Segmented Merge: A New Primitive for Parallel Sparse Matrix Computations

Haonan Ji, Shibo Lu, Kaixi Hou, Hao Wang, Weifeng Liu, Brian Vinter

Segmented Merge: A New Primitive for Parallel Sparse Matrix Computations

Haonan Ji[1], Shibo Lu[1], Kaixi Hou[2], Hao Wang[3], Weifeng Liu[1], and Brian Vinter[4]

[1] Super Scientific Software Laboratory, Department of Computer Science and Technology, China University of Petroleum-Beijing
[2] Department of Computer Science, Virginia Tech
[3] Department of Computer Science and Engineering, The Ohio State University
[4] Faculty of Technical Sciences, Aarhus University
haonan_ji@yeah.net, lslslsb@yeah.net, kaixihou@vt.edu, wang.2721@osu.edu, weifeng.liu@cup.edu.cn, vinter@au.dk

Abstract— Segmented operations, such as segmented sum, segmented scan and segmented sort, are important building blocks for parallel irregular algorithms. We in this work propose a new parallel primitive called segmented merge. Its function is in parallel merging q sub-segments to p segments, both of nonuniform lengths. We implement the segmented merge primitive on GPUs and demonstrate its efficiency on parallel sparse matrix transposition (SpTRANS) and sparse matrix-matrix multiplication (SpGEMM) operations.

Index Terms— Parallel computing, segmented merge, sparse matrix, GPU

1 Introduction

Since Blelloch et al. [1] reported that segmented operations can achieve better load balancing than row-wise approaches in parallel sparse matrix-vector multiplication (SpMV), several new parallel segmented primitives, such as segmented sum [12], segmented scan [4] and segmented sort [8] have been developed for replacing their ordinary counterparts, i.e., sum, scan and sort, in a few irregular sparse matrix algorithms on many-core platforms such as GPUs.

However, merge, another important fundamental routine in computer science, has not received much attention from the view point of segmented operation. Actually, it can be quite useful when both the input and the output matrices are stored with indirect indices. One important higher level algorithm example is sparse matrix-matrix multiplication (SpGEMM). It multiplies two sparse matrices A and B, and generates one resulting sparse matrix C. When the nonzeros in each row of the input sparse matrix B are sorted in the ascending order according to their column indices, the basic operation needed is actually merge [6,10,11].

We in this paper first define a new primitive called segmented merge. It merges q sub-segments to p segments, both of nonuniform lengths. The elements

in each sub-segment are ordered in advance. When all the sub-segments in one segment are merged into one sub-segment of the same length as the segment containing it, the operation is completed. In the SpGEMM scenario, the rows of B involved can be seen as the sub-segments, and the rows of C can be seen as the segments.

Although the definition and a serial code of the segmented merge can be both straightforward, designing an efficient parallel algorithm is not trivial. There are two major challenges. The first one is the load balancing problem. It happens when the lengths of the sub-segments and the segments are nonuniform, meaning that roughly evenly assigning them to tens of processing units can be difficult. The second challenge is the vectorization problem. It can be also hard to carefully determine a similar amount of elements processed by thousands of SIMD lanes on many-core processors such as GPUs.

To address the two challenges, we design an efficient parallel algorithm for the segmented merge operation on GPUs. The algorithm first preprocesses the segments and sub-segments, and records the boundaries of them for dividing the tasks. Then our method uses a binary tree for merging sub-segments in a bottom-up manner. The algorithm works in an iterative way and completes when each segment only has one sub-segment. In the procedure, each pair of sub-segments are merged independently by utilizing SIMD lanes, i.e., threads running on CUDA GPUs, and each SIMD lane merges a similar amount of elements with serial merge.

We benchmark two sparse kernels, sparse matrix transposition (SpTRANS) and SpGEMM, utilizing the segmented merge primitive on two NVIDIA Turing GPUs, an RTX 2080 and a Titan RTX. By testing 956 sparse matrices downloaded from the SuiteSparse Matrix Collection [2], the experimental results show that compared to the NVIDIA cuSPARSE library, our algorithm accelerates performance of SpTRANS and SpGEMM operation by a factor of on average 3.94 and 2.89 (and up to 13.09 and 109.15), respectively.

2 Related Work

In this section, we introduce three existing segmented primitives: segmented sum [1,12], segmented scan [4] and segmented sort [8].

2.1 Parallel Segmented Sum

When a parallel algorithm processes irregular data such as sparse matrices, it is quite common to in parallel deal with arrays of different lengths. For example, in the SpMV operation, multiplying a sparse matrix A and a dense vector x basically equals computing the dot product of every sparse row of A with x. When the number of nonzero entries in the rows are nonuniform, it is easy to encounter the load imbalance issue on parallel processors, and thus to degrade performance [13].

To make the parallel SpMV operation more balanced, Blelloch et al. [1] proposed the segmented sum primitive. The parallel operation has five steps: (1) first gathers intermediate products into an array, (2) labels the indices and values in the same row as a segment, (3) equally assigns the entries to independent threads, (4) then sums the values belonging to the same segment up into one value and saves it , and (5) finally sums the values across multiple threads to finish the operation. Figure 1 plots an example of the parallel segmented sum. To further make the segmented sum suitable for sparse matrices with empty rows, Liu and Vinter proposed a speculative segmented sum [12].

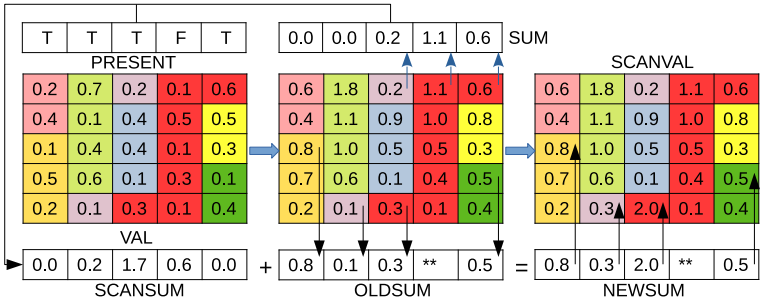


Fig. 1. An example of Blelloch's algorithm using five threads (each processes one column) to find the sum of eight segments (filled with the same color), and uses SUM and PRESENT arrays to store the intermediate results of the split segments.

2.2 Parallel Segmented Scan

The scan (also known as prefix-sum) operation sums all prefixes. For example, the row pointer array of a sparse matrix in the compressed sparse row (CSR) format is the result of scanning an array storing the number of nonzeros of the rows. Parallel segmented scan is to scan multiple segments in parallel, and the result of each segment is the same as that of a single scan. Besides the same load balancing problem as the segmented sum, segmented scan is relatively more complex because of dependencies between the prefixes.

Dotsenko et al. [4] proposed a relatively load balanced segmented scan algorithm by using a novel data structure and reduced the consumption of shared memory on GPU. This algorithm is divided into three steps: (1) stores the segments in an array and divides it into blocks of the same size, (2) each thread scan one segment and rescans when it encounters a new segment, (3) if there is a segment spanning blocks, the result of this segment is propagated. Figure 2 plots an example of the parallel segmented scan.

2.3 Parallel Segmented Sort

The parallel segmented sort operation simultaneously makes keys or key-value pairs in multiple segments ordered. It uses two arrays for storing a list of

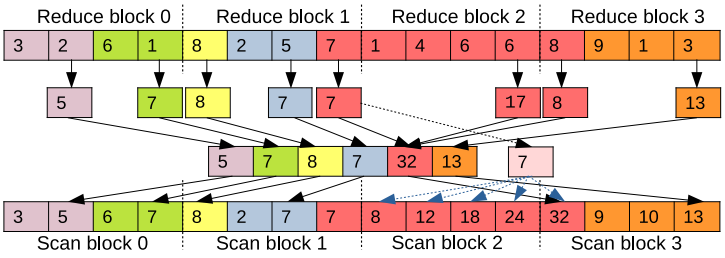


Fig. 2. An example of Dotsenko's algorithm using four threads (each for one block) to find the scan of six segments (filled with the same color)

keys and a group of header pointers of segments. In the procedure, each processing unit obtains the information of the corresponding segment by accessing the segment pointer array, then sorts each segment in serial or in parallel. Because of the nonuniform lengths of the segments, load balancing issues often restrict the efficiency of the algorithm.

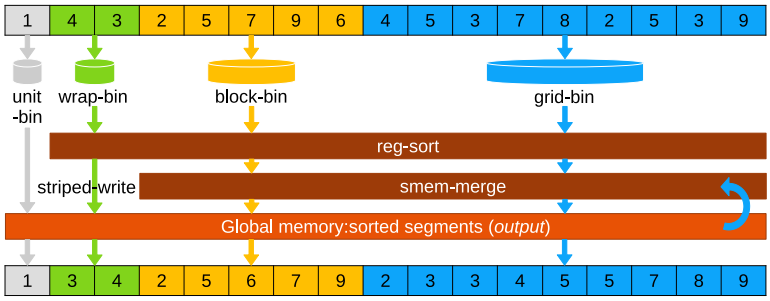


Fig. 3. An example of Hou's algorithm using four bins to store four segments (filled with the same color), and calling reg-sort and smem-merge to sort the segments.

Hou et al. [8] presented an adaptive segmented sort mechanism on GPUs. It proposed a differentiated method for eliminating irregularity in data distribution and a register-based sorting method for accelerating short segments. The register-based sorting algorithm has four steps: (1) gets the amount of tasks per thread, (2) converts data into specific sequence by using shuffle functions, (3) stores data in threads into different registers to perform swap operations, and (4) swaps the data in registers to make them ordered. Figure 3 plots an example of the segmented sort.

3 Segmented Merge and Its Parallel Algorithm

3.1 Definition of Segmented Merge

We first define the segmented merge primitive here. Assuming we have a key-value array

$$S = \{S_1, S_2, \ldots, S_p\}, \tag{1}$$

that includes p segments (i.e., sub–arrays[5]), and further a segment

$$S_i = \{S_{i,1}, S_{i,2}, \ldots, S_{i,q_i}\}, i \in [1, p], \tag{2}$$

contains q_i sub-segments. So we have

$$S = \{S_{1,1}, S_{1,2}, \ldots, S_{1,q_1}, S_{2,1}, S_{2,2}, \ldots S_{2,q_2}, \ldots, S_{p,1}, S_{p,2}, \ldots, S_{p,q_p}\} \tag{3}$$

Each sub-segment $S_{i,j}$ includes $n_{i,j}$ key-value pairs already sorted according to their keys. The objective of the segmented merge operation is to let $n_i = \sum_{j=1}^{q_i} n_{i,j}$ key-value pairs in each segment S_i ordered. Thus S eventually consists of p sorted segments.

3.2 Serial Algorithm for Segmented Merge

A serial segmented merge algorithm can be represented as a multi-way merge, meaning that each sub-segment of the same segment is regarded as a leaf node of a binary tree and is merged in a bottom-up manner. Figure 4 shows an example of merging eight sub-segments in to three segments using the segmented merge.

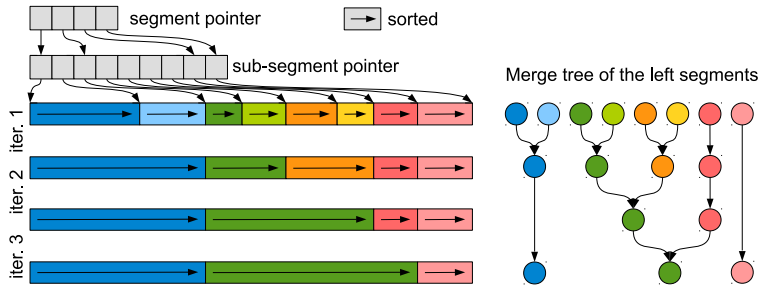


Fig. 4. An example showing the segmented merge algorithm. Here the *segment pointer* (including three segments) points to the *sub-segment pointer* pointing to eight ordered sub-segments continuously stored in the array. After the segmented merge, the eight sub-segments are merged into three ordered segments.

[5] We in this paper call "sub-array" "segment", since each segment further includes at least one "sub-segment". In this way, we can avoid using terms like "sub-array" and "sub-sub-array".

3.3 Simple Parallel Algorithm for Segmented Merge

Algorithm 1 shows a simple parallel pseudocode working in an iterative fashion (lines 3–14): each segment (lines 4–12) can be processed in parallel, and each pair of sub-segments (lines 7–11) can be also merged in parallel until each segment has only one sub-segment.

Algorithm 1 A simple parallel segmented merge algorithm.

```
 1: m_split ← get_split_row_num()
 2: snum ← get_seg_num()
 3: while m_split ≠ snum do
 4:     for each segment in parallel do
 5:         // Challenge 1: The combination of sub-segments of different
 6:           lengths in different segments may bring load imbalance
 7:         for each pair of sub-segments in segment in parallel do
 8:             // Challenge 2: Merging sub-segments of unequal length
 9:               is difficult to vectorize
10:             serial_merge(buff, segment_info, segment_id)
11:         end for
12:     end for
13:     snum ← get_seg_num()
14: end while
```

It can be seen that there are two `for` loops in lines 4 and 7 respectively. Although the two loops can be parallelized straightforward, their simple implementation may bring suboptimal performance. In particular on many-core processors running a large amount of threads, this approach may bring load imbalance problem since the lengths of segments and sub-segments may be imbalanced (see lines 5–6 and 8–9, respectively). This is the first challenge we face. Such performance degradation from irregular data distribution is actually not unusual in sparse matrix algorithms [11,13].

3.4 Improved Parallel Algorithm for Segmented Merge on GPU

We propose a parallel algorithm for segmented merge. The main idea is to fix the number of elements processed by a thread, and to merge the sub-segments in the same segment in the form of binary tree until all sub-segments are merged. The algorithm consists of three steps: (1) data preprocessing, (2) merging two sub-sequences of each thread by using merge path, (3) merging sub-segments on binary tree in an iterative way. Algorithm 2 shows a pseudocode of the parallel segmented merge, and Figure 5 gives an example.

The first step is data preprocessing (lines 4–15 in Algorithm 2). Its objective is to record starting and ending positions of segments and sub-segments. We construct two arrays of size $p + 1$ and $q + 1$, respectively, as two-level pointers (see the top left of Figures 4 and 5). Then, we set each thread's workload to

Algorithm 2 A parallel segmented merge algorithm

```
1: m_split ← get_split_row_num()
2: snum ← get_seg_num()
3: while m_split ≠ snum do
4:     for each pair of sub-segments in segment in parallel do
5:         tnum_local ← get_local_thread_num(segment_ptr, sub-segment_ptr)
6:         tnum_total ← get_total_thread_num(tnum_local)
7:     end for
8:     malloc(thread_info, tnum_total)
9:     for each pair of sub-segments in segment in parallel do
10:        thread_id ← get_thread_id()
11:        scatter_thread_info(thread_info, thread_id)
12:    end for
13:    for each thread in parallel do
14:        gather_thread_info(thread_info, thread_id)
15:    end for
16:    for each thread in parallel do
17:        serial_merge_using_merge_path(buff, thread_info, thread_id)
18:    end for
19:    snum ← get_seg_num()
20:    free(thread_info)
21: end while
```

a fixed amount *nnzpt*, and count the number of threads required in merging every pair of sub-segments (line 5). As a result, each iterative step knows the total number of threads to be issued (line 6). Then each sub-segment scatters a group of information, such as global memory offset and segments size, to threads working on it (lines 9–12). After that, each thread gathers information by running the partitioning strategy of the standard merge path algorithm (lines 13-15).

The second step is merging sub-sequences of each thread by using merge path (lines 16–18 in Algorithm 2). The merge operation of sub-segments is often completed by multiple threads and these threads cooperate with each other and merge sub-sequences through the merge path algorithm [5]. The merge path algorithm is an efficient merge algorithm and has excellent parallelism. In its procedure, each thread is responsible for processing partial sub-sequences without data correlation. First, the data in the sub-sequence corresponding to the thread is compared multiple times to generate boundaries for splitting the two sub-sequences. Then based on the target matrix, the data in the two sub-sequences are placed in the output sequence according to the 'path' to complete the merge operation. In Figure 5, two examples of merging two groups of sub-sequences is shown in the pink dotted box, and the two pairs of sub-sequences are merged with three threads.

The third step is merging sub-segments on the binary tree in an iterative way (the while loop of lines 3–21 in Algorithm 2). According to the information obtained from the above two steps, each sub-segment of the same segment can be seen as a leaf node of a binary tree to merge, and multiple iterations may be

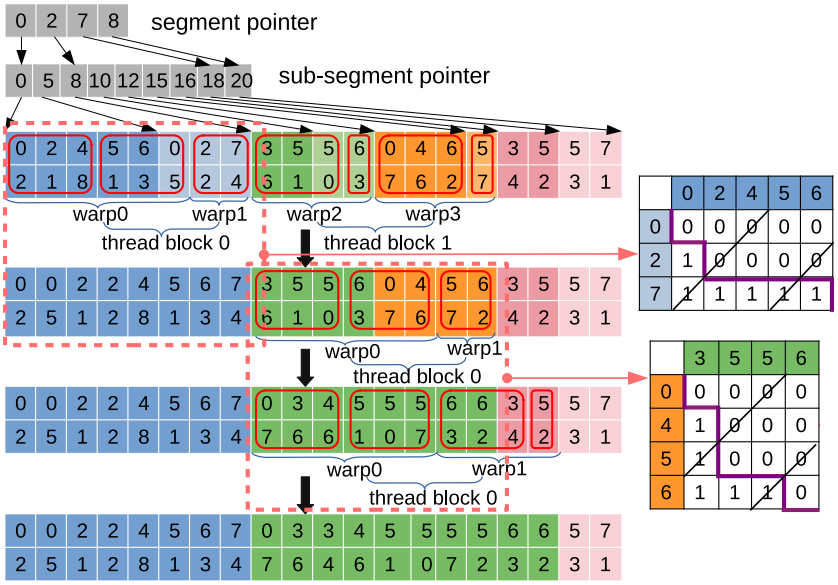


Fig. 5. An example showing the proposed segmented merge algorithm. Here the *segment pointer* (including three segments) points to the *sub-segment pointer* pointing to eight ordered sub-segments continuously stored in the array. There are two levels of arrays, with indexes on the top and values at the bottom. After the segmented merge, the eight sub-segments are merged into three ordered segments. In the computation, two thread blocks of two warps are used.

needed. When the pointers of segment and sub-segment are completely aligned, the iteration ends and the sorting is completed. If there is a single sub-segment, it will not be processed at all in the iteration (see the segment in light pink of Figure 5). When the sub-segments in the same segment are merged, they can be processed by the same warp of 32 threads in CUDA, but the sub segments in different segments can not cross different warps. For example, when combining the blue and light blue sub-segments in Figure 5, although the thread elements do not reach a fixed number, they will not cross warps. So, a large amount of threads can be saved for data in a power-law fashion (i.e., several segments have much more sub-segments than the others). Moreover, because all cores can be saturated, our segmented merge method can achieve good load balancing on massively parallel GPUs.

4 Performance Evaluation

4.1 Experimental Setup

We on two NVIDIA Turing GPUs benchmark the SpTRANS and SpGEMM functions calling the segmented merge primitive proposed, and compare them

with the corresponding functions in the NVIDIA cuSPARSE library v10.2. Table 1 lists the two testbeds and the participating algorithms.

The testbeds	The participating SpTRANS and SpGEMM algorithms
(1) An NVIDIA GeForce RTX 2080 (Turing TU104, 2944 CUDA cores @ 1.8 GHz, 10.59 SP TFlops, 331.2 DP GFlops, 4 MB LLC, 8 GB GDDR6, 448 GB/s bandwidth, driver v440.89). (2) An NVIDIA Titan RTX (Turing TU102, 4608 CUDA cores @ 1.77 GHz, 16.31 SP TFlops, 509.76 DP GFlops, 6 MB LLC, 24 GB GDDR6, 672 GB/s bandwidth, driver v440.89).	(1) The SpTRANS function `cusparse?csr2csc()` in cuSPARSE v10.2. (2) The SpTRANS method using segmented merge proposed in this work. (3) The SpGEMM function `cusparse?csrgemm()` in cuSPARSE v10.2. (4) The SpGEMM algorithm proposed by Liu and Vinter [11]. (5) The SpGEMM method using segmented merge proposed in this work.

Table 1. The testbeds and participating SpTRANS and SpGEMM algorithms.

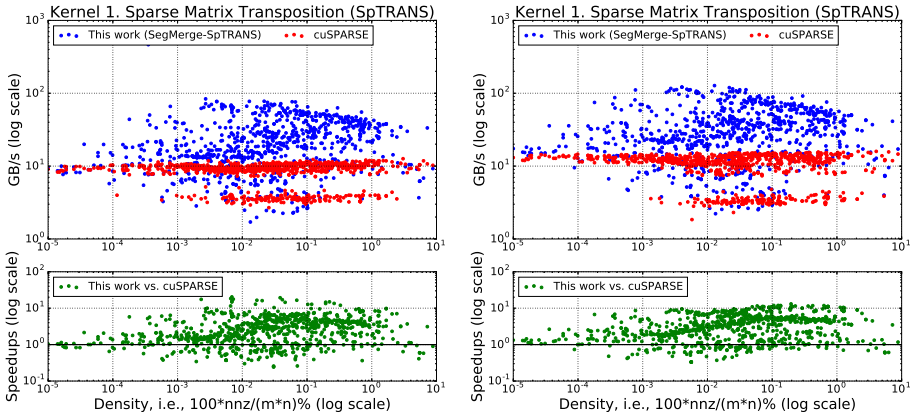
The matrix dataset is downloaded from the SuiteSparse Matrix Collection (formerly known as the University of Florida Sparse Matrix Collection [2]). We select all 956 relatively large matrices of no less than 100,000 and no more than 200,000,000 nonzero entries for the experiment.

4.2 Performance of SpTRANS using Segmented Merge

The SpTRANS operation transpose a sparse matrix A in the CSR format to its transpose A^T also in the CSR format. From the data structure point of view, the operation is the same as converting A's CSR format to its compressed sparse column (CSC) format. Wang et al. [15] proposed two vectorized and load balanced SpTRANS algorithms for x86 processors.

To utilize higher computational power and bandwidth on GPUs, we design a new SpTRANS algorithm using segmented merge proposed in this work. Specifically, the number of nonzeros are calculated firstly, then the nonzeros are inserted into the transpose matrix in an unordered way using atomic operation. To keep the indices of the nonzeros in each row of the transpose sorted, the segmented merge primitive is called. For the long rows, they are cut into smaller pieces that can be sorted independently by utilizing on-chip shared memory. Then the long rows can be seen as segments, and the sorted pieces are processed as subsegments. Thus the segmented merge primitive can be used naturally.

Figure 6 shows the abstract performance (in GB/s) and relative speedups of SpTRANS in cuSPARSE and using our segmented merge method on NVIDIA RTX 2080 and Titan RTX GPUs. It can be seen that in most cases our method is faster than cuSPARSE. The average speedup on the two GPUs can reach 3.55x (up to 9.64x) and 3.94x (up to 13.09x), respectively. For matrices with many short rows and balanced distribution, such as *igbt3* matrix, our algorithm dynamically determines the optimal number of threads to be used in each iteration. Therefore, our algorithm has a good performance for *igbt3* matrix and achieves the speedup of 13.09x over cuSPARSE.



(a) RTX 2080 (b) Titan RTX

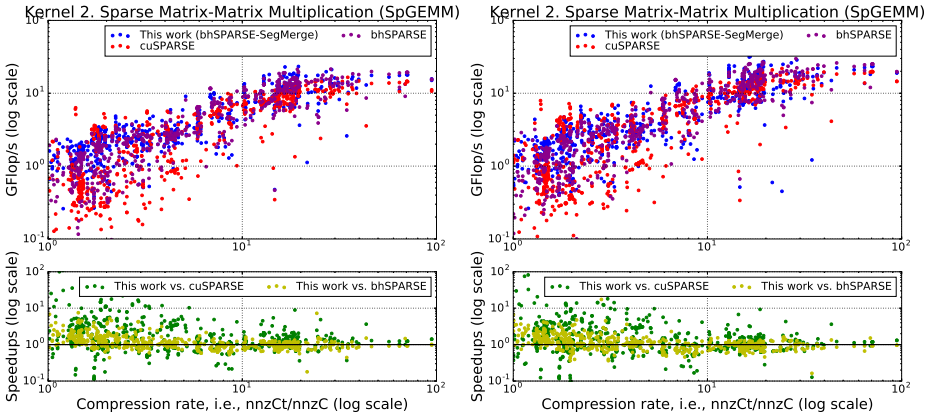
Fig. 6. Comparison of two SpTRANS methods in cuSPARSE and using the segmented merge primitive on two NVIDIA GPUs. The x-axis represents the density (the ratio of the number of nonzeros to the multiply of the number of rows and columns) of the matrices tested.

4.3 Performance of SpGEMM using Segmented Merge

The SpGEMM operation computes $C = AB$, where the three matrices are all sparse. The most used fundamental approach for SpGEMM is the row-row method proposed by Gustavson [7]. Its parallel implementation can be straightforward. Each thread traverses the nonzeros of a row of A, and uses the values to scale all entries of the corresponding rows of B, then merges the scaled entries into the row of C. The function for this procedure is also called sparse accumulator and has been studied by much research [3,9,10,11,14,16].

The SpGEMM algorithm tested is an improved version of the SpGEMM approach in bhSPARSE developed by Liu and Vinter [11]. In the original implementation, the authors assign the workload for computing the rows of C to 37 bins according to the floating point operations needed for the rows. The first 36 bins process relatively short rows, and the last bin is designed to process the rows of a large mount of nonzeros. For the rows in the last bin, the entries cannot be placed into on-chip shared memories, thus global memory has to be used. Hence the segmented merge is used for calculating the long rows, by first saving the scaled rows from B (as sub-segments) onto global memory and then merging the sub-segments belonging to the same row of C (as a segment). Note that all the rows, i.e., segments, in the last bin are involved in one segmented merge computation.

Figure 7 plots the performance of SpGEMM in cuSPARSE, bhSPARSE and bhSPARSE with segmented merge on NVIDIA RTX 2080 and Titan RTX GPUs. It can be seen that the performance of our method is significantly better than



(a) RTX 2080 (b) Titan RTX

Fig. 7. Comparison of three SpGEMM methods computing A^2: cuSPARSE, bhSPARSE [11] and bhSPARSE using segmented merge. The x-axis is compression rate, i.e., the ratio of the number of intermediate nonzeros to nonzeros in C.

cuSPARSE and bhSPARSE. On RTX 2080, the speedups over the two methods reach on average 2.89x (up to 109.15x) and 1.26x (up to 7.5x), respectively. On Titan RTX, the speedups are on average 2.53x (up to 81.85x) and 1.22x (up to 17.38x), respectively. Taking the *webbase-1M* matrix with many long rows as an example, cuSPARSE cannot evenly distribute the data to cores, and bhSPARSE can only use one thread block for each long row, meaning the two libraries actually underuse the GPUs. But because our algorithm avoids dealing with a long row in a thread by evenly dividing all elements to thread, our algorithm obtain 5.85x and 2.31x speedups over cuSPARSE and bhSPARSE, respectively.

5 Conclusion

In this paper, we have defined a new primitive called segmented merge, and presented an efficient parallel algorithm achieving good load balancing and SIMD unit utilization on GPUs. The experimental results show that two sparse matrix algorithms, SpTRANS and SpGEMM, using our parallel segmented merge are greatly faster than existing methods in cuSPARSE and bhSPARSE.

Acknowledgments

We would like to thank the invaluable comments from all the reviewers. Weifeng Liu is the corresponding author of this paper. This research was supported by the Science Challenge Project under Grant No. TZZT2016002, the National Natural Science Foundation of China under Grant No. 61972415, and

the Science Foundation of China University of Petroleum, Beijing under Grant No. 2462019YJRC004, 2462020XKJS03.

References

1. Blelloch, G.E., Heroux, M.A., Zagha, M.: Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors. Tech. rep., CMU (1993)
2. Davis, T.A., Hu, Y.: The University of Florida Sparse Matrix Collection. ACM Trans. Math. Softw. 38(1), 1:1–1:25 (2011)
3. Deveci, M., Trott, C., Rajamanickam, S.: Multithreaded Sparse Matrix-Matrix Multiplication for Many-Core and GPU Architectures. Parallel Computing. 78, 33–46 (2018)
4. Dotsenko, Y., Govindaraju, N.K., Sloan, P.P., Boyd, C., Manferdelli, J.: Fast Scan Algorithms on Graphics Processors. In: Proceedings of the 22nd Annual International Conference on Supercomputing. pp. 205–213. ICS '08 (2008)
5. Green, O., McColl, R., Bader, D.A.: GPU Merge Path: A GPU Merging Algorithm. In: Proceedings of the 26th ACM International Conference on Supercomputing. pp. 331–340. ICS '12 (2012)
6. Gremse, F., Küpper, K., Naumann, U.: Memory-Efficient Sparse Matrix-Matrix Multiplication by Row Merging on Many-Core Architectures. SIAM Journal on Scientific Computing 40(4), C429–C449 (2018)
7. Gustavson, F.G.: Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. ACM Trans. Math. Softw. 4(3), 250–269 (Sep 1978)
8. Hou, K., Liu, W., Wang, H., Feng, W.c.: Fast Segmented Sort on GPUs. In: Proceedings of the International Conference on Supercomputing. ICS '17 (2017)
9. Liu, J., He, X., Liu, W., Tan, G.: Register-Based Implementation of the Sparse General Matrix-Matrix Multiplication on GPUs. In: Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 407–408. PPoPP '18 (2018)
10. Liu, J., He, X., Liu, W., Tan, G.: Register-Aware Optimizations for Parallel Sparse Matrix-Matrix Multiplication. International Journal of Parallel Programming pp. 403–417 (2019)
11. Liu, W., Vinter, B.: A Framework for General Sparse Matrix-matrix Multiplication on GPUs and Heterogeneous Processors. Journal of Parallel and Distributed Computing (2015)
12. Liu, W., Vinter, B.: Speculative Segmented Sum for Sparse Matrix-Vector Multiplication on Heterogeneous Processors. Parallel Computing. pp. 179–193 (2015)
13. Liu, W., Vinter, B.: CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In: Proceedings of the 29th ACM on International Conference on Supercomputing. pp. 339–350. ICS '15 (2015)
14. Nagasaka, Y., Nukada, A., Matsuoka, S.: High-Performance and Memory-Saving Sparse General Matrix-Matrix Multiplication for NVIDIA Pascal GPU. In: 2017 46th International Conference on Parallel Processing (ICPP). pp. 101–110 (2017)
15. Wang, H., Liu, W., Hou, K., Feng, W.c.: Parallel Transposition of Sparse Data Structures. In: Proceedings of the 2016 International Conference on Supercomputing. pp. 33:1–33:13. ICS '16 (2016)
16. Xie, Z., Tan, G., Liu, W., Sun, N.: IA-SpGEMM: An Input-Aware Auto-Tuning Framework for Parallel Sparse Matrix-Matrix Multiplication. In: Proceedings of the ACM International Conference on Supercomputing. pp. 94–105. ICS '19 (2019)