayer, H.-P. Seidel, and M. Steinberger, “Adaptive sparse matrix-matrix multiplication on the GPU,” in Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP ’19), New York, NY, USA, Association for Computing Machinery, 2019, pp. 68-81.

[25] L. Wang, W. Wu, Z. Xu, J. Xiao, and Y. Yang, “BLASX: A High Performance Level-3 BLAS Library for Heterogeneous Multi-GPU Computing,” in Proceedings of the 2016 International Conference on Supercomputing (ICS ’16), New York, NY, USA, Association for Computing Machinery, 2016, Art. 20, pp. 1-11.

[26] R. Yuster and U. Zwick, “Detecting short directed cycles using rectangular matrix multiplication and dynamic programming,” in Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms (SODA ’04), USA, Society for Industrial and Applied Mathematics, 2004, pp. 254-260.